

CM2Y-MAN-PGB-M5049-R04C0

**CMS-2Y PROGRAMMER'S
REFERENCE MANUAL
FOR THE AN/UYK-7 AND
AN/UYK-43 COMPUTERS**

FLEET COMBAT DIRECTION
SYSTEMS SUPPORT ACTIVITY
San Diego, California 92147

REVISION NOTICE 4

THIS PUBLICATION REPLACES M-5049, DATED
15 FEBRUARY 1984 AND ALL CHANGES THERETO.
THE SUPERSEDED PUBLICATION SHOULD BE
REMOVED FROM FILES AND DESTROYED.

TECHNICAL LIBRARY (TOPSIDE)
NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CA 92162

1 OCTOBER 1986

Release approved
This document is required for official use or
administrative or operational purposes. Distri-
bution is limited to US Agencies only. Other
requests for this document must be referred
to: FCDSSA, San Diego, CA 92147-5081

This document reflects
CMS-2Y Revision 16.

121

/ (U) CM2Y-MAN-PGR-M5049-R04C0

Document No. MAN5049
Config. ID No. / (U) CM2Y-MAN-PGR-M5049-R04C0
Date Of Original Issue 1 December 1978
Revision/Change No. 4/0
Date Of Revision/Change 1 October 1986
THIS DOCUMENT UNDER CONFIGURATION CONTROL

CMS-2Y PROGRAMMER'S REFERENCE MANUAL
FOR THE AN/UYK-7 AND AN/UYK-43 COMPUTERS

Prepared By:

FLEET COMBAT DIRECTION SYSTEMS SUPPORT ACTIVITY
CODE 8
SAN DIEGO, CA 92147-5081

APPROVAL: (Signature)

David D. Peterson for Robin K. Gillett
Originator

[Signature]
Reviewing Authority

[Signature]
Approving Authority

LIST OF EFFECTIVE PAGES

Insert latest changed pages; dispose of superseded pages in accordance with applicable standards.

NOTE: On a changed page, the portion of the text affected by the latest change is indicated by a vertical line in the outer margin of the page.

Total number of pages in this manual is 648 consisting of the following:

Page No.	# Change No.	Page No.	# Change No.
Title	0	6-120 Blank	0
A	0	7-1 - 7-9	0
i	0	7-10 Blank	0
ii Blank	0	8-1 - 8-11	0
iii	0	8-12 Blank	0
iv Blank	0	9-1 - 9-52	0
v - xiv	0	10-1 - 10-5	0
1-1 - 1-7	0	10-6 Blank	0
1-8 Blank	0	A-1 - A-28	0
2-1 - 2-12	0	B-1 - B-42	0
3-1 - 3-19	0	C-1 - C-8	0
3-20 Blank	0	D-1 - D-46	0
4-1 - 4-133	0	E-1 - E-7	0
4-134 Blank	0	E-8 Blank	0
5-1 - 5-82	0	X-1 - X-43	0
6-1 - 6-119	0	X-44 Blank	0

#Zero in this column indicates an original page.

RECORD OF CHANGES

CHANGE NUMBER	DATE	ABSTRACT OF CHANGE
Original	1 December 1978	
Revision 1	15 April 1981	
Revision 2	30 September 1981	
Revision 3	15 February 1984	
Revision 4	1 October 1986	

ABSTRACT

This document contains the information required to use the Compiler Monitor System-2 (CMS-2Y) which operates on the AN/UYK-7 computer and generates code for the AN/UYK-7 and AN/UYK-43 computers. This capability is referred to as CMS-2Y(7) throughout this manual.

CM2Y-MAN-PGR-M5049-R04C0 contains a comprehensive description of the CMS-2Y(7) language statements and their usage. A basic knowledge of both programming and AN/UYK-7 and AN/UYK-43 computer characteristics has been assumed.

The "CMS-2Y Programmer's Reference Manual for the AN/UYK-7 and AN/UYK-43 Computers" is an unclassified document produced by the Systems Programming and Production Services Department at the Fleet Combat Direction Systems Support Activity, San Diego (FCDSSA San Diego).

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
1	INTRODUCTION.....	1-1
1.1	Purpose and Scope.....	1-1
1.2	Applicable Documents.....	1-2
1.3	Conventions.....	1-3
1.3.1	Semantic Conventions.....	1-3
1.3.2	Symbolic Conventions.....	1-5
1.4	Section Summary.....	1-6
2	FUNDAMENTAL CONCEPTS.....	2-1
2.1	Program Format.....	2-1
2.2	Comments.....	2-2
2.3	Spaces and Notes.....	2-4
2.4	Modes and Types.....	2-5
2.4.1	Modes and Simple Types.....	2-5
2.4.2	Universal Type.....	2-7
2.4.3	Structured Types.....	2-8
2.5	Scopes and Scope Rules.....	2-9
2.6	Input/Output and Files.....	2-10
2.7	Debugging Aids.....	2-12
3	BASIC CONSTRUCTS.....	3-1
3.1	Characters.....	3-2
3.1.1	Letters.....	3-3
3.1.2	Digits.....	3-4
3.1.3	Delimiters.....	3-5
3.2	Strings.....	3-6
3.3	Names.....	3-7
3.4	Constants.....	3-9
3.4.1	Numeric Constants.....	3-10
3.4.1.1	Decimal Constants.....	3-11
3.4.1.2	Octal Constants.....	3-14
3.4.2	Boolean Constants.....	3-16
3.4.3	Character Constants.....	3-17
3.4.4	Status Constants.....	3-18
3.5	Direct Code Block.....	3-19
4	DECLARATIONS.....	4-1
4.1	Declaration Modifiers.....	4-2
4.2	Constant Mode Declaration.....	4-5
4.3	Simple Type Specification.....	4-7
4.3.1	Numeric Types.....	4-8
4.3.2	Boolean Type.....	4-12

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
4.3.3	Character Type.....	4-13
4.3.4	Status Type.....	4-14
4.4	Type Declarations.....	4-15
4.5	Default Type Specifications.....	4-20
4.6	Variable Declaration.....	4-24
4.7	Parameter Variable Declaration.....	4-29
4.8	Table Declaration.....	4-32
4.8.1	Field Declaration.....	4-39
4.8.2	Field Overlay Declaration.....	4-44
4.8.3	Like-Table Declaration.....	4-47
4.8.4	Subtable Declaration.....	4-49
4.8.5	Item-Area Declaration.....	4-54
4.9	Array Declaration.....	4-56
4.10	Preset Value Declaration.....	4-60
4.11	Overlay Declaration.....	4-63
4.12	Text Substitution Declaration.....	4-66
4.13	Compile-Time Constant Declaration.....	4-68
4.14	Load-Time Variable Declaration.....	4-72
4.15	Address Declaration.....	4-75
4.16	System Index Declaration.....	4-77
4.17	Local Index Declaration.....	4-78
4.18	Procedure Declaration.....	4-79
4.19	Executive Procedure Declaration.....	4-82
4.20	Function Declaration.....	4-84
4.21	Label Switch Declaration.....	4-86
4.21.1	Indexed Label Switch Declaration.....	4-87
4.21.2	Double Label Switch Declaration.....	4-89
4.21.3	Item Label Switch Declaration.....	4-91
4.22	Procedure Switch Declarations.....	4-93
4.22.1	Indexed Procedure Switch Declaration.....	4-94
4.22.2	Double Procedure Switch Declaration.....	4-96
4.22.3	Item Procedure Switch Declaration.....	4-98
4.23	File Declaration.....	4-101
4.24	Format Declaration.....	4-108
4.24.1	Interpretation of Format Items.....	4-110
4.24.1.1	Format Descriptors.....	4-110
4.24.1.2	Numeric Conversion (I, O, F, and E Types).....	4-111
4.24.1.3	Character Conversion (A and L Types).....	4-112
4.24.1.4	Character Constant Format Item.....	4-113
4.24.1.5	Format Positioners.....	4-113
4.25	Stringform Declaration.....	4-116
4.25.1	Interpretation of Stringform Items.....	4-117
4.25.1.1	Stringform Descriptors.....	4-118
4.25.1.2	D-Type Conversion, Internal to Character... ..	4-118
4.25.1.3	D-Type Conversion, Character to Internal... ..	4-119
4.25.1.4	I-Type Conversion, Internal to Character... ..	4-119
4.25.1.5	I-Type Conversion, Character to Internal... ..	4-119

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
4.25.1.6	B-Type, O-Type, and X-Type Conversions, Internal to Character.....	4-120
4.25.1.7	B-Type, O-Type, and X-Type Conversions, Character to Internal.....	4-120
4.25.1.8	C-Type Conversion, Internal to Character...	4-121
4.25.1.9	C-Type Conversion, Character to Internal...	4-121
4.25.1.10	E-Type Conversion, Internal to Character...	4-122
4.25.1.11	E-Type Conversion, Character to Internal...	4-122
4.25.1.12	Stringform Positioners.....	4-122
4.25.1.13	Z-Type Positioning.....	4-122
4.25.1.14	T-Type Positioning.....	4-123
4.25.1.15	Character Constant Conversion.....	4-123
4.26	Inputlist Declaration.....	4-125
4.27	Outputlist Declaration.....	4-127
4.28	Debug Enabling Declaration.....	4-129
4.29	Range Declaration.....	4-131
5	DATA REFERENCES.....	5-1
5.1	Data Unit.....	5-2
5.1.1	Single-Valued Data Unit.....	5-3
5.1.1.1	Restrictions on Forms.....	5-4
5.1.1.2	Attributes of a Single-Valued Data Unit...	5-4
5.1.2	Multivalued Data Unit.....	5-6
5.1.3	Word Data Unit.....	5-7
5.1.3.1	Restrictions on Forms.....	5-7
5.1.3.2	Word Specification.....	5-8
5.1.3.3	Resolution of Ambiguity.....	5-8
5.1.4	Modified Data Unit.....	5-9
5.1.4.1	Bit Modified Data Unit.....	5-10
5.1.4.2	Character Modified Data Unit.....	5-12
5.2	Function Reference.....	5-14
5.2.1	User Function Reference.....	5-15
5.2.2	Intrinsic Function Reference.....	5-17
5.2.2.1	Absolute Value Function Reference.....	5-18
5.2.2.2	Core Address Function Reference.....	5-19
5.2.3	Predefined Function Reference.....	5-21
5.2.3.1	Floating-Point Arithmetic Function Reference.....	5-22
5.2.3.2	Fixed-Point Arithmetic Function Reference..	5-25
5.2.3.3	Status Operation Function Reference.....	5-28
5.2.3.3.1	Successor Function Reference.....	5-29
5.2.3.3.2	Predecessor Function Reference.....	5-30
5.2.3.3.3	Initial Value Function Reference.....	5-31
5.2.3.3.4	Final Value Function Reference.....	5-32
5.2.3.4	Bit String Function References.....	5-33
5.2.3.5	Scaling Specification Function Reference...	5-36

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
5.2.3.6	Conversion Function Reference.....	5-38
5.2.3.7	Temporary Definition Function Reference....	5-40
5.2.3.8	Remaindering Function Reference.....	5-42
5.2.3.9	Bit Count Function Reference.....	5-44
5.2.3.10	Subfile Number Function Reference.....	5-45
5.2.3.11	Subfile Position Function Reference.....	5-46
5.2.3.12	Record Length Function Reference.....	5-47
5.3	Expressions.....	5-48
5.3.1	Numeric Expression.....	5-49
5.3.1.1	Expression Evaluation.....	5-50
5.3.1.2	Numeric Conversions.....	5-50
5.3.1.3	Fixed-Point Scaling Algorithm.....	5-51
5.3.1.3.1	Symbols Used in Scaling Algorithm.....	5-52
5.3.1.3.2	The Value of the Scaling Controller.....	5-53
5.3.1.3.3	Results of Binary Operations.....	5-53
5.3.1.3.4	Floating-Point Arithmetic.....	5-54
5.3.1.4	Sign of Fixed-Point Operations.....	5-55
5.3.1.5	Constant Arithmetic.....	5-55
5.3.1.6	MSCALE Scaling Algorithm.....	5-58
5.3.1.6.1	Integer Arithmetic Scaling Algorithm.....	5-58
5.3.1.6.2	Fixed-Point Arithmetic Scaling Algorithm.....	5-58
5.3.1.7	Numeric Constant Expression.....	5-62
5.3.1.8	Numeric Constant Value.....	5-63
5.3.2	Boolean Expression.....	5-64
5.3.2.1	Expression Evaluation.....	5-65
5.3.2.2	Meaning Of Operators.....	5-65
5.3.2.3	Numeric Relational Expression.....	5-67
5.3.2.4	Boolean Relational Expression.....	5-69
5.3.2.5	Character Relational Expression.....	5-71
5.3.2.6	Status Relational Expression.....	5-73
5.3.2.7	Conditional Expression.....	5-74
5.3.2.8	Conditional I/O Expression.....	5-76
5.3.3	Character Expression.....	5-77
5.3.4	Status Expression.....	5-78
5.3.5	Bit String Expression.....	5-79
5.3.5.1	Expression Evaluation.....	5-79
5.3.5.2	Meaning of Operators.....	5-80
5.3.6	Structured Expression.....	5-82
6	STATEMENTS.....	6-1
6.1	Simple Phrases.....	6-3
6.1.1	Imperative Phrases.....	6-4
6.1.1.1	Assignment Phrase (Classes and Compatibility).....	6-5
6.1.1.1.1	Simple Assignment.....	6-7
6.1.1.1.2	Untyped Assignment.....	6-14

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
6.1.1.1.3	Word Assignment.....	6-15
6.1.1.1.4	Value Flush Assignment.....	6-16
6.1.1.1.5	Multivalued Assignment.....	6-16
6.1.1.2	Swap Phrase.....	6-18
6.1.1.3	Branch Phrase.....	6-20
6.1.1.4	Indexed Branch Phrase.....	6-22
6.1.1.5	Item Branch Phrase.....	6-25
6.1.1.6	Procedure Call Phrase.....	6-28
6.1.1.6.1	User Procedure Call Phrase (Parameter Passage Style.....	6-29
6.1.1.6.2	Supplied Procedure Call Phrase.....	6-34
6.1.1.7	Indexed Procedure Call Phrase.....	6-38
6.1.1.8	Item Procedure Call Phrase.....	6-41
6.1.1.9	Stop Phrase.....	6-43
6.1.1.10	Return Phrase.....	6-45
6.1.1.11	Exit Phrase.....	6-48
6.1.1.12	Resume Phrase.....	6-50
6.1.1.13	Executive Call Phrase.....	6-52
6.1.1.14	Shift Phrase.....	6-53
6.1.1.15	Open Phrase.....	6-56
6.1.1.16	Close Phrase.....	6-58
6.1.1.17	Endfile Phrase.....	6-59
6.1.1.18	Define Label Phrase.....	6-60
6.1.1.19	Check Label Phrase.....	6-61
6.1.1.20	File Positioning Phrase.....	6-62
6.1.1.21	Record Positioning Phrase.....	6-64
6.1.1.22	Output Phrase.....	6-66
6.1.1.22.1	Extended Subscript Data Unit.....	6-67
6.1.1.22.2	The Format Scan.....	6-69
6.1.1.22.3	Output to the Printer.....	6-71
6.1.1.22.4	Record Size with Unformatted Input and Output.....	6-72
6.1.1.23	Input Phrase.....	6-74
6.1.1.24	Encode Phrase.....	6-76
6.1.1.25	Decode Phrase.....	6-78
6.1.1.26	Convertin Phrase.....	6-79
6.1.1.26.1	Run-Time Stringforms.....	6-80
6.1.1.27	Convertout Phrase.....	6-82
6.1.1.28	Display Phrase.....	6-84
6.1.1.29	Snap Phrase.....	6-87
6.1.1.30	Trace Phrase.....	6-89
6.1.1.31	End-Trace Phrase.....	6-91
6.1.1.32	Null Phrase.....	6-92
6.1.2	Statement Blocks.....	6-93
6.1.2.1	Begin Block.....	6-94
6.1.2.2	Loop Block.....	6-96
6.1.2.3	Case Block.....	6-105

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
6.1.2.4	End Phrase.....	6-110
6.2	Conditional Statements.....	6-112
6.2.1	If Statement.....	6-113
6.2.2	Find Statement.....	6-116
7	SUBPROGRAMS.....	7-1
7.1	Subprogram Block.....	7-2
7.1.1	Procedure Block.....	7-3
7.1.2	Executive Procedure Block.....	7-4
7.1.3	Function Block.....	7-5
7.2	Subprogram Body.....	7-7
7.2.1	Subprogram Data Block.....	7-8
8	SYSTEM ELEMENTS.....	8-1
8.1	System Data Element.....	8-2
8.2	System Procedure Element.....	8-4
8.3	Local Data Block.....	8-6
8.4	Automatic Data Declaration.....	8-8
8.5	Minor Header.....	8-10
9	COMPILATION MODULES.....	9-1
9.1	Major Header.....	9-3
9.2	Options Declarations.....	9-6
9.2.1	Source Specification.....	9-10
9.2.2	Object Specification.....	9-12
9.2.3	Listing Specification.....	9-16
9.2.4	Message Level Specification.....	9-18
9.2.5	Monitor Specification.....	9-19
9.2.6	Nonreal-Time Specification.....	9-20
9.2.7	Structured Specification.....	9-21
9.2.8	Mode Variable Specification.....	9-22
9.2.9	Scaling Specification.....	9-23
9.3	Compiler Directives.....	9-24
9.3.1	Parameter Passage Directive.....	9-25
9.3.1.1	Routine Linkage.....	9-26
9.3.1.2	Function Value Return.....	9-26
9.3.1.3	Direct Passage.....	9-26
9.3.1.4	Register Passage Algorithm.....	9-26
9.3.1.5	Register Passage, Calling Only.....	9-27
9.3.2	Single Precision Directive.....	9-28
9.3.3	Executive Directive.....	9-29
9.3.4	Spill Directive.....	9-30
9.3.5	Pooling Directive.....	9-31
9.4	Address Counter Separation Declaration.....	9-35

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
9.5	Compiler Input and Output Files.....	9-37
9.5.1	ISCM File Elements.....	9-38
9.5.2	Compiler Input ISCM Files.....	9-38
9.5.2.1	Library Declaration.....	9-39
9.5.2.2	Source Retrieval Declaration.....	9-40
9.5.2.3	Compool Retrieval Declaration.....	9-44
9.5.3	Compiler Output ISCM Files.....	9-46
9.5.3.1	ISCM File Specification With The Options Declaration.....	9-46
9.5.3.2	Key Specification.....	9-48
9.5.3.3	Dependent Element Declaration.....	9-51
10	CONDITIONAL COMPILATION.....	10-1
10.1	Conditional Compilation Brackets.....	10-2
10.2	Compilation Selection Directives.....	10-4
10.3	Cswitch Delete Declaration.....	10-5
APPENDIX A	ERROR AND WARNING MESSAGES.....	A-1
A.1	Source Error and Source Warning Messages.....	A-1
A.2	Library Retrieval Diagnostic Messages.....	A-19
A.3	Object Error and Object Warning Messages.....	A-21
A.4	Reference Listings Error Messages.....	A-26
A.5	Other Errors.....	A-28
A.5.1	Compiler Phase Errors.....	A-28
A.5.2	Allocation Errors.....	A-28
APPENDIX B	DIRECT CODE.....	B-1
B.1	Basic Constructs.....	B-2
B.1.1	Direct Code Characters.....	B-2
B.1.2	Delimiters.....	B-3
B.1.3	Names.....	B-4
B.1.3.1	Text Substitution Declaration Names.....	B-4
B.1.3.2	Compile-Time Constant Declaration Names.....	B-4
B.1.3.3	Load-Time Variable Declaration.....	B-4
B.1.3.4	System and Local Index Names.....	B-4
B.1.4	Operation Codes.....	B-5
B.1.5	Direct Code Constants.....	B-15
B.1.5.1	Direct Code Numeric Constants.....	B-16
B.1.5.2	Direct Code Character Constants.....	B-21
B.1.5.3	Direct Code Literal.....	B-22
B.2	Direct Code Expressions.....	B-23
B.2.1	Direct Code Numeric Constant Expressions.....	B-24
B.2.2	Direct Code Address Expressions.....	B-25
B.3	Direct Code Statements.....	B-27

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
B.3.1	Direct Code Name.....	B-28
B.3.2	Addressable Direct Code Statement.....	B-30
B.3.2.1	Direct Code Instruction.....	B-31
B.3.3	Direct Code Preset.....	B-32
B.3.4	Direct Code Directives.....	B-35
B.3.4.1	Abs Directive.....	B-36
B.3.4.2	Byte Directive.....	B-37
B.3.4.3	Char Directive.....	B-38
B.3.4.4	Do Directive.....	B-39
B.3.4.5	Form Directive.....	B-40
B.3.4.6	Res Directive.....	B-41
B.3.5	Form Preset.....	B-42
APPENDIX C	TARGET MACHINE INTERFACES.....	C-1
C.1	Compiled Forms of Inputlist and Outputlist Declarations.....	C-1
C.1.1	Control Words.....	C-3
C.2	Compiled Form of Stringforms.....	C-7
APPENDIX D	LISTING FORMATS.....	D-1
D.1	Source Listings.....	D-2
D.1.1	Compiler Source Listing.....	D-3
D.1.2	Source Listing.....	D-5
D.2	Object Listing.....	D-8
D.2.1	Compiler Diagnostic Listing.....	D-9
D.2.2	Symbol Analysis Listing (SA).....	D-11
D.2.2.1	Files.....	D-16
D.2.2.2	Formats.....	D-17
D.2.2.3	Types.....	D-18
D.2.2.4	Tables.....	D-19
D.2.2.5	Switches.....	D-21
D.2.2.6	Variables.....	D-22
D.2.2.7	Inputlists/Outputlists.....	D-23
D.2.2.8	Stringforms.....	D-24
D.2.2.9	Procedures and Functions.....	D-25
D.2.2.10	Local Indexes.....	D-26
D.2.3	Source Mnemonic Listing.....	D-27
D.2.4	Local Address Cross Reference Listing (CR,CRL).....	D-30
D.2.5	Local Source Cross Reference Listing (SCR,SCRL).....	D-33
D.2.6	Global Address Cross Reference (CR, CRG).....	D-36
D.2.7	Global Source Cross Reference (SCR, SCRG).....	D-38
D.2.8	Compile Summary.....	D-41
D.3	System Listings.....	D-43

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
D.3.1	SHARE/7 System Summary.....	D-43
D.3.2	Batch System Summary.....	D-45
APPENDIX E	FORMAT OF THE SYMBOL ANALYSIS DUMP.....	E-1
INDEX OF SYNTAX SYMBOLS.....		X-1

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
1-01	Narrative Symbols.....	1-4
4-01	CMODE Declaration Examples.....	4-6
4-02	Allocation of Fields with Compiler-Specified Packing.....	4-19
4-03	Allocation of Typed Variables.....	4-26
4-04	Table Storage Addressing Sequence.....	4-35
4-05	Vertical Table Layout (Table TEST).....	4-43
4-06	Internal Structure of Subtable HORIZST.....	4-52
4-07	Internal Structure of Subtable VERTST.....	4-53
4-08	Parent Table Relationships.....	4-55
4-09	A 3-Dimensional Array.....	4-59
5-01	Floating-Point Arithmetic Conversions.....	5-51
5-02	Boolean Operators.....	5-66
6-01	Simple Assignment Operation Types.....	6-17
6-02	Example of Bit Assignments for the Display Phrase.....	6-86
6-03	Vary Block Control Flow.....	6-102
9-01	Some Options Parameter Combinations and Results..	9-9
D-01	An Example of a Compiler Source Listing.....	D-4
D-02	An Example of a Source Listing.....	D-6
D-03	An Example of a Compiler Diagnostic Listing.....	D-10
D-04	An Example of a Symbol Analysis Listing.....	D-12
D-05	An Example of a Source Mnemonic Listing.....	D-28
D-06	An Example of a Local Address Cross Reference Listing.....	D-31
D-07	An Example of a Local Source Cross Reference Listing.....	D-34
D-08	An Example of a Global Address Cross Reference Listing.....	D-37
D-09	An Example of a Global Source Cross Reference Listing.....	D-39
D-10	An Example of a Compile Summary.....	D-42
D-11	An Example of a SHARE/7 System Summary.....	D-44
D-12	An Example of a Batch System Summary.....	D-46

SECTION 1. INTRODUCTION

1.1 Purpose and Scope

This programmer's reference manual describes the syntax and semantics of the CMS-2Y(7) language. The description is stated in terms of the syntactic forms permissible in a correctly written program and the effect of executing such a program. The meanings of incorrect programs cannot be inferred from this description.

This manual assumes that the reader has some prior knowledge of both programming and AN/UYK-7 and AN/UYK-43 computer characteristics. It should be viewed as a language reference manual, not as a CMS-2Y primer nor as a tutorial text.

1.2 Applicable Documents

The following documents complement this manual to provide a complete description of CMS-2Y(7) interfaces:

- a. CM2Y-MAN-CPC-M5040-R00C0 "CP-642A/B Computer Characteristics," FCDSSA, San Diego, 1 May 1979.
- b. CM2Y-MAN-PGR-M5044-R01C0 "CMS-2Y Programmer's Reference Manual for the Transferable Subset," FCDSSA, San Diego, 1 October 1986.
- c. CM2Y-MAN-PGR-M5045-R04C0 "CMS-2Y Programmer's Reference Manual for the AN/UYK-20 and AN/AYK-14 Computers," FCDSSA, San Diego, 1 October 1986.
- d. CM2Y-MAN-CPC-M5046-R00C3 "AN/UYK-20 Computer Characteristics," FCDSSA, San Diego, 1 October 1986.
- e. CM2Y-MAN-PGR-M5047-R01C0 "CMS-2Y Programmer's Reference Manual for the CP-642 Computer," FCDSSA, San Diego, 1 October 1986.
- f. CM2Y-MAN-CPC-M5048-R00C1 "AN/UYK-7 Computer Characteristics," FCDSSA, San Diego, 15 April 1981.
- g. CM2Y-MAN-PGR-M5050-R01C3 "CMS-2Y Supporting Subsystems," FCDSSA, San Diego, 1 October 1986.
- h. CM2Y-MAN-PGR-M5131-R02C0 "Assemblers and Macro Assemblers," FCDSSA, San Diego, 1 October 1986.
- i. CM2Y-NTS-PGR-N1144-R01C0 "Guide to Efficient CMS-2Y Programming," FCDSSA, San Diego, 15 April 1981.

1.3 Conventions

1.3.1 Semantic Conventions

The word must, as used in this manual, describes a condition required of a correct program.

The word undefined, as used in this manual, describes a condition to which no meaning is attached. For a program actually compiled by the CMS-2Y(7) compiler, an undefined condition will usually have some meaning. A user may discover the meaning and subsequently write a program that relies on that meaning, but the program usually cannot be transported to a different target machine or even another compiler for the same target machine. Furthermore, the CMS-2Y(7) compiler's handling of such a condition might reasonably change from one revision to the next.

The phrase no limit, as used in this manual, describes a quantity that is not limited by the definition of the CMS-2Y(7) language. In reality the quantity is not truly unlimited. The limit typically depends on some resource of the compiler, often the amount of memory available for its symbol table, and is usually quite large. However, in extreme cases (when the pertinent resource has been depleted during a compilation) the limit may be small. In such cases the usual effect is an aborted compilation -- aborted because of the insufficient resource -- rather than an indication that the quantity has exceeded a limit.

The phrase effect of is used in this manual when describing semantics in a manner that might appear to be suggesting the method of implementation. No such suggestion is intended.

Symbols used in the narrative of this manual to assist in the functional description of a process are not necessarily used the same as CMS-2 language symbols nor as the symbols used in syntactical productions. Narrative symbols appearing in descriptive text are interpreted as follows:

SYMBOL	FUNCTION
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
>	greater-than relation
<	less-than relation
(or [initial expression enclosure
) or]	terminal expression enclosure

Figure 1-01. Narrative Symbols

1.3.2 Symbolic Conventions

A modified Backus Naur form (BNF) is used to present all syntax productions. The symbols used in this specification are:

- < > - Angle brackets isolate syntactic symbols. Each symbol contained within the brackets is subject to further definition through substitution of the symbol either by other syntactic symbols or, ultimately, by symbols for which there can be no further definition. Those symbols for which substitutions may be made are called nonterminal symbols. Those for which there can be no further definition are known as terminal symbols; these are not enclosed in angle brackets (examples include IF, SET, +, and \$).
- ::= - Ideogram that separates a nonterminal symbol from its definition. Alternative definitions for the nonterminal are presented on subsequent lines, with the ::= repeated. If a definition cannot be contained on a single line, any continuation lines are indented and are not preceded by the ::= ideogram.
- [] - Square brackets indicate an optional entity.
- @ - "At" sign indicates that the preceding nonterminal may be repeated an arbitrary number of times, separated by commas.
- & - Ampersand indicates that the preceding nonterminal may be repeated an arbitrary number of times. No commas are used.

This manual has been organized with the reader of the text in mind. Because of this organization, the BNF is in a "nonstandard" presentation sequence, making heavy use of forward references. The page on which the definition of each nonterminal appears is marked in the index of syntax symbols. Use of this index should aid the reader of the BNF.

1.4 Section Summary

This manual is presented in several subdivisions, each of which discusses a functional area of the CMS-2Y(7) language.

Section 1. Introduction

This section sets forth the purpose of this manual, describes the background behind CMS-2Y, lists documents related to this one, presents the conventions used herein, and summarizes the significant functions of each of the manual's sections.

Section 2. Fundamental Concepts

This section discusses the grammatical format of CMS-2Y(7) programs, internal program documentation, data classes, and name scopes.

Section 3. Basic Constructs

This section defines the CMS-2Y(7) character set, character strings, and names, and it sets forth the constraints which define numeric, Boolean, and character constants.

Section 4. Declarations

This section describes the various declarations governing the organization and attributes of data within the user's program.

Section 5. Data References

This section discusses how data is referenced, either by retrieving values or by comparing two values. References can be to basic data units, constants, and functions, or to combinations of these (expressions).

Section 6. Statements

This section describes the various actions which may be performed to manipulate data and to specify logic.

Section 7. Subprograms

This section discusses the three ways that statements may be grouped together in a CMS-2Y(7) program.

Section 8. System Elements

This section describes how data declarations are grouped together and how data declarations and statements may be blocked to form elements of a CMS-2Y(7) program.

Section 9. Compilation Modules

This section defines all the declarations required by the compiler which affect its code generation and the kinds of output it creates.

Section 10. Conditional Compilation

This section describes how to bracket blocks of source statements and how to direct their inclusion or exclusion by the compiler.

SECTION 2. FUNDAMENTAL CONCEPTS

2.1 Program Format

Compilation modules are presented to the CMS-2Y(7) compiler as a sequence of 80-character units, called lines. The character positions within each line are called columns and are numbered from left to right, beginning with 1.

Columns 1-10 of each line have no meaning in a CMS-2Y(7) program and may be used for any purpose by the programmer. However, the suggested use is for a card identification field, as follows:

<u>Columns</u>	<u>Entry</u>
1-4	Program identification
5-8	Card sequence number
9-10	Insert number

Columns 11-80 of the lines of a compilation module are considered to be a continuous stream of characters; column 80 of each line is followed immediately by column 11 of the next. Thus, if it is necessary to break a token (paragraphs 3.1.3, 3.3, and 3.4) between two lines, the first part of the token must end in column 80 of the first line and the remainder must begin in column 11 of the next. If, however, a line break occurs between two tokens, the first can end in any column from 11 through 80, and the second can begin in any column (because spaces generally have no effect in a CMS-2Y(7) program).

CMS-2Y(7) programs are written in free format; columns 11-80 have no significance on the effect of a program. Certain comment forms (paragraph 2.2) do have some positional significance, but comments do not affect the meaning of a program.

2.2 Comments

Syntax

<comment statement>
 ::= COMMENT [<comment>] \$

<comment>
 ::= <comment character>&

<comment character>
 ::= <letter>
 ::= <digit>
 ::= <delimiter>
 ::= <space>
 ::= <special character>
 ::= \$\$

Semantics

A comment statement provides program documentation information or controls the form of the program listing.

COMMENT - A language keyword indicating a comment statement.

<comment> - Optional. The string of characters (not including a single \$) that provides the documentation information.

A comment statement has no effect on the execution of a CMS-2Y(7) program.

In general, a comment statement may be written following any \$ in a CMS-2Y(7) program (possibly with intervening blanks). There are some exceptions to this rule:

- a. A comment statement may not appear in a direct code block (paragraph 3.5). In particular, a comment statement may not follow the direct code head (DIRECT \$).
- b. A comment statement may not follow the end-system declaration (Section 9). The end-system declaration defines the end of the program; any following comment would not be in the program.

Comment statements do not form part of the CMS-2Y(7) language proper. Because of this, they are not mentioned elsewhere in this manual. In particular, they do not appear in any other syntax productions of this manual.

There is virtually no limit to the number of characters permitted in a comment statement; a comment may exceed the character limit for a line and continue for as many lines as the user deems appropriate.

Three special forms of the comment statement are used to control the program listing. Each requires the keyword COMMENT in columns 11-17 of a line, a space in column 18, two left parentheses in columns 19 and 20, and a 5-character string in columns 21-25. The terminating \$ may appear in any column thereafter. The meanings of the special forms are:

COMMENT ((EJECT - The program listing is ejected to the top of a new page.

COMMENT ((SKIPn - n lines will be skipped on the program listing, where n is an integer from 1 to 9.

COMMENT ((LINE* - A line of asterisks is printed across the program listing.

In all three cases, the comment line itself is not printed.

If any other comment statement has the keyword COMMENT in columns 11 through 17, a blank line will appear on the output listing followed by the printing of the comment statement, with COMMENT replaced by an asterisk followed by six blanks.

2.3 Spaces and Notes

Syntax

<note>
 ::= ''<comment>''

Semantics

Generally, space characters may be used freely to improve the readability of a program listing. Any number of spaces may be inserted between any two tokens (paragraphs 3.1.3, 3.3, and 3.4) of the language. Free spaces may not be inserted into tokens. (Spaces may appear within a token only in character constants and status constants. In both of these cases, the spaces are significant -- that is, they have value in the program.)

Notes, which consist of two consecutive apostrophes, a string of characters other than an isolated \$, and two more consecutive apostrophes, may be used to improve program documentation. A note has the effect of a single space character.

2.4 Modes and Types

2.4.1 Modes and Simple Types

CMS-2Y(7) contains four predefined modes of data: numeric, Boolean, character, and status.

Each mode, except for Boolean mode, has a number of variable attributes. Each datum of a CMS-2Y(7) program is of one of the modes with the variable attributes fixed. Fixing the variable attributes of a mode defines a simple type.

Because of the essential similarity, there is a natural conversion between simple types of the same mode. The different types can be considered to be representing the same kinds of information, although in possibly different internal forms. Thus implicit conversion between two types of the same mode is allowed.

By contrast, there is no natural conversion between simple types of different modes, and implicit conversions are not permitted.

The numeric mode contains three submodes: integer, fixed-point, and floating-point. In specifying a numeric type, the submode must first be specified, and then the variable attributes of that submode must be fixed.

The following paragraphs describe the characteristics and attributes of the modes.

Numeric: Integer

The integer submode represents integer numeric data. The representation is exact. The variable attributes are the total number of bits necessary to express a datum of the type and whether the type is to be able to represent signed (positive, negative, or zero) or unsigned (non-negative only) values.

Numeric: Fixed-Point

The fixed-point submode represents those rational numeric data that can be represented as terminating binary fractions. The representation is exact. The variable attributes are the total number of bits necessary to express a datum of the type, the position of the implied binary point, and whether the type is to be able to represent signed or unsigned values.

Numeric: Floating-Point

There are two floating-point types if the target computer is an AN/UYK-7, four if it is an AN/UYK-43. For the AN/UYK-7 there is only one variable attribute: The rounding property -- whether or

not calculations using a datum of the type are to be rounded. For the AN/UYK-43 there is, in addition, either single precision or double precision AN/UYK-43 floating-point format internal representation.

Boolean

The Boolean mode represents the Boolean values of true and false. They are represented in a single target machine bit, which is on (1) to represent true and off (0) to represent false. There is only one Boolean type; it has no variable attributes.

Character

The character mode represents character strings of fixed length. The only attribute of a character type is the number of characters in the string.

Status

The status mode represents data that can assume only a finite number of values, with the values having no mathematical significance. The attributes of a status type are the values that a datum of the type can assume, and the order of those values.

Note

An integer type and a fixed-point type having the same length, the same signed-unsigned attribute, and a fractional bits value of zero can represent exactly the same mathematical values. However, when the MSCALE scaling specifier has been requested, the two types are not identical. Their most striking dissimilarity arises in the context of division. The quotient of two fixed-point quantities, both having zero fractional bits, usually has a nonzero fractional bits value. The quotient of two integer quantities, by contrast, is always an integer quantity. When the MSCALE scaling rules are not in effect, the integer type is merely a notational convenience for the programmer.

2.4.2 Universal Type

There is one additional type used in CMS-2Y(7) -- the universal type. Unlike the other types, it cannot be declared. It arises in contexts of interpreting any one of the simple types as a string of bits. If a datum of universal type is used in an arithmetic context, it is treated as an unsigned integer type. Its application is in performing specific operations which require universal typing before the operation can be performed.

2.4.3 Structured Types

A structured datum is one whose values are made up of one or more parts, called fields. Fields can be defined with the attributes of any of the four modes of CMS-2Y(7) data.

The attributes of a structured type are its item allocation (paragraph 4.8), the names and types of its fields, and the allocation of the fields within the type. For a compiler-packed type (paragraph 4.8) the allocation of the fields is equivalent to the order in which the fields are declared and the allocation imposed by field overlay declarations.

2.5 Scopes and Scope Rules

CMS-2Y(7) supports four name scopes: universal, global, local, and subprogram. There is one universal scope, one global scope for each compilation module, and any number of local and subprogram scopes.

The universal scope is a scope that contains every CMS-2Y(7) program. It contains names that are predefined to the compiler.

The global scope of a CMS-2Y(7) program consists of all names whose attributes are known throughout a compilation module. A name has global scope if it is declared in a major header or a system data block, or if its declaration is preceded by a scope modifier (paragraph 4.1). (Note that not all declarations may be preceded by these modifiers.)

The local scopes of a CMS-2Y(7) program consist of names whose attributes are known throughout a single system data block or system procedure block. A name has local scope if it is declared in a minor header, it is declared in a local data block or automatic data block and its declaration is not preceded by a scope modifier, or it is a statement name.

The subprogram scopes of a CMS-2Y(7) program consist of names whose attributes are known throughout a single subprogram. Local index names and names declared in subprogram data blocks have subprogram scope.

Scopes may overlap only if they are nested. If a name denotes different entities in two scopes, one contained in the other, then up to the point of the first declaration of that name in the inner scope the name refers to the entity in the outer scope. From the point of that declaration to the end of the inner scope the name refers to the entity in the inner scope.

2.6 Input/Output and Files

CMS-2Y(7) contains a number of input/output statements whose primary function is transferring data between the program's data areas and external media (magnetic tape, punched cards, etc.). The statements also have a secondary function: conversion of the data between character string form (the form appropriate for communicating the data to humans) and the internal form used by the computer. Each of these functions can be performed separately; it is also possible to write single CMS-2Y(7) statements that perform both. The conversion process is called formatting.

All input/output statements require the use of certain subroutines that are on the CMS-2Y system library. These subroutines call on the CMS-2Y monitor for the actual communication with the various peripheral devices. As a result, the MONITOR option must be specified in any program that uses any of these statements.

Data external to a program is organized by the programmer into logical structures called files. For some peripheral devices a file is simply all of the data that is input or output by means of that device during execution of the program; e.g., card readers and printers. Magnetic tape files, however, consist of the data contained on a single tape reel. Although it is possible to have more than one file associated with a single magnetic tape unit during the execution of a program, only one file can be associated with the unit at any given moment. Magnetic disks can contain several files simultaneously, but the CMS-2Y monitor treats each file as a single magnetic tape file. In any discussion of input/output statements, any statement about magnetic tape files will apply equally to disk files. (The manner in which disk packs are subdivided into file areas is determined at the time each CMS-2Y system is generated.)

Magnetic tape files can be divided into subfiles, each having most of the properties of a file. The subfiles are separated from each other by end-of-file marks that are placed into the file at the time it is created. The end-of-file marks can be detected as the file is read. Dividing a file into subfiles is a purely logical operation whose meaning is determined by the programmer. In any discussion of input/output statements, any statement about files will apply equally to subfiles unless otherwise noted.

Files are subdivided into two types of records that in turn consist of data items. Data items are values that are transferred between the peripheral device and variables and fields in the program. Usually the data items transferred during the execution of a single input/output statement make up a record known as a logical record. It is possible, however, to transfer more than one logical record with a single statement in some circumstances.

A logical record, then, is a logical subdivision of a file. A file can also be physically subdivided into physical records. For some peripheral devices the size of a physical record is fixed. One line on the printer and one card on the card reader are physical records for those devices. For the magnetic tape, however, the size of physical records varies and can be controlled by the programmer. For those devices that have fixed physical record size, the concepts of physical record and logical record are identical. For the other devices they can be either identical or independent, as determined by the programmer.

When data is transferred between the program and a peripheral device, an intermediate storage area in the computer memory, called a buffer, is usually used. On input, physical records are read into the buffer when needed and the data is moved from the buffer into the user data areas. On output, data is moved from the user data areas into the buffer, which is then written to the external medium at some appropriate time. It is this use of buffers that permits, among other things, the distinction between logical and physical records. Buffers are not program data areas; they are automatically defined by the compiler based on information about the file supplied by the programmer.

A count of the subfiles of a file and the physical records in a subfile (or the file, if it is not divided into subfiles) is maintained during the execution of a program, beginning with zero for a file or a subfile. For example, the first record of the second subfile of a file is record 0 of subfile 1. This pair of counts can be used to position magnetic tape files at any physical record during the execution of a program. When such positioning is desired, it is usually necessary to have logical records be the same as physical records for the file.

A magnetic tape file can have a header record, which is the first record of the file (the first record of subfile 0). This record, in a special format, is 30 words long, and is intended to contain information that identifies the file. The record is created at the time the file is created, and can be checked any time the file is read. All of this can be done without resort to a special record, but the header record has two additional properties that make it particularly useful:

- a. Special statements for creating and checking the header record are supplied; if the check fails, the message WRONG TAPE MOUNTED is displayed to the operator, followed by a request that another tape be mounted.
- b. The header record is not counted; record 0 of subfile 0 is the second record of the file.

2.7 Debugging Aids

CMS-2Y(7) contains several statements and declarations that can be used to assist in debugging a program. These aids can be used to trace program execution (either at the statement level or the subprogram invocation level), and to display data under various conditions.

The debugging aids are controlled at three different levels. At the lowest level are the debug data declarations and statements, which are inserted into the body of the program at points appropriate to the debugging task. These statements and declarations are divided into four classes. Each of these classes must be enabled by a declaration that appears in the major header of the compilation module. If a particular class is not enabled, the statements and declarations of that class are ignored by the compiler; the effect is as if they did not appear at all. Finally, the portions of the CMS-2Y monitor that support the functioning of the classes must be enabled at load time by including the appropriate parameters on the \$LOAD call. If the portion of the monitor that supports a class is not enabled, no output will be generated during program execution by statements of that class.

Thus a programmer can insert debugging aids into his source program and leave them there after the program becomes operational, and not incur any penalty at execution time by not enabling the aids in the major header during the final compilation. Similarly, if a program has been compiled with enabled debugging statements and is found (when tested) to be debugged, it can be used immediately without recompilation by not enabling the supporting software in the monitor. In this case there is an execution penalty because, although no debugging output is being produced, the code generated by the debugging aids is being executed.

All debugging statements and declarations require the use of the CMS-2Y monitor. As a result, the MONITOR option must be specified in any program that uses any of these aids.

SECTION 3. BASIC CONSTRUCTS

The basic constructs used in CMS-2Y(7) programs are defined in this section.

Some of these constructs are tokens (paragraphs 3.1.3, 3.3, and 3.4). A token is a construct with no subunits whose meanings (if any) are related to the meaning of the token. For example, the sequence of characters PAX12 is a name and X1 is also a name, but the interpretation of X1 is not inherently related to the interpretation of PAX12. In contrast, XY-12 is not a token, because XY, -, and 12 all have meanings, and those meanings are inherently related to the meaning of XY-12.

3.1 Characters

Syntax

<character>
 ::= <letter>
 ::= <digit>
 ::= <delimiter>
 ::= <space>
 ::= <terminator>
 ::= <special character>

<space>
 ::= blank

<terminator>
 ::= \$

<special character>
 ::= implementation dependent character

Semantics

The nonterminal <space> represents the blank, or space, character.

A special character is any character other than a letter, digit, delimiter, blank, or terminator that can be input and output through the I/O devices of a particular installation.

Not every system can accept every special character.

3.1.1 Letters

Syntax

<letter>

::= A
::= B
::= C
::= D
::= E
::= F
::= G
::= H
::= I
::= J
::= K
::= L
::= M
::= N
::= O
::= P
::= Q
::= R
::= S
::= T
::= U
::= V
::= W
::= X
::= Y
::= Z

Semantics

A letter may be any letter of the English alphabet.

3.1.2 Digits

Syntax

<digit>
 ::= 0
 ::= 1
 ::= 2
 ::= 3
 ::= 4
 ::= 5
 ::= 6
 ::= 7
 ::= 8
 ::= 9

Semantics

A digit may be any one of the decimal digits 0-9.

3.1.3 Delimiters

Syntax

<delimiter>
::= +
::= -
::= /
::= *
::= .
::= (
::=)
::= ;
::= ';

Semantics

Delimiters are tokens that have predefined meanings. They can be used to separate (delimit) other tokens.

The uses of the delimiters are:

- + - Add operator and unary plus operator.
- - Subtract operator, unary minus operator, and hyphen.
- / - Divide operator.
- * - Multiply operator. Two consecutive asterisks form the exponentiation operator.
- . - Radix point and label delimiter. Two consecutive periods form the in-line scaling specifier. Three consecutive periods form the extended index indicator.
- (- Initial enclosure for expressions, lists, or other syntactic units.
-) - Terminal enclosure for expressions, lists, or other syntactic units.
- ; - List separator (comma).
- ' - Status constant delimiter (apostrophe). Two consecutive apostrophes form the programmer notes delimiter.

3.2 Strings

Syntax

<character string>
 ::= <character>&

<simple string>
 ::= <simple character>&

<simple character>
 ::= <letter>
 ::= <digit>
 ::= <delimiter>
 ::= <space>

Semantics

A character string is a string of any of the characters valid in a CMS-2Y(7) program.

A simple string is a character string that contains no terminators (\$) or special characters.

3.3 Names

Syntax

```

<name>
    ::= <letter>[<alphanumeric character>]&

<alphanumeric character>
    ::= <letter>
    ::= <digit>
    
```

Semantics

Names are tokens made up of strings of alphanumeric characters (letters and digits). The first character is a letter. Names may be no more than eight characters long.

Names are used to identify various entities in a CMS-2Y(7) program.

A name cannot be the same as any of the following reserved words:

ABS	DECODE	FORMAT	NOT	SPILL
ALG	DEFID	FROM	OCM	STOP
AND	DENSE	FUNCTION	ODDP	SWAP
BASE	DEP	GOTO	OPEN	SWITCH
BEGIN	DIRECT	GT	OPTIONS	SYSTEM
BIT	DISPLAY	GTEQ	OR	TABLE
BY	ELSE	HEAD	OUTPUT	THEN
CAT	ELSIF	IF	OVERFLOW	THRU
CHAR	ENCODE	INDIRECT	OVERLAY	TO
CHECKID	END	INPUT	PRINT	TRACE
CIRC	ENDFILE	INTO	PTRACE	TYPE
CLOSE	EQ	INVALID	PUNCH	UNTIL
CMODE	EQUALS	LIBS	RANGE	USING
COMMENT	EVENP	LOG	READ	VALID
COMP	EXCHANGE	LT	REGS	VARY
CORAD	EXEC	LTEQ	RESUME	VARYING
CORRECT	EXIT	MEANS	RETURN	VRBL
CSWITCH	FIELD	MEDIUM	SAVING	WHILE
DATA	FILE	MODE	SET	WITH
DATAPool	FIND	NITEMS	SHIFT	WITHIN
DEBUG	FOR	NONE	SNAP	XOR

Note:

The following names have universal scope (are predefined). They may be redefined.

ACOS2	BAMS	FIRST	ORF	SCALF
ACOS	CNT	ICOS	POS	SIN
ALOG	COMPF	IEXP	PRED	SUCC
ANDF	CONF	ISIN	RAD	TDEF
ASIN2	COS	LAST	ROTATEHP	VECTORHP
ASIN	EXP	LENGTH	REM	VECTORP
ATAN2	FIL	LN	ROTATEP	XORF
ATAN				

3.4 Constants

Syntax

<constant>
 ::= <numeric constant>
 ::= <boolean constant>
 ::= <character constant>
 ::= <status constant>

Semantics

Constants are tokens that specify values that cannot change during execution of a CMS-2Y(7) program.

The nonterminal <constant> in this manual is reserved for literal constants -- constants whose values are described by the manner in which they are written. CMS-2Y(7) also supports symbolic (named) constants (paragraph 4.13) and constants whose value is specified when the program is loaded for execution (paragraph 4.14).

3.4.1 Numeric Constants

Syntax

```
<numeric constant>  
 ::= <decimal constant>  
 ::= <octal constant>
```

Semantics

A numeric constant specifies a numeric value.

<decimal constant> - A numeric constant expressed in base 10 notation.

<octal constant> - A numeric constant expressed in base 8 notation.

Many numeric constants can be written in more than one manner. The value of the constant is independent of the manner in which it is written. The value is always expressed with maximum accuracy in 63 bits.

A numeric constant is of fixed-point mode. Its scaling (number of fractional bits) is determined by the manner in which it is written (paragraphs 3.4.1.1 and 3.4.1.2) and must be in the range [-127,127]. When the context of a numeric constant requires more fractional bits than its scaling implies, the extra bits are obtained from its stored value, up to the maximum 63 bits.

3.4.1.1 Decimal ConstantsSyntax

```

<decimal constant>
    ::= <decimal number>
    ::= D(<decimal number>)
    ::= <decimal number>D

<decimal number>
    ::= <decimal mantissa>[E<decimal exponent>]

<decimal mantissa>
    ::= <decimal integer>
    ::= <decimal integer>.<decimal integer>
    ::= .<decimal integer>

<decimal exponent>
    ::= [<unary numeric operator>]<decimal integer>

<decimal integer>
    ::= <decimal digit>&

<decimal digit>
    ::= <digit>

<unary numeric operator>
    ::= +
    ::= -

```

Semantics

A decimal constant specifies a numeric value in base 10 notation. The value may be written in conventional decimal notation or in a form of scientific or engineering notation -- a decimal value multiplied by a power of 10.

- D - Optional. A language keyword indicating a decimal constant.
- <decimal mantissa> - A numeric constant in base 10 notation in the form of a string of decimal digits and an optional decimal point, which may be before, after, or embedded in the string of digits.

- E - Optional. A language keyword indicating that an exponent follows.
- <decimal exponent> - Optional. A decimal integer, optionally preceded by a unary plus or minus sign, representing the power of ten by which the value of the mantissa is to be multiplied to obtain the value of the constant.
- <decimal integer> - A numeric constant in base 10 notation in the form of a string of decimal digits.
- <unary numeric operator> - Optional. The unary plus or unary minus operator.

The character string that represents a decimal constant may not contain embedded blanks.

Numeric constants are decimal by default in CMS-2Y(7). Either form D(<decimal number>) or <decimal number>D is required only if the implied constant type has been specified to be octal (paragraph 4.2).

The number of fractional bits in a fixed-point decimal constant is determined in the following steps:

- a. The number of fractional bits is initially calculated according to the formula $[\text{Log}_2(10) * F] + 1$, where $\text{Log}_2(10)$ is the logarithm of 10 to the base 2 (approximately equal to 3.3219280949), F is the number of fractional digits of the number as written, and [] represents the "truncation to integer" function.
- b. The value obtained in step (a) is reduced by the number of trailing zero bits in the bit string consisting of the constant, converted to base 2, and including the number of fractional bits determined in step a.
- c. If the number of bits required to contain the integer part of the value and the number of fractional bits obtained in step (b) is greater than 63, then excess bits are truncated on the right of the bit string to reduce its length to the maximum number of bits, and the number of fractional bits is reduced by the number of bits truncated.

Examples

990
D(990)
990D
.99E3

The above examples all represent the decimal number 990.

Notes

- a. Trailing zeros are often significant in decimal fractions in CMS-2Y. This is because decimal fractions can seldom be converted exactly to binary and step (a) says that the accuracy of approximation is dependent on the number of fractional digits. To illustrate this, consider the decimal number 0.1, which in octal has the non-terminating representation 0.06310631... Then the inherent accuracy of three equivalent decimal representations of this number is:

Source Program Form	Number of Fractional Bits	Internal Octal Value	Internal Decimal Value
0.1	4	0.04	0.0625
0.10	7	0.06	0.09375
0.100	10	0.063	0.099609

- b. In both step (b) and step (c) above, the number of fractional bits can become negative.

3.4.1.2 Octal Constants

Syntax

<octal constant>
 ::= 0(<octal number>)
 ::= <octal number>

<octal number>
 ::= <octal mantissa>[E<octal exponent>]

<octal mantissa>
 ::= <octal integer>
 ::= <octal integer>.[<octal integer>]
 ::= .<octal integer>

<octal exponent>
 ::= [<unary numeric operator>]<octal integer>

<octal integer>
 ::= <octal digit>&

<octal digit>
 ::= 0
 ::= 1
 ::= 2
 ::= 3
 ::= 4
 ::= 5
 ::= 6
 ::= 7

Semantics

An octal constant specifies a numeric value in base 8 notation. The value may be written in conventional octal notation or in a form of scientific or engineering notation -- an octal value multiplied by a power of 8.

- 0 - Optional. A language keyword indicating an octal constant.
- <octal mantissa> - A numeric constant in base 8 notation in the form of a string of octal digits and an optional octal point, which may be before, after, or embedded in the string of digits.
- E - Optional. A language keyword indicating that an exponent follows.

<octal exponent> - Optional. An octal integer, optionally preceded by a unary plus or minus sign, representing the power of 8 by which the value of the mantissa is to be multiplied to obtain the value of the constant.

<octal integer> - A numeric constant in base 8 notation in the form of a string of octal digits.

The character string that represents an octal constant may not contain embedded blanks.

Numeric constants are decimal by default in CMS-2Y(7) and the optional 0 form is necessary to specify an octal constant. An octal constant can be written without the 0 only if the implied constant type has been changed to octal (paragraph 4.2).

The number of fractional bits of a fixed-point octal constant is obtained by the same three-step process as is used for decimal constants, except that the number of fractional bits is initially given by the formula $3 * F$, where F is the number of fractional digits in the number as written (step a of paragraph 3.4.1.1).

Examples

0(144)

This character sequence generates a binary value equal to 100 decimal.

-0(100)

This character sequence generates a binary value equal to negative 64 decimal.

3.4.2 Boolean Constants

Syntax

```
<boolean constant>  
 ::= 0  
 ::= 1
```

Semantics

The Boolean constant 1 represents the value true and the Boolean constant 0 represents the value false.

3.4.3 Character Constants

Syntax

<character constant>
 ::= H(<character string>)

Semantics

A character constant denotes a character string.

H - A language keyword indicating a character constant.

<character string> - A string that represents the value of the constant.

No spaces may appear between the keyword H and the following left parenthesis of a character constant.

Each character of the character string represents a character of the value of the constant, except that a right parenthesis is represented by two consecutive right parentheses.

The character string that represents the value of the constant begins with the character that immediately follows the first left parenthesis and ends with the character that immediately precedes the closing right parenthesis.

The maximum length of the value of a character constant is 132.

Examples

H(ABC))DE) produces the constant "ABC)DE"
H(ABC) produces the constant "ABC "
H((ABC))) produces the constant "(ABC) "

Notes

A right parenthesis cannot immediately follow a character constant, because the two consecutive right parentheses would be interpreted as part of the constant. At least one blank character must be inserted between the terminating right parenthesis and the following right parenthesis in such a case.

3.4.4 Status Constants

Syntax

<status constant>
 ::= '<character string>'

Semantics

A status constant represents a nonmathematical value.

The value of a status constant is the string of characters between the enclosing apostrophes. The value begins with the character immediately following the left apostrophe and ends with the character immediately preceding the right apostrophe. Leading and trailing blanks in the value are significant.

The value of a status constant may not contain the apostrophe character.

The value of a status constant may be no longer than eight characters.

Examples

'SYSTEM'
'ALERT'
'READY'

These examples all illustrate status constants.

3.5 Direct Code Block

Syntax

<direct code block>
 ::= <direct code phrase> \$

<direct code phrase>
 ::= <direct code head> <direct code> <cms-2 phrase>

<direct code head>
 ::= DIRECT \$

<direct code>
 ::= <direct code statement> &

<cms-2 phrase>
 ::= CMS-2

Semantics

A direct code block specifies a sequence of machine code mnemonic statements.

- DIRECT - A language keyword indicating that one or more machine code mnemonic statements follows.
- <direct code> - Machine code mnemonic statements.
- CMS-2 - A language keyword indicating that the sequence of machine code mnemonic statements is ended.

The direct code of a direct code block may be any sequence of valid direct code statements (Appendix B). Although mixing CMS-2Y(7) data declarations and statements is impossible, there is no such restriction on mixing the corresponding direct code statements. If the use of a direct code block causes data declarations to appear among executable code, it is the programmer's responsibility to control the program flow around the data areas.

SECTION 4. DECLARATIONS

Declarations are the CMS-2Y(7) constructs used to define the attributes of the names in a CMS-2Y(7) program. The attributes of all names other than statement labels (Section 6), procedures, and functions must be defined before they are referenced. (The attributes of procedure names can be defined implicitly in procedure switch declarations.)

For many entities, CMS-2Y(7) allows two types of declarations: attribute declarations and allocation declarations. An attribute declaration defines the attributes of the entity. An allocation declaration defines the attributes of the entity and specifies that the target machine memory necessary to contain the entity is to be allocated as part of the system element in which the declaration appears.

If an entity is one of those for which both attribute and allocation declarations are allowed, it may be declared any number of times in a CMS-2Y(7) program, but only one of the declarations may be an allocation declaration. The attributes of the entity must be the same on all of its declarations, except for the fields of a user-packed table (paragraph 4.8).

Names must be unique in the scope in which they are declared, except for field names (see below). (Note that at the point of declaration, the name might already denote an entity in a larger scope. See paragraph 2.5.) In addition, within a single element a name cannot be declared to denote two different entities, one with local scope and the other with global scope.

Field names must be unique within the structured type in which they are defined. They do not have to be distinct from any other names in any scope. The same name may be used as a field name in any number of structured types in a scope and may also be declared as the name of some other entity in that scope.

Note

While the interaction of the scope rules, implicit declaration of procedures in procedure switches, and forward-referencing is well-specified in this manual, the results can occasionally be surprising. These surprises can be avoided by explicitly declaring all entities before referencing them.

4.1 Declaration Modifiers

Syntax

```
<declaration modifier>  
 ::= <scope modifier>  
 ::= <allocation modifier>
```

```
<scope modifier>  
 ::= (EXTDEF)  
 ::= (EXTREF)  
 ::= (TRANSREF)
```

```
<allocation modifier>  
 ::= (EXTREF)  
 ::= (LOCREF)  
 ::= (TRANSREF)
```

Semantics

A declaration modifier is used to indicate that the name whose attributes are being declared has global scope (when the position of the declaration might imply that it has local scope) or that the declaration is an attribute declaration.

- (EXTDEF) - A scope modifier. Signifies that the name being declared has global scope.
- (EXTREF) - A scope modifier and allocation modifier. Signifies that the name being declared has global scope and that the entity is not to be allocated because of this declaration.
- (LOCREF) - An allocation modifier. Signifies that the entity being declared is not to be allocated because of this declaration. The entity will be allocated in the system procedure (paragraph 8.2) containing this declaration and the name of the entity will have local scope.
- (TRANSREF) - A scope modifier and allocation modifier. Signifies that the name being declared has global scope and that the entity is located in another system element that cannot be assigned a permanent base register.

Any declaration of a variable, table, procedure, executive procedure, function, procedure switch, file, or format that is not preceded by an allocation modifier is an allocation declaration.

The (EXTDEF) scope modifier may be used with a declaration that appears in a system data block (paragraph 8.1). The name of an entity declared in a system data block has global scope by default, but the redundancy is permitted.

A name whose attributes are declared using the (EXTREF) allocation modifier need not be allocated in the same system block (Section 9) containing the attribute declaration. Its allocation declaration may appear in another system block.

The (LOCREF) allocation modifier may only be used with subprogram attribute declarations in a local data block or an automatic data block. The named subprogram must be defined in the same system procedure block.

The (TRANSREF) scope modifier is used to establish an attribute definition for an entity allocated in another system element that cannot be assigned a target machine base register with a permanent fixed value. Each reference made to this entity's name will cause a transient base register to be loaded with a value appropriate to the referenced system element. Since referencing names using a transient base register is less efficient than using a base register with a fixed value, the (TRANSREF) scope modifier should be used only when a program's combined instruction and data size exceed 65,536 locations, or when the program has otherwise exhausted the available number of base registers.

Declaration modifiers may not appear in subprogram data blocks.

Examples

```

TDAT      SYS-DD $
          VRBL JORG B $
          END-SYS-DD TDAT $
          .
          .
          .
TPROC    SYS-PROC $
          LOC-DD $
(EXTREF) VRBL JORG B $
          END-LOC-DD $
          .
          .
          .
          SET JORG TO 0 $
          .
          .
          .
          END-SYS-PROC TPROC $
    
```

/(U) CM2Y-MAN-PGR-M5049-R04C0

In this example, system element TPROC may be compiled with or without system element TDAT. Because of the (EXTREF) scope modifier, there is no duplicate declaration.

```
      LOC-DD $
      VRBL ALGAE A 8 S 0 $
(LOCREF) PROCEDURE HICHK INPUT ALGAE $
      END-LOC-DD $
      .
      .
      HICHK INPUT 7 $
      .
      .
      PROCEDURE HICHK INPUT ALGAE $
```

In this example the (LOCREF) allocation modifier is used because procedure HICHK is called before it is declared and also because it includes a formal input parameter.

```
GARDOG      SYS-PROC $
      .
      .
(TRANSREF) VRBL FLOATER F $
```

References made to variable FLOATER in system procedure GARDOG cause the generation of code to use a transient base register.

4.2 Constant Mode Declaration

Syntax

```
<constant mode declaration>  
 ::= CMODE [<constant mode>] $
```

```
<constant mode>  
 ::= O  
 ::= D
```

Semantics

A constant mode declaration specifies the mode of numeric constants whose modes are not explicitly indicated.

CMODE - A language keyword indicating a constant mode declaration.

<constant mode> - The language keywords O and D, indicating octal and decimal mode, respectively.

If the constant mode O is specified or the optional constant mode is omitted, all numeric constants other than those immediately followed by the letter D, or those enclosed in parentheses and preceded by the letter D, are interpreted as octal constants.

If the constant mode D is specified, all numeric constants other than those enclosed in parentheses and preceded by the letter O are interpreted as decimal constants.

A constant mode declaration has no effect on constants appearing in direct code blocks.

Examples

Figure 4-01 illustrates the effects of the cmode declaration on various forms of constant representation.

Constant	Under CMODE D	Under CMODE O
12	12 decimal	12 octal
O(12)	12 octal	12 octal
D(12)	12 decimal	12 decimal
12D	12 decimal	12 decimal
329	329 decimal	Illegal

Figure 4-01. CMODE Declaration Examples

4.3 Simple Type Specification

Syntax

```
<simple type specification>  
  ::= <numeric type specification>  
  ::= <boolean type specification>  
  ::= <character type specification>  
  ::= <status type specification>
```

Semantics

Simple type specifications are used in type declarations, variable declarations, parameter variable declarations, table declarations, field declarations, array declarations, ltag declarations, function declarations, conversion function references, temporary definition function references, and case blocks to specify the attributes of the data.

4.3.1 Numeric Types

Syntax

```
<numeric type specification>
  ::= <integer type specification>
  ::= <fixed-point type specification>
  ::= <floating-point type specification>

<integer type specification>
  ::= I <bit length> <sign specification>

<fixed-point type specification>
  ::= A <bit length> <sign specification> <fractional bits>

<floating-point type specification>
  ::= F[( <floating-point attribute> )]

<floating-point attribute>
  ::= T
  ::= R
  ::= S
  ::= D

<bit length>
  ::= <numeric constant expression>

<fractional bits>
  ::= <numeric constant expression>

<sign specification>
  ::= S
  ::= U
```

Semantics

A numeric type specification specifies an instance of the three numeric modes of CMS-2Y(7). A value of one numeric type can always be converted to another numeric type, provided that the attributes of the type being converted to permit representation of the most significant part of the value being converted.

- | | |
|---|---|
| I | - A language keyword (not reserved) indicating integer type. |
| A | - A language keyword (not reserved) indicating fixed-point type. |
| F | - A language keyword (not reserved) indicating floating-point type. |

- T - Optional. A language keyword (not reserved) indicating that the AN/UYK-7 internal format is to be used and the values of arithmetic operations involving two data units of the type are to be approximated by truncation.
- R - Optional. A language keyword (not reserved) indicating that the AN/UYK-7 internal format is to be used and the value of arithmetic operations involving two data units of the type are to be approximated by rounding.
- S - Optional. A language keyword (not reserved) indicating that the industry standard single-precision internal format is to be used.
- D - Optional. A language keyword (not reserved) indicating that the industry standard double-precision internal format is to be used.
- <bit length> - A numeric constant expression that specifies the total number of bits required to represent a datum of integer or fixed-point type.
- <fractional bits> - A numeric constant expression that specifies the number of bits to the right of the implied binary point of a datum of fixed-point type.
- <sign specification> - An S or U indicating that the integer or fixed-point datum is signed (may assume negative values) or unsigned (may only assume non-negative values) respectively.

The values of the numeric constant expressions that specify the bit length and the number of fractional bits must be an integer. The value of the bit length expression must be in the range [1,64]. The value of the fractional bits expression must be in the range [-127,127].

If a data type is signed, one bit is required for the sign itself. The bit length must be one more than the number of bits required to represent the magnitude of a value of the type.

The floating-point types F(S) and F(D) are valid only if the target computer is an AN/UYK-43. The values of arithmetic operations involving two data units of one of these types are always approximated by rounding. All other floating point types are valid when the target computer is either AN/UYK-7 or AN/UYK-43.

If the optional floating-point attribute is omitted, the attribute T is implied.

Examples

I 4 U

A datum of this type is an unsigned integer, four bits in length. Its range of possible integer values is 0 through 15.

I 4 S

A datum of this type is a signed integer of three data bits and one sign bit (leftmost). Its range of values is -7 through +7.

A 3 U 1

A datum of this type is unsigned fixed-point of two integer bits and one fractional bit. Its possible values are 0, $\pm.5$, ± 1 , ± 1.5 , ± 2 , ± 2.5 , ± 3 , and ± 3.5 .

A 3 U -4

A datum of this type is fixed-point with negative scaling. This example has three magnitude bits, seven integer bits, and no fractional bits. The rightmost four integer bits cannot be accessed, and are functionally zero. The values that a datum of this type may have are 0, 16, 32, 48, 64, 80, 96, and 112.

A 3 U 5

This type illustrates that the number of fractional bits (in this case five) may be greater than the bit length of a datum. The values that this datum may have are 0, $1/32$, $1/16$, $3/32$, $1/8$, $5/32$, $3/16$, and $7/32$.

A 7 U 35

A datum of this type is fixed-point, with seven magnitude bits, no integer bits, and 35 fractional bits. The radix point is functionally 35 bits to the left of the rightmost magnitude bit. This example illustrates that the number of fractional bits may be greater than the number of magnitude bits. The nonzero values that a datum of this type may have are in the range $[2^{*-29}$,

2**-28 - 2**-35]. Graphically, the maximum value in binary format would appear as a string of 28 zeros followed by seven ones.

A 14 S 8

A datum of this type is signed fixed-point of five integer bits, eight fractional bits, and one sign bit (leftmost).

Assume that a datum of signed integer type with a range of -63 through +57 decimal is desired. To determine the proper definition, the programmer should convert the number with the largest absolute magnitude to octal, count the bits, and add 1 for the sign. Thus -63 becomes |-63|, which becomes 63. It then equals 77 octal, which requires six bits plus a sign bit. Therefore, I 7 S is the type declaration under the given criteria. It is important to go through this procedure if maximum accuracy without errors is desired in the arithmetic operations that will involve this datum.

Implementation Note

A variable defined as I 32 U requires two words; a variable defined as I 64 U is illegal.

4.3.2 Boolean Type

Syntax

```
| <boolean type specification>  
  ::= B
```

Semantics

A Boolean type specification specifies that a datum is of Boolean type. Boolean type has no attributes other than the type itself.

B - A language keyword (not reserved) indicating Boolean type.

4.3.3 Character Type

Syntax

<character type specification>
 ::= H <character length>

<character length>
 ::= <numeric constant expression>

Semantics

A character type specification specifies that a datum is of character type.

H - A language keyword indicating character type.

<character length> - A numeric constant expression that specifies the number of characters required to represent a datum of the type.

The value of the character length expression must be an integer in the range [1,132].

Examples

H 1

A datum of this type has a length of one character.

H 45

A datum of this type has a length of 45 characters.

4.3.4 Status Type

Syntax

```
| <status type specification>  
  ::= S <status constant>@
```

Semantics

A status type specification specifies that a datum is of a status type. The specification contains the list of values that the datum may assume.

S - A language keyword (not reserved) indicating status type.

<status constant> - One of the values that may be assumed by a datum of the type being specified.

A status constant may not appear more than once in a single status type specification. The same constant may appear in different status type specifications.

Examples

```
S 'RED', 'AN/UYK-7', '$', '**', 'TOO BIG'
```

A datum of this type can assume five values and is three bits long. The value '**' is represented internally by 3.

Implementation Note

The values of a status type are represented internally by the integer values 0, 1, 2, ..., n-1, where n is the number of values, where 0 is the internal representation of the first value in the list, 1 is the representation of the second value, etc. The length of the type is the number of bits required to express n-1.

Note

There is no fixed limit on the number of values of a status type. That number is limited only by the amount of symbol table available to the compiler.

4.4 Type Declarations

Syntax

```

<type declaration>
  ::= <simple type declaration>
  ::= <structured type declaration>

<simple type declaration>
  ::= [(EXTDEF)] TYPE <simple type name> <simple type
    specification> $

<simple type name>
  ::= <name>

<structured type declaration>
  ::= <untyped structure declaration>
  ::= <typed structure declaration>

<untyped structure declaration>
  ::= [(EXTDEF)] <untyped structure head> [<structure
    information>&] <structured type end>

<untyped structure head>
  ::= TYPE <untyped structure name> <structure allocation> $

<untyped structure name>
  ::= <name>

<structure allocation>
  ::= NONE
  ::= MEDIUM
  ::= DENSE
  ::= <number of words>
  ::= (<untyped structure name>)

<number of words>
  ::= <numeric constant expression>

<structure information>
  ::= <field declaration>
  ::= <field overlay declaration>
  ::= <range declaration>

<structured type end>
  ::= END-TYPE <name> $

<typed structure declaration>
  ::= [(EXTDEF)] <typed structure head> [<structure
    information>&] <structured type end>

```

<typed structure head>
 ::= TYPE <typed structure name> (<typed structure>) \$

<typed structure name>
 ::= <name>

<typed structure>
 ::= <simple type>
 ::= <typed structure name>

<simple type>
 ::= <simple type specification>
 ::= <simple type name>

Semantics

A type declaration specifies a type and declares a name to be used in referring to the type. No memory is allocated as a result of a type declaration.

- | | |
|--------------------------|--|
| (EXTDEF) | - Optional. Signifies that the name being declared has global scope (paragraph 4.1). |
| TYPE | - A language keyword indicating a type declaration. |
| <simple type name> | - The name by which a simple type can be referenced. |
| <untyped structure name> | - The name by which an untyped structure can be referenced. |
| <typed structure name> | - The name by which a typed structure can be referenced. |
| <structure allocation> | - An indication of the manner in which an entity of the structured type is to be allocated. |
| <structure information> | - Optional. Field declarations, field overlay declarations, and range declarations, used when declaring a structured type. |
| END-TYPE | - A language keyword indicating the end of a structured type declaration. |

An untyped structure head and a typed structure head can contain two type names. The name being declared is the name immediately

following the keyword TYPE. The other type name is being referenced in these contexts. The name being referenced may not be the same as the name being declared.

The name that appears in a structured type end must be the name being declared in the structured type declaration.

A type name cannot be A, B, D, F, H, I, O, P, or S.

A simple type name is an alias for the underlying simple type specification. Any appearance of a simple type name outside of its declaration has the same effect as if the underlying simple type specification appeared instead.

An untyped structure head and a typed structure head can reference a type name (in parentheses). If that name is an untyped structure name, the declaration is an untyped structure declaration. If that name is a typed structure name, the declaration is a typed structure declaration.

Structured types have the concept of size, which affects the amount of memory allocated to entities of these types.

If the structure allocation of an untyped structure declaration is NONE, MEDIUM, or DENSE, the structured type is called compiler-packed. The size of a compiler-packed structured type is an integral number of target machine words, which depends on the properties of the fields and the packing algorithm. The packing algorithms affect only those fields that do not appear as field overlay siblings. NONE causes those fields to be allocated in such a manner that no part of any two of them share a target machine word. MEDIUM permits those fields to share a word, but in such a manner that they can be accessed using the target machine k-designator; that is, there is no access time penalty paid when accessing a field of an entity allocated according to the MEDIUM packing algorithm. DENSE permits those fields to share target machine words in such a manner that the type's size is minimized; in this case there can be an access time penalty caused by using the target machine's indirect addressing capability and/or shifting instructions. Figure 4-02 shows details of the three packing algorithms.

If the structure allocation of an untyped structure declaration is a numeric constant expression, the structured type is called user-packed. The value of the expression must be a positive integer. This value specifies the size of the type in target machine words.

The size of a typed structure is the smallest integral number of target machine words that can contain a value of the underlying simple type.

All field declarations appearing in a user-packed type declaration or a typed structure declaration must be user-packed; no field declarations are required in such type declarations. All field declarations appearing in a compiler-packed type declaration must be compiler-packed; at least one field declaration is required in such a type declaration.

If the structure allocation of an untyped structure declaration is the name of an untyped structure enclosed in parentheses, the structure information of the untyped structure declaration is merged with that of the referenced type and the untyped structure being declared inherits the structure allocation of the referenced type. If the typed structure of a typed structure declaration is the name of a typed structure, the structure information of the typed structure declaration is merged with that of the referenced type. Such types are called grown types. The structure information of the referenced type is said to be inherited by the grown type.

Field overlay declarations and range declarations appearing in a grown type declaration may not reference inherited fields. Except for this, for grown types the effect of the merging of structure information is as if the structure allocation or typed structure of the grown type declaration were replaced by the corresponding entry in the referenced type declaration and the structure information of the referenced type were inserted into the current type declaration in the same order as it appeared in the referenced type declaration between the type head and the structure information of the current declaration.

The scope of the referenced type name in a grown type cannot be smaller than the scope of the type being declared.

A field declaration in a structured type declaration may not be preset.

Note

One result of the way structure information is merged for grown types is that an inherited compiler-packed field is allocated in the same position in both types.

		Compiler Packing Descriptor			
		NONE	MEDIUM	DENSE	
Field Type and Allo- cation	Signed integer or fixed- point type with length:	≤ 16 bits	1 word	16 bits #	Number of bits specified
		> 16 bits < 32 bits	1 word	1 word	Number of bits specified
		> 32 bits	2 words	2 words	2 words
	Unsigned integer or fixed- point type with length:	< 8 bits	1 word	8 bits ##	Number of bits specified
		> 8 bits < 15 bits	1 word	16 bits #	Number of bits specified
		> 15 bits < 31 bits	1 word	1 word	Number of bits specified
		> 31 bits	2 words	2 words	2 words
	Floating-point type T,R,&D		2 words	2 words	2 words
	Floating-point type S		1 word	1 word	1 word
	Boolean type		1 word	8 bits ##	1 bit
	Status type N values		1 word	8 bits ## $N < 255$ 16 bits $N > 255$	X bits $2^{**X} > N > 2^{**}(X-1)$
	Charac- ter type with number of charac- ters:	=1	1 word **	8 bits ##	8 bits*
		=2	1 word **	16 bits #	16 bits*
		=3	1 word **	1 word **	24 bits*
> 3		Number of words = [(number of characters-1)/4] + 1; left-justified; truncated to an integer			

Not necessarily aligned on byte boundaries, i.e., assigned to the first available 8, 16, or 24 bits as required.

* Left-justified /// # (half-word) /// ## (quarter word)

Figure 4-02. Allocation of Fields with Compiler-Specified Packing

4.5 Default Type Specifications

Syntax

```
<default type specification>
 ::= MODE VRBL [<simple type>] [P <preset value>] $
 ::= MODE FIELD [<simple type>] $

<preset value>
 ::= <numeric constant expression> [<preset magnitude>]
 ::= <boolean constant>
 ::= <character constant>
 ::= <status constant>
 ::= CORAD(<addressable name>)
 ::= <tag name>

<preset magnitude>
 ::= V (<magnitude value>, <magnitude bit>)

<magnitude value>
 ::= <numeric constant expression>

<magnitude bit>
 ::= <numeric constant expression>

<addressable name>
 ::= <variable name>
 ::= <table name>
 ::= <switch name>
 ::= <procedure switch name>
 ::= <procedure name>
 ::= <function name>
 ::= <stringform name>
 ::= <inputlist name>
 ::= <outputlist name>
 ::= <statement name>

<switch name>
 ::= <label switch name>
 ::= <item label switch name>

<procedure switch name>
 ::= <indexed procedure switch name>
 ::= <item procedure switch name>
```

Semantics

A default type specification specifies the type to be assumed in any variable, field, or function declaration in which the type has been omitted.

- MODE VRBL - A language keyword indicating a default type specification for variables and functions.
- MODE FIELD - A language keyword indicating a default type specification for fields.
- <simple type> - Optional. The type to be used as a default type.
- P - Optional. A language keyword (not reserved) indicating that a preset value is being specified.
- <preset value> - Optional. A preset value to be used with variables whose type is not specified.

Default type specifications occur in two forms. The form MODE VRBL specifies a default mode for variables and functions and a default preset value for variables. The form MODE FIELD specifies a default type for fields.

In the absence of a default type specification for either of the forms, the default type for variables, fields, and functions is A 16 S 0. There is no default preset value for variables.

Any number of default type specifications of either form can appear in a system block. The effect of multiple default type specifications of the same form depends on the sequence of source statements that make up the system block, not the execution sequence of the program. A default type specification will affect only the variable, field, or function declarations that follow it, subject to the limitations stated below, and will override any preceding default type specifications of the same form.

The last default type specification of each of the forms in the major header becomes the default type of that form for the remainder of the system block. A default type specification in a minor header or a system element can override the established default type, but at the end of the system element the established default type again becomes the default type.

If the simple type of the specification is a type name, it must be the name of a simple type.

If the optional type is omitted in a default type specification, the current default type remains the default type. If the optional P and preset value are omitted from a default type declaration, the default of no presetting is established.

If a default preset value is specified, it must be assignment-compatible (paragraph 6.1.1.1) with the default type being specified, or with the default type in effect if none is being specified.

If a preset value is specified and the optional preset magnitude is present, the value is to be converted into a modified binary system in which the specified magnitude value is converted as the specified magnitude bit. The bit to the right of the specified magnitude bit then corresponds to a value that is one-half of the specified magnitude value, the bit to the left corresponds to twice the magnitude value, etc.

Magnitude bits are numbered from right to left; the rightmost bit is bit 0. This numbering refers only to the bits of the variable and does not include additional bits allocated by the compiler.

Examples

```
MODE VRBL A 24 U 13 $
```

| If this specification is included in a system element, all succeeding variables that are defined within the system element without an explicit type specification will receive a type of A 24 U 13 until another default type specification for variables is encountered.

| If the specification appears in a major header, it will affect all subsequent system elements that do not contain a default type specification for variables.

```
MODE FIELD A 32 S 16 $
TABLE LOCALTRK H 11 100 $
  FIELD TRKNO I 32 S 0 31 $
  FIELD ID S 'FRIEND', 'FOE', 'UNKNOWN' 1 31 $
  FIELD X1                                     2 31 $
  FIELD Y1                                     3 31 $
  FIELD T1                                     4 31 $
END-TABLE LOCALTRK $
```

In this example, type is not explicitly specified for fields X1, Y1 and T1; the field mode declaration attributes of A 32 S 16 are assigned for each. It is still necessary to specify the location for these three fields because compiler packing is not specified.

MODE VRBL A 64 S 12 P .999755859375 V(1,6) \$
VRBL ABC \$

The decimal fraction preset here converts to .7777 octal, which is 12 bits of precision. However, because of preset magnitude, in which the sixth bit from the right represents 1, only six bits (.77 octal) are stored in the least significant bit positions of the first word of variable ABC.

4.6 Variable Declaration

Syntax

<variable declaration>
 ::= [<scope modifier>] VRBL <variable list> [<type>]
 [P <preset value>] \$

<variable list>
 ::= <variable name>
 ::= (<variable name>@)

<variable name>
 ::= <name>

<type>
 ::= <simple type>
 ::= <structured type>

<structured type>
 ::= <untyped structure name>
 ::= <typed structure name>

Semantics

A variable declaration specifies that the names of the variable list refer to variables of the specified type. A preset (initial value) may be specified for the variables.

- <scope modifier> - Optional. Refer to scope modifier definition (paragraph 4.1).
- VRBL - A language keyword indicating a variable declaration.
- <variable list> - The names of the variables being declared.
- <type> - Optional. The type of the variables being declared.
- P - Optional. A language keyword (not reserved) indicating that a preset value is being specified.
- <preset value> - Optional. The value to which the variables are preset.

A maximum of 25 names may appear in the variable list of a single variable declaration.

If the optional type is omitted, the type of the variables being declared is the default type for variables in effect at the time of the variable declaration (paragraph 4.5). If the optional type is omitted and no preset value is specified, the default preset value in effect at the time of the variable declaration, if any, becomes the preset value of the variables being declared. If the optional type is omitted and a preset value is specified, the specified value overrides any default preset value in effect at the time of the variable declaration.

If a preset is specified or implied, all variables of the variable list are preset to that value. A preset may be specified only if the type is simple or a typed structure. The preset value must be assignment-compatible with the underlying simple type of the variables being declared (paragraph 6.1.1.1). The effect of the preset is the same as if the preset value were assigned to the variables at the beginning of program execution.

If the preset value has character type and contains more characters than the variable can hold, the rightmost excess characters will be truncated. If the preset value has fewer characters than the variable can hold, it will be left-justified in the variable and excess character positions will be filled with spaces.

If the preset is a numeric value of greater magnitude than the variable can hold, the excess most significant bits will be truncated. If the preset is a value of greater precision than is provided for in the variable, the excess least significant bits will be truncated.

If a preset value is of the form CORAD(<addressable name>), the corresponding variable must be of integer type, and must have at least 16 magnitude bits.

A variable declaration that is preceded by an EXTREF scope modifier may not include a preset.

A variable whose type is simple is allocated according to its type, as illustrated in Figure 4-03. A variable whose type is structured is allocated the number of target computer words that is the size of its type.

Examples

```
VRBL I3S I 3 S $
```

This is an example of an integer type variable, which is a signed string of bits three bits long, and which can assume integer values in the range -3 through +3 inclusively.

```
VRBL I21SP I 21 S P TAG $
```

		Size	
	Signed fixed-point type with length	≤ 16 bits	16 bits (half word)
		> 16 bits ≤ 32 bits	1 word
		> 32 bits	2 words
Variable Type and Allocation	Unsigned fixed-point type with length:	≤ 8 bits	8 bits (quarter word)
		> 8 bits ≤ 15 bits	16 bits (half word)
		> 15 bits ≤ 31 bits	1 word
		> 31 bits	2 words
		Floating-point type T, R, & D	2 words
	Floating-point type S	1 word	
	Boolean type	8 bits (quarter word)	
	Status type	8 bits (quarter word) (N values: $N \leq 255$) 16 bits (half word) (N values: $N > 255$)	
	Character type with number of characters:	= 1	8 bits (quarter word)
		= 2	16 bits (half word)
		> 2	Number of words = $\lceil (\text{number of characters} - 1) / 4 \rceil + 1$; left-justified; truncated to an integer

Figure 4-03. Allocation of Typed Variables

This is an integer type variable of 21 bits, of which 20 are magnitude bits and the leftmost is a sign bit. At compilation time the variable is preset to the value associated with the name TAG. TAG may be specified as either an ntag (paragraph 4.13) or an ltag (paragraph 4.14).

VRBL I16U I 16 U P CORAD(I3S) \$

This integer type variable is preset at compilation time to the address of an addressable unit, I3S. The bit length, 16, is the minimum valid length for a variable that is to be preset to a core address.

VRBL A4U0 A 4 U 0 \$

This is an example of a fixed-point type variable which is four bits long with no fractional bits. It can assume positive values in the range 0 through 15 inclusively.

VRBL A4S0 A 4 S 0 P 5 \$

This is a fixed-point type variable which has three magnitude bits and one sign bit. It has no fractional bits and can assume values in the range -7 through +7 inclusively. It is preset at compilation time to the value 5.

VRBL A4S1 A 4 S 1 \$

This is a fixed-point type variable which has a sign bit and three magnitude bits, one of which is a fractional bit. It can assume the values 0, ± 0.5 , ± 1 , ± 1.5 , ± 2 , ± 2.5 , ± 3 , and ± 3.5 exactly.

VRBL A4SM1 A 4 S -1 \$

This fixed-point type variable is four bits long, including the sign bit, and has one bit of negative scaling. The -1 effectively establishes a range of five bits, except that the ones-position bit is excluded. It can assume the values 0, ± 2 , ± 4 , ± 6 , ± 8 , ± 10 , ± 12 , and ± 14 exactly.

VRBL HOLRITH H 1 P H(5) \$

This is an example of a character type variable declaration specifying one character. It is preset at compilation time to the character code which represents the character 5. This variable does not have the value 5 for numeric computation, and may not be used for numeric computation.

/(U) CM2Y-MAN-PGR-M5049-R04C0

```
VRBL BOOL B $
```

This is a Boolean type variable declaration.

```
VRBL FLT F $
```

This is an example of a floating-point type variable declaration.

```
VRBL STATX S 'LOW', 'MEDIUM', 'HIGH' $
```

This is an example of a status type variable declaration.

```
TYPE VSTRUC (H 40) $  
  FIELD FH3 H 3 0 31 $  
  FIELD FH20 H 20 1 31 $  
END-TYPE VSTRUC $
```

```
VRBL VX VSTRUC P H(THIS IS TYPED H 40) $
```

This is an example of a structured variable. It has the underlying simple type of H 40. The variable contains the fields FH3 and FH20, which are accessed as VX(FH3) and VX(FH20).

4.7 Parameter Variable Declaration

Syntax

```
<parameter variable declaration>  
 ::= [<scope modifier>] PARAMETER <variable list> [<type>]  
    [P <preset value>], <register number> $  
  
<register number>  
 ::= <numeric constant expression>
```

Semantics

A parameter variable declaration specifies the attributes of one or more variables and a target machine A register or register pair to be used when the variable is used as a formal parameter of a procedure.

- <scope modifier> - Optional. Refer to scope modifier definition (paragraph 4.1).
- PARAMETER - A language keyword indicating a parameter variable declaration.
- <variable list> - The names of the variables being declared.
- <type> - Optional. The type of the variables being declared.
- P - Optional. A language keyword (not reserved) indicating that a preset value is being specified.
- <preset value> - Optional. The value with which the variables are preset.
- <register number> - Specification of the target machine A register or the first of an A register pair to be used in parameter passage if one of the variables being declared is used as a formal parameter of a procedure.

The meanings and restrictions on the variable list and preset value are the same as in a variable declaration. The type cannot be an untyped structure. If the underlying type is character, its maximum length is eight.

The value of the register number expression must be in the range [0,7].

A parameter variable declaration is a special form of a variable declaration. Throughout this manual, any reference to variables includes variables declared by means of a parameter variable declaration.

Examples

	<u>Using</u> <u>Parameter Variable Declarations</u>	<u>Using</u> <u>Variable Declarations</u>
Formal Parameter Definition	PARAMETER XX I 24 S , 0 \$ PARAMETER YY I 24 S , 6 \$:	VRBL XX I 24 S \$ VRBL YY I 24 S \$:
Procedure Definition	PROCEDURE P INPUT XX OUTPUT YY \$ SA A0,XX,K3 : LA A6,YY,K3 END-PROC P \$:	PROCEDURE P INPUT XX OUTPUT YY \$: : END-PROC P \$:
Procedure Call	P INPUT Q OUTPUT R \$ LA A0,Q,K3 LBJ B6,P SA A6,R,K3	P INPUT Q OUTPUT R \$ LA A3,Q,K3 SA A3,XX,K3 LBJ B6,P LA A4,YY,K3 SA A4,R,K3

This example illustrates the use of parameter variables to pass procedure parameters in a procedure call. The difference between the two sequences is the location of the store instruction in input parameter passage, and the location of the load instruction in output parameter passage.

```
(EXTREF) PARAMETER BUFADDR I 16 U, 6 $
(EXTREF) PARAMETER NWORDS I 9 U, 7 $
(EXTREF) PARAMETER DSKSECTR I 16 U, 0 $
(EXTREF) PARAMETER ERROR B, 0 $
(EXTREF) PROCEDURE DISKREAD INPUT BUFADDR,
          DSKSECTR,NWORDS OUTPUT ERROR $
```



```
TABLE DATAREC V 512 1 $  
END-TABLE DATAREC $  
VRBL NXTSECTR I 14 U $  
VRBL RDERROR B $
```

```
DISKREAD INPUT CORAD(DATAREC), NXTSECTR, 512  
OUTPUT RDERROR $
```

```
LA A6, DATAREC, K0  
LA A0, NXTSECTR, K2  
LA A7, 512  
LBJ B6, DISKREAD  
SA A0, RDERROR, K5
```

The parameter variable declarations and program attribute declaration would be useful for facilitating procedure calls from a CMS-2 program to an assembly language disk read routine that expects input parameters in fixed A registers and produces an output in a specific A register. The declaration and procedure call illustrate the generated code that could be produced from a call to the assembly language routine.

Note

A parameter variable cannot be used as a formal parameter of a function.

Implementation Notes

The target machine A registers are used during expression evaluation. If the actual argument list of a procedure invocation (paragraph 6.1.1.6) contains expressions that must be evaluated using the A registers, and the formal parameters of the procedure contain one or more parameter variables, conflicts in the use of the A registers can occur.

Conflicts can also occur if the formal parameters of a procedure contain two or more parameter variables that use the same register. If either of these cases occur, the compiler will issue a warning message.

4.8 Table Declaration

Syntax

```
<table block>
  ::= <table declaration> [<table information>&] <end-table
      declaration>

<table declaration>
  ::= [<scope modifier>] TABLE <table name> <table type> <item
      allocation> [INDIRECT] <table subscript declaration>
      [<major index>] $

<table name>
  ::= <name>

<table type>
  ::= H
  ::= V

<item allocation>
  ::= NONE
  ::= MEDIUM
  ::= DENSE
  ::= <number of words>
  ::= (<type>)

<table subscript declaration>
  ::= <number of items>
  ::= <status type>
  ::= <tag name>

<number of items>
  ::= <numeric constant expression>

<status type>
  ::= <status type specification>
  ::= <status type name>

<status type name>
  ::= <simple type name>

<major index>
  ::= <name>

<table information>
  ::= <structure information>
  ::= <like-table declaration>
```

```
 ::= <subtable declaration>
 ::= <item-area declaration>

<end-table declaration>
 ::= END-TABLE <table name> $
```

Semantics

A table is an ordered set of homogeneous data called items. The items may be simple, structured, or both. The table block specifies the name of the table, the manner of allocating the table to memory, and the number of items in the table.

- | | |
|-------------------------------|---|
| <scope modifier> | - Optional. Refer to the scope modifier definition (paragraph 4.1). |
| TABLE | - A language keyword indicating a table declaration. |
| <table name> | - The name of the table being declared. |
| <table type> | - An H or V, specifying that the table is to be allocated to memory horizontally or vertically, respectively. |
| <item allocation> | - An indication of the manner in which an item of the table is to be allocated. |
| INDIRECT | - Optional. A language keyword indicating that the table is not to be allocated and its name is to be used as a surrogate for other addressable data. |
| <table subscript declaration> | - A declaration of the values that can be used to access items of the table. |
| <major index> | - Optional. Declaration of a variable that will be used to indicate the number of active entries in the table. |
| <table information> | - Optional. Structure information defining the structure of the table items. |

item-area declarations declaring related variables, and like-table declarations and subtable declarations declaring related tables.

END-TABLE

- A language keyword indicating the end of a table block.

The table name that appears on the end-table declaration must be the same as the table name that appears on the table declaration.

A table name cannot be D, H, or O.

Only the table declaration and the structure information of a table block specify attributes of items of the table. Item-area declarations, like-table declarations, and subtable declarations declare entities whose attributes are related to those of the table being declared.

A table block has the effect of declaring a unique anonymous structured type whose name is different from all other names in the compilation module in the form

```
TYPE name <item allocation> $  
    <structure information> &  
END-TYPE name $
```

where the item allocation and structure information are exact copies of the corresponding text from the table block, then declaring the table without structure information and with its item allocation being the name of the anonymous type in parentheses. The anonymous type declaration must be valid.

Each item of the table has the attributes specified by the associated anonymous type. In particular, each item is allocated the number of target machine words that is the size of the anonymous type.

If the anonymous type is compiler-packed, then the table is compiler-packed. If the anonymous type is user-packed, then the table is user-packed. If the anonymous type is a typed structure, then the table is called item-typed.

A vertical table, indicated by a table type of V, is allocated to memory in sequential memory addresses. All words of the first item are allocated to sequential memory addresses, followed by all words of the second item, etc.

A horizontal table, indicated by a table type of H, is allocated to memory in sequential memory addresses. The first word of the first item is followed by the first word of the second item, which is followed by the first word of the third item, etc. The first word of the last item is followed by the second word of the first item, which is followed by the second word of the second item. In general, word k of item n is followed by word k of item $n+1$, unless item n is the last item. Word k of the last item is followed by word $k+1$ of the first item, unless word k is the last word.

No field of a horizontal table may cross a word boundary.

Figure 4-04 illustrates the difference in address allocation between horizontal and vertical storage for a table consisting of three words per item.

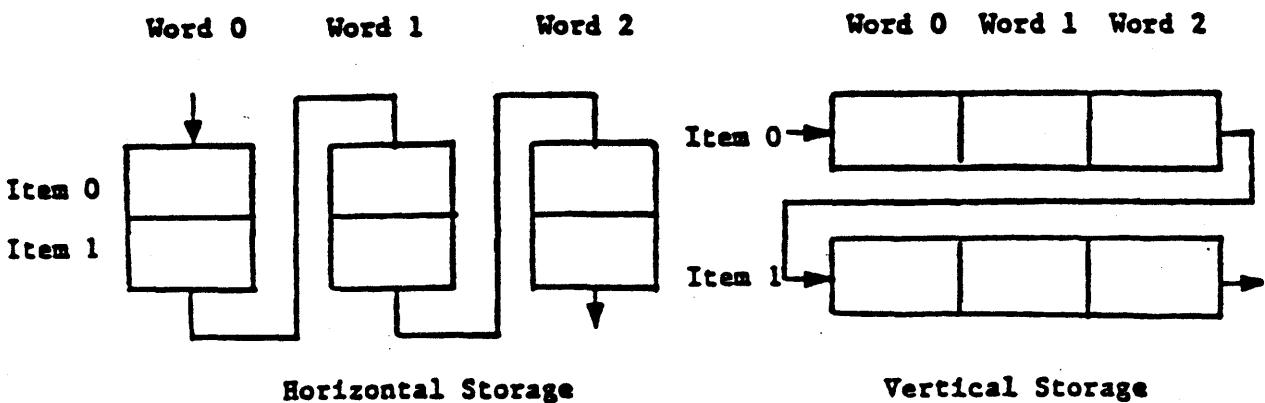


Figure 4-04. Table Storage Addressing Sequence

If the table subscript declaration is a numeric constant expression, its value specifies the number of items of the table. The value must be a nonnegative integer. If the number of items is n , the items are numbered from 0 through $n-1$ inclusive. The items may be accessed by integer subscript values in the range $[0, n-1]$.

If the table subscript declaration is a status type, there is one item of the table for each value of the type. The first item

corresponds to the first value of the type, the second item corresponds to the second value, etc. The items may be accessed only by status expressions of the same status type.

If an ltag name appears as a table subscript declaration, the number of items of the table is determined by the value of the name at load time -- that is, the value specified at load time, or, if no such value is specified, the compile time value. The value of the name at load time must be nonnegative.

If an ltag name appears as a table subscript declaration, the allocation of the table cannot be specified by an address declaration and fields of the table cannot be preset.

The scope of the ltag must be greater than or equal to the scope of the table.

The optional major index may not appear if the table subscript declaration is a status type. The major index implicitly declares a variable whose value is the number of active items in the table. The active items are the first items of the table. The major index is of type I 16 S. The name of the major index has the same scope as the name of the table. The name of the major index may not appear in a variable declaration or item-area declaration in the same scope as the table declaration. A major index name may be used with only one table in a scope. It is the programmer's responsibility to maintain the value of the major index. The major index is not preset.

A table is called indirect if the keyword INDIRECT appears in its declaration. No memory is allocated for an indirect table itself, but one target machine word which corresponds to the table name is allocated. An indirect table must not be referenced before the address of another addressable datum has been assigned to its name. The assignment of an address to an indirect table name is achieved through CORAD assignment, either directly or through parameter passage (paragraph 6.1.1.1, and paragraphs 5.2 and 6.1.1.6). When an indirect table is referenced, the actual reference is to the addressable name whose address was last assigned to the indirect table name. All references are based on the structure of the indirect table, without regard for the structure of the actual addressable name being referenced.

Examples

```
TABLE TVA V 4 200 $
END-TABLE TVA $
```

Table TVA is a vertical table containing 200 items of four words each.

```
LOAD-VRBL LV19U 1 9 U P 6 $
TABLE TH7LV6 V (H 7) LV19U $ ''INIT=6''
END-TABLE TH7LV6 $
```

Unless directed otherwise by control statements to the loader program, table TH7LV6 will be allocated six items as specified by the mandatory preset value of 6 in the declaration of ltag LV19U. By default, then, the table is compiled as 12 words long.

```
TABLE TV2 V (I 5 U) 6 $
END-TABLE TV2 $
```

Table TV2 specifies items typed integer, five bits, unsigned. This means that any reference to an item of this table will be treated as though it were an I 5 U variable. The table has six items.

```
TABLE TH3 H (A 15 S 8) TAG $
END-TABLE TH3 $
```

Table TH3 specifies items typed fixed-point with six integer bits, eight fractional bits, and a sign bit. The number of items is designated by the value associated with the name TAG, where TAG is an ntag, ltag, status type, or string name.

```
TABLE TCP V DENSE 5 $
  FIELD ---
  FIELD ---
END-TABLE TCP $
```

This table illustrates the compiler packing designator that will pack in the most compact way allowed by the compiler, with whatever fields are defined. The minimum number of words required to contain this data will determine the number of words per item for the five items defined. It is the most memory-efficient method to construct a data design when only the data is being considered. However, dynamic references to packed data may be less efficient than NONE or MEDIUM packed tables, depending on the bit lengths required by the specified field definitions.

```
TYPE STC S 'OFF', 'INACTIVE', 'ACTIVE', 'DOWN' $
TABLE TST H MEDIUM STC $
  FIELD FLD1 I 8 U $
  FIELD FLD2 I 8 U $
END-TABLE TST $
```

Table TST is a horizontal, medium-packed table with status STC for its subscript declaration. There are four table items corresponding to the four possible STC status values. Items of this

/(U) CM2Y-MAN-PGR-M5049-R04C0

table can only be accessed using subscripts which are status expressions of type STC.

```
TYPE STRUC NONE $  
  FIELD SF1 H 20 $  
  FIELD SF2 I 3 U $  
  FIELD SF3 B $  
END-TYPE STRUC $
```

```
TABLE TVC V (STRUC) INDIRECT 3 $  
  FIELD TF A 13 S 4 $  
END-TABLE TVC $
```

Table TVC is an indirect, vertical table with three items. Each item is compiler-packed (NONE) and contains the fields SF1, SF2, SF3 and TF.

4.8.1 Field Declaration

Syntax

<field declaration>
 ::= FIELD <field name> [<simple type>] [<starting word>
 <starting bit>] [P <preset item>] \$

<field name>
 ::= <name>

<starting word>
 ::= <numeric constant expression>

<starting bit>
 ::= <numeric constant expression>

<preset item>
 ::= <preset value>
 ::= <repeat value> (<preset value>)

<repeat value>
 ::= <numeric constant expression>

Semantics

A field declaration specifies the name of a field of a structured type and the properties of that field.

- | | |
|-----------------|--|
| FIELD | - A language keyword indicating a field declaration. |
| <field name> | - The name of the field being declared. |
| <simple type> | - Optional. The type of the field being declared. |
| <starting word> | - Optional. The number of the word in which the leftmost bit of the field is to be allocated. |
| <starting bit> | - Optional. The bit position of the leftmost bit of the field. |
| P | - Optional. A language keyword (not reserved) indicating that preset values are being specified. |

<preset item> - A value, optionally enclosed in parentheses and preceded by a repeat value, with which the fields of one or more items are to be preset.

The following discussion is stated in terms of the type in which the field declaration appears. This is either an explicit type declaration or the implicit anonymous type declaration associated with a table or array declaration.

Field names must be unique within the type in which they are declared. They do not have to be distinct from any other names in any scope; the same name may be used as a field name in any number of types in a scope, and may also be declared as the name of some other entity in that scope.

If the starting word and starting bit of a field are specified, the field is user-packed. If the starting word and starting bit of a field are not specified, the field is compiler-packed.

Words of a type are numbered sequentially first to last, beginning with 0. Bits of a word are numbered from right to left, from 0 through 31.

The values of the numeric constant expressions that specify the starting word and starting bit must be non-negative integers. The starting word and starting bit must be such that the entire field lies within the type. The starting word must be less than 256.

User-packed fields of a type can be specified in such a manner that they overlap. It is not necessary that all bits of a type be allocated through user-packing.

Any user-packed field whose length is no greater than 32 bits (one target machine word) must be positioned so that it does not cross a word boundary. For any longer user-packed field, the starting bit must be bit 31.

A user-packed integer or fixed-point field whose length is greater than 32 bits (which therefore requires two words of storage and a starting bit of 31) is allocated to two consecutive words of memory in the "folded" representation of the AN/UYK-7. Thus the specified starting word actually contains the least significant 32 bits of the value. (For details of this representation, see M-5048.) Furthermore, the value of the field is manipulated using the AN/UYK-7 double word instructions. In particular, the value is accessed using the double load instruction, which accesses all 64 bits of the two words, and a new value is assigned using the double store instruction, which modifies all 64 bits of

the two words. Thus the apparently unused bits of the two words can be used for other fields only with the greatest caution on the part of the programmer.

No field of a horizontal table may cross a word boundary. In particular, fields of a type that require more than one target machine word may not appear in horizontal tables.

If the optional type is omitted, the type of the field being declared is the default type for fields in effect at the time of the field declaration (paragraph 4.5).

The preset items specify values that are to be used to preset the field in successive items of a table. The first field preset value is used to preset the field in the first item, the second field preset value is used to preset the field in the second item, etc. A preset item may not be used in a field declaration in an explicit type declaration.

The value of the repeat value must be a positive integer. The effect of a preset item that contains a repeat value is the same as if the preset value had been written n times consecutively, where n is the value of the repeat value.

The number of preset values specified, taking into account the effect of repeat values, must not be more than the number of items of the table and must be in the range [1,256]. Specifying fewer preset values than the number of items of the table is permitted; only the first items of the table will be preset.

The preset values must be assignment-compatible with the type of the field being declared (paragraph 6.1.1.1). The effect of the presets is the same as if the preset values were assigned to the fields of the corresponding items at the beginning of execution of the program.

The constraints for CORAD presets of fields are the same as CORAD presets of variables.

Examples

```
TABLE E1 V NONE 400 $
    FIELD ALPHA I 10 S P 5 $
    FIELD SETT A 10 U 4 P 1, 3(4.5), 2, 7.3 $
END-TABLE E1 $
```

/(U) CM2Y-MAN-PGR-M5049-R04C0

E1 is a compiler-packed table. Field ALPHA is a signed integer type field, 10 bits long. It is preset to 5 in item 0. Field SETT is an unsigned fixed-point type field, 10 bits long, including four fractional bits. Field SETT in items 0 through 5 will be preset to 1, 4.5, 4.5, 4.5, 2, and 7.3 respectively.

```
TABLE E2 V (I 12 U) 20 $
  FIELD BETA A 10 U 4 0 13 P 3 $
END-TABLE E2 $
```

E2 is a user-packed table. The table has 20 items of unsigned integer type, each 12 bits long. Field BETA has unsigned fixed-point type and a length of 10 bits, including four fractional bits. This field is defined as being in the first word of each item of the table, starting in bit 13. Field BETA of item 0 is preset to 3.

```
TABLE TEST V 2 5 $
  FIELD VALUE1 I 3 U 0 4 $
  FIELD VALUE2 I 3 U 0 3 $
  FIELD VALUE3 I 2 U 1 2 $
END-TABLE TEST $
```

These declarations could be viewed in storage as shown in Figure 4-05.

```
TABLE NOTYP V 5 4 $
  FIELD FL1 I 14 S 0 13 $
  FIELD FL2 I 11 U 1 10 $
END-TABLE NOTYP $
```

```
TYPE AAA I 14 S $
TYPE BBB I 11 U $
```

```
TABLE TYP V 5 4 $
  FIELD FL1 AAA 0 13 $
  FIELD FL2 BBB 1 10 $
END-TABLE TYP $
```

User-packed tables NOTYP and TYP have the same field structure. The typing of the fields in table TYP is specified by the types AAA and BBB. Field FL1 is a signed integer type field, 14 bits long. Field FL2 is an unsigned integer type field, 11 bits long.

Implementation Note

Only the upper or lower half of a target machine word, and not both, may be preset to a CORAD value.

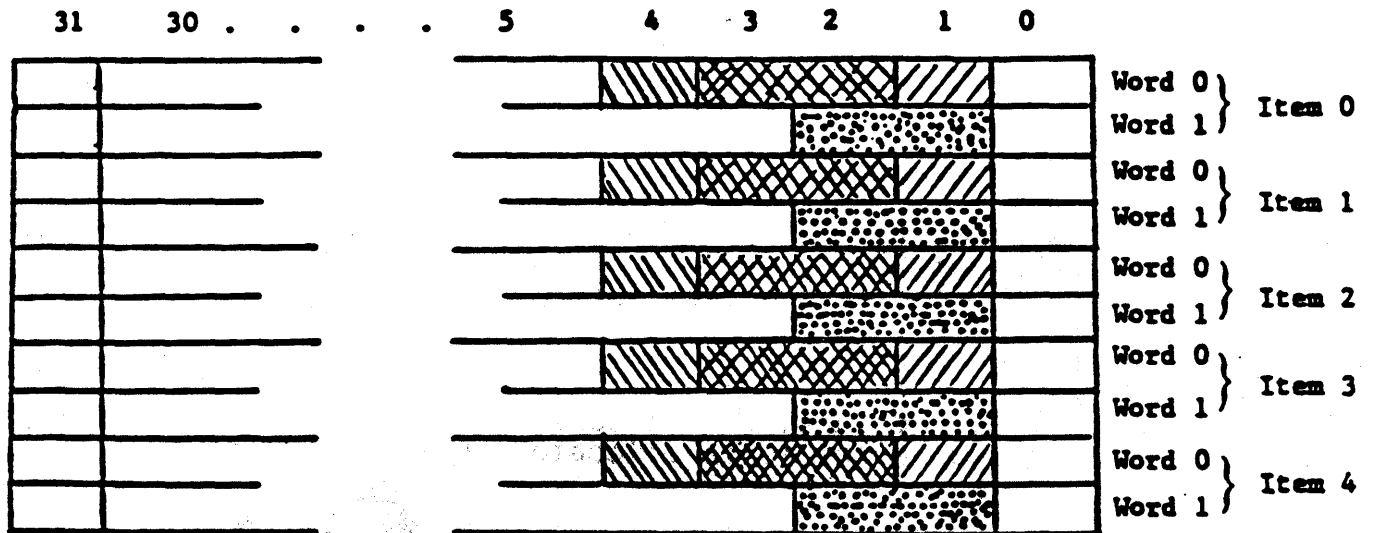



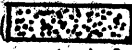


Figure 4-05. Vertical Table Layout (Table TEST)

Notes

- Table TEST is a vertical table (all words of an item are together) of 5 items of 2 words each.
- Fields VALUE1  and VALUE2  are identified with word 0 of every item within the table. Notice the overlapping of the fields .
- Field VALUE3  is identified with word 1 of every item within the table.

4.8.2 Field Overlay Declaration

Syntax

<field overlay declaration>
 ::= <field overlay parent> OVERLAY <field overlay
 sibling>@ \$

<field overlay parent>
 ::= <field name>

<field overlay sibling>
 ::= <field name>
 ::= <numeric constant expression>

Semantics

A field overlay declaration indicates that certain fields of a compiler-packed table are to be allocated in such a manner that they share memory.

- <field overlay parent> - The name of a field in which the bits are to be allocated to the field overlay siblings.
- OVERLAY - A language keyword indicating an overlay declaration.
- <field overlay sibling> - A numeric constant expression or the name of a field that is to be allocated in such a manner that it occupies some or all of the bits of the field overlay parent.

A field overlay declaration may only appear in the type declaration of a compiler-packed type.

In allocating the field overlay siblings, the field overlay parent is considered to be a string of bits, without regard for its structure. The first field overlay sibling is allocated so that its leftmost bit is the leftmost bit of the field overlay parent, the second field overlay sibling is allocated so that its leftmost bit is the first bit to the right of the rightmost bit of the first overlay sibling, etc. In general, each field overlay sibling other than the first is allocated so that its leftmost bit is immediately to the right of the rightmost bit of the previous field overlay sibling. No sibling of character type can be positioned such that any character crosses a word boundary.

If a field overlay sibling is a numeric constant expression, its value must be a non-negative integer. Such a sibling is

interpreted as an unnamed field whose length is given by the numeric constant value. Thus the effect is as if the allocation process skips the number of bits specified by the value.

The field overlay parent must be long enough to contain all of the bits of all field overlay siblings after allocation.

A field name may appear as a field overlay sibling only once in a structured type declaration. There is no limit on the number of times a field name may appear as a field overlay parent.

If a field name appears as both a field overlay parent and a field overlay sibling, its appearance as a sibling must precede all of its appearances as a parent.

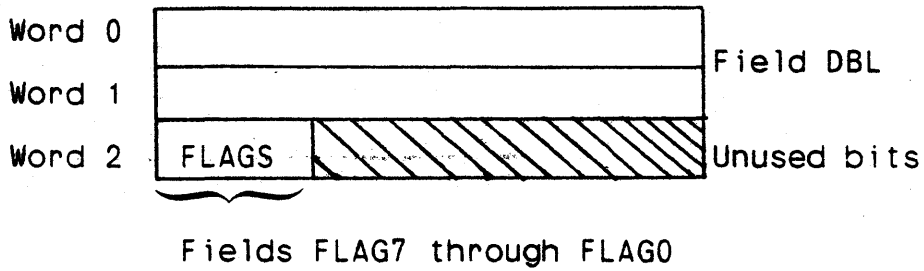
A field overlay sibling whose length is no greater than 32 bits may not cross a target machine word boundary. A field overlay sibling that is a multiword data unit must be allocated on a target machine word boundary.

If a name appearing as a field overlay sibling is the name of both a field in the table block and an ntag, the sibling is interpreted to be the field.

Examples

```
TABLE SAMPLE V MEDIUM 14 $
  FIELD FLAGS I 8 U $
    FIELD FLAG7 B $
    FIELD FLAG6 B $
    FIELD FLAG5 B $
    FIELD FLAG4 B $
    FIELD FLAG3 B $
    FIELD FLAG2 B $
    FIELD FLAG1 B $
    FIELD FLAG0 B $
  FLAGS OVERLAY FLAG7, FLAG6, FLAG5, FLAG4, FLAG3,
    FLAG2, FLAG1, FLAG0 $
  FIELD DBL I 64 S $
END-TABLE SAMPLE $
```

Each item of table SAMPLE will have the following structure:



Because MEDIUM compiler packing is declared, field FLAGS is allocated a full quarter-word. The eight overlay sibling Boolean type fields are densely packed, with each one occupying only a single bit.

Note

A field overlay declaration is syntactically identical to an overlay declaration.

4.8.3 Like-Table Declaration

Syntax

```
<like-table declaration>
 ::= [<scope modifier>] LIKE-TABLE <table name> [<table
      subscript declaration>] [<major index>] $
```

Semantics

A like-table declaration specifies the name of a table whose items have the same attributes as its parent table.

<scope modifier>	- Optional. Refer to the scope modifier definition (paragraph 4.1).
LIKE-TABLE	- A language keyword indicating a like-table declaration.
<table name>	- The name of the table being declared.
<table subscript declaration>	- Optional. A declaration of the values that can be used to access items of the table.
<major index>	- Optional. A declaration of a variable that will be used to indicate the number of active entries in the table.

A like-table declaration may only appear in a table block. The table being declared by the table block is called the parent table of the table being declared by the like-table declaration.

A table declared by a like-table declaration has the same table type and item allocation as its parent table.

There is no relation between the type of the parent table's subscript declaration and the type of the like-table's subscript declaration. If the table subscript declaration is omitted, the table subscript declaration of the parent table is used by default.

The meanings of and constraints on the number of items and the major index in a like-table declaration are the same as in a table declaration.

If an ltag name appears as a subscript declaration, the scope of the ltag must be greater than or equal to the scope of the like-table.

A like-table declaration may not appear in the table block of an indirect table.

The scope of a table declared by a like-table declaration must be no larger than the scope of its parent table. Either may be declared with an attribute declaration (i.e., EXTREF) while the other is declared with an allocation declaration (i.e., EXTDEF).

The like-table declaration is a means of defining a table. Throughout this manual, any reference to a table includes tables declared by means of a like-table declaration.

Example

```
TABLE FLAGS V (B) 10 $
  LIKE-TABLE TFLAGS 5 $
END-TABLE FLAGS $
```

FLAGS and TFLAGS each contain a series of Boolean items. FLAGS contains 10 items and TFLAGS contains five items.

```
TABLE PARENT H MEDIUM 14 $
  FIELD ---
  FIELD ---
  FIELD ---
  LIKE-TABLE LT1 $
  LIKE-TABLE LT2 204 $
END-TABLE PARENT $
```

All of the fields named will be allocated for both the parent table PARENT and the two like-tables. Like-table LT2 has 204 items, while LT1 has 14 items (same as the parent table).

4.8.4 Subtable Declaration

Syntax

```
<subtable declaration>
 ::= [<scope modifier>] SUB-TABLE <table name> <starting
    item> <table subscript declaration> [<major index>] $

<starting item>
 ::= <numeric constant expression>
 ::= <status constant>
```

Semantics

A subtable declaration specifies the name of a table which is part of a larger table. Items of the two tables have the same attributes.

- | | |
|-------------------------------|--|
| <scope modifier> | - Optional. Refer to the scope modifier definition (paragraph 4.1). |
| SUB-TABLE | - A language keyword indicating a subtable declaration. |
| <table name> | - The name of the subtable being declared. |
| <starting item> | - A specification of the item of the larger table that corresponds to the first item of the subtable being declared. |
| <table subscript declaration> | - A declaration of the values that can be used to access items of the subtable. |
| <major index> | - Optional. A declaration of a variable that will be used to indicate the number of active entries in the table. |

A subtable declaration may only appear in a table block. The table being declared by the table block is called the parent table of the table being declared by the subtable declaration.

A table declared by a subtable declaration has the same table type and item allocation as its parent table.

There is no relation between the type of the parent table's subscript declaration and the type of the subtable's subscript declaration.

If the parent table's subscript declaration is a number of items or an ltag name, then the subtable's starting item must be a numeric constant expression whose value is one of the subscript values of the parent table. If the parent table's subscript declaration is a status type, then the subtable's starting item must be a value of that status type.

The first item of the subtable is allocated to the same memory address as the item of the parent table specified by the starting item. As a result, the item of the parent table after the starting item and the second item of the subtable are allocated at the same memory address, the second item of the parent table after the starting item and the third item of the subtable are allocated at the same memory address, etc.

The starting item and the number of items of the subtable declaration must be such that the entire subtable is allocated within the memory that is allocated to the parent table. In addition, the subtable, after any adjustments caused by specifying the values of ltag names at load time (affecting either the subtable, its parent table, or both) must lie entirely within its parent table. The loader will not verify this restriction.

The name of the subtable has the same scope as the name of the parent table. If the parent table declaration is an attribute declaration, then the subtable declaration is also an attribute declaration.

The meanings of and constraints on the number of items and the major index in a subtable declaration are the same as in a table declaration.

If an ltag appears as the subscript declaration, the ltag must have a scope greater than or equal to the subtable.

A subtable declaration may not appear in the table block of an indirect table.

The subtable declaration is a means of defining a table with special allocation properties. Throughout this manual, any reference to tables is also a reference to subtables.

Examples

```

TABLE HORIZ H 3 10 HMI $
  SUB-TABLE HORIZST 5 4 HMIST $
  FIELD ---
  FIELD ---
END-TABLE HORIZ $

```

Subtable HORIZST starts in the sixth item of parent table HORIZ and overlays the rest of the table except for the last item. (HMIST is the major index for subtable HORIZST.) Figure 4-06 illustrates the literal sequence of subtable HORIZST in memory.

```

TABLE VERT V 4 5 $
  SUB-TABLE VERTST 0 3 $
  FIELD ---
  FIELD ---
END-TABLE VERT $

```

Subtable VERTST overlays the first three items of parent table VERT. Figure 4-07 illustrates the literal sequence of subtable VERTST in memory.

```

TABLE TSTAT V 4 S 'ZERO','ONE','TWO','THREE' $
  SUBTABLE STSTAT 'ONE' 2 $
END-TABLE TSTAT $

```

Subtable STSTAT will start in the second item of table TSTAT and overlay a total of two items. Subtable STSTAT must be referenced with numeric subscripts.

```

TABLE TSTAT1 V 4 S 'ZERO','ONE','TWO','THREE' $
  SUBTABLE STSTAT1 'ONE' S 'BLACK','WHITE' $
END-TABLE TSTAT1 $

```

Subtable STSTAT will start in the second item of table TSTAT and overlay a total of two items. Subtable STSTAT must be referenced with the status subscripts 'BLACK' and 'WHITE'.

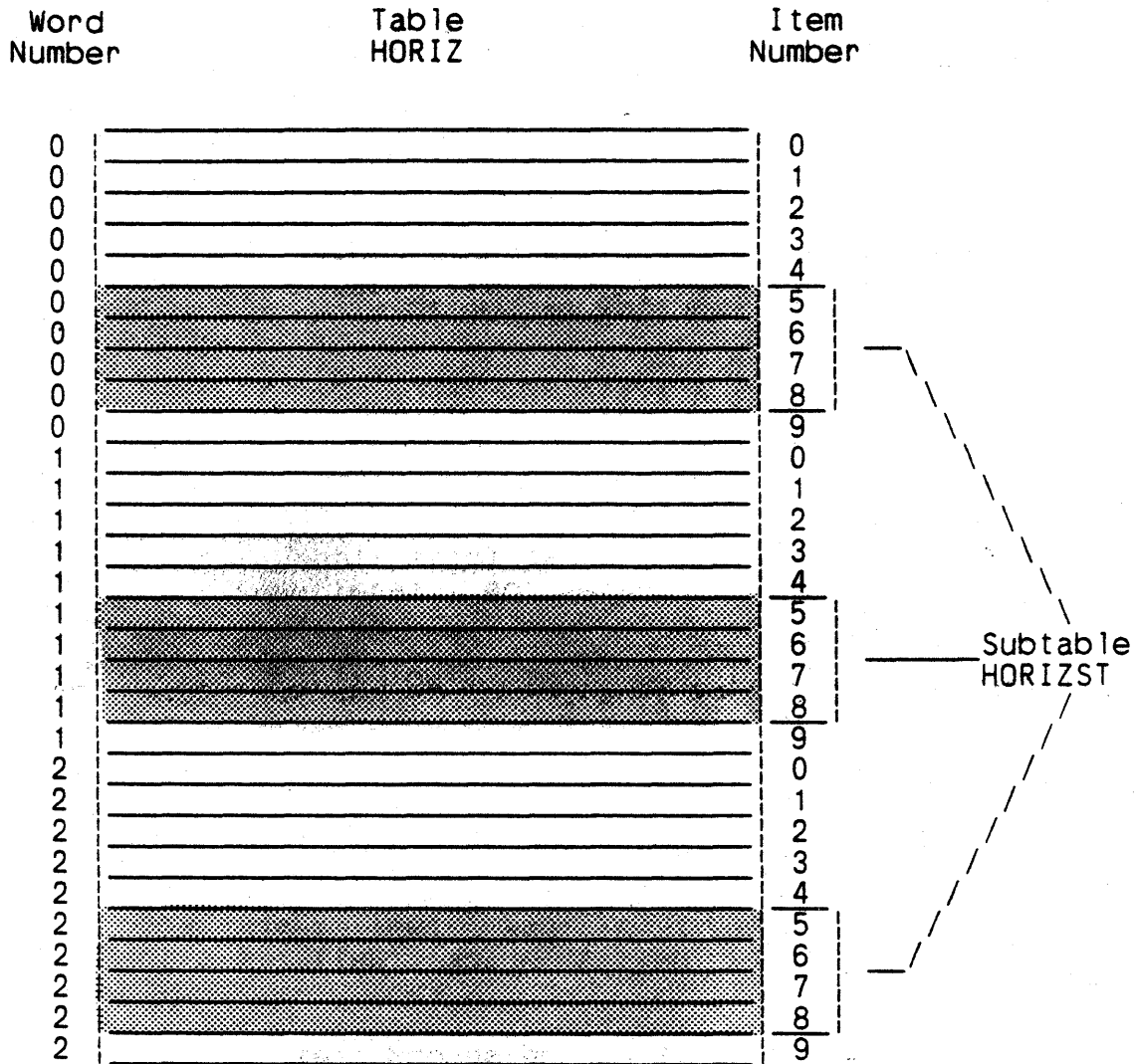


Figure 4-06. Internal Structure of Subtable HORIZST

Word Number	Table VERT	Item Number
0		0
1		0
2		0
3		0
0		1
1		1
2		1
3		1
0		2
1		2
2		2
3		2
0		3
1		3
2		3
3		3
0		4
1		4
2		4
3		4

Subtable VERTST

Figure 4-07. Internal Structure of Subtable VERTST

4.8.5 Item-Area Declaration

Syntax

```
<item-area declaration>  
 ::= [<scope modifier>] ITEM-AREA <variable name>@ $
```

Semantics

An item-area declaration specifies one or more names to be the names of variables having the same attributes as the items of a table.

<scope modifier> - Optional. Refer to the scope modifier definition (paragraph 4.1).

ITEM-AREA - A language keyword indicating an item-area declaration (see Figure 4-08).

<variable name> - The name of a variable being declared.

An item-area declaration may only appear in a table block or an array block. The table or array being declared by the block is called the parent table of the variables being declared by the item-area declaration (see Figure 4-08).

The effect of an item-area declaration is the same as declaring variables in a variable declaration using the type (explicit or anonymous implicit) of the parent table's declaration.

The scope of the variables declared by an item-area declaration must not be larger than the scope of their parent table. Either may be declared with an attribute declaration (i.e., EXTREF) while the other is declared with an allocation declaration (i.e., EXTDEF).

An item-area declaration is a means of declaring a variable. Throughout this manual, any reference to a variable includes variables declared by means of an item-area declaration.

Examples

```
TABLE TABA H 32 256 $  
  ITEM-AREA ITM $  
END-TABLE TABA $
```

Table TABA is a horizontal table composed of 256 items, each of which is 32 words long. Item-area ITM is a separate block of 32 words.

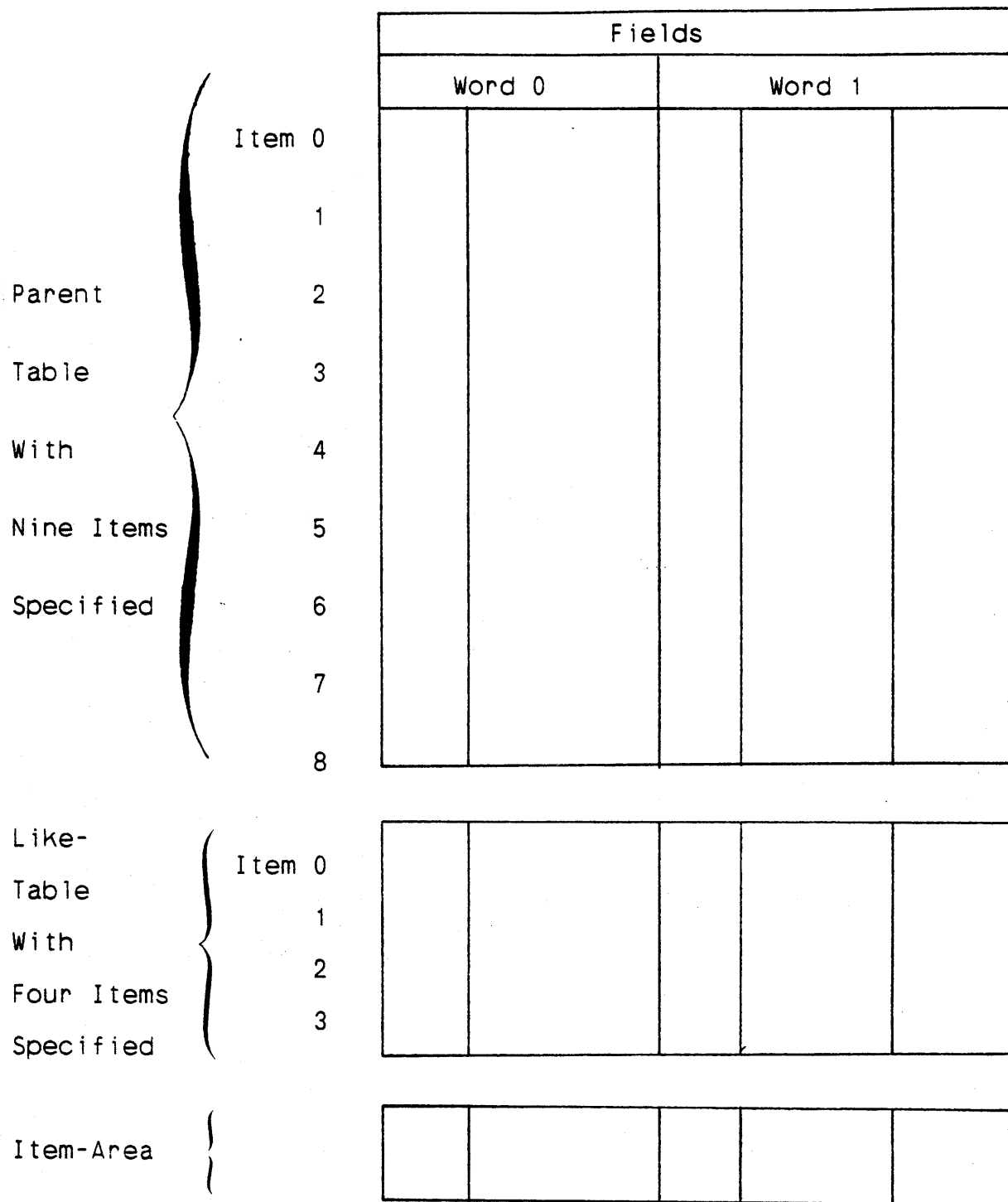


Figure 4-08. Parent Table Relationships

4.9 Array Declaration

Syntax

<array block>
 ::= <array declaration> [<array information>] <end-table
 declaration>

<array declaration>
 ::= [<scope modifier>] TABLE <table name> A <item
 allocation> [INDIRECT] <subscript declaration>@ \$

<subscript declaration>
 ::= <numeric constant expression>
 ::= <status type>

<array information>
 ::= <structure information>
 ::= <item-area declaration>

Semantics

An array is a multidimensioned table. An array block specifies the name of the array, the subscripts to be used in accessing its items, and the manner of allocating it to memory.

- | | |
|-------------------------|---|
| <scope modifier> | - Optional. Refer to the scope modifier definition (paragraph 4.1). |
| TABLE | - A language keyword indicating a table declaration. |
| <table name> | - The name of the array being declared. |
| A | - A language keyword (not reserved) indicating an array declaration. |
| <item allocation> | - An indication of the manner in which an item of the table is to be allocated. |
| INDIRECT | - Optional. A language keyword indicating that the array is not to be allocated, and its name is to be used as a surrogate for other addressable units. |
| <subscript declaration> | - A declaration of the values that can be used in a subscript position to access items of the array. |

<array information> - Optional. Declarations defining the structure and attributes of items of the array or related variables.

The table name that appears on the end-table declaration must be the same as the table name that appears on the array declaration.

A table name cannot be D, H, or O.

An array declaration must contain at least one, and no more than seven, subscript declarations. The minimum value of a numeric subscript declaration is 0 and the maximum value is 65,535. The number of the subscript declaration(s) is the dimension of the array. An array of dimension one is identical to a vertical table.

The product of the number of words per item and the values of subscript declarations is the total number of words allocated for any given array. The maximum number of words permitted in an array is 65,535.

The meaning of the keyword INDIRECT is the same as its meaning in a table declaration.

The first item of an array is the item corresponding to all numeric subscripts equal to zero and all status subscripts equal to their first values. The items of an array are allocated to memory in sequential memory locations, with the first subscript varying most rapidly, the second subscript varying next most rapidly, etc. Figure 4-09 illustrates this process and the correspondence between the way an item is referenced and its address. If the value of a numeric subscript declaration is n , a subscript in that position must be in the range $[0, n-1]$. For a status subscript declaration, a subscript in that position must be a status value that is assignment-compatible with the type of the declaration.

An array always has vertical table type. Figure 4-09 illustrates one way to conceptualize a 3-dimensional array.

Like-tables cannot be declared in an array block.

An array is a special form of table. Throughout this manual, any reference to tables is also a reference to arrays.

Examples

```
TABLE ARY A (I 13 S) INDIRECT 3, 4, 5 $  
END-TABLE ARY $
```

/(U) CM2Y-MAN-PGR-M5049-R04C0

Table ARY is an indirect array of three dimensions. The items are typed I 13 S.

```
TABLE ARA A MEDIUM 4,4 $
  FIELD A12S8 A 12 S 8 $
  FIELD I3U I 3 U $
END-TABLE ARA $
```

ARA is a direct 4-by-4 (16 items) array with medium compiler packing of fields.

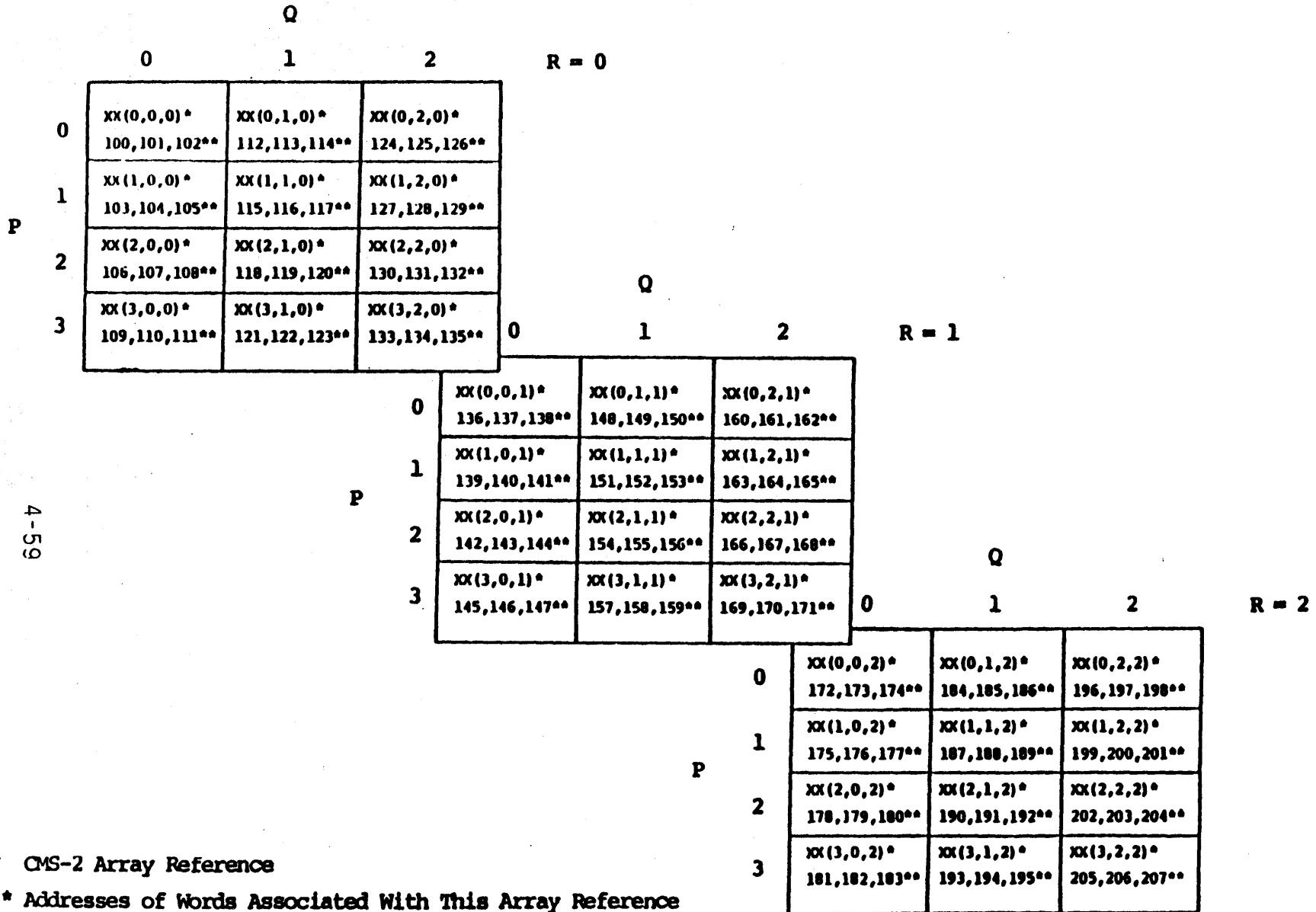
```
TABLE MATRIX A (F) 8,8 $
  ITEM-AREA ITM $
END-TABLE MATRIX $
```

Table MATRIX is an 8-by-8 array of two-word items. Each item, along with the item-area, represents a single floating-point operand.

```
TABLE XX A 3 4,3,3 $
END-TABLE XX $
```

A reference to an item of this array, XX(P,Q,R), indicates which three-word piece of this array is chosen. Figure 4-09 illustrates the correspondence between the reference and the actual words selected in memory.

Assume that the first word in the array is at location 100.



* CMS-2 Array Reference

** Addresses of Words Associated With This Array Reference

Figure 4-09. A 3-Dimensional Array

4.10 Preset Value Declaration

Syntax

```
<preset value declaration>
  ::= [<pre-settable name>] DATA <preset entry> $

<pre-settable name>
  ::= <variable name>
  ::= <table name>

<preset entry>
  ::= <preset semi-entry> [<preset semi-entry>]
  ::= <character constant>

<preset semi-entry>
  ::= <numeric constant value> [,<fractional bits>]
  ::= CORAD (<addressable name>)
```

Semantics

A preset value declaration specifies a preset value.

- <pre-settable name> - Optional. The name of a variable or table whose value is to be preset with the first preset entry.
- DATA - A language keyword indicating a preset value declaration.
- <preset entry> - Specification of the value of one or more target machine words.
- <preset semi-entry> - Specification of the value of a target machine half-word.

Each preset value declaration specifies a preset value for an integral number of target machine words. Only a preset value declaration whose preset entry is a character constant can specify a preset value for more than one word.

If the optional pre-settable name is present, the preset value declaration specifies a preset value for the first target machine word(s) allocated to that name. If the pre-settable name is omitted and the preset value declaration immediately follows another preset value declaration, a preset value is being specified for one or more target machine words that immediately follow the word(s) preset by the previous preset value declaration. If the pre-settable name is omitted and the preset value declaration does not immediately follow another preset value declaration, the effect is undefined.

If a preset entry consists of a single numeric constant value, the value is right-justified in the word. It may be scaled and/or a fraction.

If a preset entry consists of two numeric constant values, the value of the first numeric constant value is right-justified in the upper half-word and the second is right-justified in the lower half-word. Both numeric constant values must be no longer than 16 bits, and each must be an integer.

If a preset entry consists of a character constant, the value of the constant is left-justified and blank-filled on the right in the minimum number of target machine words necessary to contain the value.

If a preset entry is of the form CORAD(<addressable name>), the indicated pre-settable name must be integer type or fixed-point type with zero fractional bits and at least 16 magnitude bits; the value is right-justified in the word.

If a preset entry consists of a numeric constant value and a CORAD value, the values of each are right-justified in their respective half-words. The numeric preset value must be no longer than 16 bits.

Examples

```
VRBL TAC I 32 S $
TAC DATA 77 $
```

The whole word located at the address allocated to TAC has an initial preset value of 77.

```
TABLE DICT V 3 1 $
END-TABLE DICT $
DICT DATA -64 $
DATA 7 0 $
DATA 11 , 5 $
```

The first word of the table DICT has an initial value of -64. The second word has an initial value of 7 in the upper half and 0 in the lower half. This third word of the table has an initial preset value of 11 scaled 5.

```
VRBL HOLVB H 6 $
HOLVB DATA H(TWOWDS) $
```

This statement will preset two words beginning at the location allocated to HOLVB with the characters TWOWDS (left-justified with two trailing blanks).

/(U) CM2Y-MAN-PGR-M5049-R04C0

```
VRBL XX I 16 S $  
VRBL ZZ A 32 S 6 $  
TABLE TABA V 2 1 $  
END-TABLE TABA $  
IW EQUALS 0(100000) $  
TABA DATA IW CORAD(ZZ) $  
DATA CORAD(XX) $
```

The first word of data unit TABA will have the octal value 100000 in the upper half-word and the 16-bit address of ZZ in the lower half-word. The second word of TABA will have 0 in the upper half-word and the 16-bit address of XX in the lower half-word.

Implementation Note

Only the upper or lower half of a target machine word, and not both, may be preset to a CORAD value.

4.11 Overlay Declaration

Syntax

<overlay declaration>
 ::= <overlay parent> OVERLAY <overlay sibling>@ \$

<overlay parent>
 ::= <variable name>
 ::= <table name>

<overlay sibling>
 ::= <variable name>
 ::= <table name>
 ::= <numeric constant expression>

Semantics

An overlay declaration indicates that certain variables and tables are to be allocated in such a manner that they share memory.

<overlay parent> - The name of a variable or table whose bits are to be overlaid with the overlay siblings.

OVERLAY - A language keyword indicating an overlay declaration.

<overlay sibling> - A numeric constant expression or the name of a variable or table that is to be allocated in such a manner that it occupies some or all of the bits of the overlay parent.

In allocating the overlay siblings, the overlay parent is considered to be a string of bits, without regard for its structure. The first overlay sibling is allocated so that its leftmost bit is the leftmost bit of the overlay parent, the second overlay sibling is allocated so that its leftmost bit is the first bit to the right of the rightmost bit of the first overlay sibling, etc. In general, each overlay sibling other than the first is allocated so that its leftmost bit is immediately to the right of the rightmost bit of the previous overlay sibling. No sibling of character type can be positioned such that any character crosses a word boundary.

If an overlay sibling is a numeric constant expression, its value must be a nonnegative integer. Such a sibling is interpreted as an unnamed variable whose length is given by the value. Thus the effect is as if the allocation process skips the number of bits specified by the value.

Any overlay sibling that is a table must be allocated on a target machine word boundary.

An overlay sibling whose length is no greater than 32 bits may not cross a target machine word boundary.

An overlay sibling that is a multiword data unit must be allocated on a target machine word boundary.

A table declared by means of a subtable declaration may not appear as an overlay sibling.

The overlay parent must be long enough to contain all of the bits of all of the overlay siblings after allocation.

The allocation declarations for the overlay parent and all the overlay siblings must appear in the same data block before the OVERLAY declaration. The overlay parent's scope must contain or be the same as the scope of all the overlay siblings.

A variable name or a table name may appear as an overlay sibling only once in a data block. There is no limit on the number of times a name may appear as an overlay parent.

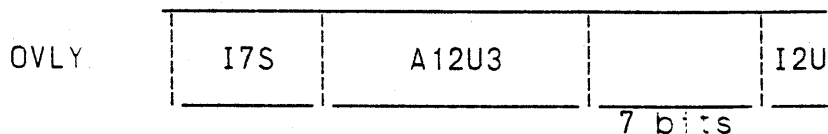
If a variable name or table name appears as both an overlay parent and an overlay sibling, its appearance as a sibling must precede all of its appearances as a parent.

A variable or table whose name appears as an overlay sibling may not be allocated by means of an address declaration. An indirect table cannot be either an overlay parent or an overlay sibling.

Examples

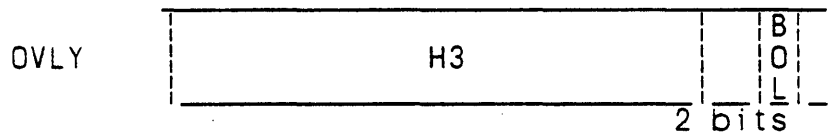
```
VRBL I7S I 7 S $
VRBL A12U3 A 12 U 3 $
VRBL I2U I 2 U $
VRBL OVLY I 28 S $
OVLY OVERLAY I7S,A12U3,7,I2U $
```

The 28-bit variable OVLY will be overlaid from left to right as follows:



VRBL H3 H 3 \$
VRBL BOL B \$
OVLY OVERLAY H3,2,BOL \$

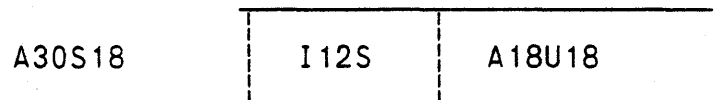
The same 28-bit variable OVLY will again be overlaid as follows:



Note that the sum of the bits needed for the overlay siblings may be less than the total number of bits allocated to the parent.

VRBL A30S18 A 30 S 18 \$
VRBL I12S I 12 S \$
VRBL A18U18 A 18 U 18 \$
A30S18 OVERLAY I12S, A18U18 \$

This overlay declaration will allow the A30S18 data unit to be referenced by its integer and fractional parts through the overlay siblings I12S and A18U18.



VRBL CARD H 80 \$
VRBL COL1TO8 H 8 "CARD COLUMNS 1 THRU 8" \$
VRBL COL13 H 1 "CARD COLUMN 13" \$
CARD OVERLAY COL1TO8, 32, COL13 \$

The use of 32 is necessary to achieve the desired overlay structure in this example.

Note

An overlay declaration is syntactically identical to a field overlay declaration.

4.12 Text Substitution Declaration

Syntax

<substitution declaration>
 ::= <string name> <substitution type> <simple string> \$

<substitution type>
 ::= MEANS
 ::= EXCHANGE

<string name>
 ::= <name>

Semantics

A text substitution declaration assigns a name to a string of characters. Subsequent appearances of the name are replaced by the character string.

<string name> - The name that is to be replaced by the character string.

MEANS - A language keyword indicating an effective text substitution declaration.

EXCHANGE - A language keyword indicating an actual text substitution declaration.

<simple string> - The character string that is to replace the string name.

The substitution type EXCHANGE specifies that the simple string is to be substituted for all occurrences of the string name, and the result of the substitution is to appear on any program listings and source output files produced during the compilation.

The substitution type MEANS specifies that the simple string is to be effectively substituted for all occurrences of the string name, but the result of the substitution is not to appear on any program listings or source output files produced during the compilation.

The string name will not be replaced by the simple string in occurrences that appear before the text substitution declaration, in COMMENT statements, or in notes.

The simple string may not contain a string name.

The simple string may consist of any characters other than the dollar sign. If the substitution type keyword is terminated by a

blank character, the simple string begins with the character following that blank. Otherwise, the keyword must be terminated by a delimiter, and the simple string begins with that delimiter. The simple string includes all characters up to (but not including) the statement terminator (\$), and may be empty.

The string name may be a reserved word.

Examples

```
ATTR MEANS A 27 S 10 $
```

The simple string begins with the first character following the space after MEANS, and ends with the last character before the \$.

```
COEFF MEANS (XX/2) + YY$
```

The character string begins with the left parenthesis and ends with YY.

The following statements illustrate the use of the above examples:

```
Statement: VRBL NAME ATTR $
Effect:    VRBL NAME A 27 S 10 $
```

```
Statement: SET FIXPT TO ZZ + COEFF $
Effect:    SET FIXPT TO ZZ + (XX/2) + YY $
```

The compiler listings and source listings will not show the change.

```
CNT EXCHANGE CNTR $
```

All references to the name CNT will be compiled as references to the name CNTR. Any source output and listings will reflect this name change. CNT is a reserved word denoting an intrinsic function. Any references to that function within the scope of this substitution declaration will also be affected by the change.

Note

The string name may not be redeclared in a nested scope. When the string name appears in a nested scope, it is replaced by the associated simple string.

4.13 Compile-Time Constant Declaration

Syntax

```
<compile-time constant declaration>
  ::= <ntag declaration>
  ::= <ntag declaration>

<ntag declaration>
  ::= <ntag name> EQUALS <ntag expression> $

<ntag name>
  ::= <name>

<ntag expression>
  ::= <ntag expression> <numeric operator> <ntag primary>
  ::= [<unary numeric operator>] <ntag primary>

<numeric operator>
  ::= <additive operator>
  ::= *
  ::= /

<additive operator>
  ::= +
  ::= -

<ntag primary>
  ::= <numeric constant>
  ::= <ntag name>

<ntag declaration>
  ::= <ntag name> EQUALS <ntag expression> $

<ntag name>
  ::= <name>

<ntag expression>
  ::= <ntag expression> <numeric operator> <ntag primary>
  ::= <ntag expression> <numeric operator> <ntag name>
  ::= <ntag expression> <numeric operator> <ntag name>
  ::= <ntag expression> - <addressable name>
  ::= [<unary numeric operator>] <ntag name>

<ntag expression>
  ::= <ntag expression> + <addressable name>
  ::= <ntag expression> + <addressable name>
  ::= <ntag expression> <additive operator> <ntag primary>
  ::= <ntag expression> <additive operator> <ntag name>
  ::= <addressable name>
```

Semantics

A compile-time constant declaration assigns a name to a numeric constant. There are two kinds of compile-time constants: ntags (numeric tags), which are pure numeric constants, and rtags (relative tags), which are numeric constants that depend on the relative addresses of one or more pairs of addressable names.

- <ntag name> - A name being declared as an ntag.
- <rtag name> - A name being declared as an rtag.
- EQUALS - A language keyword indicating a compile-time constant declaration, load-time constant declaration, or an address declaration.
- <ntag expression> - An expression that defines an ntag.
- <rtag expression> - An expression that defines an rtag.
- <atag expression> - An expression whose value is an offset to the address of an addressable name.
- <numeric operator> - An operator indicating one of the numeric operations of addition, subtraction, multiplication, division, or exponentiation (paragraph 5.3.1).
- <additive operator> - An operator indicating one of the numeric operations of addition or subtraction (paragraph 5.3.1).

Expressions that appear in a compile-time constant declaration must be parenthesis-free, and are interpreted from left to right (all operators have equal precedence). The expressions are evaluated using the rules for constant arithmetic (paragraph 5.3.1).

An atag expression must have a nonnegative integer value.

The value of an addressable name, when used in an rtag expression or an atag expression, is the target machine sy-address assigned to the name.

An ntag name may be used as a primary in any numeric expression. An rtag name may be used as a primary in any numeric expression other than a numeric constant expression.

The allocation declarations of all addressable names in the compile-time constant declaration must appear in the same system element as the compile-time constant declaration.

When an ntag is formed by subtracting an addressable name from an atag expression, the name and the expression must have the same allocation attributes. The allocation attributes of an addressable name and an atag expression are determined by recursively applying the following definitions:

- a. The allocation attributes of an addressable name are the system element in which its allocation declaration appears and the type of the data block in which the name is allocated.
- b. The allocation attributes of an atag expression formed by adding an addressable name to an ntag expression or an rtag expression are the allocation attributes of the addressable name.
- c. The allocation attributes of an atag expression formed by combining an atag expression and an ntag primary or an rtag name using an additive operator are the allocation attributes of the atag expression.

Examples

```
VRBL ALO I 8 S $
VRBL AL1 I 7 U $
TABLE TMED V MEDIUM 5 $
  FIELD ---
  FIELD --- $
END-TABLE TMED $
VRBL AL2 A 15 S 3 $
```

```
NT1 EQUALS 4 $
```

NT1 is an ntag of value 4.

```
NT2 EQUALS NT1 + 3 $
```

The value of the ntag NT2(7) is derived from the value of NT1.

```
NT3 EQUALS -NT2 $
```

This declaration makes use of the unary numeric operator to complement the value of NT2, giving -7 for the value of the ntag NT3.

NT4 EQUALS NT1 + NT2 * NT3 \$

This is an example of the left-to-right evaluation of an ntag expression. The value of the ntag NT4 is $-77 = (4 + 7) \times (-7)$.

RT2 EQUALS TMED - AL0 \$

RT2 is an rtag made up of an atag expression (TMED) minus an allocatable name (AL0). The value is the difference between the two addresses for the allocatable names.

RT3 EQUALS RT2 + NT2 \$

This illustrates an rtag expression plus an ntag expression.

RT4 EQUALS NT3 * RT2 \$

RT5 EQUALS RT3 - RT4 \$

RT6 EQUALS -RT5 \$

RT7 EQUALS NT1 + 2 + AL2 - AL1 \$

RT8 EQUALS RT3 + RT2 + AL2 - AL1 \$

RT9 EQUALS RT3 + RT2 + AL2 + RT8 - AL1 \$

The six examples listed above illustrate several of the combinations of ntag, rtag, and atag expressions that may be used in defining an rtag.

Implementation Note

The two types of compile-time constant declarations are syntactically identical to each other and to the address declaration.

4.14 Load-Time Variable Declaration

Syntax

```
<ltag declaration>
  ::= <loadvrbl form>
  ::= <nitems form>

<loadvrbl form>
  ::= LOAD-VRBL <ltag list> <integer type> P <numeric constant
  expression> $

<ltag list>
  ::= <ltag name>
  ::= (<ltag name>@)

<ltag name>
  ::= <name>

<integer type>
  ::= <integer type specification>
  ::= <simple type name>

<nitems form>
  ::= NITEMS (<ltag name>) EQUALS <ntag expression> $
```

Semantics

A load-time variable declaration specifies one or more integer variables whose values can be changed only at load time.

- | | |
|-------------------------------|---|
| LOAD-VRBL | - A language keyword indicating a load-time variable declaration. |
| <ltag list> | - The names of the load-time variables being declared. |
| <integer type> | - A specification of the type of the load-time variables. |
| P | - A language keyword indicating that a preset value is being specified. |
| <numeric constant expression> | - The compile-time value of the variables being declared. |

- | | |
|-------------------|---|
| NITEMS | - A language keyword indicating a load-time variable declaration. |
| <ltag name> | - The name of the load-time variable being declared. |
| EQUALS | - A language keyword indicating a compile-time constant declaration, an address declaration, or a load-time variable declaration. |
| <ntag expression> | - An expression whose value is the compile-time value of the variable being declared. |

If an ltag name appears in a numeric expression, its behavior with respect to the scaling rules is the same as any other variable of the same type; that is, although the value of the ltag is constant during program execution, the constant scaling rules do not apply.

If the integer type of a load-time variable form is a simple type name, it must be the name of a simple integer type.

If no value is specified for an ltag name at load-time, the value of the ltag name is the value of the numeric constant expression that appears in its declaration.

The value of the numeric constant expression that appears in an ltag declaration must be an integer capable of being represented by the specified type.

The type of a load-time variable declared using the nitems form is I 15 U.

Examples

```
LOAD-VRBL LVI3U I 3 U P 7 $
LOAD-VRBL LVI16S I 16 S P -642 $
LOAD-VRBL (LVI1, LVI2, LVI3) I 5 S P 12 $
```

In these examples, LVI3U represents the value 7 and LVI16S the value -642. The string of ltags LVI1, LVI2, and LVI3 have a value of 12. They may be changed at load time.

```
NITEMS (LVI) EQUALS 8 $
```

LVI represents the value 8, unless it is changed at load time.

Implementation Note

An ltag name satisfies the usual CMS-2 scope rules at compilation time. That is, its scope is global if its declaration appears in the major header or a system data block, its scope is local if its declaration appears in a minor header or a local data block and its scope is subprogram if it appears in a subprogram data block. However, at load time all ltag names are treated as having global scope and must therefore be distinct from each other (they are automatically distinct from all other names having global scope).

4.15 Address Declaration

Syntax

<address declaration>
 ::= <allocatable name> EQUALS <atag expression> \$

<allocatable name>
 ::= <variable name>
 ::= <table name>
 ::= <switch name>
 ::= <procedure switch name>
 ::= <file name>
 ::= <format name>
 ::= <inputlist name>
 ::= <outputlist name>
 ::= <stringform name>

Semantics

An address declaration specifies a target machine address for a variable, table, switch, procedure switch, file, format, inputlist, outputlist, or stringform. The address specified is an offset to the address of some other addressable name.

<allocatable name> - The name of the variable, table, switch, procedure switch, file, format, inputlist, outputlist, or stringform whose address is being specified.

EQUALS - A language keyword indicating a compile-time constant declaration, load-time constant declaration, or an address declaration.

<atag expression> - An expression whose value is the address assigned to the allocatable name.

An address declaration must appear in the same system element as the allocation declaration of the allocatable name whose address is being specified.

The address specified for an allocatable name cannot depend on the address of the allocatable name in any fashion.

An allocatable name whose address is specified by means of an address declaration cannot appear as an overlay sibling (paragraph 4.11).

The address of a table specified by means of a subtable declaration may not be specified by means of an address declaration.

Examples

```
TABSIZ EQUALS 100 $
TABLE TAB V 4 TABSIZ $
  ITEM-AREA FIRST, LAST $
END-TABLE TAB $
FIRST EQUALS TAB $
LAST EQUALS TABSIZ-1*4+TAB $
```

Item-areas FIRST and LAST are allocated over the first and last items of table TAB. The number of items in table TAB is defined by the ntag name TABSIZ. Note the left-to-right evaluation of the second allocation declaration. A more obvious statement

```
LAST EQUALS TAB + TABSIZ*4-4 $
```

erroneously attempts to produce the same result. This tag expression is equivalent to the parenthesized formula

```
((TAB+TABSIZ)*4)-4.
```

This formula not only yields an undesired result, it would also be flagged as erroneous by the compiler because of the attempted multiplication involving an atag expression.

Note

Address declarations and compile-time constant declarations are syntactically identical.

Implementation Note

No memory is reserved by the compiler for an allocatable name whose address is specified in an address declaration.

4.16 System Index Declaration

Syntax

```
<system index declaration>  
 ::= SYS-INDEX <system index specification>@ $  
  
<system index specification>  
 ::= <register number> <system index name>  
  
<system index name>  
 ::= <name>
```

Semantics

A system index declaration specifies the names of integer variables whose values are to be held in target machine B registers during execution of the program.

SYS-INDEX - A language keyword indicating a system index declaration.

<system index specification> - The name of a system index, preceded by a register number.

<register number> - A numeric constant expression specifying a target machine B register that is to hold the value of the corresponding system index.

<system index name> - The name of a system index being declared.

A system index is of type I 16 U.

The value of the register number expression must be an integer in the range [1,5].

A given index register number may appear in only one system index specification.

Example

```
SYS-INDEX 1 XX, 2 YY, 3 ZZ $
```

The names XX, YY, and ZZ specify target machine hardware registers B1, B2, and B3, respectively.

4.17 Local Index Declaration

Syntax

```
<local index declaration>  
 ::= LOC-INDEX <local index name>@ $
```

```
<local index name>  
 ::= <name>
```

Semantics

A local index declaration specifies the names of integer variables whose values are to be held in target machine B registers during execution of a subprogram.

LOC-INDEX - A language keyword indicating a local index declaration.

<local index name> - The name of a local index being declared.

A local index is of type I 16 U.

Any number of local indexes may be declared in a subprogram. If the number of target machine B registers available to hold local index values is less than the number of local index names, the extra local indexes will be assigned to memory locations.

Examples

```
PROCEDURE EXAMPLE $  
  LOC-INDEX J,IND $
```

J and IND will be the names of local indexes for the duration of procedure EXAMPLE.

4.18 Procedure DeclarationSyntax

```

<procedure declaration>
    ::= [<declaration modifier>] PROCEDURE <procedure name>
        [<formal procedure parameters>] $

<procedure name>
    ::= <name>

<formal procedure parameters>
    ::= <formal i/o parameters> [EXIT <formal exit parameter>@]

<formal i/o parameters>
    ::= [INPUT <formal input parameter>@] [OUTPUT -<formal output
        parameter>@]

<formal input parameter>
    ::= <variable name>
    ::= <table name>
    ::= <system index name>
    ::= <core address receptacle>

<formal output parameter>
    ::= <variable name>
    ::= <table name>
    ::= <system index name>

<formal exit parameter>
    ::= <name>

```

Semantics

A procedure declaration specifies the name of a user procedure and its formal parameters.

- | | |
|------------------------|--|
| <declaration modifier> | - Optional. Refer to declaration modifier definition. |
| PROCEDURE | - A language keyword indicating a procedure declaration. |
| <procedure name> | - The name of the procedure being declared. |
| INPUT | - Optional. A language keyword indicating that one or more formal input parameters is being specified. |

- <formal input parameter> - Specification of a variable, table, system index, or table surrogate whose value is to be replaced by the value of the corresponding actual input parameter at procedure invocation (paragraph 6.1.1.6).
- OUTPUT - Optional. A language keyword indicating that one or more formal output parameters is being specified.
- <formal output parameter> - Specification of a variable, table, or system index whose value is to be transmitted to the calling procedure at the end of procedure execution (paragraph 6.1.1.6).
- EXIT - Optional. A language keyword indicating that one or more formal exit parameters is being specified.
- <formal exit parameter> - Specification of a name that can be used in a procedure return phrase.

If the optional declaration modifier is omitted or is (EXTDEF), the procedure declaration may only appear at the beginning of a procedure block.

If the (EXTREF) or (TRANSREF) allocation modifier appears, the declaration may appear in either a system data block or a local data block. If the (LOCREF) allocation modifier appears, the declaration may only appear in a local data block.

A name used as a formal input parameter or a formal output parameter must be known in a scope that contains the scope of the procedure name. That is, if a procedure name has global scope, the formal input parameters and formal output parameters must have global scope.

Formal exit parameters have subprogram scope.

A procedure declaration may contain a maximum of 25 formal input parameters, 25 formal output parameters, and 10 formal exit parameters.

Examples

(EXTDEF) PROCEDURE ERROR \$

ERROR is defined as a global procedure. It has no formal parameters.

PROCEDURE TEST INPUT V1,V2 OUTPUT V3 \$

TEST is defined as a local procedure. It has two formal input parameters, V1 and V2, which will contain input values when the first statement of the procedure is executed. The contents of the formal output parameter V3 will be transferred to the actual output parameter on return from the procedure.

PROCEDURE ALPHA EXIT KHI, PSI, OMEGA \$

Procedure ALPHA has four exit points, only one of which returns control to the statement following the call to ALPHA. The other three exit points--KHI, PSI, and OMEGA--return control to statement labels identified in the procedure call statement within the calling subprogram (paragraph 6.1.1.6).

4.19 Executive Procedure Declaration

Syntax

```
<executive procedure declaration>  
 ::= [<declaration modifier>] EXEC-PROC <procedure name>  
    [INPUT <formal input parameter>@] $
```

Semantics

An executive procedure declaration specifies the name and formal input parameters of an executive procedure, which is a procedure that executes in the target machine task state but is called from the executive state.

- <declaration modifier> - Optional. Refer to the declaration modifier definition.
- EXEC-PROC - A language keyword indicating an executive procedure declaration.
- <procedure name> - The name of the executive procedure being declared.
- INPUT - Optional. Language keyword indicating that one or more formal input parameters is being specified.
- <formal input parameter> - Specification of a variable, table, table surrogate, or system index whose value is to be replaced by the value of the corresponding actual input parameter prior to the executive state program calling the procedure.

A name used as a formal input parameter or a formal output parameter must be known in a scope that contains the scope of the procedure name. That is, if a procedure name has global scope, the formal input parameters must have global scope.

Executive procedure linkage is the responsibility of the programmer, since no return linkage is generated at either the entry point or the exit point.

A procedure declaration may contain a maximum of 25 formal input parameters, 25 formal output parameters, and 10 formal exit parameters.

Examples

```
EXEC-PROC EXPROC1 $  
EXEC-PROC EXP2 INPUT VI10S,A12S3,ITEMA1 $
```

These statements declare the beginning of executive procedure blocks named EXPROC1 and EXP2. EXP2 has three formal inputs.

Note

An executive program is defined as a program that performs executive functions. It probably executes in the executive state. However, an executive procedure is a user procedure, which therefore executes in the task state.

4.20 Function Declaration

Syntax

```
<function declaration>
 ::= [<declaration modifier>] FUNCTION <function name>
    ([<formal input parameter>@]) [<function type>] $

<function name>
 ::= <name>

<function type>
 ::= <type>
```

Semantics

A function declaration specifies the name of a function, its formal input parameters, and, optionally, the type of the value it returns.

- <declaration modifier> - Optional. Refer to the declaration modifier definition.
- FUNCTION - A language keyword indicating a function declaration.
- <function name> - The name of the function being declared.
- <formal input parameter> - Optional. Specification of a variable, table, table surrogate, or system index whose value is to be replaced by the value of the corresponding actual input parameter when the function is evaluated (paragraph 5.2).
- <function type> - Optional. The type of the value returned by the function.

If the optional declaration modifier is omitted or is (EXTDEF), the function declaration may only appear at the beginning of a function block.

The function type cannot be an untyped structure. If the function type is a typed structure, the function value has the attributes of the associated simple type.

If the (EXTREF) or (TRANSREF) allocation modifier appears, the declaration may appear in either a system data block or a local data block. If the (LOCREF) allocation modifier appears, the declaration may only appear in a local data block.

A name used as formal input parameter must be known in a scope that contains the scope of the function name.

A function declaration may contain a maximum of 25 formal input parameters.

If the optional function type is omitted, the type of the function being declared is the default type for variables in effect at the time of the function declaration (paragraph 4.5). The default preset value for variables in effect at the time of the function declaration, if any, has no effect on the declaration.

Examples

```
FUNCTION FUN (A1) A 12 S 5 $
```

In this example FUN is defined as a function with one formal input parameter, A1, and an output value type of A 12 S 5.

```
TYPE FTYPE F $
```

```
FUNCTION LESSER (VB1,VB2) FTYPE $
```

In this example LESSER has two inputs and its floating point type is specified using a type declaration.

```
TYPE I12U I 12 U $
```

```
FUNCTION RANDOM () I12U $
```

In this example RANDOM has no inputs and its type is specified using a type declaration.

4.21 Label Switch Declaration

Syntax

```
<label switch block>  
 ::= <indexed label switch block>  
 ::= <double label switch block>  
 ::= <item label switch block>
```

Semantics

A label switch block specifies one or two label switches and the names of the statements to which control is transferred when a switch branch phrase of the appropriate type is executed.

4.21.1 Indexed Label Switch Declaration

Syntax

```
<indexed label switch block>
  ::= <label switch declaration> <label switch point>&
     <end-switch declaration>

<label switch declaration>
  ::= SWITCH <label switch name> $

<label switch name>
  ::= <name>

<label switch point>
  ::= [S] <statement name> $

<end-switch declaration>
  ::= END-SWITCH <label switch name> $
```

Semantics

An indexed label switch block specifies the name of an indexed label switch and the names of the statements to which control is transferred when a corresponding indexed branch phrase is executed.

- SWITCH - A language keyword indicating a label switch declaration.
- <label switch name> - The name of the indexed label switch being declared.
- <switch point> - The name of a statement (Section 6), optionally preceded by S, to which control is to be transferred by use of an indexed branch phrase.
- END-SWITCH - A language keyword indicating the end of a label switch block or a procedure switch block.

The label switch name that appears on the end-switch declaration must be the same as the label switch name that appears on the label switch declaration.

An indexed label switch block can appear in a local data block or a subprogram data block. The statement names declared in the block must be the names of statements in the system procedure containing the block.

/(U) CM2Y-MAN-PGR-M5049-R04C0

There is no limit on the number of switch points in an indexed label switch block.

Examples

The following example illustrates an indexed switch block named CHOICE. The indexed switch defined by this block has three switch points (names defined by labels in the subsequent procedure).

```
LOC-DD $  
SWITCH CHOICE $  
  ALPHA $  
  BETA $  
  GAMMA $  
END-SWITCH CHOICE $  
END-LOC-DD $  
PROCEDURE SAMPLE $
```

GAMMA.

ALPHA.

BETA.

4.21.2 Double Label Switch Declaration

Syntax

```
<double label switch block>  
 ::= <double label switch declaration> <double switch point>&  
      <end double switch declaration>
```

```
<double label switch declaration>  
 ::= SWITCH <label switch name>, <label switch name> $
```

```
<double switch point>  
 ::= [S] <statement name> [, <statement name>] $
```

```
<end double switch declaration>  
 ::= END-SWITCH <label switch name>, <label switch name> $
```

Semantics

A double label switch declaration specifies two indexed label switches simultaneously.

- | | |
|-----------------------|--|
| SWITCH | - A language keyword indicating a label switch declaration. |
| <label switch name> | - The name of one of the indexed label switches being specified. |
| <double switch point> | - An optional S, followed by one or two statement names, which specify switch points of the indexed label switch blocks being specified. |
| END-SWITCH | - A language keyword indicating the end of a label switch block or a procedure switch block. |

The label switch names in the double label switch declaration are the names of the indexed label switches being declared.

The first statement name in each double switch point is a label switch point of the first indexed label switch named in the double label switch declaration. The second statement name in each double switch point, if present, is a label switch point of the second indexed label switch named in the double label switch declaration.

If one of the indexed label switches being declared has more switch points than the other, it must be the first named switch. The double switch points that contain two statement names must be the first double switch points of the double label switch block.

/(U) CM2Y-MAN-PGR-M5049-R04C0

The indexed label switch names that appear on the end double switch declaration must be the same switch names that appear on the double label switch declaration, and must appear in the same order.

A double label switch block can appear in a local data block or a subprogram data block. The statement names declared in the block must be the names of statements in the system procedure containing the block.

There is no limit on the number of switch points in a double label switch block.

Examples

```
SWITCH DOLOOP, SEARCH $  
  LOOP5, SHORT $  
  AGAIN, LONG $  
  DONE $  
END-SWITCH DOLOOP, SEARCH $
```

Switch DOLOOP has three switch points; switch SEARCH has two switch points:

4.21.3 Item Label Switch Declaration

Syntax

```
<item label switch block>
  ::= <item label switch declaration> <item label switch
      point>& <end-switch declaration>

<item label switch declaration>
  ::= SWITCH <item label switch name> (<switch selector>) $

<item label switch name>
  ::= <name>

<switch selector>
  ::= <variable name>

<item label switch point>
  ::= <switch value>, <statement name> $

<switch value>
  ::= <numeric constant expression>
  ::= <constant>
```

Semantics

An item label switch block specifies the name of an item label switch, the name of a variable whose value governs the setting of the switch, and the names of statements to which control is transferred when a corresponding item branch phrase is executed.

- | | |
|--------------------------|---|
| SWITCH | - A language keyword indicating a label switch declaration. |
| <item label switch name> | - The name of the item label switch being declared. |
| <switch selector> | - The name of a variable whose value at the time of execution of a corresponding item branch phrase specifies the statement to be executed. |
| <switch value> | - A value to be compared to the value of the switch selector at the time of execution of a corresponding item branch phrase. |

- <statement name> - The name of a statement (Section 6) to which control is to be transferred by use of an item branch phrase.
- END-SWITCH - A language keyword indicating the end of a label switch block or a procedure switch block.

The item switch name that appears on the end-switch declaration must be the same as the switch name that appears on the item switch declaration.

A switch selector must be of a simple type. The switch values must be of the same mode as the switch selector.

An item switch block can appear in a local data block or in a subprogram data block. The statement names declared in the block must be the names of statements in the system procedure containing the block.

There is no limit on the number of switch points in an item switch block.

Examples

```
VRBL FINISH H 4 $
SWITCH SWOFF (FINISH) $
  H(END), ELEMENT $
  H(STOP), UNCOND $
  H(TERM), DONE $
END-SWITCH SWOFF $
```

This declaration defines item switch SWOFF with switch points ELEMENT, UNCOND and DONE. A reference to switch SWOFF will transfer control to one of these switch points, depending upon the value of the variable FINISH. If, for example, the value of FINISH is H(TERM), control will transfer to the statement named DONE.

4.22 Procedure Switch Declarations

Syntax

```
<procedure switch block>  
  ::= <indexed procedure switch block>  
  ::= <double procedure switch block>  
  ::= <item procedure switch block>
```

Semantics

A procedure switch block specifies one or two procedure switches, which are groups of user procedure names, and common formal input-output parameters for them. One of the procedures is invoked by the execution of a procedure switch call of the appropriate type.

The appearance of a name in a procedure switch point within a procedure switch block constitutes an attribute declaration of a procedure having the common formal input-output parameters as its formal procedure parameters. If the name has not been declared previously in the system block, the scope of the procedure name depends on the scope of the switch name: If the switch name is global the procedure name is global; otherwise the procedure name is local to the system procedure in which the procedure switch is declared. If the procedure name has been declared previously, its scope must contain the scope of the procedure switch name. Thus a procedure name having global scope may be a procedure switch point in a procedure switch having local scope, but not vice versa.

4.22.1 Indexed Procedure Switch Declaration

Syntax

```
<indexed procedure switch block>
  ::= <indexed procedure switch declaration> <indexed
      procedure switch point>& <end-procedure-switch
      declaration>

<indexed procedure switch declaration>
  ::= [<scope modifier>] P-SWITCH <indexed procedure switch
      name> [<formal i/o parameters>] $

<indexed procedure switch name>
  ::= <name>

<indexed procedure switch point>
  ::= [P] <procedure name> $

<end-procedure-switch declaration>
  ::= END-SWITCH <indexed procedure switch name> $
  ::= END-P-SW <indexed procedure switch name> $
```

Semantics

An indexed procedure switch block specifies the name of an indexed procedure switch, the names of user procedures that are invoked when a corresponding indexed procedure call phrase is executed, and the formal input and output parameters of those procedures.

- | | |
|---------------------------------|--|
| <scope modifier> | - Optional. Refer to the scope modifier definition (paragraph 4.1). |
| P-SWITCH | - A language keyword indicating a procedure switch declaration. |
| <indexed procedure switch name> | - The name of the indexed procedure switch being declared. |
| <formal i/o parameters> | - Optional. A declaration of the formal input and output parameters for the procedures named in the procedure switch points. |

- <indexed procedure switch point> - The name of a procedure, optionally preceded by P, that is to be invoked by use of an indexed procedure call phrase.
- END-SWITCH - A language keyword indicating the end of a label switch block or a procedure switch block.
- END-P-SW - A language keyword indicating the end of a procedure switch block.

The indexed procedure switch name that appears on the end-procedure-switch declaration must be the same as the indexed procedure switch name that appears on the indexed procedure switch declaration.

There is no limit on the number of procedure switch points in a single procedure switch block.

All the procedures declared as procedure switch points in a procedure switch block must have the same formal I/O parameters, which are those specified in the procedure switch declaration.

Examples

```
P-SWITCH TRIG INPUT ANG, SIDE OUTPUT SOL $
    SIN $
    COS $
    TAN $
END-SWITCH TRIG $
```

This declaration defines procedure switch TRIG, whose formal input parameters are ANG and SIDE, and the formal output parameter is SOL. A reference to procedure switch TRIG transfers control to one of the procedures SIN, COS, or TAN, depending upon an index value of 0, 1, or 2, respectively, in the procedure switch call. If the definition of TRIG appears in a system data block, SIN, COS, and TAN will be defined as global procedures. If TRIG appears in a local data block, SIN, COS, and TAN will be defined as local procedures unless the definition of TRIG is preceded by a global definition of SIN, COS, or TAN.

4.22.2 Double Procedure Switch Declaration

Syntax

```
<double procedure switch block>
 ::= <double procedure switch declaration> <double procedure
      switch point> <end double procedure switch
      declaration>

<double procedure switch declaration>
 ::= [<scope modifier>] P-SWITCH <indexed procedure switch
      name>, <indexed procedure switch name> $

<double procedure switch point>
 ::= [P] <procedure name> [, <procedure name>] $

<end double procedure switch declaration>
 ::= END-SWITCH <indexed procedure switch name>, <indexed
      procedure switch name> $
 ::= END-P-SW <indexed procedure switch name>, <indexed
      procedure switch name> $
```

Semantics

A double procedure switch declaration specifies two indexed procedure switches simultaneously.

- | | |
|---------------------------------|--|
| <scope modifier> | - Optional. Refer to the scope modifier definition (paragraph 4.1). |
| P-SWITCH | - A language keyword indicating a procedure switch declaration. |
| <indexed procedure switch name> | - The name of one of the indexed procedure switches being specified. |
| <double procedure switch point> | - An optional P, followed by one or two procedure names, which specifies switch points of the indexed procedure switch blocks being specified. |
| END-SWITCH | - A language keyword indicating the end of a label switch block or a procedure switch block. |

END-P-SW

- A language keyword indicating the end of a procedure switch block.

The procedure switch names in the double procedure switch declaration are the names of the double procedure switches being declared.

The first procedure name in each double switch point is a procedure switch point of the first indexed procedure switch named in the double procedure switch declaration. The second procedure name in each double switch point, if present, is a procedure switch point of the second procedure switch named in the double procedure switch declaration.

If one of the indexed procedure switches being declared has more switch points than the other, it must be the first named switch. The double switch points that contain two procedure names must be the first double switch points of the double procedure switch block.

The indexed procedure switch names that appear on the end double procedure switch declaration must be the same switch names that appear on the double procedure switch declaration and must appear in the same order.

There is no limit on the number of switch points in a double procedure switch block.

Examples

```
P-SWITCH PLANE, TRAIN $
  PROP, PULLMAN $
  TURBO, FREIGHT $
  JET, DIESEL $
  LAG, STEAM $
  FOG $
END-SWITCH PLANE, TRAIN $
```

Switch PLANE references five procedures; switch TRAIN references four.

Note

The procedures declared in a double procedure switch block can not have formal parameters of any kind.

4.22.3 Item Procedure Switch Declaration

Syntax

```
<item procedure switch block>
  ::= <item procedure switch declaration> <item procedure
      switch point> <end-procedure-switch declaration>

<item procedure switch declaration>
  ::= [<scope modifier>] P-SWITCH <item procedure switch name>
      (<switch selector>) [<formal i/o parameters>] $

<item procedure switch name>
  ::= <name>

<item procedure switch point>
  ::= <switch value>, <procedure name> $
```

Semantics

An item procedure switch block specifies the name of an item procedure switch, the name of a variable whose values govern the setting of the switch, the names of user procedures that are invoked when a corresponding item procedure call phrase is executed, and the common formal input and output parameters of the procedures.

- | | |
|------------------------------|---|
| <scope modifier> | - Optional. Refer to the scope modifier definition (paragraph 4.1). |
| P-SWITCH | - A language keyword indicating a procedure switch declaration. |
| <item procedure switch name> | - The name of the item procedure switch being declared. |
| <switch selector> | - The name of a variable whose value at the time of execution of a corresponding item procedure switch call phrase specifies the procedure to be invoked. |
| <formal i/o parameters> | - Optional. A declaration of the formal input and output parameters for the procedures named in the procedure switch points. |

- <switch value> - A value to be compared to the value of the switch selector at the time of execution of a corresponding item procedure call phrase.
- <procedure name> - The name of a procedure to be invoked by use of an item procedure call phrase.
- END-SWITCH - A language keyword indicating the end of a label switch block or a procedure switch block.
- END-P-SW - A language keyword indicating the end of a procedure switch block.

The item switch name that appears on the end-procedure-switch declaration must be the same as the item switch name that appears on the item procedure switch declaration.

The scope of the switch selector must be the same as the scope of the item procedure switch.

A switch selector must be of a simple type. The switch values must be of the same mode as the switch selector.

There is no limit on the number of switch points in an item procedure switch block.

Examples

```
VRBL ERTYPE S 'N', 'W', 'E', 'F' $
P-SWITCH ERROR (ERTYPE) OUTPUT MESSAGE $
  'E' , ERRORMES $
  'W' , WARNMES $
  'F' , FATALMES $
END-SWITCH ERROR $
```

This data structure is an item procedure switch declaration named ERROR. Three procedures are declared within the block: ERRORMES, WARNMES, FATALMES. Each produces the formal output MESSAGE. When the procedure switch is invoked, the current value of the status type variable ERTYPE is compared to the status constants 'E', 'W', and 'F'; one of the three procedures will be called if a match is found.

/(U) CM2Y-MAN-PGR-M5049-R04C0

```
P-SWITCH LINK (MTYPE) $  
O(12) ; MTPA $  
O(22) ; MTPB $  
O(32) ; MTPC $  
END-SWITCH LINK $
```

Procedure MTPA, MTPB, or MTPC will be invoked when procedure switch LINK is called, if variable MTYPE matches one of the switch values.

4.23 File Declaration

Syntax

```
<file declaration>
  ::= <standard file declaration>
  ::= <nonstandard file declaration>

<standard file declaration>
  ::= [<scope modifier>] FILE <file name> <file specification>
     <standard hardware name> [<file status>] $

<nonstandard file declaration>
  ::= [<scope modifier>] FILE <file name> <file specification>
     <nonstandard hardware name> [<file status>]
     [WITHLBL] $

<file name>
  ::= <name>

<file specification>
  ::= <file type> <record limit> <file structure> <record
     size>

<file type>
  ::= H
  ::= B

<record limit>
  ::= <numeric constant expression>

<file structure>
  ::= R
  ::= V
  ::= S

<record size>
  ::= <numeric constant expression>

<standard hardware name>
  ::= PRINT
  ::= PUNCH
  ::= READ
  ::= OCM

<nonstandard hardware name>
  ::= MT1
  ::= MT2
  ::= MT3
  ::= MT4
  ::= MT5
```

::= MT6
::= MT7
::= MT8
::= MT9
::= MT10
::= MT11
::= MT12
::= MT13
::= MT14
::= MT15
::= MT16
::= PPTR
::= PPTP
::= <installation hardware name>

<installation hardware name>
::= <name>

<file status>
::= <status constant>@

Semantics

A file declaration specifies the name by which the file is referenced in the program, the form of the data in the file, the maximum number of records in any subfile, the relation between physical records and logical records, the size of the buffer for the file, the absence or presence of a header record, and a means for recognizing various conditions that can occur during input/output operations.

- | | |
|------------------|--|
| <scope modifier> | - Optional. Refer to the scope modifier definition (paragraph 4.1). |
| FILE | - A language keyword indicating a file declaration. |
| <file name> | - The name by which the file is referenced in the program. |
| <file type> | - An H or B, indicating that the data of the file consists entirely of character data or that it is in the target machine internal encoded form, respectively. |

- <record limit> - A numeric constant expression that specifies the maximum number of records permitted in any subfile of the file.
- <file structure> - An R, V, or S, indicating that the file consists of records having rigid length or variable length, or that the file has a stream organization, respectively.
- <record size> - A numeric constant expression that specifies the length of the buffer associated with the file.
- <nonstandard hardware name> - A specification of the target machine peripheral device associated with a nonstandard file.
- <standard hardware name> - A specification of the target machine peripheral device associated with a standard file.
- <file status> - A list of status constants to be used in testing conditions that can arise during an input/output operation.
- WITHLBL - Optional. A language keyword indicating that the nonstandard file being declared contains a header record.
- PRINT - A language keyword specifying the standard hardcopy device.
- PUNCH - A language keyword specifying the standard output device.
- READ - A language keyword specifying the standard input device.
- OCM - A language keyword specifying the operator's terminal (operator communication medium).

A file declaration may appear in a program only if the compiler option MONITOR has been specified.

Files are classified as standard or nonstandard. The standard files are the printer, card reader, card punch, and operator communication medium. Their names are PRINT, READ, PUNCH, and OCM, respectively; their names are identical to their standard hardware names as used in a standard file declaration. The properties of standard files are predefined and no file declaration is required for them. The values of the predefinitions are given at the end of this section.

The properties specified in a standard file declaration will override the predefined properties for that file.

No properties of nonstandard files are predefined. A file declaration is required for each nonstandard file referenced in a program.

The file name is the name by which the file is referenced in the program. The language keywords PRINT, READ, PUNCH, and OCM are reserved to the predefined standard files. Even when one of the standard files is redefined by means of a standard file declaration, its file name cannot be one of these four keywords.

The file type specifies the basic form of the data in the file. H signifies that all of the data are in the form of character strings, which usually means that the data must be converted as they are transferred between the file and the program's data areas. B signifies that the data is in the target machine internal form.

The value of the record limit expression must be a nonnegative integer. It specifies the maximum number of records permitted in any subfile of the file. A value of 0 is a convention indicating that any number is permitted.

The file structure specifies the form of physical records and the relation between physical and logical records. A file structure of R specifies that all physical records have the same size (rigid length), specified by the record size. A file structure of V specifies that the size of physical records varies (variable length), depending on the amount of data transferred as the record is created, with the maximum size specified by record size. A file structure of S specifies that the entire file is to be treated as a single stream of data (stream organization), with the length of physical records specified by record size.

For file structures of rigid length and variable length records, logical records and physical records are identical. For a file with stream organization, there is no relation between physical

and logical records--each logical record begins where the previous one ended in the stream. The value of the record size expression specifies the size of the buffer associated with the file and is related to the size of the physical records, as described above. If the file type is binary, the value of the record size expression is the number of target machine words in the buffer. If the file type is character, the value of the record size expression is the number of characters in the buffer.

A record size of 0 is valid only for files with rigid length or variable length records, provided no formatted input or output is done with the file. In this case no buffer is used; data are transferred directly between the file and the program's data areas.

A nonstandard or standard hardware name is a name by which the user specifies the hardware device containing the file. The names MT1 through MT16 (magnetic tape units), PPTR (paper tape reader), and PPTP (paper tape punch) are common to all CMS-2Y installations.

An installation hardware name is a name by which a hardware device is referenced at a particular installation. The name must be no longer than three characters, and must be defined to the CMS-2Y monitor prior to execution of a program that uses the name. If the name has not been defined to the monitor at the time the file is opened, the message 'HARDWARE NAME NOT RECOGNIZED' will be output to the operator and the execution will be aborted.

A specific hardware name can be present in more than one file declaration, but two files with the same hardware name cannot be open at the same time.

The optional file status provides a means for testing the various states which can occur as a result of an input or output operation. The actual testing is performed using conditional I/O statements. The status constants of the file status are associated with the values of a control word returned by the CMS-2Y monitor after any input or output operation. The first constant is associated with the value 0, the second with the value 1, etc. The meanings of the values are given in the following table:

<u>Status Value</u>	<u>Meaning</u>
0	- Indicates that I/O is in progress.
1	- Indicates that I/O has completed normally.
2	- Indicates that a sentinel (instead of data), e.g., an end of file (EOF), has been encountered.
3	- Indicates an unrecoverable hardware error.
4	- Indicates an invalid I/O request packet or sequence of such packets. (These packets are built and maintained by the CMS-2Y monitor for standard peripheral device assignments.)
5	- Indicates either that there is no more data to be input from the peripheral device, or that the device has insufficient room for the requested output.
6	- Indicates that a legal logical unit number has not been assigned to a specific I/O device.

It is not necessary to assign status constants to all of the states. If a particular state is of no interest it can be bypassed by not specifying a status constant at its position, resulting in two successive commas in the list. A string of trailing commas can itself be omitted, with the result that the specified constants are assigned to the lowest values of the status word and the conditions corresponding to the highest values are not testable.

The option WITHLBL specifies that the file has been, or is to be, created with a header record (label). It is not necessary to specify WITHLBL for existing files that have header records, but it is the programmer's responsibility to bypass the header record in such a case, and to realize that the first data record is record 1, not record 0.

The following table presents the effect of the predefined file properties for the standard files:

<u>Standard File</u>	<u>Effective Declaration</u>
CARD READER	FILE READ H 0 R 80 READ \$
CARD PUNCH	FILE PUNCH H 0 R 80 PUNCH \$
PRINTER	FILE PRINT H 0 R 132 PRINT \$
OCM	FILE OCM H 0 R 80 OCM \$

Notes

Since all I/O operations are completed in some fashion before the monitor returns control to the program, the busy condition should not be detectable.

The optional file status has been omitted in all of the standard file predefinitions. To test any states for one of these files, the programmer must include a standard file declaration in the program with the file status specified.

4.24 Format Declaration

Syntax

<format declaration>
 ::= [<scope modifier>] FORMAT <format name> <format list> \$

<format name>
 ::= <name>

<format list>
 ::= <format item>@

<format item>
 ::= [<item replicator>]<format descriptor>
 ::= <format positioner>
 ::= [<item replicator>]<character constant>
 ::= <item replicator>(<format list>)
 ::= [<format item>]/[<format item>]

<item replicator>
 ::= <numeric constant value>

<format descriptor>
 ::= I <format specification>
 ::= O <format specification>
 ::= F <format specification>
 ::= E <format specification>
 ::= A <field width>
 ::= L <field width>

<format positioner>
 ::= <field width> X
 ::= T <position>

<format specification>
 ::= <field width> [.<fraction size>]

<field width>
 ::= <numeric constant value>

<fraction size>
 ::= <numeric constant value>

<position>
 ::= <numeric constant value>

Semantics

A format declaration specifies the types of conversions to be performed when converting data between its internal form and a character string form.

- <scope modifier> - Optional. Refer to the scope modifier definition (paragraph 4.1).
- FORMAT - A language keyword indicating a format declaration.
- <format name> - The name of the format being declared.
- <item replicator> - A numeric constant value that specifies the number of times that the following item is to be repeated.
- <format descriptor> - A specification of the type of conversion to be performed.
- <format positioner> - A specification of the position of the character string to be considered next in the conversion process.
- <character constant> - A value that is to be transmitted unchanged during the conversion process.
- <field width> - A numeric constant that specifies the length of the character string to be used during the conversion process.
- <fraction size> - A numeric constant that specifies the fractional part of a number being converted.
- <position> - A numeric constant that specifies the next position during a format scan (paragraph 6.1.1.22).

All the numeric constants that can appear in a format declaration--the repeat value, field width, fraction size, and position--must be integers.

A format list has the effect of a list of format descriptors, format positioners, and character constants, separated by commas and the virgule (/) character. The use of the item replicator permits a shortened form of such a list to be written in certain

cases. A format descriptor preceded by an item replicator has the same effect as consecutive repetitions of the format descriptor, repeated the number of times specified by the item replicator. A similar interpretation is made for a character constant preceded by an item replicator. A format item consisting of a format list enclosed in parentheses and preceded by an item replicator has the same effect as consecutive repetitions of the format list (without the parentheses) separated by commas.

An item replicator must be positive.

If a field width, item replicator, or position is written as a string of digits, it and any preceding letter that makes up part of the format item may be written without intervening spaces, even though a string that has the form of a name might be written. This juxtaposition of the letter and the number is not possible if the number appears as a compile-time constant, a string name, or as the letter D or O followed by a string of digits enclosed in parentheses.

4.24.1 Interpretation of Format Items

When data is converted in CMS-2Y(7) using a format, the conversion is either from the target machine internal form into a character string or from a character string into the internal form. A single datum being converted is converted to or from a substring of that string; the substrings are called fields.

The positions of the fields within the character string are controlled by a pointer, called the conversion cursor. The conversion cursor always points to the leftmost character of the field. The positions of the master character string are numbered from left to right, beginning with zero. At the beginning of a conversion process, the conversion cursor is positioned at the first character of the master character string.

The conversion cursor must always point to a character position within the master character string.

4.24.1.1 Format Descriptors

Each format descriptor contains a field width expression. The value of this expression is the number of characters in the field that participates in the conversion controlled by the descriptor. The value must be positive. After the conversion is performed, the conversion cursor is updated to point one position to the right of the rightmost character of the field.

4.24.1.2 Numeric Conversion (I, O, F, and E Types)

Conversion of numeric data is controlled by the I, O, F, and E format descriptors. Their effects are summarized in the following table, where w represents the field width and d represents the fraction size part of the format specification:

<u>Format Descriptor</u>	<u>Internal Form</u>	<u>Character String Form</u>
Iw[.d]	Fixed-point binary	Fixed-point decimal
Ow[.d]	Fixed-point binary	Fixed-point octal
Fw[.d]	Floating-point binary	Fixed-point decimal
Ew[.d]	Floating-point binary	Floating-point decimal

The three character string forms are similar to the corresponding source program constants. They are defined by:

<fixed-point decimal string>
 ::= [<unary numeric operator>] <decimal mantissa>

<fixed-point octal string>
 ::= [<unary numeric operator>] <octal mantissa>

<floating-point decimal string>
 ::= <fixed-point decimal string> E [<unary numeric operator> <decimal digit> [<decimal digit>]]

The character string forms differ from the source program constants in that the strings may begin with a plus or minus sign. The minus sign is necessary to indicate a negative value, but a plus sign is optional. Except for the optional sign, a fixed-point decimal string is the same as a decimal number without a decimal exponent, and a fixed-point octal string is the same as an octal number without the octal exponent. A floating-point decimal string is the same as a decimal number with a decimal exponent except for the following:

- a. The string can be preceded by a sign.
- b. The string's exponent can be indicated by the letter E alone.
- c. If the exponent is indicated by a decimal integer, it must be one or two digits preceded by a plus or minus sign (i.e., the sign in the exponent is not optional).

The fraction size part of the format specification specifies the number of digits to the right of the radix point in the fixed-point decimal string or the fixed-point octal string. For E type conversion, the characters of the exponent have no effect on the fraction size. If the fraction size is omitted, the effect is the same as if $d=0$.

On input, the character string form of the number can have leading and trailing blanks, which have no effect on the value, for I type, O type, and F type conversions. For E type conversion on input, leading blanks are permitted, having no effect on the value, but the character string terminates either after w characters or after the last nonblank character, whichever occurs first. The one exception to this rule is the conversion of the character string form which ends with E (the exponent is omitted); such forms must be right-justified in a field of w characters.

For any numeric conversion on input, the radix point can be omitted, in which case its implied position is specified by d . If the radix point appears in the string its position overrides the specified value of d ; an explicit radix point can appear at any valid position without regard for d .

On output, the character string forms are right-justified in the field of w characters. If the value of d specifies less accuracy in the character string form than in the internal form, the value is rounded during the conversion process. Negative internal values cause a leading minus sign to be generated, but leading plus signs are never generated.

For I type, O type, and F type conversions a 0 is generated to the left of the radix point if the value lies between -1 and 1 . For these conversions a radix point is not generated if $d=0$.

For E type conversions a standard form is generated: the absolute value of the fixed-point decimal string lies in the interval $(0.1,1)$, a 0 is generated to the left of the decimal point, and the decimal integer that specifies the exponent is two digits.

4.24.1.3 Character Conversion (A and L Types)

Conversion of character data is controlled by the A and L format descriptors. In the following text, to avoid confusion, external form will be used in opposition to internal form when referring to character data.

The field width part of the format specification, denoted w , specifies the number of characters in the external form of the datum. In the following discussion, n denotes the number of characters in the internal form of the datum.

For A type conversion on input, if $w < n$ or $w = n$, the w characters of the external form replace the first w characters of the internal form; any remaining internal characters are ignored. If $w > n$, the first n characters of the external form replace the internal form; the remaining $w-n$ characters of the external form are ignored.

For A type conversion on output, if $w < n$ or $w = n$, the first w characters of the internal form become the external form. If $w > n$, the external form consists of the n characters of the internal form padded with $w-n$ trailing blanks.

L type conversions are similar to A type, except that last replaces first in describing the effect on the internal form.

4.24.1.4 Character Constant Format Item

The character constant format item stands alone during the scan of a format; it does not specify the conversion of any internal datum.

Assume that the character constant specifies a string of n characters. On output, when a character constant format item is encountered during the scan of a format, the n characters specified by the constant are generated in a field of width n (as specified by the conversion cursor) and the conversion cursor is updated to point to the first position to the right of the generated field. On input, the conversion cursor is updated to point n positions to the right of its position at the time the format item is encountered, thus skipping n characters of the input record.

4.24.1.5 Format Positioners

As data is being converted under the control of a format, the characters of the record are being processed in a left-to-right manner (paragraph 6.1.1.22.2). The format positioners provide a means of modifying this processing.

A wX format item, where w represents the field width, specifies that the next w characters of the record are to be skipped. On input, the characters in these positions are ignored. On output, the effect is as if w blanks were generated.

A Tp format item, where p represents the position, specifies that the conversion cursor is to be positioned at position p . On output, if position p is to the right of the current position of the conversion cursor, the effect is as if the intervening character positions were filled with blanks. However, on output, if position p is to the left of the current position of the conversion cursor, the intervening characters are not blanked out.

/(U) CM2Y-MAN-PGR-M5049-R04C0

Examples

Given the string of characters 350274-0162E+050703 with format

FORMAT CORE I2.0, F4.1, E9.2, O4.0 \$

the quantities stored on input are:

35, 27.4, -1.62x10**5 , 0(703)

Given the internal quantities 417, -320, 0.536x10**3 and octal 627 with format

FORMAT DRAB H(1), I3.0, F6.2, E10.3, O5.0 \$

the string of characters resulting from an output is:

1417*****+0.536E+03 627

where the asterisks indicate that the value -320 cannot be converted as directed by an F6.2 format descriptor.

Given the internal quantities 27, H(XYZ), 74.51 , H(JKLM) with the format

FORMAT HAN F6.2, H(RAG), L2, F6.2, H(MODY), A2 \$

the string of characters resulting from an output is:

27.00RAGYZ 74.51MODYJK

Note that this character string is preceded by a blank (i.e., a space character).

Given the internal quantities 12, 3, H(ABCD), 4, 56, H(EF) with the format

FORMAT FMTI 2(F6.2,I2,A2) \$

the string of characters resulting from an output is:

12.00 3AB 4.0056EF

Note that this character string is preceded by two blanks.

Examples of format lists with item replicator and the equivalent forms without item replicator are as follows:

Repeated Form

3I7.2
2(F6.0,/A3)
3(H(ABC))

Equivalent Form

I7.2, I7.2, I7.2
F6.0,/A3,F6.0,/A3
H(ABC), H(ABC), H(ABC)

4.25 Stringform Declaration

Syntax

```
<stringform declaration>
  ::= [<scope modifier>] STRINGFORM <stringform name>
     <stringform list> $

<stringform name>
  ::= <name>

<stringform list>
  ::= <stringform item>@

<stringform item>
  ::= [<item replicator>]<stringform descriptor>
  ::= <stringform positioner>
  ::= [<item replicator>]<character constant>
  ::= [<item replicator>](<stringform list>)

<stringform descriptor>
  ::= D<field width>.<fraction size>[.<exponent size>]
  ::= I<field width>
  ::= B<field width>
  ::= O<field width>
  ::= X<field width>
  ::= C<field width>
  ::= E<field width>

<stringform positioner>
  ::= Z<field width>
  ::= T[<direction>]<position>

<exponent size>
  ::= <numeric constant value>

<direction>
  ::= +
  ::= -
```

Semantics

A stringform declaration specifies the types of conversions to be performed when converting data between its internal (target machine) form and character string form.

<scope modifier> - Optional. Refer to scope modifier description.

STRINGFORM - A language keyword indicating a stringform declaration.

<stringform name> - The name of the stringform being declared.

<stringform item> - A stringform conversion specifier.

All of the numeric constants that can appear in a stringform declaration -- the item replicator, field width, fraction size, exponent size, and position -- must be integers.

A stringform list has the effect of a list of stringform descriptors, stringform positioners, and character constants. The use of the item replicator permits a shortened form of such a list to be written in certain cases. A stringform descriptor preceded by an item replicator has the same effect as consecutive repetitions of the stringform descriptor, the number of times specified by the item replicator. A similar interpretation is made for a character constant preceded by an item replicator. A stringform item consisting of a stringform list enclosed in parentheses and preceded by an item replicator has the same effect as consecutive repetitions of the stringform list (without the parentheses).

An item replicator must be positive.

4.25.1 Interpretation of Stringform Items

Stringforms are used in conjunction with convertout phrases and convertin phrases. Convertout phrases cause the values of data to be converted from their internal forms to character strings and the converted value to be inserted into another data unit. Convertin phrases cause character strings that represent values to be selected from a data unit, converted to an internal format, and assigned to another data unit. A stringform statement controls both the position of the character strings in the data unit and the type of conversion performed.

The position of the character strings is controlled by a pointer, called the conversion cursor. The conversion cursor always points to the leftmost character of the string. The positions of the data unit of which the strings are part are numbered from left to right, beginning with zero. At the beginning of execution of a convertout or convertin phrase, the conversion cursor is positioned at position 0, which is the first character of the data unit.

The conversion cursor must always point to a character position within the data unit.

4.25.1.1 Stringform Descriptors

Each stringform descriptor contains a field width expression. The value of this expression is the number of characters in the character string that participates in the conversion; the value must be positive. After the conversion is performed, the conversion cursor is updated to point one position to the right of the rightmost character of the character string.

Each stringform descriptor begins with a single letter; that letter is used to name the stringform descriptor. For example, I-type conversion means conversion controlled by a stringform descriptor of the form I<field width>.

Each stringform descriptor functions in conjunction with an internal value or a data unit. If the internal value or data unit is given by a word specification, its type is context-dependent. If a numeric type is required, the word type is I 32 S. If a character type is required, the word type is H 4.

4.25.1.2 D-Type Conversion, Internal to Character

The value being converted must be a numeric type. The value of the fraction size expression must be nonnegative. The value of the exponent size expression, if present, must be positive.

If the exponent size expression is not present, the internal value is converted into a character string in the form of a decimal mantissa with a decimal point, preceded by a minus sign if the value is negative. The number of fractional digits is specified by the value of the fraction size expression.

If the exponent size expression is present, the internal value is converted into a character string in the form of a decimal mantissa with a decimal point, followed by a decimal exponent, and preceded by a minus sign if the value is negative. The number of fractional digits in the mantissa is specified by the value of the fraction size expression. The number of digits in the exponent is specified by the value of the exponent size expression.

If the exponent size expression is present and the value to be converted is not zero, one nonzero digit appears to the left of the decimal point. If the value is zero, one zero digit appears to the left of the decimal point, the number of zero digits specified by the fraction size expression appear to the right of the decimal point, and the exponent value is zero.

If the field width is greater than the number of characters in the character string, the character string is right-justified in the field. and the extra positions on the left are filled with blanks.

4.25.1.3 D-Type Conversion, Character to Internal

The data unit that receives the converted value must be of a numeric type. The value of the fraction size expression must be nonnegative. The value of the exponent size expression, if present, must be positive.

The field may consist of a string of nonblank characters, optionally preceded and/or followed by strings of blanks. The string to be converted is the string of nonblank characters. It may be any of the valid forms for decimal constants, optionally preceded by a unary plus or minus sign.

The fraction size expression has an effect only if the string to be converted has no explicit decimal point. In this case, the value of the fraction size expression specifies the number of rightmost digits of the mantissa that form the fractional part of the mantissa.

The exponent size expression has no effect in this conversion.

4.25.1.4 I-Type Conversion, Internal to Character

The value being converted must be of a numeric type.

The integer part of the internal value is converted into a character string in the form of a decimal integer, preceded by a minus sign if the value is negative.

If the field width is greater than the number of characters in the character string, the character string is right-justified in the field and the extra positions on the left are filled with blanks.

4.25.1.5 I-Type Conversion, Character to Internal

The data unit that receives the converted value must be of a numeric type.

The field may consist of a string of nonblank characters, optionally preceded and/or followed by strings of blanks. The string to be converted is the string of nonblank characters. It must be in the form of a decimal integer constant, optionally preceded by a unary plus or minus sign.

4.25.1.6 B-Type, O-Type, and X-Type Conversions, Internal to Character

The value being converted may be of any simple type.

The pattern of bits that represents the internal value is converted to a character string in binary (B-type), octal (O-type), or hexadecimal (X-type) notation. All bits of the bit pattern are represented in the string; leading and trailing zeros are not suppressed. A negative numeric value is represented in the complement notation of the value.

In hexadecimal notation, the letters A through F represent the bit patterns 1010, 1011, 1100, 1101, 1110, and 1111 respectively.

If the value being converted is of fixed-point type, a radix point appears in the string in the proper position. The number of digits to the left of the radix point is the minimum number needed in the specified notation system to express the integer bits of the internal value. The number of digits to the right of the radix point is the minimum number needed in the specified notation system to express the fractional bits of the internal value. Integer bits and fractional bits include, in this case, any implied bits.

If the value being converted is not of fixed-point type, the number of digits of the character string is the minimum number needed in the specified notation system to denote the bits of the internal value. For O-type and X-type conversions, the internal value is considered to be padded on the left with the minimum number of zero bits necessary to make the length of the internal value of a multiple of 3 or 4 respectively.

If the field width is greater than the number of characters in the character string, the character string is right-justified in the field, and the extra positions on the left are filled with blanks.

4.25.1.7 B-Type, O-Type, and X-Type Conversions, Character to Internal

The data unit that receives the converted value may be of any simple type.

The field may consist of a string of nonblank characters, optionally preceded and/or followed by strings of blanks. The string to be converted is the string of nonblank characters. It may consist of 0s and 1s (B-type), octal digits (O-type), or hexadecimal digits (X-type), and (at most) one radix point.

The string to be converted represents the bit pattern of the internal value. If no radix point is present and the character string specifies fewer bits than are required to represent a value, the character string specifies the rightmost bits of the value, and extra bits on the left are set to 0s. If no radix point is present and the character string specifies more bits than are required to represent a value, the value is assumed to be specified in the rightmost bits, and the extraneous bits on the left must be 0s.

If the radix point is present, the data unit that receives the value must be of fixed-point type. The character string then represents a value of the data unit in base 2 (B-type), 8 (O-type), or 16 (X-type). If the value is negative, it must be expressed in ones complement notation consistent with the type definition of the data unit.

4.25.1.8 C-Type Conversion, Internal to Character

The value being converted must be of a character type.

The characters of the internal value are placed into the field, beginning with the first character of the internal value and the first position of the field.

If the number of characters in the internal value is less than the width of the field, the extra positions on the right of the field are filled with blanks. If the number of characters in the internal value is greater than the width of the field, only the initial characters of the internal value are placed in the field which is filled.

4.25.1.9 C-Type Conversion, Character to Internal

The data unit that receives the converted value must be of a character type.

The entire field represents the string to be converted.

The characters of the field are placed into the data unit, beginning with the first position of the field and the first character of the data unit.

If the width of the field is less than the number of characters in the data unit, the extra characters at the end of the data unit are set to blanks.

4.25.1.10 E-Type Conversion, Internal to Character

The value being converted must be of a status type.

The internal value is converted to the corresponding character string representation, as specified in the type declaration of the value, but without the enclosing apostrophes. If leading and/or trailing blanks are specified in the type declaration as part of the value, they are significant in the character string.

The character string is left-justified in the field. If the width of the field is greater than the length of the string, the extra positions on the right are filled with blanks.

4.25.1.11 E-Type Conversion, Character to Internal

The data unit that receives the converted value must be of status type.

The field may be in one of two forms:

- a. A string of characters, enclosed in apostrophes, optionally preceded and/or followed by strings of blanks.
- b. A string of characters not enclosed in apostrophes, optionally followed by blanks.

The string to be converted in the first case is the enclosed string, not including the apostrophes. The string to be converted in the second case is the initial string, not including the optional trailing blanks.

The string to be converted must be one of the values of the data unit that is to receive the converted value, as specified in the type declaration of that data unit. If the specified value contains significant blanks, they must appear in the string.

4.25.1.12 Stringform Positioners

The stringform positioners cause the position of the conversion cursor to change, thereby affecting the position of the next field, but they do not of themselves cause any conversions to take place.

4.25.1.13 Z-Type Positioning

The value of the field width expression specifies a character string, beginning at the current position of the conversion cursor. When converting from internal forms to character strings, the specified string is filled with blanks and the conversion

cursor is updated to point to the right of the rightmost character of the string. When converting from character strings to internal forms, the conversion cursor is simply updated to point to the right of the rightmost character of the string; thus the characters of the specified string are skipped.

4.25.1.14 I-Type Positioning

If the optional direction is not present, the value of the position expression specifies a character position in the character data unit, and the conversion cursor is updated to point to that position.

If the optional direction is present, the value of the position expression is the number of character positions that the conversion cursor is to move. If the direction is "+", the conversion cursor is to move that number of positions to the right; if the direction is "-", it is to move that number of positions to the left.

4.25.1.15 Character Constant Conversion

A character constant stringform item specifies a field whose width is the number of characters in the value of the constant.

When converting from internal forms to character strings, the specified string is filled with the value of the constant and the conversion cursor is updated to point to the right of the rightmost character of the string. When converting from character strings to internal forms, the conversion cursor is simply updated to point to the right of the rightmost character of the string; thus, the characters of the specified string are skipped.

Examples

STRINGFORM A1 3X4 \$

The stringform item 3X4 indicates twelve hexadecimal characters, treated as three groups of four characters each.

STRINGFORM A2 D5,2 \$

Stringform A2 implies different formats between input (character to internal) and output (internal to character). On input, a 5-digit string is converted, with the two rightmost digits comprising a fraction if no decimal point appears in the string. On output, a character string comprised of two integral digits, a decimal point, and two fractional digits is created.

STRINGFORM A3 I1, T+3, I1 \$

A3 identifies the conversion specifying two single integer digits separated by three spaces.

Implementation Notes

The results of conversions are undefined in the following cases:

- a. When converting from internal forms to character strings, and the field is not wide enough to contain the string (except in the case of a C-type conversion).
- b. When converting from character strings to internal forms, and the form of the character string is invalid.
- c. When converting from character strings to internal forms, and the attributes of the data unit that is to receive the value are inadequate for storing the converted value. This includes a C-type conversion in which the character string is longer than the data unit that receives the value.
- d. When there is a mismatch between a stringform descriptor and the type of the corresponding value or data unit.

4.26 Inputlist Declaration

Syntax

```
<inputlist declaration>
  ::= [<scope modifier>] INPUTLIST <inputlist name>
     <inputlist> $
```

```
<inputlist name>
  ::= <name>
```

```
<inputlist>
  ::= <inputlist item>@
```

```
<inputlist item>
  ::= <input receptacle>      -
  ::= <inputlist name>
  ::= *<single-valued data unit>
```

```
<input receptacle>
  ::= <data unit>
  ::= <core address receptacle>
```

Semantics

An inputlist declaration specifies a list of receptacles to be used in a convertin phrase and a name by which the list can be referenced.

<scope modifier> - Optional. Refer to scope modifier description.

INPUTLIST - A language keyword indicating an inputlist declaration.

<inputlist name> - The name of the inputlist being declared.

<inputlist item> - The name of a data unit, or another inputlist specifying a data unit, as a receptacle for a converted character string.

The list of receptacles specified in an inputlist declaration is the list of inputlist items, if that list does not contain the name of an inputlist or a single-valued data unit preceded by an asterisk. If it contains an inputlist name, the effect is as if the list specified by the named inputlist were inserted in place of the inputlist name. If it contains a single-valued data unit preceded by an asterisk, that data unit must be of an integer type and must contain the address of an inputlist name at the time a convertin phrase referencing the inputlist is executed;

the effect is as if the addressed inputlist were inserted in place of the single-valued data unit and asterisk.

The scope of any name, other than an ntag name, appearing in the inputlist of an inputlist declaration must be at least as great as the scope of the inputlist name being declared.

Inputlists may not be used recursively, either at compile time or at execution time.

If a typed table name appears as an inputlist item, the effect is as if each item of the table appeared in the list in sequence, beginning with the first item. If an untyped table name or an untyped item-area name appears as an inputlist item, the effect is as if each word of the table or item-area appeared in the list in sequence, beginning with the first word.

CORAD(<inputlist name>) is valid as a CORAD function reference.

Examples

```
TABLE TABVM2 V MEDIUM 2 $  
  FIELD FBOOL B $  
END-TABLE TABVM2 $
```

```
INPUTLIST INL2 TABVM2(1,FBOOL) $
```

The name INL2 specifies Boolean field FBOOL in the second item of table TABVM2 as the receptacle for a convertin phrase.

```
TABLE TABI12U V (I 12 U) INDIRECT 3 $  
END-TABLE TABI12U $  
VRBL I16U I 16 U P CORAD(INL2) $
```

```
INPUTLIST INL3 CORAD(TABI12U), *I16U $
```

INL3 specifies a list of receptacles, which is the core address of the table TABI12U, and the receptacles listed in an inputlist whose address is contained in the variable I16U.

4.27 Outputlist Declaration

Syntax

<outputlist declaration>
 ::= [<scope modifier>] OUTPUTLIST <outputlist name>
 <outputlist> \$

<outputlist name>
 ::= <name>

<outputlist>
 ::= <outputlist item>@

<outputlist item>
 ::= <expression>
 ::= <table name>
 ::= <single-valued data unit>
 ::= <outputlist name>
 ::= *<single-valued data unit>

Semantics

An outputlist declaration specifies a list of values to be used in a convertout phrase and a name by which the list can be referenced.

<scope modifier> - Optional. Refer to scope modifier description.

OUTPUTLIST - Language keyword indicating an outputlist declaration.

<outputlist name> - The name of the outputlist being declared.

<outputlist item> - A value or group of values to be converted into a character string.

The list of values specified in an outputlist declaration is the list of outputlist items, if that list does not contain the name of an outputlist or a single-valued data unit preceded by an asterisk. If it contains an outputlist name, the effect is as if the list specified by the named outputlist were inserted in place of the outputlist name. If it contains a single-valued data unit preceded by an asterisk, that data unit must be of an integer type and must contain the address of an outputlist name at the time a convertout phrase referencing the outputlist is executed; the effect is as if the addressed outputlist were inserted in place of the single-valued data unit and asterisk.

The scope of any name, other than an ntag name, appearing in the outputlist of an outputlist declaration must be at least as great as the scope of the outputlist name being declared.

Outputlists may not be used recursively, either at compile time or at execution time.

If a typed table name appears as an outputlist item, the effect is as if each item of the table appeared in the list in sequence, beginning with the first item. If an untyped table name or an untyped item-area name appears as an inputlist item, the effect is as if each word of the table or item-area appeared in the list in sequence, beginning with the first word.

CORAD(<outputlist name>) is valid as a CORAD function reference.

Examples

The following data declarations are referenced in the subsequent examples.

```
NTAG EQUALS 1023 $
VRBL A30S5 A 30 S 5 $
VRBL BOOL B $
VRBL H4 H 4 $
VRBL I16U I 16 U $
```

```
OUTPUTLIST OUTL1 H(NAME IS) $
```

This outputlist declaration specifies that the character constant NAME IS is identified by the name OUTL1 for use in a convertout phrase.

```
OUTPUTLIST OUTL2 5, 6/2, 8/3, NTAG S
```

This outputlist declaration identifies OUTL2 as the name which specifies a list of the four constants, 5, 3, 8/3, and 1023.

```
OUTPUTLIST OUTL3 3*A30S5, I16U/NTAG, A30S5/I16U $
```

This outputlist declaration identifies OUTL3 as the name which specifies the list of three values resulting from the three declared numeric expressions.

```
OUTPUTLIST OUTL4 COMP(BOOL),
A30S5 GTEQ I16U, H4 NOT H(XX) $
```

This outputlist declaration specifies a list of three Boolean values identified by the name OUTL4.

4.28 Debug Enabling Declaration

Syntax

```
<debug enabling declaration>
 ::= DEBUG <debug parameter>@ $
```

```
<debug parameter>
 ::= SNAP
 ::= DISPLAY
 ::= TRACE
 ::= RANGE
 ::= PTRACE
 ::= DELETE
```

Semantics

A debug enabling declaration specifies either the classes of debug phrases or declarations that are to be enabled, or that those debug phrases or declarations that have not been enabled are to be deleted from the source programs.

- DEBUG - A language keyword indicating a debug enabling declaration.
- SNAP - A language keyword indicating that snap phrases are to be enabled.
- DISPLAY - A language keyword indicating that display phrases are to be enabled.
- TRACE - A language keyword indicating that trace and end-trace phrases are to be enabled.
- RANGE - A language keyword indicating that range declarations are to be enabled.
- PTRACE - A language keyword indicating that subprogram tracing should be enabled.
- DELETE - A language keyword indicating that all debug phrases or declarations that have not been enabled are to be deleted from the source and listing outputs.

If a class of debug phrase or declaration is not enabled, all occurrences of debug phrases or declarations of that class are ignored by the compiler. If, in addition, the debug parameter DELETE has been specified, all phrases or declarations of that class are deleted from any listings and source file output during the compilation. (The source input file is never changed.)

/(U) CM2Y-MAN-PGR-M5049-R04C0

There is no debug phrase corresponding to the parameter PTRACE. If that parameter is specified, every procedure call or user function reference will result in a display of the form

```
PROCEDURE xxxxxxxx CALLING PROCEDURE yyyyyyyy
```

on the system hardcopy device, where xxxxxxxx is the name of the calling procedure or function and yyyyyyyy is the name of the called procedure or function.

Note

The debug enabling declaration may only appear in a major header.

Examples

```
DEBUG RANGE, TRACE $
```

In this example, only range declarations and trace phrases are enabled and no calls will be generated by the compiler for other debug phrases.

```
DEBUG DELETE $
```

In this example, if there are no other debug enabling declarations in the major header, all debug phrases and declarations will be deleted from the listing and source outputs.

4.29 Range Declaration

Syntax

```
<range declaration>
  ::= <ranged name> RANGE <maximum value> [...<minimum
      value>] $
```

```
<ranged name>
  ::= <variable name>
  ::= <field name>
```

```
<maximum value>
  ::= <numeric constant expression>
```

```
<minimum value>
  ::= <numeric constant expression>
```

Semantics

A range declaration specifies a range of values for a variable or a field, and that all assignments to that variable or field are to be checked during execution to ensure that the assigned value lies in that range.

- RANGE - A language keyword indicating a range declaration.
- <ranged name> - The name of a variable or field for which a range is being specified.
- <maximum value> - The upper limit of the range of acceptable values.
- <minimum value> - Optional. The lower limit of the range of acceptable values.

The specified variable or field must be of a numeric type. Its variable or field declaration must precede the range declaration.

If the ranged name is a field, the range declaration must appear within the same type declaration, table block or array block as the field declaration. All assignments to the specified field will be checked, including assignments to like-tables, subtables, and item areas.

If no minimum value is specified, a minimum value of zero is assumed.

Each time the ranged name is assigned a value, the value is compared to the specified range. If it lies outside the specified range, an appropriate message is printed on the system hardcopy device identifying the statement in which the range violation occurs. The statement is identified in the same manner as in the execution of a trace phrase.

Examples

```
1.      VRBL COUNT I 5 U $
        COUNT RANGE 25 $
        .
        .
        .
        LOC-INDEX INDEX $
PASS1.  VARY INDEX THRU 26 $
        SET COUNT TO INDEX + 1 $
        .
        .
        .
```

Execution of the complete vary block loop index range would result in the following printout, assuming the RANGE debug facility is enabled at both compilation time and load time:

```
** RANGE: COUNT
EXCEEDS RANGE AT PASS1 + 1
```

```
2.      TABLE TRIG V MEDIUM 10 $
        FIELD ANGLE F $
        FIELD SINE F $
        .
        .
        .
        SINE RANGE 1...-1 $
        END-TABLE TRIG $
        .
        .
        .
EVAL.  VARY TIX WITHIN TRIG $
        SET TRIG(TIX,SINE) TO SIN( TRIG(TIX,ANGLE)) $
        .
        .
        .
        END EVAL $
```

If any of the results of the multiple calls to function SINF produce a value outside the range of -1 through +1, an appropriate message will be produced. In particular, if the function SINF were malfunctioning and always returning the value 3.14159, the following line would be printed 10 times in succession:

TRIG(TIX,SINE) EXCEEDS RANGE AT EVAL+1

Notes

Assignment operations that are checked by the compiler as a result of a range declaration are:

- a. Explicit assignment as a result of executing a set phrase, swap phrase, pack phrase, or shift phrase.
- b. Implicit assignment as a result of executing a procedure call or function reference in which the ranged name is a formal input parameter or actual output parameter, or a vary block or find statement in which the ranged name is the loop index.

SECTION 5. DATA REFERENCES

Data references are the basis on which a CMS-2Y(7) program is built. Data is referenced when a value is calculated. Values are calculated (the calculation might be as simple as retrieving the value of a single entity) when two values are compared in order to make a decision or when an entity is assigned a new value. These two processes of making decisions and assigning values are the fundamental processes of programming.

The basic data references are references to constants, data units (variables, fields, etc.), and functions. Expressions are built up of these basic data references.

Data references are either single-valued (e.g., variables) or multivalued (e.g., tables). Single-valued data references are either simple or structured. A simple data reference is a reference to an entity that is declared to be of a simple type or has universal type.

5.1 Data Unit

Syntax

```
<data unit>  
 ::= <single-valued data unit>  
 ::= <multivalued data unit>  
 ::= <word data unit>  
 ::= <modified data unit>
```

Semantics

Data units are the stored data of a CMS-2Y(7) program. They are capable of being changed, although a particular program may treat a particular data unit as a constant.

A single-valued data unit is a data unit that has one value associated with it; the data unit may be structured. If the data unit is simple, it can appear as an operand of an expression.

A multivalued data unit is a data unit that has one or more values associated with it. The individual values can only be referenced by the use of subscripts. The data units can be structured; if they are simple, the individual values can appear as operands of expressions, but a multivalued data unit cannot itself appear as an operand of an expression in CMS-2Y(7).

A word data unit is a target machine word. It has a single value, which can appear as an operand of an expression. The type of the value is context-dependent.

A modified data unit is a part of another data unit.

5.1.1 Single-Valued Data Unit

Syntax

```
<single-valued data unit>
  ::= <variable name>[( <field name>)]
  ::= <subscripted data unit>
  ::= <system index name>
  ::= <local index name>

<subscripted data unit>
  ::= <table name>( <subscript expression>@[ , <field name>])

<subscript expression>
  ::= <numeric expression>
  ::= <status expression>
```

Semantics

A single-valued data unit specifies a variable or part of a variable, a table item or part of a table item, a system index, or a local index.

<field name> - Optional. A specification of a part of a data unit.

<subscripted data unit> - Specification of an item of a table, or a field of an item of a table.

<subscript expression> - A numeric expression used to specify a table item.

The number of subscript expressions used to select an item of a table must be the same as the number of subscript declarations in the table declaration. The first subscript expression corresponds to the first subscript declaration, the second subscript expression corresponds to the second subscript declaration, etc.

A numeric subscript expression is evaluated according to the rules for numeric expression evaluation. If the type of the value is not integer, the value is converted to integer (paragraph 5.3.1), giving the subscript value; if the type of the value is integer, the value is the subscript value. The subscript value must be in the range specified in the corresponding subscript declaration, which must be numeric.

The value of a status subscript expression must be assignment compatible with the type of the corresponding subscript declaration, which must be of a status type.

A subscripted data unit of the form

<table name>(<subscript expression>@)

specifies an item of the named table. The data unit is simple if items of the table have a typed structure.

5.1.1.1 Restrictions on Forms

The form

<variable name>(<field name>)

is valid only if the named variable is structured and the named field is a field of the named variable.

The form

<table name>(<subscript expression>@,<field name>)

is valid only if the named table has structured items and the named field is one of its fields.

5.1.1.2 Attributes of a Single-Valued Data Unit

A data unit consisting of a variable name alone has the attributes specified in the declaration of that variable.

A data unit consisting of a system index name or a local index name has the implied attributes I 16 U.

Data units of the forms

<variable name>(<field name>)
<table name>(<subscript expression>@,<field name>)

have the attributes of the named fields.

Examples

```
TABLE TWOWAY A 1 4,4 $  
END-TABLE TWOWAY $
```

```
SET TWOWAY(3,2) TO 0(1776) $
```

The one word of the item in row 3 of column 2 of the array TWOWAY is set to the octal value 1776.

```
TABLE TABTYP V MEDIUM 10 $  
  FIELD A14S6 A 14 S 6 $  
  ITEM-AREA TEMPTYPE $  
END-TABLE TABTYP $
```

```
SET TEMPTYP(A14S6) TO 3 $
```

The integer type constant is converted to fixed-point type and assigned to field A14S6 in item-area TEMPTYPE.

```
SET TABTYP(5,A14S6) TO 3 $
```

The integer type constant is converted to fixed-point type and assigned to field A14S6 in the sixth item of table TABTYP.

5.1.2 Multivalued Data Unit

Syntax

```
<multivalued data unit>  
 ::= <table name>
```

Semantics

A multivalued data unit specifies a table.

Examples

```
TABLE TABB H 1 100 $  
    LIKE-TABLE TABC $  
END-TABLE TABB $  
  
SET TABB TO TABC $
```

Each word of table TABB is set to the corresponding word of TABC without conversion. A shorter table is transferred to the larger table to the extent of the shorter table's length. A larger table is transferred to the shorter table's length; cells beyond the end of the shorter table are left unchanged.

5.1.3 Word Data Unit

Syntax

```
<word data unit>  
 ::= <variable name>(<word specification>)  
 ::= <table name>(<subscript expression>@,<word  
 specification>)  
  
<word specification>  
 ::= <numeric expression>
```

Semantics

A word data unit specifies a target machine word.

<word specification> - A numeric expression whose value specifies a target machine word as part of a variable or table item.

A word data unit specifies a target machine word as part of a variable or item (see below); it does not specify an absolute machine address.

A word specification specifies a numbered target machine word within a variable or item. If the variable or item requires n target machine words, they are numbered from 0 through n-1 from first to last.

The word specification expression is evaluated according to the rules for numeric expression evaluation. If the type of the value is not integer, the value is converted to integer (paragraph 5.3.1), giving the word number; if the type of the value is integer, the value is the word number. The word number must be in the range [0,n-1].

The requirements on the number and value of subscript expressions are the same as for single-valued data units.

5.1.3.1 Restrictions on Forms

The form

```
<variable name>(<word specification>)
```

is valid only if the named variable is structured.

The form

<table name>(<subscript expression>@,<word specification>)

is always valid.

5.1.3.2 Word Specification

The forms

<variable name>(<word specification>)

<table name>(<subscript expression>@,<word specification>)

specify the numbered word of the variable or table item, respectively.

5.1.3.3 Resolution of Ambiguity

If a field name and a variable name in a scope are identical, certain forms of single-valued data units (using the name as a field name) are identical to certain word data units (using the name as a variable name in a word specification expression consisting of the variable only). In such cases, the name is always interpreted as a field name, yielding a single-valued data unit.

Examples

```
TABLE WORDS V 1 400 $  
END-TABLE WORDS $
```

```
SET WORDS(4,0) TO 0(1.4) $
```

This SET statement causes the value 1 to be assigned to the fifth item word of table WORDS.

The fractional bits of the octal constant are truncated because the word data unit assumes the type I 32 S.

5.1.4 Modified Data Unit

Syntax

```
<modified data unit>  
 ::= <bit modified data unit>  
 ::= <character modified data unit>
```

Semantics

A modified data unit represents part of the bit string that makes up the value of another data unit. It can be used to select a substring of bits or a substring of characters of the value of that other data unit.

Some instances of modified data units require the use of a compiler-generated procedure call. In these cases, one of the options MONITOR or NONRT must be specified. For details, see the implementation notes in the sections that describe each of the modified data units.

5.1.4.1 Bit Modified Data Unit

Syntax

<bit modified data unit>
 ::= BIT(<bit string start>[,<bit string length>]) (<parent
 unit>)

<bit string start>
 ::= <numeric expression>

<bit string length>
 ::= <numeric expression>

<parent unit>
 ::= <single-valued data unit>
 ::= <word data unit>

Semantics

A bit modified data unit specifies a substring of bits of another data unit.

- BIT - A language keyword indicating a bit modified data unit.
- <bit string start> - A numeric expression whose value specifies the leftmost bit of the bit string to be selected.
- <bit string length> - Optional. A numeric expression whose value specifies the length of the bit string to be selected.
- <parent unit> - The data unit from which the bit string is to be selected.

The string of bits that make up the parent unit are numbered from left to right, beginning with zero. This numbering references only the string of bits that make up the data unit; it does not include any extra bits allocated by the compiler to contain the value of the data unit.

If the value of the bit string start expression is not integer, it is converted to integer (paragraph 6.1.1.1) and the converted value is the starting bit number of the string to be selected; if the expression value is integer, it is the starting bit number. If the bit string length is present and its value is not integer, it is converted to integer and the converted value is the length, in number of bits, of the string to be selected; if the expression value is integer, it is the string length.

If the optional bit string length expression is omitted, the bit string length is 1.

The starting bit number must be nonnegative. The bit string length must be in the range [1,64]. Together the starting bit number and the bit string length must specify a bit string that lies entirely within the parent data unit.

The string of bits representing a bit modified data unit is interpreted as an unsigned fixed-point value with zero fractional bits and a length equal to the bit string length, if that length is constant. If the bit string length is not constant, the length of the bit modified data unit is 64.

If the bit string length expression is omitted or has the constant value 1, the bit modified data unit may be used in a context that requires a Boolean data unit. If the specified bit is on (1), the bit modified data unit has the value true; if the specified bit is off (0), the bit modified data unit has the value false.

Examples

```
VRBL A32S0 A 32 S 0 $  
VRBL I64S I 64 S $  
VRBL MESSAGE1 H 23 $
```

```
BIT(63)(I64S)  
BIT(183)(MESSAGE1)  
BIT(0,A32S0)(I64S)
```

In the first BIT reference, the rightmost (or least significant) bit in I64S is indicated. In the second BIT reference, the rightmost bit of the twenty-third character in MESSAGE1 is indicated. In the third BIT reference, any number of bits from I64S may be indicated in the range [0,64], depending on the value in A32S0. In all cases (except zero) the bit string starts with the most significant bit of I64S. The compiler generates a call to a CMS-2Y(7) monitor procedure to select the bit string.

Implementation Note

If the bit string start expression is a numeric constant value, or if the bit string length expression is a numeric constant value or is omitted and the specified bit string does not cross a target machine word boundary, the compiler will generate in-line code to select the specified bit string. In all other cases a compiler-generated procedure call is used.

5.1.4.2 Character Modified Data Unit

Syntax

<character modified data unit>
 ::= CHAR(<character string start>[,<character string
 length>]) (<parent unit>)

<character string start>
 ::= <numeric expression>

<character string length>
 ::= <numeric expression>

Semantics

A character modified data unit specifies a substring of characters of another data unit.

- CHAR - A language keyword indicating a character modified data unit.
- <character string start> - A numeric expression whose value specifies the leftmost character of the character string to be selected.
- <character string length> - Optional. A numeric expression whose value specifies the number of characters of the character string to be selected.
- <parent unit> - The data unit from which the character string is to be selected.

The bit string making up the parent unit is interpreted as a sequence of 8-bit characters, numbered from left to right, beginning with 0. The bit string considered does not include any extra bits allocated by the compiler to contain the parent unit.

If the value of the character string start expression is not integer, it is converted to integer (paragraph 6.1.1.1) and the converted value is the starting character number of the string to be selected; if the expression value is integer, it is the starting character number. If the character string length expression is present and its value is not integer, it is converted to integer and the converted value is the length, in number of characters, of the string to be selected; if the expression value is integer, it is the string length.

If the optional character string length expression is omitted, the character string length is 1.

The starting character number must be nonnegative. The character string length must be positive. Together, the starting character number and the character string length must specify a character string that lies entirely within the parent data unit.

The string of bits representing a character modified data unit is interpreted as a character value with a length, in characters, equal to the character string length.

Examples

```
VRBL MESSAGE1 H 23 P H(ABCDEF-%WHEAT!*XYZ) $
```

```
...CHAR(10,4)(MESSAGE1)...  
...CHAR(17,2)(MESSAGE1)...
```

The first line of these examples references the character string EAT! while the second line references the letter Z followed by a blank.

Implementation Note

If the character string start expression is a numeric constant value, the character string length expression is a numeric constant value or is omitted, and the specified character string either begins on a target machine word boundary or does not cross a word boundary, the compiler will generate in-line code to select the specified character string. In all other cases a compiler-generated procedure call is used.

5.2 Function Reference

Syntax

```
<function reference>  
 ::= <user function reference>  
 ::= <predefined function reference>  
 ::= <intrinsic function reference>
```

Semantics

A function reference is written as an operand of an expression. During evaluation of the expression, the function reference is replaced by a value. The manner in which the value is determined is specified in the definition of the function.

CMS-2Y(7) supports three classes of functions: (1) user functions, which are declared with function declarations and defined in function blocks, (2) predefined functions, which are known to the compiler and may be referenced without being declared, and (3) intrinsic functions, which use the function notation but are an integral part of the syntax of the language.

5.2.1 User Function Reference

Syntax

<user function reference>
 ::= <function name>([[<actual input parameter>@]])

<actual input parameter>
 ::= [<expression>]

Semantics

A user function reference specifies an invocation of a function declared in a function declaration and defined in a function block.

- <function name> - The name of the function being referenced.
- <actual input parameter> - Optional. An expression whose value will be the value of a formal input parameter at the beginning of execution of the function body.

The evaluation of a user function reference comprises three steps:

- a. The value of each actual input parameter is assigned to the corresponding formal input parameter.
- b. The body of the function is executed. The execution of the function body is terminated by the execution of a function return phrase.
- c. The value of the expression specified on the function return phrase becomes the value of the function reference.

The first actual input parameter corresponds to the first formal input parameter, the second actual input parameter corresponds to the second formal input parameter, etc. Each actual input parameter must be assignment-compatible (paragraph 6.1.1.1) with its corresponding formal parameter.

If an actual input parameter is omitted in a function reference, and the corresponding formal input parameter was not declared using a parameter declaration, the value of the formal input parameter is unchanged prior to execution of the function body. If the formal input parameter is declared using a parameter declaration, its value is undefined at the beginning of execution of the

/(U) CM2Y-MAN-PGR-M5049-R04C0

function body; in this case, omitting an actual input parameter implies that the value of the corresponding formal input parameter is irrelevant for that function reference.

The names of all formal parameters must be known in the scope containing the function reference.

Examples

```
N EQUALS 5 $
TABLE T1 V (I 10 S) N $
END-TABLE T1 $
VRBL (J,K) I 6 U $

FUNCTION IXOK (J,K) B $
  IF J GT N-1 OR K GT N-1 THEN
    RETURN (0)      "INDEX INVALID" $
  ELSE
    RETURN (1)      "INDEX OK" $
END-FUNCTION IXOK $
```

Referencing the above function, the statement

```
IF COMP IXOK (L,M) THEN
  SET L,M TO 0 $
```

causes L and M to be set to zero if L or M are out of range as input to function IXOK.

5.2.2 Intrinsic Function Reference

Syntax

```
<intrinsic function reference>  
 ::= <abs function reference>  
 ::= <corad function reference>
```

Semantics

An intrinsic function is an integral part of the CMS-2Y(7) language; the name of each intrinsic function is a reserved keyword.

The intrinsic functions cannot be declared in function declarations.

The formal input parameters of intrinsic functions have subprogram scope.

An intrinsic function reference does not affect the value of any data unit in a CMS-2Y(7) program.

5.2.2.1 Absolute Value Function Reference

Syntax

<abs function reference>
 ::= ABS(<numeric expression>)

Semantics

The value of an absolute value function reference is the absolute value of its actual input parameter.

ABS - A language keyword indicating an absolute value function reference.

<numeric expression> - The expression whose absolute value is the value of the function reference.

If the type of the actual input parameter of an absolute value function reference is integer or fixed-point, the function reference is unsigned. In all other aspects, the type of the function value is the type of its actual input parameter.

Examples

```
VRBL ATYPE A 13 S 9 $
```

The value of the function reference

```
ABS(ATYPE)
```

is the absolute value of the variable ATYPE. The type of the value is A 12 U 9.

5.2.2.2 Core Address Function Reference

Syntax

<corad function reference>
 ::= CORAD(<addressable unit>)

<addressable unit>
 ::= <variable name>[(<field name>)]
 ::= <subscripted data unit>
 ::= <multivalued data unit>
 ::= <word data unit>
 ::= <stringform name>
 ::= <inputlist name>
 ::= <outputlist name>
 ::= <switch name>
 ::= <procedure switch name>
 ::= <procedure name>
 ::= <function name>
 ::= <statement name>

Semantics

A core address function reference returns an A 16 U 0 value that is the target machine sy-address of its actual input parameter.

CORAD - A language keyword indicating a core address function reference.

<addressable unit> - Any entity that is assigned to memory and can be referenced during execution of a CMS-2Y program.

If the addressable unit is allocated to part of a target machine word, the value of the core address function reference is the same as if it were allocated that entire word.

If the addressable unit is the name of an indirect table, the value of a core address function reference is the value most recently assigned to the indirect table by a core address assignment (paragraphs 5.2, 6.1.1.1, and 6.1.1.6).

/(U) CM2Y-MAN-PGR-M5049-R04C0

Examples

```
VRBL A16S2 A 16 S 2 $
VRBL CX I 6 U $
TABLE DAT1 V MEDIUM 10 $
  FIELD FLD0 A 16 S 0 $
  FIELD FLD1 I 32 S $
END-TABLE DAT1 $
```

CORAD(A16S2)

This CORAD function reference returns the A 16 U 0 value that is the target machine address of A16S2.

CORAD(DAT1(CX,FLD0))

This example returns the address of the word in item CX of table DAT1 that contains field FLD0.

5.2.3 Predefined Function Reference

Syntax

```

<predefined function reference>
 ::= <floating-point arithmetic function reference>
 ::= <fixed-point arithmetic function reference>
 ::= <status operation function reference>
 ::= <bit string function reference>
 ::= <scalf function reference>
 ::= <conf function reference>
 ::= <tdef function reference>
 ::= <rem function reference>
 ::= <cnt function reference>
 ::= <fil function reference>
 ::= <pos function reference>
 ::= <length function reference>

```

Semantics

A predefined function is known to the compiler at the beginning of each compilation. Its name has universal scope.

The name of a predefined function in many cases is generic, representing a class of functions. The attributes of the formal input parameters of the function can vary for such generic functions, usually depending on attributes of the actual input parameters. (For example, the LN function does not have a fixed type for its formal input parameter. Therefore, LN can be considered to represent a class of functions, one for each possible formal input type.) In some cases, what appears to be an actual input parameter is a specification of a member of a class of functions.

The predefined functions cannot be declared in function declarations. Because their names have universal scope, a declaration of a function with one of their names would be considered a declaration of a user-defined function in a smaller scope.

The formal input parameters of predefined functions have subprogram scope.

A predefined function reference does not affect the value of any data unit in a CMS-2Y(7) program.

5.2.3.1 Floating-Point Arithmetic Function Reference

Syntax

<floating-point arithmetic function reference>
 ::= SIN(<angle>)
 ::= COS(<angle>)
 ::= ASIN(<numeric expression>)
 ::= ACOS(<numeric expression>)
 ::= ATAN(<numeric expression>)
 ::= ASIN2(<ordinate>, <magnitude>)
 ::= ACOS2(<abscissa>, <magnitude>)
 ::= ATAN2(<abscissa>, <ordinate>)
 ::= EXP(<numeric expression>)
 ::= ALOG(<numeric expression>)

<angle>
 ::= <numeric expression>

<ordinate>
 ::= <numeric expression>

<abscissa>
 ::= <numeric expression>

<magnitude>
 ::= <numeric expression>

Semantics

A floating-point arithmetic function reference specifies the calculation of one of several common arithmetic values in floating-point.

- | | |
|-------|--|
| SIN | - A predefined identifier indicating that the sine function is to be evaluated. |
| COS | - A predefined identifier indicating that the cosine function is to be evaluated. |
| ASIN | - A predefined identifier indicating that the inverse sine function is to be evaluated. |
| ACOS | - A predefined identifier indicating that the inverse cosine function is to be evaluated. |
| ATAN | - A predefined identifier indicating that the inverse tangent function is to be evaluated. |
| ASIN2 | - A predefined identifier indicating that the inverse sine function is to be evaluated. |

- ACOS2 - A predefined identifier indicating that the inverse cosine function is to be evaluated.
- ATAN2 - A predefined identifier indicating that the inverse tangent function is to be evaluated.
- EXP - A predefined identifier indicating that the exponential function (to the base e) is to be evaluated.
- ALOG - A predefined identifier indicating that the natural logarithm function is to be evaluated.
- <angle> - A numeric expression whose value represents an angle measured in radians.
- <abscissa> - A numeric expression whose value represents the abscissa (x-value) of a point in the coordinate plane.
- <ordinate> - A numeric expression whose value represents the ordinate (y-value) of a point in the coordinate plane.
- <magnitude> - A numeric expression whose value represents the distance from a point (x,y) in the coordinate plane to the origin (0,0).

The anonymous formal parameters and the values of the floating-point arithmetic functions are of floating-point type. If the target computer is the AN/UYK-43, that type is the industry standard single-precision type. If the target computer is the AN/UYK-7, that type is the AN/UYK-7 floating-point type without rounding.

If an actual parameter of a floating-point arithmetic function reference is not of the specified floating-point type, it will be converted to that type according to the rules for assignment.

If the target computer is the AN/UYK-7, the values of the functions are calculated by means of library routines and the NONRT option must be in effect.

The absolute value of the numeric function that is the actual parameter of the ASIN and ACOS functions must not exceed 1.

For the ASIN2, ACOS2, and ATAN2 functions, let x denote the value of the abscissa expression, y denote the value of the ordinate expression, and r denote the value of the magnitude expression. Then the value of $ASIN2(\langle\text{ordinate}\rangle, \langle\text{magnitude}\rangle)$ is $ASIN(y/r)$ and the value of $ACOS2(\langle\text{abscissa}\rangle, \langle\text{magnitude}\rangle)$ is $ACOS(x/r)$. The

value of ATAN2(<abscissa>,<ordinate>) is $\text{ATAN}(\frac{x}{y})$ if y is non-zero; if $y = 0$, then x must be non-zero and the value is $\pm \pi/2$, where the sign agrees with the sign of x . The constraints $|x| \leq r$, $|y| \leq r$, and $r > 0$ must be satisfied.

The value of the numeric expression that is the actual parameter of the EXP function must not exceed 177.447 if the target computer is the AN/UYK-43 or 22713.05 if the target computer is the AN/UYK-7. (Both figures are approximate.)

The value of the numeric expression that is the actual parameter of the ALOG function must be positive.

Examples

```
VRBL (FLT1,FLT2,FLT3) F $
```

```
SET FLT1 TO SIN(FLT2) $
```

The floating point sine of the floating point variable FLT2 is stored in the floating point variable FLT1.

```
SET FLT1 TO ACOS(FLT2) $
```

The floating point inverse cosine of the floating point variable FLT2 is stored in the floating point variable FLT1.

```
SET FLT1 TO ATAN2(FLT2,FLT3) $
```

The floating point inverse tangent of the floating point value FLT2/FLT3 is stored in the floating point variable FLT1.

```
SET FLT1 TO ALOG(FLT2) $
```

The floating point natural logarithm of the floating point value FLT2 is stored in the floating point variable FLT1.

5.2.3.2 Fixed-Point Arithmetic Function Reference

Syntax

```
<fixed-point arithmetic function reference>  
 ::= LN(<numeric expression>)  
 ::= IEXP(<numeric expression>)  
 ::= ISIN(<numeric expression>)  
 ::= ICOS(<numeric expression>)  
 ::= BAMS(<numeric expression>)  
 ::= RAD(<numeric expression>)
```

Semantics

A fixed-point arithmetic function reference specifies the calculation of one of several common arithmetic values in fixed-point.

- LN - A predefined identifier indicating that the natural logarithm function is to be evaluated.
- IEXP - A predefined identifier indicating that the exponential function (to the base e) is to be evaluated.
- ISIN - A predefined identifier indicating that the sine function is to be evaluated.
- ICOS - A predefined identifier indicating that the cosine function is to be evaluated.
- BAMS - A predefined identifier indicating that a radians-to-BAMS conversion is to be evaluated.
- RAD - A predefined identifier indicating that a BAMS-to-radians conversion is to be evaluated.

If the target computer is the AN/UYK-7, the values of the fixed-point arithmetic functions are calculated by means of library routines and the NONRT option must be in effect.

For many of the fixed-point arithmetic functions, the type of the anonymous formal parameter depends on the type of the actual parameter expression. For purposes of the following descriptions, an integer type is considered to be equivalent to a fixed-point type having the same bit length and sign specification and zero fractional bits.

The type of the anonymous formal parameter of the logarithm function depends on the type of the actual argument expression. If the actual argument expression is of a floating-point type, the type of the formal parameter is A 32 S 28. Otherwise the type of the formal parameter is A 30 U \underline{x} , where \underline{x} is in the range [0,30].

If the CMS-2Y scaling rules are in effect (paragraph 5.3.1.3), then \underline{x} is the number of fractional bits of the actual parameter. If the MSCALE scaling rules are in effect (paragraph 5.3.1.6), the value of \underline{x} is determined by the actual parameter of the function reference. Let \underline{m} denote the number of magnitude bits of the actual parameter value and let \underline{f} denote its number of fractional bits. If $\underline{m} \leq 30$, then $\underline{x} = \underline{f}$. If $\underline{m} > 30$, then $\underline{x} = \underline{f} - (\underline{m} - 30)$ and $\underline{m} = 30$; that is, the value of the actual parameter is shifted right until it has 30 magnitude bits and the number of fractional bits is adjusted appropriately. In either case, if \underline{x} is negative, the result of the function reference is undefined. The value of the function reference is of type A 32 S 30.

The type of the anonymous formal parameter of the exponential function is A 32 S 31; that is, the value of the actual parameter expression (y) must lie in the range $-1 < y < 1$. The type of the value of the function reference is A 31 U 29.

The type of the anonymous formal parameters of the sine and cosine function is A 32 U 32; that is, the value of the actual parameter expression must lie in the range (0,1). This value represents an angle measured in the BAMS system, in which the value 0.25 represents a right angle. The type of the value of the function reference is A 32 S 30.

The type of the anonymous formal parameter of the radians-to-BAMS function depends on the type of the actual argument expression. If the actual argument expression is of a floating-point type, the type of the formal parameter is A 32 S 28. Otherwise the type of the formal parameter is A 32 U \underline{x} , where \underline{x} is determined by the actual parameter of the function reference. If the CMS-2Y scaling rules are in effect (paragraph 5.3.1.3), then \underline{x} is the number of fractional bits of the actual parameter. If the MSCALE scaling rules are in effect (paragraph 5.3.1.6), the value of \underline{x} is determined as follows: let \underline{m} denote the number of magnitude bits of the actual parameter value and let \underline{f} denote its number of fractional bits. If $\underline{m} \leq 32$, then $\underline{x} = \underline{f}$. If $\underline{m} > 32$, then $\underline{x} = \underline{f} - (\underline{m} - 32)$ and $\underline{m} = 32$; that is, the value of the actual parameter is shifted right until it has 32 magnitude bits and the number of fractional bits is adjusted appropriately. The value of the actual parameter expression must lie in the range (0, 2). This value represents an angle measured in the radians system. The type of the value of the function reference is A 32 U 32.

The type of the anonymous formal parameter of the BAMS-to-radians function is A 32 U 32. The value of the actual parameter expression must lie in the range (0,1). This value represents an angle measured in the BAMS system. The type of the value of the function reference is A 31 U 28.

Examples

```
VRBL A32S10 A 32 S 10 $
VRBL A32S15 A 32 S 15 $
```

```
SET A32S15 TO LN(A32S10) $
```

This example computes the natural logarithm of the value in A32S10 and stores the result in the variable A32S15.

```
SET A32S15 TO IEXP(A32S10) $
```

This example computes the exponential function of the value in A32S10 and stores the result in the variable A32S15.

```
SET A32S15 TO ISIN(A32S10) $
```

This example computes the sine of the value in A32S10 and stores the result in the variable A32S15.

```
SET A32S15 TO BAMS(A32S10) $
```

This example assumes that the value in A32S10 is in radians, converts it to BAMS, and stores the result in the variable A32S15.

```
SET A32S15 TO RAD(A32S10) $
```

This example assumes that the value in A32S10 is in BAMS and converts it to radians. The result is stored in the variable A32S15.

In all of the above examples, the result is aligned to 15 bits of scaling before being stored in the variable A32S15.

| 5.2.3.3 Status Operation Function Reference

| Syntax

| <status operation function reference>
| ::= <successor function reference>
| ::= <predecessor function reference>
| ::= <initial value function reference>
| ::= <final value function reference>

| Status operation function references provide operations on status.
| type values and status types.

| Examples

| The examples of the following paragraphs will make use of the
| following declarations:

| TYPE DAY S 'SUN', 'MON', 'TUE', 'WED', 'THU',
| 'FRI', 'SAT' \$
| VRBL TODAY DAY \$

5.2.3.3.1 Successor Function Reference

Syntax

<successor function reference>
 ::= SUCC(<status expression>)

Semantics

The value of a successor function reference is the value that follows the value of its argument, in the order in which the values are listed in the declaration of the type of the argument.

SUCC - A predefined identifier indicating a successor function reference.

<status expression> - The expression whose successor value is the value of the function reference.

The value of a successor function reference is undefined if the value of the argument expression is the last value of its type.

Example

If TODAY has the value 'MON' (paragraph 5.2.3.3), then SUCC(TODAY) has the value 'TUE'. If TODAY has the value 'SAT', then SUCC(TODAY) is undefined.

5.2.3.3.2 Predecessor Function Reference

Syntax

<predecessor function reference>
 ::= PRED(<status expression>)

Semantics

The value of a predecessor function reference is the value that precedes the value of its argument, in the order in which the values are listed in the declaration of the type of the argument.

PRED - A predefined identifier indicating a predecessor function reference.

<status expression> - The expression whose predecessor value is the value of the function reference.

The value of a predecessor function reference is undefined if the value of the argument expression is the first value of its type.

Example

If TODAY has the value 'MON' (paragraph 5.2.3.3), then PRED(TODAY) has the value 'SUN'. If TODAY has the value 'SUN', then PRED(TODAY) is undefined.

5.2.3.3.3 Initial Value Function Reference

Syntax

<initial value function reference>
 ::= FIRST(<status type>)

Semantics

The value of an initial value function reference is the first value of its argument type.

FIRST - A predefined identifier indicating initial value function reference.

<status type> - The status type whose first value is the value of the function reference.

Example

The value of FIRST(DAY) (paragraph 5.2.3.3) is 'SUN'.

| 5.2.3.3.4 Final Value Function Reference

| Syntax

| <final value function reference>
| ::= LAST(<status type>)

| Semantics

| The value of a final value function reference is the last value of its argument type.

| LAST - A predefined identifier indicating a final value function reference.

| <status type> - The status type whose last value is the value of the function reference.

| Example

| The value of LAST(DAY) (paragraph 5.2.3.3) is 'SAT'.

5.2.3.4 Bit String Function References

Syntax

```

<bit string function reference>
 ::= <bit string sum function reference>
 ::= <bit string product function reference>
 ::= <bit string difference function reference>
 ::= <bit string complement function reference>

<bit string sum function reference>
 ::= ORF(<bit string operand 1>,<bit string operand 2>)

<bit string product function reference>
 ::= ANDF(<bit string operand 1>,<bit string operand 2>)

<bit string difference function reference>
 ::= XORF(<bit string operand 1>,<bit string operand 2>)

<bit string complement function reference>
 ::= COMPF(<bit string operand 1>)

<bit string operand 1>
 ::= <simple expression>

<bit string operand 2>
 ::= <simple expression>

```

Semantics

A bit string function reference specifies a manipulation of the bits that represent one or two CMS-2Y values.

- ORF - A predefined identifier specifying the bit-by-bit logical sum of two bit strings.
 - ANDF - A predefined identifier specifying the bit-by-bit logical product of two bit strings.
 - XORF - A predefined identifier specifying the bit-by-bit logical symmetric difference of two bit strings.
 - COMPF - A predefined identifier specifying the bit-by-bit logical complement of a bit string.
- <bit operand 1> - Expressions whose values are the bit strings to be manipulated.
 <bit operand 2>

The anonymous formal parameters of the bit string functions are of universal type; the bit strings that represent the values of the actual parameters are not converted as part of a bit string function reference.

The type of a bit string function reference is universal. Each function reference has a length, in bits, which depends on the function and the lengths of its actual arguments.

If the actual arguments of a bit string sum function reference, a bit string product function reference, or a bit string difference function reference have different lengths, the shorter is effectively padded on the left with 0-bits to the length of the longer.

The value of a bit string sum function reference is 0 at those bit positions where both actual arguments have 0-bits; the value has 1-bits at all other bit positions. The length of the function reference is the length of the longer of its actual arguments.

The value of a bit string product function reference is 1 at those bit positions where both actual arguments have 1-bits; the value has 0-bits at all other positions. The length of the function reference is the length of the shorter of its actual arguments.

The value of a bit string difference function reference is 1 at those bit positions where one of the actual arguments has a 1-bit and the other has a 0-bit; the value has 0-bits at all other positions (where the bits of the actual arguments are the same). The length of the function reference is the length of the longer of its actual arguments.

The value of a bit string complement function reference is 1 at those bit positions where its actual argument has a 0-bit and 0 at those bit positions where its actual argument has a 1-bit. The length of the function is the length of its actual argument.

Examples

```
VRBL V1 I 12 U P O(0770) $  
VRBL V2 I 12 U P O(77)  $
```

```
SET VB1 TO ORF(V1,V2) $
```

Results in VB1 being set to o(0777).

```
SET VB1 TO XORF(V1,V2) $
```

Results in VB1 being set to O(0707).

SET VB1 TO ANDF(V1,V2) \$

Results in VB1 being set to 0(0070).

SET VB1 TO COMPF(V1) \$

Results in VB1 being set to 0(7007).

5.2.3.5 Scaling Specification Function Reference

Syntax

<scalf function reference>
 ::= SCALF(<scale factor>,<controlled expression>)

<scale factor>
 ::= <numeric constant expression>

<controlled expression>
 ::= <numeric expression>

Semantics

A scaling specification function reference specifies a fixed-point numeric expression and a scale factor to be used in evaluating the expression and aligning its value.

SCALF - A predefined identifier indicating a scaling specification function reference.

<scale factor> - A numeric constant expression specifying the scaling to be used during evaluation of the controlled expression and the final alignment of its value.

<controlled expression> - A numeric expression whose evaluation is controlled by the scale factor.

The controlled expression must be a numeric expression whose operands are fixed-point values or constants. Exponentiations are permitted only if the exponent is a constant expression whose value is integer; the expression must have at least two operands. At least one operand must be nonconstant.

All operations in the evaluation of the controlled expression that involve at least one nonconstant operand will be performed in fixed-point arithmetic, subject to the scaling rules specified below. Operations involving only constant operands will be performed according to the constant arithmetic rules (paragraph 5.3.1).

Only operations on the primaries of the controlled expression are affected by the SCALF function reference. The evaluations of subscript expressions and actual argument expressions are unaffected.

The value of the scale factor expression must be integer in the range [-127,127]. It must be possible to execute the operations using the fixed-point (integer) instruction set of the target machine.

For addition and subtraction, both operands will be aligned to the scale factor before the operation.

For multiplication (including multiplication executed during the evaluation of an exponentiation), no prealignment will be performed. The product will be aligned to the scale factor.

For division, the numerator will be aligned to a scaling of S_f+S_d prior to the division operation, where S_f is the value of the scale factor and S_d is the scaling of the divisor. As a result of this prealignment, the scaling of the quotient will be S_f .

Examples

```
VRBL A11S4 A 11 S 4 $  
VRBL A5S2 A 5 S 2 $  
VRBL A11S3 A 11 S 3 $
```

```
SCALF(3,A11S4 + A5S2)
```

The contents of variable A11S4 are loaded into a register and right-shifted one bit (scaled 3). The contents of variable A5S2 are loaded into a register and left-shifted one bit (scaled 3). The two values are then added.

```
SCALF(2,A11S3/A11S4)
```

The contents of variable A11S3 are loaded into a register and left-shifted three bits (scaled 6) and then divided by the contents of variable A11S4. The result is scaled 2.

5.2.3.6 Conversion Function Reference

Syntax

<conf function reference>
 ::= CONF(<target conversion type>, <conversion source>)

<target conversion type>
 ::= <numeric type specification>
 ::= <simple type>
 ::= <typed structure name>

<conversion source>
 ::= <numeric expression>

Semantics

A conversion function reference specifies the conversion of a numeric value to another numeric type.

- CONF - A predefined identifier indicating a conversion function reference.
- <target conversion type> - A specification of the numeric type to which the value is to be converted.
- <conversion source> - A numeric expression whose value is to be converted to the target type.

The value of a conversion function reference is the value of the source expression, converted to the target type (paragraph 5.3.1).

If the target conversion type is in the form of a name, it must be the name of a simple numeric type or a typed structure whose underlying simple type is numeric. In the latter case, the type to which the value is converted is the underlying simple type.

The value of a conversion function reference is undefined if the conversion of the source expression value to the target type is invalid.

Examples

```
VRBL A11S3 A 11 S 3 $  
VRBL A8S5 A 8 S 5 $
```

```
CONF(I 8 S, A11S3)
```

This function reference has the effect of discarding the fractional bits of A11S3.

```
CONF(A 10 S 2, A11S3)
```

This function reference effectively discards one fractional bit of A11S3.

```
CONF(A 9 S 7, A8S5)
```

This function reference has the effect of adding two fractional bits to A8S5 and discarding the most significant bit of A8S5. This function reference can be undefined for some values of A8S5.

Implementation Note

Use of a conversion function reference is an implicit declaration that the target type is capable of expressing the value of the source expression, subject to the rules of numeric conversion (paragraph 5.3.1). Therefore, for example, if the target type requires fewer significant bits than the type of the source expression, the compiler does not generate code to mask out the unneeded significant bits; nor to verify that the new sign bit, if the target type is signed, correctly represents the sign of the source value.

5.2.3.7 Temporary Definition Function Reference

Syntax

```
<tdef function reference>  
 ::= TDEF(<target redefinition type>,<redefinition source>)  
  
<target redefinition type>  
 ::= <numeric type specification>  
 ::= <simple type>  
 ::= <typed structure name>  
  
<redefinition source>  
 ::= <simple expression>
```

Semantics

A temporary definition function reference causes a bit string representing a value of one type to be treated as though it were representing a value of another type.

- | | |
|----------------------------|---|
| TDEF | - A predefined identifier indicating a temporary definition function reference. |
| <target redefinition type> | - A specification of the type to which the bit string is to be converted. Type must be either integer type, fixed-point type, or floating-point type. |
| <redefinition source> | - A simple expression whose value is the bit string to be converted. |

The effect of a temporary definition function reference is as if the value of the source expression were converted to universal type, and the resulting bit string were then treated as a value of the target type.

If the target redefinition type is in the form of a name, it must be the name of a simple numeric type or a typed structure whose underlying simple type is numeric. In the latter case, the type to which the bit string is converted is the underlying simple type.

The number of bits required by the target type must not be greater than the number of bits of the source expression.

Examples

VRBL I14S I 14 S \$

TDEF(A 12 S 3, I14S)

The bit pattern that is the value of I14S is used as though it were a value of type A 12 S 3.

5.2.3.8 Remaindering Function Reference

Syntax

<rem function reference>
 ::= REM(<remaindering expression>)

<remaindering expression>
 ::= <numeric expression>

Semantics

A remaindering function reference returns the remainder of a fixed-point division operation.

REM - A predefined identifier indicating a remaindering function reference.

<remaindering expression> - A numeric expression containing the fixed-point division operation whose remainder is the value of the function reference.

The remaindering expression must contain only fixed-point operands, and exactly one fixed-point division operation. The division operation must be indicated by the division operator (/).

The operands of the division operation of the remaindering expression cannot both be constant expressions.

The value of a remaindering function reference is the remainder of the explicit division operation in the remaindering expression.

All constant arithmetic in the remaindering expression is performed according to the constant arithmetic rules (paragraph 5.3.1).

The sign of the remainder is the sign of the numerator. The number of magnitude bits in the remainder is the number of magnitude bits in the denominator, after it has been aligned prior to the division operation (paragraph 5.3.1). The number of fractional bits in the remainder is the number of fractional bits in the numerator, after it has been aligned prior to the division operation (paragraph 5.3.1).

Examples

```
VRBL A12S4 A 12 S 4 $  
VRBL A4S2 A 4 S 2 $
```

The value of the function reference

```
REM(A12S4/A4S2)
```

is the remainder obtained by dividing the variable A12S4 by variable A4S2. The attributes of this value depend on the fixed-point scaling rules being used (paragraph 5.3.1.3). If the MSCALE rules are being used, the value has 14 magnitude bits and seven fractional bits.

Note

A division operation on two fixed-point operands may be performed in floating-point due to scaling rules (paragraph 5.3.1). In this case the remaindering function reference will cause an error message, since the function applies to fixed-point division only.

5.2.3.9 Bit Count Function Reference

Syntax

<cnt function reference>
 ::= CNT(<simple expression>)

Semantics

The value of a bit count function reference is the number of "on" bits (1 bits) in its actual input parameter.

CNT - A predefined identifier indicating a bit count function reference.

<simple expression> - An expression whose "on" bits are to be counted.

The simple expression must not be longer than 32 bits. The type of the value of a bit count function reference is I 32 S.

Examples

```
VRBL V1V-I 23 U P O(16) $
```

```
IF CNT(V1V) EQ 3 THEN RETURN $
```

This bit count function reference results in conversion of the variable V1V to a 32-bit universal type operand before the bits are counted. If V1V contains three "on" bits, the return statement will be executed.

5.2.3.10 Subfile Number Function Reference

Syntax

<fil function reference>
 ::= FIL(<file name>)

Semantics

The value of a subfile number function reference is the number of the current subfile in the named file.

FIL - A predefined identifier indicating a subfile number function reference.

<file name> - The name of the file whose subfile number is the value of the function reference.

The named file must be open at the time of the subfile number function reference.

The type of the value of a subfile number function reference is A 32 S 0.

5.2.3.11 Subfile Position Function Reference

Syntax

<pos function reference>
 ::= POS(<file name>)

Semantics

The value of a subfile position function reference is the number of the current record in the current subfile of the named file.

POS - A predefined identifier indicating a subfile position function reference.

<file name> - The name of the file whose record number is the value of the function reference.

The named file must be open at the time of the subfile position function reference.

The type of the value of a subfile position function reference is A 32 S 0.

5.2.3.12 Record Length Function Reference

Syntax

<length function reference>
 ::= LENGTH(<file name>)

Semantics

The value of a record length function reference is the length of the current record in the named file.

LENGTH - A predefined identifier indicating a record length function reference.

<file name> - The name of the file whose current record length is the value of the function reference.

The named file must be open at the time of the record length function reference.

If the type of the named file is B, the value of the function reference is the length of the current record in target machine words. If the type of the named file is H, the value of the function reference is the length of the current record in characters.

The type of the value of a record length function reference is A 32 S 0.

Examples

LENGTH(INSTRUCT)

LENGTH specifies the length of the current record for the file named INSTRUCT.

5.3 Expressions

Syntax

<expression>
 ::= <simple expression>
 ::= <structured expression>

<simple expression>
 ::= <numeric expression>
 ::= <boolean expression>
 ::= <character expression>
 ::= <status expression>
 ::= <bit string expression>

Semantics

An expression is the means by which values are generated in a CMS-2Y(7) program. Expressions include constants, data units, and function references as special cases.

The type of the value of a simple expression is simple. The type of the value of a structured expression is structured.

5.3.1 Numeric Expression

Syntax

```

<numeric expression>
  ::= <numeric expression> + <numeric term>
  ::= <numeric expression> - <numeric term>
  ::= <numeric term>

<numeric term>
  ::= <numeric term> * <numeric factor>
  ::= <numeric term> / <numeric factor>
  ::= <numeric factor>

<numeric factor>
  ::= <numeric primary> ** <numeric factor>
  ::= [<unary numeric operator>] <numeric primary>

<numeric primary>
  ::= (<numeric expression>) [<scaling specifier>]
  ::= <single-valued data unit> [<scaling specifier>]
  ::= <word data unit> [<scaling specifier>]
  ::= <function reference> [<scaling specifier>]
  ::= <bit modified data unit>
  ::= <ntag name>
  ::= <rtag name>
  ::= <ltag name>
  ::= <numeric constant>

<scaling specifier>
  ::= ..<numeric constant value>

```

Semantics

A numeric expression specifies the calculation of a numeric value using the numeric operations of unary plus, unary minus (negation), addition, subtraction, multiplication, division, and exponentiation. A numeric expression can also consist of a single numeric value, with no operation performed.

The type of any single-valued data unit or the value of any function reference used as a primary in a numeric expression must be numeric.

A word data unit used as a primary in a numeric expression is assumed to be of type I 32 S.

5.3.1.1 Expression Evaluation

The semantics of a numeric expression are implied by the productions that specify its syntax. There is a hierarchy of operators, which is given in the following table:

<u>Operator</u>	<u>Operation</u>	<u>Hierarchy</u>
+	Unary Plus	1
-	Unary Minus (Negation)	1
**	Exponentiation	1
*	Multiplication	2
/	Division	2
+	Addition	3
-	Subtraction	3

The operators "+" and "-" are unary operators whenever they are not immediately preceded by an operand. In all other cases they represent the binary operations of addition and subtraction respectively.

Numeric expressions are evaluated according to the operator hierarchy. The operation indicated by an operator may be performed provided that the immediately preceding operator, if present, and the immediately succeeding operator, if present, have a higher hierarchy number. When an operation is performed, the operator and its operand or operands are replaced in the expression by the value of the operation, and the resulting expression is then evaluated.

A parenthesized expression appearing as an operand of a numeric operator must be evaluated before the operation indicated by the operator can be performed.

If two consecutive numeric operators both have hierarchy number 2 or both have hierarchy number 3, the operation indicated by the left operator is performed first. If two consecutive numeric operators both have hierarchy number 1, the operation indicated by the right operator is performed first.

5.3.1.2 Numeric Conversions

If one operand of an addition, subtraction, multiplication, division, or comparison is of a fixed-point type and the other is of a floating-point type, the value of the fixed-point operand will be converted to the floating-point type before the operation is performed. If the operands of one of these operations are of different floating-point types, the value of one of the operands will be converted to the other type before the operation is performed, as specified in Figure 5-01. If neither operand is floating-point and the scaling rules require that they be

converted to floating-point, they are converted to the form having floating-point attribute T. If the operands of an addition or subtraction are of different fixed-point types, the values of one or both will be converted (by shifting) to a different fixed-point type before the operation is performed.

<u>Operand Attributes</u>	<u>Floating-Point Conversion</u>
T & R	T to R
T & S	S to T
T & D	D to T
R & S	S to R
R & D	D to R
S & D	S to D

See page 4-8 for a discussion of T, R, D AND S attributes.

Figure 5-01. Floating-point Arithmetic Conversions

The non-floating-point base operand of an exponentiation is converted to floating-point unless both operands are integers or the exponent is a positive integer less than or equal to 4. The exponent operand is converted to floating-point unless it is an integer. The conversion will be from fixed-point to AN/UYK-7 floating-point unless the other operand has the floating-point attribute S or D, in which case the conversion is to that type.

In a conversion from one numeric type to another, the conversion algorithm is such that each value of the source type is converted to its most accurate approximation in the target type. The method of approximation is by truncation on the right. If the attributes of the target type do not permit representation of the most significant bit of the source value, the effect of the conversion is undefined.

5.3.1.3 Fixed-Point Scaling Algorithm

The CMS-2Y fixed-point scaling algorithm is context-sensitive; that is, identical expressions appearing in different contexts might be evaluated in different manners. The number of fractional bits in the result of a numeric operation with fixed-point operands is determined by the operator, by the attributes of the operands, and by the value of a scaling controller, as defined in the scaling rules below. The alignment used to arrive at the resultant scaling is also defined.

5.3.1.3.1 Symbols Used in Scaling Algorithm

If X and Y are two numeric values, the evaluation of $R = X \text{ op } Y$ (where op means arithmetic operation) is performed in accordance with the scaling rules below, using the following symbols:

- A1 = number of fraction bits in X
- I1 = number of integer bits in X
- A2 = number of fraction bits in Y
- I2 = number of integer bits in Y
- Ar = number of fraction bits in R
- Ir = number of integer bits in R (not including the sign bit)
- N1 = number of bits in X (sign included)
- N2 = number of bits in Y (sign included)
- Nr = number of bits in R (sign included)
- Z = value of the scaling controller
- M1 = $I1 + A1$
- M2 = $I2 + A2$
- C = If Y is a constant, the number of significant bits of Y minus 1 (i.e., not including leading zeros); otherwise, 0.

If either operand is a single-valued data unit followed by a scaling specifier, a word data unit followed by a scaling specifier, or a function reference followed by a scaling specifier, the operand's number of fractional bits is the value of the scaling specifier, which must be integer. The value of the data unit or function reference is not converted to the scaling specified; the bit string representing the value is interpreted as representing a value of the specified number of bits.

5.3.1.3.2 The Value of the Scaling Controller

If a numeric expression contains only fixed-point operands, is enclosed in parentheses, and is followed by the optional scaling specifier, the numeric constant value in the scaling specifier is the value of Z. If a numeric expression contains only fixed-point operands, is not enclosed in parentheses and is not followed by a scaling specifier, and is the source in an assignment statement whose receptacle is of a fixed-point type, the number of fractional bits of the receptacle is the value of Z. For all other cases of numeric expressions containing only fixed-point operands not enclosed in parentheses and followed by a scaling specifier (in a numeric relational expression, a case selector (paragraph 6.1.2.3), or an assignment phrase with a floating-point receptacle), the value of Z is determined as follows:

If A1 and A2 are both nonzero, then $Z = \min(A1, A2)$.

If A1 and A2 are both zero, then $Z = 0$.

Otherwise, Z is the nonzero one of A1 and A2.

5.3.1.3.3 Results of Binary Operations

a. Addition and Subtraction

1. If $A1 = A2$, then $A_r = A1 = A2$.

If A1 and A2 are not equal and $\min(A1, A2) > Z$, then $A_r = \min(A1, A2)$. Otherwise, $A_r = Z$.

2. X and Y are prealigned to A_r fractional bits.

b. Multiplication

1. If $N1 > 32$ or $N2 > 32$, both operands are converted to floating-point and the operation is performed in floating-point.
2. No prealignment of operands occurs in fixed-point expressions that form part of a relational expression.
3. If $A1 > Z$ and X is the result of a previous multiplication, X is prealigned to Z fractional bits and $A1' = Z$. Otherwise, X is not prealigned and $A1' = A1$.

4. If $A2 > Z$ and Y is the result of a previous multiplication, Y is prealigned to Z fractional bits and $A2' = Z$. Otherwise, Y is not prealigned and $A2' = A2$.

5. $A_r = A1' + A2'$.

c. Division

1. If $N2 > 32$, the operands are converted to floating-point and the operation is performed in floating-point.

2. If $A2 > Z$, Y is prealigned to Z and $A2' = Z$. Otherwise, Y is not prealigned and $A2' = A2$.

3. X is prealigned to $A2' + Z$ and $A1' = A2' + Z$.

4. $A_r = Z$.

d. Exponentiation

1. $A_r = A1$.

Implementation Note

If the exponent is a positive integer less than or equal to 4, the coding will be in-line; otherwise, a run-time routine is called.

5.3.1.3.4 Floating-Point Arithmetic

A binary numeric operation is executed in floating-point whenever either of its operands is of floating-point type. Certain multiplications and divisions involving only fixed-point operands are executed in floating-point (see above). If an operand of a floating-point operation is of fixed-point type, it is converted to floating-point before the operation.

If either operand of a floating-point operation has the rounding attribute, the operation is executed using the target machine floating-point rounding instructions, as specified in Figure 5-01. A fixed-point operand converted to floating-point does not have the rounding attribute.

The result of a floating-point operation is of floating-point type. If either operand has the rounding attribute, the result has the rounding attribute.

5.3.1.4 Sign of Fixed-Point Operations

The value of a fixed-point addition, multiplication, or division operation is unsigned if both operands are unsigned. The value of a fixed-point unary plus operation is unsigned if its operand is unsigned. Any other fixed-point value that arises from a numeric operation is signed.

5.3.1.5 Constant Arithmetic

If both operands of an addition, subtraction, multiplication, or division are constants, or are the values of an expression consisting of these operators and constant operands, the operation is performed during compilation to an accuracy of 63 bits. The scaling (number of fractional bits) of one of these operations depends on the scaling of its operands. For an addition, subtraction, or division, the scaling of the result is the larger of the scalings of the two operands. For a multiplication, the scaling of the result is the sum of the scalings of the two operands.

The compiler does not rearrange any numeric expressions in an effort to discover opportunities for constant arithmetic.

Examples

The following variables are referenced in the scaling rule examples below:

```
VRBL A5S2 A 5 S 2 $
VRBL A6S2 A 6 S 2 $
VRBL A6S3 A 6 S 3 $
VRBL A8U7 A 8 U 7 $
VRBL A4U0 A 4 U 0 $
VRBL A9U5 A 9 U 5 $
VRBL A4U1 A 4 U 1 $
VRBL A48U16 A 48 U 16 $
VRBL (FLTA,FLTB) F $
VRBL FLTC F (R) $
VRBL A5U4 A 5 U 4 $
```

a. SET A4U1 TO A6S2 - A8U7. \$

The fractional bits of the first and second operand (2 and 7) are unequal, $\min(A1,A2) = \min(2,7) = 2$, and $Z = 1$ (the number of fractional bits of the receptacle, A4U1). Since $\min(A1,A2) > Z$, then $A_r = \min(A1,A2) = 2$. Therefore, A6S2 and A8U7 are prealigned to two fractional bits before the subtraction is performed.

- b. SET A9U5 TO A6S2 - A8U7 \$

This is similar to the above example except that $Z = 5$, and thus $\min(A1, A2) < Z$. Therefore $A_r = Z = 5$, and A6S2 and A8U7 are prealigned to five fractional bits before the subtraction is performed.

- c. SET A8U7 TO A48U16 * A4U0 \$

Here $N1 = 48$, $N2 = 4$, $A1 = 16$, and $A2 = 0$. Because $N1 > 32$, both operands are converted to floating-point, and the multiplication is performed in floating-point.

- d. IF A6S2 * A8U7 GT 127 THEN ... \$

This fixed-point multiplication is a part of a relational expression. As a result, there is no prealignment of operands.

- e. SET A5U4 TO A5S2 * A6S3 * A8U7 \$

Because multiplications are done from left to right, the above statement is equivalent to

SET A5U4 TO (A5S2 * A6S3) * A8U7 \$

For the parenthesized multiplication, $A1 = 2$, $A2 = 3$, and $Z = 4$. Since $A1 \leq Z$, X is not prealigned and $A1' = A1 = 2$. Since $A2 \leq Z$, Y is not prealigned and $A2' = A2 = 3$. $A_r = A1' + A2' = 5$.

For the second multiplication, $A1 = 5$, $A2 = 7$, and $Z = 4$. Since $A1 > Z$ and X is the result of a previous multiplication, X (the value of the first multiplication) is prealigned to four fractional bits, and $A1' = 4$. Although $A2 > Z$, A8U7 is not the result of a previous multiplication. Therefore A8U7 is not prealigned, $A2' = A2 = 7$, and $A_r = 11$.

- f. SET A8U7 TO A6S2 * A6S3 \$

$A1 = 2$, $A2 = 3$, and $Z = 7$. Since $A1 < Z$, the value of A6S2 is not prealigned, and $A1' = A1 = 2$. Since $A2 < Z$, the value of A6S3 is not prealigned, and $A2' = A2 = 3$. Therefore, $A_r = 5$.

- g. SET A6S3 TO A48U16 / A9U5 \$

Here $N1 = 48$, $N2 = 9$, $A1 = 16$, and $A2 = 5$. Because $N1 > 32$, both operands are converted to floating-point, and the division is performed in floating-point.

h. SET A6S2 TO A6S2 / A5S2 \$

$A1 = 2$, $A2 = 2$, and $Z = 2$. Since $A2 \leq Z$, the Y operand (A5S2) is not prealigned and $A2' = \bar{A2} = 2$. The X operand (A6S2) is prealigned to four fractional bits ($A2' + Z = 4$) and $A1' = 4$. $Ar = Z = 2$.

i. SET A5S2 TO A6S3 / A8U7 \$

Here $A1 = 3$, $A2 = 7$, and $Z = 2$. Since $A2 > Z$, the Y operand (A8U7) is prealigned to two fractional bits, and $A2' = Z = 2$. The X operand (A6S3) is prealigned to five fractional bits ($A2' + Z = 5$) and $A1' = 5$. $Ar = Z = 2$.

j. SET A9U5 TO A5S2 ** 2 \$

The scaling of the result of this exponentiation is $Ar = A1 = 2$.

k. SET A6S3 TO FLTA * A5S2 \$

Because one of its operands is floating-point, the other operand (A5S2) is converted to floating-point and the multiplication is executed in floating-point. The value is converted to fixed-point with three fractional bits prior to being assigned to A6S3.

l. SET FLTA TO FLTB / FLTC \$

Since operand FLTC has the rounding attribute, the division is executed using the target machine hardware floating-point rounding instructions.

m. SET A9U5 TO A5S2..0 + A6S3..0 \$

Because of the two scaling specifiers, $A1 = 0$ and $A2 = 0$. Since $A1 = A2$, $Ar = A1 = A2 = 0$. Thus neither operand is prealigned prior to the addition, which means that the ones bit of A6S3 is aligned with the twos bit of A5S2. After the addition is performed in this manner, the value obtained is shifted left five places before being assigned to A9U5.

n. SET A9U5 TO A5S2..6 + A6S3..7 \$

Because of the two scaling specifiers, $A1 = 6$, $A2 = 7$, and $Z = 5$. Since $\min(A1, A2) = \min(6, 7) = 6 > Z$, then $Ar = \min(A1, A2) = 6$. Thus both operands are prealigned to six fractional bits prior to the addition. However, in evaluating this expression, A5S2 is treated as having six fractional bits and A6S3 is treated as having seven

fractional bits, because of the scaling specifiers. Therefore, the value of A6S3 is shifted right one bit and the value of A5S2 is not shifted. After the addition, the resulting value is shifted right one bit prior to being assigned to A9U5.

5.3.1.6 MSCALE Scaling Algorithm

The MSCALE scaling rules apply when the scaling specification appears in an options declaration. These rules are context-free. That is, the number of magnitude bits (integer bits and fractional bits) in the result of a numeric operation is determined only by the operator and the attributes of the operands, as defined in the scaling rules below. The alignment used to arrive at the resultant scaling is also defined below. The symbols used in the MSCALE scaling algorithm are the same as those listed in paragraph 5.3.1.3.1.

5.3.1.6.1 Integer Arithmetic Scaling Algorithm

The integer arithmetic rules apply only if X and Y are both of integer type.

a. Addition and Subtraction

$$N_r = \max(N_1, N_2)$$

b. Multiplication

1. $N_r = \min(N_1, 32) + \min(N_2, 32)$

2. If $N_1 > 32$, the rightmost 32 bits of X are used to perform the operation.

3. If $N_2 > 32$, the rightmost 32 bits of Y are used to perform the operation.

c. Division

1. $N_r = \min(N_1 - C, 32)$

2. If $N_2 > 32$, the rightmost 32 bits of Y are used to perform the operation.

5.3.1.6.2 Fixed-Point Arithmetic Scaling Algorithm

The fixed-point arithmetic rules apply if X or Y is of fixed-point type and neither is of floating-point type.

The fixed-point scaling rules are defined to retain the most significant bits of the largest value that can result from an

operation (except that one bit may be lost on the left if an addition or subtraction overflows beyond the left bit of the number with the largest number of integer bits). In aligning to retain the most significant bits, bits may be lost from the right of an operand or from the result of an operation. In some cases when alignment is done to retain the most significant bits, the entire nonzero value of an operand or operation may be lost, and the result will be zero.

a. Addition and Subtraction

1. $I_r = \max(I_1, I_2)$
2. $A_r = \min(63 - I_r, \max(A_1, A_2))$
3. X and Y are aligned to A_r and then used to perform the operation.

b. Multiplication

1. If M_1 or M_2 is greater than 31, both operands are converted to floating-point and the multiplication is performed in floating-point, giving a floating-point result.
2. Otherwise, the resultant values for I_r and A_r are as follows:

$$\begin{aligned} I_r &= I_1 + I_2 \\ A_r &= A_1 + A_2 \end{aligned}$$

c. Division

1. If $M_1 + M_2 > 31 + C$, both operands are converted to floating-point and the division is performed in floating-point, giving a floating-point result.
2. Otherwise, left shift X by $S = \min(M_2, 31 + C - M_1)$ (right shift if S is negative) and adjust A_1 to $A_1 + S$.
3. The divide operation is performed using X and Y as modified in step 2.
4. The resultant values for I_r and A_r depend on the magnitude of the original operands and the alignment

performed as noted under step 2. The values are given below in terms of the revised values of A1, A2, I1, I2.

$$\begin{aligned} I_r &= I_1 + A_2 - C \\ A_r &= A_1 - A_2 \end{aligned}$$

d. Exponentiation

If the exponent is a constant 2, 3, or 4, the exponentiation is executed as the necessary number of multiplications according to the MSCALE scaling rules for multiplication. Otherwise, a run-time routine is called.

Examples

The following expressions will illustrate the MSCALE scaling algorithm.

a. $A\ 20\ S\ 5 \pm A\ 18\ U\ 5$

$$\begin{aligned} I_1 &= 14, A_1 = 5, M_1 = 19 \\ I_2 &= 13, A_2 = 5, M_2 = 18 \\ I_r &= \text{MAX}(I_1, I_2) = 14 \quad A_r = \text{MAX}(A_1, A_2) = 5 \\ \text{Resultant type} &= A\ 20\ S\ 5 \end{aligned}$$

b. $A\ 13\ S\ 5 \pm A\ 10\ U\ 8$

$$\begin{aligned} I_1 &= 7, A_1 = 5, M_1 = 12 \\ I_2 &= 2, A_2 = 8, M_2 = 10 \\ I_r &= 7, A_r = 8 \text{ (operand 1 is aligned to operand 2)} \\ \text{Resultant type} &= A\ 16\ S\ 8 \end{aligned}$$

c. $A\ 22\ S\ 8 \pm A\ 18\ U\ -4$

$$\begin{aligned} I_1 &= 13, A_1 = 8, M_1 = 21 \\ I_2 &= 22, A_2 = -4, M_2 = 18 \\ I_r &= 22, A_r = 8 \text{ (operand 2 is aligned to operand 1)} \\ \text{Resultant type} &= A\ 31\ S\ 8 \end{aligned}$$

d. $A\ 14\ U\ 8 \pm A\ 20\ U\ -4$

$$\begin{aligned} I_1 &= 6, A_1 = 8, M_1 = 14 \\ I_2 &= 24, A_2 = -4, M_2 = 20 \\ I_r &= 24, A_r = 8 \\ \text{Resultant type} &= A\ 32\ U\ 8 \end{aligned}$$

e. $A\ 46\ U\ 6 * A\ 42\ U\ 0$

$$\begin{aligned} I_1 &= 40, A_1 = 6, M_1 = 46 \\ I_2 &= 42, A_2 = 0, M_2 = 42 \end{aligned}$$

Since $M1 > 31$, convert both operands to floating-point to do the multiply. The resultant type is floating-point.

f. A 18 U 6 * A 10 U 0

I1 = 12, A1 = 6, M1 = 18
 I2 = 10, A2 = 0, M2 = 10
 Ir = 22, Ar = 6
 Resultant type is A 28 U 6

g. A 14 S 3 * A 12 U -1

I1 = 10, A1 = 3, M1 = 13
 I2 = 13, A2 = -1, M2 = 12
 Ir = 23, Ar = 2
 Resultant type is A 26 S 2

h. A 18 S 4 / A 12 S 2

I1 = 13, A1 = 4, M1 = 17
 I2 = 9, A2 = 2, M2 = 11
 Shift X left 11 bits, resulting in A1 = 15
 Ir = 15, Ar = 13
 Resultant type is A 29 S 13

i. A 39 S 4 / A 32 S 6

I1 = 34, A1 = 4, M1 = 38
 I2 = 25, A2 = 6, M2 = 31
 Since $M1 + M2 > 31+C$, convert both operands to floating point and do the divide. The resultant type is floating point.

Note

Two numeric operators may not appear without an intervening operand and unless the right operator is a unary plus or unary minus.

5.3.1.7 Numeric Constant Expression

Syntax

```
<numeric constant expression>  
 ::= <numeric constant expression> + <numeric constant term>  
 ::= <numeric constant expression> - <numeric constant term>  
 ::= <numeric constant term>
```

```
<numeric constant term>  
 ::= <numeric constant term> * <numeric constant factor>  
 ::= <numeric constant term> / <numeric constant factor>  
 ::= <numeric constant factor>
```

```
<numeric constant factor>  
 ::= <numeric constant primary> ** <numeric constant factor>  
 ::= [<unary numeric operator>] <numeric constant primary>
```

```
<numeric constant primary>  
 ::= (<numeric constant expression>)  
 ::= <ntag name>  
 ::= <numeric constant>
```

Semantics

A numeric constant expression specifies the calculation of a numeric value that can be determined at compile time. A numeric constant expression is calculated using the numeric operations of unary plus, unary minus, addition, subtraction, multiplication, division and exponentiation. A numeric constant expression may also consist of a single numeric value with no operation performed. A numeric constant factor used as an exponent must be an integer value. Expression evaluation is in the same order as for a numeric expression, but the calculations are carried out according to the constant arithmetic rules.

5.3.1.8 Numeric Constant Value

Syntax

```
<numeric constant value>  
 ::= [-]<numeric constant>  
 ::= <ntag name>
```

Semantics

A numeric constant value is a limited form of numeric constant expression. It is required in only a few contexts (e.g., DATA, STRINGFORM, FORMAT) where use of a general numeric constant expression would create an ambiguity.

5.3.2 Boolean Expression

Syntax

<boolean expression>
 ::= <boolean expression> OR <boolean term>
 ::= <boolean term>

<boolean term>
 ::= <boolean term> AND <boolean factor>
 ::= <boolean factor>

<boolean factor>
 ::= COMP <boolean primary>
 ::= <boolean primary>

<boolean primary>
 ::= (<boolean expression>)
 ::= <relational expression>
 ::= <conditional expression>
 ::= <conditional i/o expression>
 ::= <single-valued data unit>
 ::= <word data unit>
 ::= <function reference>
 ::= <ntag name>
 ::= <boolean constant>

<relational expression>
 ::= <numeric relational expression>
 ::= <boolean relational expression>
 ::= <character relational expression>
 ::= <status relational expression>

Semantics

A Boolean expression specifies the calculation of a Boolean value using the operations of logical sum, logical product, and logical complement. A Boolean expression can also consist of a single Boolean value, with no operation performed, or a relational expression, which is an expression having a Boolean value resulting from a comparison of two operands.

The type of any single-valued data unit or the value of any function reference used as a primary in a Boolean expression must be Boolean.

A word data unit used as a primary in a Boolean expression is considered to be of universal type. The Boolean value of the word data unit is represented by its rightmost bit.

An ntag name used as a primary in a Boolean expression must have the value 0 (false) or 1 (true).

5.3.2.1 Expression Evaluation

The semantics of Boolean expressions are implied by the productions that specify their syntax. There is a hierarchy of operators, which is given in the following table:

<u>Operator</u>	<u>Operation</u>	<u>Hierarchy</u>
COMP	Logical Complement	1
AND	Logical Product	2
OR	Logical Sum	3

Boolean expressions are evaluated according to the operator hierarchy. The operation indicated by an operator may be performed provided that the immediately preceding operator, if present, and the immediately succeeding operator, if present, have a higher hierarchy number. When an operation is performed, the operator and its operand or operands are replaced in the expression by the value of the operation, and the resulting expression is then evaluated.

A parenthesized expression appearing as an operand of a Boolean operator must be evaluated before the operation indicated by the operator can be performed.

A relational expression appearing as an operand of a Boolean operator must be evaluated before the operation indicated by the operator can be performed.

If two consecutive Boolean operators both have hierarchy number 2 or both have hierarchy number 3, the operation indicated by the left operator is performed first.

5.3.2.2 Meaning Of Operators

The operators AND, OR, and COMP result in combined conditions whose values follow the rules of Boolean algebra in which OR is inclusive. Figure 5-02 is a truth table wherein the Boolean type value results from the combination of the operators with the possible Boolean type condition values (1 = true, 0 = false).

A	B	COMP A	A AND B	A OR B
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Figure 5-02. Boolean Operators

Note

Two Boolean operators may not appear without an intervening operator unless the right operator is COMP.

5.3.2.3 Numeric Relational Expression

Syntax

<numeric relational expression>
 ::= <numeric comparand> <relational operator> <numeric comparand>

<numeric comparand>
 ::= <numeric expression>

<relational operator>
 ::= EQ
 ::= NOT
 ::= LT
 ::= GT
 ::= LTEQ
 ::= GTEQ

Semantics

A numeric relational expression specifies the calculation of a Boolean value of 1 (true) or 0 (false) as the result of comparing two numeric values. The comparands may be of different numeric types.

There are six relational operators. When used with numeric or Boolean comparands, their meanings are:

<u>Operator</u>	<u>Meaning</u>
EQ	Is Equal To
NOT	Is Not Equal To
LT	Is Less Than
GT	Is Greater Than
LTEQ	Is Less Than or Equal To
GTEQ	Is Greater Than or Equal To

The value of the relational expression is 1 (true) if the values of the comparands satisfy the stated relation, and is 0 (false) if they do not satisfy the relation.

Each numeric relational expression contains exactly one relational operator. There is no hierarchy among the relational operators.

If one or both of the numeric comparands is a word reference, it is interpreted as a primary of a numeric expression, and is therefore considered to be of type I 32 S.

If one of the comparand values is of floating-point type and the other is of some fixed-point type, the latter is converted to floating-point before performing the comparison (paragraph 5.3.1.2). If the comparand values are of different floating-point types, one is converted to the type of the other before performing the comparison (paragraph 5.3.1.2).

If the comparands are of fixed-point types they are aligned prior to the comparison, according to the rules for fixed-point addition (paragraph 5.3.1.2). This alignment can cause nonzero bits to be lost from the right of one of the comparands and can thus affect the meanings of the relational operators.

Two constants cannot be compared in a numeric relational expression.

Examples

...SPEED GT 55...

This expression is evaluated as false if the value of data unit SPEED is equal to 55 or less.

Implementation Note

The AN/UYK-7 and AN/UYK-43 use a ones complement representation for negative numbers, which means (among other things) that they have two representations for the value zero, commonly called 0 and -0; the AN/UYK-7 and AN/UYK-43 arithmetic hardware do not recognize these as representing the same mathematical value. Therefore, the relational operators EQ, NOT, LTEQ, and GTEQ must be used with great care in those cases when one of the comparands might be zero, as mathematical equality might not be recognized.

5.3.2.4 Boolean Relational Expression

Syntax

<boolean relational expression>
 ::= <boolean comparand> <relational operator> <boolean comparand>

<boolean comparand>
 ::= <single-valued data unit>
 ::= <word data unit>
 ::= <function reference>
 ::= <ntag name>
 ::= <boolean constant>

Semantics

A Boolean relational expression specifies the calculation of a Boolean value of 1 (true) or 0 (false) as the result of comparing two Boolean values.

The Boolean comparands consist of a subset of the Boolean primaries. They must satisfy the same requirements when used as Boolean comparands as they do when used as Boolean primaries (paragraph 5.3.2).

The comparison specified in a Boolean relational expression is a comparison of the numeric values of 1 and 0 that represent the Boolean values of true and false respectively. With this interpretation, the meanings of the relational operators in a Boolean relational expression are the same as their meanings in numeric relational expressions.

Each Boolean relational expression contains exactly one relational operator. There is no hierarchy among the relational operators.

Two Boolean constants cannot be compared in a Boolean relational expression.

Examples

```
VRBL BTYPE B $  
VRBL ITYPE I 10 U $
```

The relational expression

```
BTYPE LT BIT(9)(ITYPE)
```

is true if BTYPE is false and ITYPE is odd.

/(U) CM2Y-MAN-PGR-M5049-R04C0

Note

If both comparands of a relational expression are word references, the expression is considered to be a numeric relational expression.

5.3.2.5 Character Relational Expression

Syntax

<character relational expression>
 ::= <character comparand> <relational operator> <character comparand>

<character comparand>
 ::= <character expression>
 ::= <word data unit>

Semantics

A character relational expression specifies the calculation of a Boolean value of 1 (true) or 0 (false) as the result of comparing two character values. The lengths of the comparands can be different.

When used with character comparands, the meanings of the relational operators are:

<u>Operator</u>	<u>Meaning</u>
EQ	Is The Same As
NOT	Is Different From
LT	Collates Before
GT	Collates After
LTEQ	Collates Before or Is The Same As
GTEQ	Collates After or Is The Same As

The value of the relational expression is 1 (true) if the values of the comparands satisfy the stated relation in the collating sequence of the target machine and is 0 (false) if they do not satisfy the relation.

Each character relational expression contains exactly one relational operator. There is no hierarchy among the relational operators.

If a character comparand is a word data unit, it is treated as a character data unit of type H 4..

The shorter of the two comparands determines the number of characters to be compared, unless the shorter comparand is a constant. When the shorter comparand is a constant, it is effectively extended on the right with blanks to the length of the longer comparand.

/(U) CM2Y-MAN-PGR-M5049-R04C0

The comparisons are performed on a character-by-character basis, beginning with the leftmost characters. The first inequality found by this process determines the result of the comparison. If no inequality is found, the comparands are determined to be the same.

At least one character comparand must be nonconstant.

Examples

VRBL CTYPE H 6 \$

The relational expression in the phrase

CTYPE NOT H()

is true if CTYPE is not all spaces.

Note

If both comparands of a relational expression are word references, the expression is considered to be a numeric relational expression.

5.3.2.6 Status Relational Expression

Syntax

<status relational expression>
 ::= <status comparand> <relational operator> <status comparand>

<status comparand>
 ::= <status expression>

Semantics

A status relational expression specifies the calculation of a Boolean value of 1 (true) or 0 (false) as the result of comparing two status values. The comparands may be of different status types.

When used with status comparands, the meanings of the relational operators are:

<u>Operator</u>	<u>Meaning</u>
EQ	Is Equal To
NOT	Is Not Equal To
LT	Is Less Than
GT	Is Greater Than
LTEQ	Is Less Than Or Equal To
GTEQ	Is Greater Than or Equal To

The value of the relational expression is 1 (true) if the values of the comparands satisfy the stated relation and is 0 (false) if they do not satisfy the relation.

The meanings of equal, less than, and greater than are relative to the encoded values of the status comparands, not to their conceptual values.

At least one comparand of a status relational expression must be nonconstant.

Examples

```
VRBL STATUSQ S 'LOW', 'MED', 'HIE' $
STATUSQ NOT 'MED'
```

This expression is evaluated as true if the value of variable STATUSQ is 'LOW' or 'HIE'.

5.3.2.7 Conditional Expression

Syntax

```
| <conditional expression>  
  ::= <checkable reference> <validity test>  
  ::= <single-valued data unit> <parity test>  
  
| <checkable reference>  
  ::= <subscripted data unit>  
  ::= <word data unit>  
  
<validity test>  
  ::= VALID  
  ::= INVALID  
  
<parity test>  
  ::= ODDP  
  ::= EVENP
```

Semantics

A conditional expression specifies the calculation of a Boolean value. The value may be the result of testing the validity of subscript values or a word specification value or the result of testing the parity of a single-valued data unit.

A validity test may only be performed on a table item reference or a word reference. For any table item reference or word reference that contains a subscript expression, the validity test tests the values of all subscript expressions in the reference. For a word reference, the value of the word specification is also tested. The value of a conditional expression containing the keyword VALID is 1 (true) if each subscript value (if any) is in its valid range for the named table, and the value of the word specification (if any) specifies a word of the table item or variable; otherwise, it is 0 (false). The value of a conditional expression containing the keyword INVALID is 1 (true) if any subscript value lies outside its valid range for the named table, or if the word specification (if present) does not specify a word of the table item or variable; otherwise, it is 0 (false). The valid range of a subscript is determined by the size of the table at load time (if the number of items of the table is specified by a load time constant) and by the value of the major index of the table, if any, at the time the conditional expression is evaluated.

The value of a conditional expression consisting of a single-valued data unit followed by the keyword ODDP is 1 (true) if the number of "on" bits (1 bits) in the value of the data unit is

odd; otherwise, it is 0 (false). The value of a conditional expression consisting of a single-valued data unit followed by the keyword EVENP is 1 (true) if the number of "on" bits (1 bits) in the value of the data unit is even; otherwise, it is 0 (false). The subject of these predicates is the value of the data unit only; Only single words or subfields thereof may be tested for parity; checking double-words is not allowed.

Examples

```
TABLE CATA H 5 2 $
  SUB-TABLE CATB 0 1 $
END-TABLE CATA $
```

```
·
·
CATA(I) VALID
```

```
·
·
CATB(J) INVALID
```

The first conditional expression is true if I has a value of 0 or 1. The second conditional expression is true if J is not 0.

```
VRBL STAT I 32 S $
```

```
·
·
STAT EVENP
```

The number of bits set to 1 in the variable STAT are counted. If their sum is an even number, the condition is true.

5.3.2.8 Conditional I/O Expression

Syntax

<conditional i/o expression>
 ::= <file name> <file status operator> <status constant>

<file status operator>
 ::= EQ
 ::= NOT

Semantics

A conditional I/O expression specifies the calculation of a Boolean value based on the status of a file operation.

The value of a conditional I/O expression is $\bar{1}$ (true) if the current file status of the named file (as set by the monitor as a result of the most recent I/O operation) and the status constant specified in the expression satisfy the specified relation; otherwise, it is 0 (false). The status constant must be one of those in the file status list of the named file's declaration.

A conditional I/O expression is valid only if the compiler option MONITOR has been specified.

Examples

```
FILE INPUTC H 500 R 120 MT5 'BUSY', 'FINISHED', 'SENTINEL',  
    'HARDWARE', 'BAD PKG', 'EMPTY', 'NODEVICE' $
```

```
IF INPUTC EQ 'SENTINEL' THEN RETURN $
```

INPUTC can assume six values, BUSY, FINISHED, SENTINEL, ..., NODEVICE, which are represented internally by the integer values 0, 1, 2, ..., 6. If the integer value for file status is equal to 2, the return statement is executed. Otherwise, control is passed to the next sequential statement.

Note

The meanings of file status values are not arbitrary; they are determined by the CMS-2Y monitor program (see paragraph 4.23).

5.3.3 Character Expression

Syntax

```
<character expression>  
 ::= <character expression> CAT <character primary>  
 ::= <character primary>
```

```
<character primary>  
 ::= <single-valued data unit>  
 ::= <function reference>  
 ::= <character modified data unit>  
 ::= <character constant>  
 ::= (<character expression>)
```

Semantics

A character expression specifies the calculation of a character value.

Any single-valued data unit or the value of any function reference used as a character primary must be of character type.

The operator CAT specifies the string operation of concatenation.

Examples

```
VRBL MESSAGE H 13 $  
VRBL FATAL H 6 P H(FATAL ) $  
VRBL ERNUM H 1 P H(2) $
```

```
SET MESSAGE TO FATAL CAT H(ERROR ) CAT ERNUM $
```

The result of this set phrase is to assign to variable MESSAGE the character expression value FATAL ERROR 2.

Implementation Note

If the referenced character strings all lie within the boundaries of a word, the compiler will generate in-line code to achieve the concatenation. If the referenced characters cross word boundaries, the compiler will generate a procedure call to a monitor routine to do the concatenation. In this case, the compiler option MONITOR must be specified.

5.3.4 Status Expression

Syntax

```
<status expression>  
  ::= <single-valued data unit>  
  ::= <function reference>  
  ::= <status constant>
```

Semantics

A status expression specifies the calculation of a status value. There are no status operators; operations on status data can only be performed through function references.

Any single-valued data unit or the value of any function reference used as a status expression must be of status type.

Examples

```
VRBL OPSTAT S 'READY', 'NOGO', 'LOAD' $
```

```
IF OPSTAT NOT 'READY' THEN RETURN $
```

This conditional statement contains status expressions as the two terms in the relational expression. The first is a status type data unit; the second is a status constant defined as one of the states of the typed data unit.

5.3.5 Bit String Expression

Syntax

```

<bit string expression>
  ::= <bit string expression> OR <bit string term>
  ::= <bit string expression> XOR <bit string term>
  ::= <bit string term>

<bit string term>
  ::= <bit string term> AND <bit string factor>
  ::= <bit string factor>

<bit string factor>
  ::= COMP <bit string primary>
  ::= <bit string primary>

<bit string primary>
  ::= (<bit string expression>)
  ::= <numeric expression>
  ::= <boolean expression>
  ::= <character expression>
  ::= <status expression>

```

Semantics

A bit string expression specifies the calculation of a bit string using the logical (bit-by-bit) operations of logical sum, logical exclusive sum, logical product, and logical complement. The primaries of a bit string expression are interpreted as strings of bits, without regard to any declared or generated attributes.

No operand of a bit string expression can exceed 64 bits in length.

5.3.5.1 Expression Evaluation

The semantics of bit string expressions are implied by the productions that specify their syntax. There is a hierarchy of operators, which is given by the following table:

<u>Operator</u>	<u>Operation</u>	<u>Hierarchy</u>
COMP	Logical Complement	1
AND	Logical Product	2
OR	Logical Sum	3
XOR	Logical Disjoint Sum	3

Bit string expressions are evaluated according to the operator hierarchy. The operation indicated by an operator may be

performed provided that the immediately preceding operator, if present, and the immediately succeeding operator, if present, have a higher hierarchy number. When an operation is performed, the operator and its operand or operands are replaced in the expression by the value of the operation, and the resulting expression is then evaluated.

A parenthesized expression appearing as an operand of a bit string operator must be evaluated before the operation indicated by the operator can be performed.

If two consecutive bit string operators both have hierarchy number 2 or hierarchy number 3, the operation indicated by the left operator is performed first.

5.3.5.2 Meaning Of Operators

The bit string operators and bit string functions are alternatives. The values of the operators AND, OR, and XOR for any two operands are identical to the values of the functions ANDF, ORF, and XORF, respectively, with those operands as actual arguments; the value of the operator COMP for any operand is identical to the value of the function COMPF with that operand as an actual argument.

Examples

```
VRBL BITS I 8 U $
```

```
·
```

```
...COMP BITS ...
```

The eight bits of variable BITS are complemented.

```
VRBL MASK I 7 U $
```

```
·
```

```
...MASK AND 0(170)...
```

The rightmost three bits of variable MASK are cleared by this operation. The value of the other four bits remains unchanged.

```
VRBL BETA F $
```

```
·
```

```
...BETA LT 0 OR 0(7070)...
```

If BETA contains a negative value, the value of the expression will be 7071 octal. Otherwise, the value of the expression is 7070 octal.

```
VRBL EPSILON A 15 S 6 P O(224.57) $
VRBL YY A 11 S 3 P O(123.4) $
VRBL ZZ A 13 S 3 P O(402.5) $
```

...EPSILON XOR(YY OR ZZ)...

The bit values of the OR expression will be 1 for any bit setting of 1 in either YY or ZZ, but the resulting intermediate octal value is 5235 (not 523.5) because it is interpreted as a universal type: i.e., a string of bits. Likewise, the string of bits in EPSILON is interpreted octally as 22457 (instead of 224.57). The result of XOR expression (5235 XOR 22457) is 27662 octal because like bits in the two operands produce a 0 bit, and unlike bits produce a 1 bit.

Notes

Many bit string expressions and Boolean expressions are syntactically identical, and certain Boolean expressions can often be thought of as special cases of bit string expressions. The primary differences between the two types of expressions are the types of the values of the expression (universal and Boolean, respectively), the operator XOR is permitted in a bit string expression but not in a Boolean expression, and a bit string expression must contain at least one bit string operator.

Two bit string operators may not appear without an intervening operand unless the right operator is COMP.

5.3.6 Structured Expression

Syntax

<structured expression>
 ::= <single-valued data unit>

Semantics

A structured expression specifies a structured value. There are no operators that operate on structured values.

Any single-valued data unit used as a structured expression must be of a structured type.

Examples

```
TABLE TBLA H 1 5 $  
FIELD XX I 3 S 0 31 $  
FIELD YY H 2 0 16 $  
  ITEM-AREA ITA $  
END-TABLE TBLA $
```

```
SET TBLA(4) TO ITA $  
SET ITA TO TBLA(3) $
```

In the two assignment phrases, both of the source data units (ITA and TBLA(3)) are structured expressions.

SECTION 6. STATEMENTS

Syntax

<statement>
 ::= [<statement label>&] <simple statement>
 ::= [<statement label>&] <conditional statement>

<simple statement>
 ::= <simple phrase> [THEN [<statement label>] <simple
 phrase>]& \$

<statement label>
 ::= <statement name>.

-

<statement name>
 ::= <name>

Semantics

A statement specifies an action to be performed during the execution of a CMS-2Y(7) program.

- <statement name> - A name by which the statement can be referenced.
- <simple statement> - A statement that can stand alone or be a part of a more complex statement.
- THEN - Optional. A language keyword used to connect a sequence of simple phrases, optionally preceded by statement labels, to form a single statement.
- <conditional statement> - A statement whose execution depends on the value of a Boolean expression. A conditional statement cannot be a part of a more complex statement.

A statement label consists of a statement name followed immediately by a period. Spaces between the name and the period are not permitted.

Statement names do not have to be declared. The appearance of a name as part of a statement label implicitly declares the name to be a statement name. The scope of the implicit declaration is the system procedure block in which the statement appears.

/(U) CM2Y-MAN-PGR-M5049-R04C0

When a statement label appears before a statement, it names the statement. If the statement is a block, the name of the statement is the name of the block.

Use of the keyword THEN to form a simple statement from a sequence of simple phrases permits the sequence of phrases to be treated syntactically as a single statement. The execution of such a statement is as if the sequence of phrases had been written using the terminator (\$) instead of THEN.

6.1 Simple Phrases

Syntax

```
<simple phrase>  
  ::= <imperative phrase>  
  ::= <statement block>
```

Semantics

A simple phrase occurs in one of two forms: an imperative phrase, which specifies the basic computational actions of the CMS-2Y(7) language, and a statement block, which is a group of statements that collectively specify a derived computational action.

6.1.1 Imperative Phrases

Syntax

```
<imperative phrase>
 ::= <assignment phrase>
 ::= <swap phrase>
 ::= <branch phrase>
 ::= <indexed branch phrase>
 ::= <item branch phrase>
 ::= <procedure call phrase>
 ::= <indexed procedure call phrase>
 ::= <item procedure call phrase>
 ::= <stop phrase>
 ::= <return phrase>
 ::= <exit phrase>
 ::= <resume phrase>
 ::= <executive call phrase>
 ::= <shift phrase>
 ::= <open phrase>
 ::= <close phrase>
 ::= <endfile phrase>
 ::= <define label phrase>
 ::= <check label phrase>
 ::= <file positioning phrase>
 ::= <record positioning phrase>
 ::= <output phrase>
 ::= <input phrase>
 ::= <encode phrase>
 ::= <decode phrase>
 ::= <convertin phrase>
 ::= <convertout phrase>
 ::= <display phrase>
 ::= <snap phrase>
 ::= <trace phrase>
 ::= <end-trace phrase>
 ::= <null phrase>
 ::= <direct code phrase>
```

Semantics

An imperative phrase specifies an elemental action of a CMS-2Y(7) program--an action that cannot be refined into subactions within the language.

6.1.1.1 Assignment Phrase (Classes and Compatibility)

Syntax

```

<assignment phrase>
    ::= SET <receptacle>@ TO <source> [<remainder phrase>]
       [<overflow phrase>]

<receptacle>
    ::= <single-valued data unit> [<scaling specifier>]
    ::= <word data unit> [<scaling specifier>]
    ::= <multivalued data unit>
    ::= <bit string receptacle>
    ::= <character string receptacle>
    ::= <core address receptacle>

<bit string receptacle>
    ::= BIT(<bit string start>[,<bit string length>]) (<parent
       receptacle>)

<character string receptacle>
    ::= CHAR(<character string start>[,<character string
       length>]) (<parent receptacle>)

<parent receptacle>
    ::= <variable name>[(<field name>)]
    ::= <subscripted data unit>

<core address receptacle>
    ::= CORAD(<table name>)

<source>
    ::= <expression>
    ::= <multivalued data unit>

<remainder phrase>
    ::= SAVING <remainder receptacle>

<remainder receptacle>
    ::= <single-valued data unit>

<overflow phrase>
    ::= OVERFLOW <statement name>
    
```

Semantics

An assignment phrase specifies the assigning of a value to one or more data units or parts of data units.

SET - A language keyword indicating an assignment phrase.

- <receptacle> - Specification of a data unit or part of a data unit that is to receive a value.
- TO - A language keyword separating the list of receptacles from the source.
- <source> - The value or values to be assigned to the receptacles.
- SAVING - Optional. A language keyword indicating that the remainder of a fixed-point division operation is to be saved.
- <remainder receptacle> - Optional. A single-valued data unit that is to receive the remainder of a fixed-point division operation.
- OVERFLOW - Optional. A language keyword indicating that program control is to be transferred if a target machine overflow condition occurs.
- <statement name> - Optional. The name of the statement to be executed next if a target machine overflow condition occurs.

The semantics of the bit string receptacle are the same as those of a bit modified data unit, except that the string of bits being specified is to receive a value as a result of the assignment phrase. The string of bits is temporarily defined to have universal type.

The semantics of the character string receptacle are the same as those of a character modified data unit, except that the string of bits being specified is to receive a value as the result of the assignment phrase. The string of bits is temporarily defined to have character type.

The parent receptacle of a bit string receptacle or a character string receptacle must not be an untyped structure. Thus, for example, a table item may not be used as a parent receptacle unless it is an item of a typed table.

The core address receptacle is limited to specifying a memory address of an indirect table. The table name in a core address

receptacle must be the name of an indirect table. Core address receptacles have A 16 U 0 type.

If the optional scaling specifier appears as part of a receptacle, it must follow a single-valued data unit of fixed-point type or a word data unit. The value of the scaling specifier specifies the number of fractional bits to be used for the data unit during execution of the assignment phrase, thus temporarily overriding the declared number of fractional bits. It does not change the value of the scaling controller (paragraph 5.3.1) used in the calculation of the value of the source expression, however. The value of the scaling controller depends only on the declared number of fractional bits of the data unit.

When more than one receptacle is listed in an assignment phrase, the source is evaluated once and its value is assigned to each of the receptacles. Each receptacle must be assignment compatible with the source, as defined below. If the source is a numeric expression, the last (rightmost) receptacle determines the value of the scaling controller used in evaluating the expression.

If the optional remainder phrase appears, the source must be a numeric expression containing at least one fixed-point division operation. The remainder of the last fixed-point division operation executed during the evaluation of the expression is assigned to the remainder receptacle, which may be any single-valued data unit as long as it is of numeric type.

If the optional overflow phrase appears, the target machine overflow indicators are tested after evaluation of the expression. If an indicator is on it is turned off, the value of the expression is assigned to the receptacle(s), and the named statement is executed next. Once turned on, the indicators tested by the overflow phrase remain on until they are tested. Thus taking the branch specified by the overflow phrase does not imply that an overflow occurred during evaluation of the source expression.

If the compiler option MONITOR has been specified, both the hardware fixed-point overflow indicator and the floating-point error indicator maintained by the CMS-2 monitor are tested to detect an overflow condition. If MONITOR has not been specified, only the fixed-point overflow indicator is tested.

There are five classes of assignment: simple, untyped, word, value flush, and multivalued.

6.1.1.1.1 Simple Assignment

An assignment is simple if the receptacle is simple. This includes receptacles whose type is universal. The semantics of,

and restrictions on, simple assignments depend on the type of the receptacle.

- a. The receptacle type is numeric. The source must be an expression of numeric or universal type.

If the source type is numeric, the value of the source is converted to the type of the receptacle before the assignment.

If the source type is universal, the bit string of the source is assigned to the bits of the receptacle. If the source is longer than the receptacle, excess high-order bits are truncated. If the source is shorter than the receptacle, the source bit string is extended on the left with zero bits to the length of the receptacle and the extended string is assigned to the receptacle.

- b. The receptacle type is Boolean. The source must be an expression of Boolean or universal type.

If the source type is Boolean, its value is assigned to the receptacle.

If the source type is universal, its value at the time of the assignment must have 0 bits at all positions except the least significant bit. The value of the least significant bit is the Boolean value of the universal type source but, for reasons of efficiency, bits of the source other than the least significant bit may be assigned to the bits allocated to the receptacle.

- c. The receptacle type is character. The source must be an expression of character type or a word reference.

If the source is a character constant longer than the receptacle, the rightmost characters of the constant are truncated to the length of the receptacle and the resultant value replaces the value of the receptacle. If the source is a character constant of the same length as the receptacle, the value of the constant replaces the value of the receptacle. If the source is a character constant shorter than the receptacle, the value of the constant is padded on the right with blank characters to generate a constant of the length of the receptacle and the generated constant replaces the value of the receptacle.

If the source is a nonconstant character expression, the lesser of the length of the source and the length of the receptacle is the number of characters to be assigned.

The assignment is effectively performed from left to right, beginning with the leftmost characters of the source and receptacle.

If the source is a word data unit, the word is interpreted as a string of four characters.

- d. The receptacle type is status. The source must be an expression of status type.

The internal numeric value that is the encoding of the source value is assigned to the receptacle. If the source and the receptacle are of different status types, assignment can result in a change of the conceptual value.

- e. The receptacle type is universal. The source must be an expression of numeric, Boolean, character, status, or universal type.

If the source type is numeric, the receptacle is treated as an integer type whose length is the length of the receptacle. The value of the source expression is converted to integer and the converted value is assigned to the bits of the receptacle.

If the source type is Boolean, the value of the source is assigned to the least significant bit of the receptacle, and the other bits of the receptacle are set to zeros. For reasons of efficiency, the compiler may assume that the value of the source expression is represented by a string of bits that are all 0, except for possibly the least significant bit.

If the source type is character, the value of the source is assigned to the receptacle and is right-justified. If the length of the source is greater than the length of the receptacle, the leftmost bits of the source are truncated. If the length of the source is less than the length of the receptacle, the leftmost bits of the receptacle are set to zeros. For reasons of efficiency, the compiler may assume, in the latter case, that the value of the source is represented by a string of bits that begins with leading zeros. The bit length of the receptacle, nevertheless, must be a multiple of 8.

If the source type is status, the internal numeric value that is the encoding of the source value is assigned to the rightmost bits of the receptacle, and the other bits of the receptacle are set to zeros.

If the source type is universal, the value of the source is assigned to the value of the receptacle and is right-justified. If the length of the source is greater than the length of the receptacle, the leftmost bits of the source are truncated. If the length of the source is less than the length of the receptacle, the leftmost bits of the receptacle are set to zeros.

Examples

a. Numeric.

```
VRBL A9S4 A 9 S 4 P .9375 $
VRBL A14S6 A 14 S 6 P .984375 $
VRBL CTYPE H 13 P H(GOOD HEAVENS!) $
VRBL FTYP F P 314159265E-8 $
VRBL FTYPE F P 1E57 $
VRBL I29U I 29 U P 536870911 $
VRBL I10U I 10 U P 1023 $
```

1. Numeric: Integer

```
SET I10U TO 3 $
```

The source constant is assigned to receptacle variable I10U.

```
SET I10U TO A14S6 $
```

The value of fixed-point type source variable A14S6 is converted to integer type and assigned to receptacle variable I10U. The six fractional bits of A14S6 are not transferred to I10U.

```
SET I10U TO FTYPE $
```

The value of floating-point type source variable FTYPE is converted to integer type and assigned to receptacle variable I10U.

```
SET I29U TO I10U $
```

The value of integer type source variable I10U is assigned to receptacle variable I29U.

```
SET I10U TO BIT(6,3)(A9S4) $
```

Source variable A9S4 is converted to universal type and the three rightmost bits of its contents are assigned to receptacle variable I10U.

SET I10U TO CNT(A9S4) \$

The expression argument A9S4 is converted to universal type, and the bits set in the converted operand are then counted and the count is assigned to receptacle variable I10U.

SET I10U TO COMP A9S4 \$

The expression argument A9S4 is converted to universal type, and the contents of the converted operand are then complemented and assigned to receptacle variable I10U.

SET I10U TO A14S6 AND A9S4 \$

Each expression argument is converted to universal type, and if corresponding bits in each operand of the argument are set, the corresponding bit of receptacle variable I10U is set (logical product). No prealignment of the values of A14S6 and A9S4 occurs, in spite of their being defined with different numbers of fractional bits. Thus, for example, the 1s bit of A14S6 and the 4s bit of A9S4 are anded.

2. Numeric: Fixed-point

SET A14S6 TO 27.2 \$

The fixed-point (arithmetic) type source constant is assigned to receptacle variable A14S6. Note that .2 is equal to an infinite series of alternating pairs of 0s and 1s in binary. The compiler computes that four fractional bits are required; however, the context of this assignment requires that the value be realized with six fractional bits.

SET A14S6 TO A9S4 \$

Fixed-point type source variable A9S4 is shifted by the compiler so that its radix aligns with the radix of receptacle variable A14S6. Thus, when its value is transferred to A14S6, the two rightmost bits of the fractional part in A14S6 are set to zero.

SET A14S6 TO FTYPE \$

The contents of floating-point type source variable FTYPE are converted to fixed-point type and assigned to receptacle variable A14S6.

SET A14S6 TO I10U \$

The contents of source variable I10U are converted to fixed-point type and assigned to receptacle variable A14S6. If the value in I10U is larger than seven bits, the assign is undefined.

SET A9S4 TO A9S4 OVERFLOW NEXT \$
NEXT. SET A9S4 TO I10U/A14S6 SAVING FTYPE OVERFLOW FOLD \$

FOLD. OVERFLO \$

The first statement clears the fixed-point overflow designator if it is set; whether it is set or not, control will always transfer to the statement labeled NEXT. The portion of the statement, SET A9S4 TO A9S4, generates no code; it exists to justify the OVERFLOW NEXT portion of the statement which guarantees that the overflow designator will be clear when statement NEXT starts execution. The quotient from the operation of dividing variable I10U by variable A14S6 is assigned to variable A9S4. The remainder is assigned to variable FTYPE. If during execution of this statement the division operation results in an overflow condition (i.e., the quotient is too big for variable A9S4), program control transfers to label FOLD wherein procedure OVERFLO is called.

Examples of assigning universal type source operands to fixed-point type receptacle operands are parallel to those shown above in assigning universal types (BIT, CNT, COMP, and logical functions) to integer types. The difference is that allowances are made for properly aligning the radix point in the fixed-point type receptacle.

3. Numeric: Floating-point

SET FTYPE TO 2.54E-22 \$

The floating-point type source constant is assigned to receptacle variable FTYPE.

SET FTYPE TO A14S6 \$

The fixed-point type source variable A14S6 is converted to floating-point type and its contents assigned to receptacle variable FTYPE.

SET FTYPE TO FTYP \$

The contents of floating-point type variable FTYP are assigned to receptacle variable FTYPE.

```
SET FTYPE TO I10U $
```

The integer type source variable I10U is converted to floating-point type and its contents assigned to receptacle variable FTYPE.

Examples of assigning universal type sources to the floating-point type receptacle are parallel to those shown above in assigning universal type sources (BIT, CNT, COMP, and logical functions) to the integer type receptacle. The difference is that object code will be generated to convert to floating-point format.

b. Boolean.

```
VRBL A5U0 A 5 U 0 P 31 $  
VRBL ATYPE A 16 U 8 P 255 $  
VRBL BTYPE B P 1 $
```

```
SET BTYPE TO 0 $
```

The Boolean type receptacle variable BTYPE is assigned the value false (0).

```
SET BTYPE TO BIT(4,1)(A5U0) $
```

Source variable A5U0 is converted to universal type and its rightmost bit is assigned to receptacle variable BTYPE.

```
SET BTYPE TO ATYPE GT 0 $
```

The numeric relational expression ATYPE GT 0 is evaluated, and the result (0 or 1) is assigned to receptacle variable BTYPE.

```
SET BTYPE TO (ATYPE OR A5U0) AND 1 $
```

Each expression argument is converted to universal type, and if corresponding bits in either or both operands of the argument are set, the corresponding bit of the universal operand is set (logical sum). This operand is then logically multiplied with a true condition and the resulting bit value is assigned to receptacle variable BTYPE.

c. Character.

```
VRBL CTYPE H 5 $  
VRBL CHARACS H 6 P H(PLANTS) $
```

```
SET CTYPE TO H( ) $
```

The character constant space is assigned to all five character positions in receptacle variable CTYPE.

```
SET CTYPE TO CHARACS $
```

The characters PLANT are assigned sequentially to each of the five character positions in receptacle variable CTYPE.

d. Status.

```
VRBL STYPE S 'A', 'B', 'C', 'D', 'E' $
```

```
SET STYPE TO 'D' $
```

Variable STYPE is assigned the value 3 because the letter D is the fourth status constant in the variable declaration.

e. Universal.

```
VRBL A26S4 A 26 S 4 $  
TABLE VALUES V (I 10 U) 20 $  
END-TABLE VALUES $
```

```
SET BIT(22,4)(A26S4) TO 0(10) $
```

The fractional portion of variable A26S4 is assigned the constant 0.5.

```
SET BIT(0)(VALUES(19)) TO 0 $
```

The most significant bit (the tenth bit from the right) in the last item of table VALUES is cleared.

6.1.1.1.2 Untyped Assignment

An assignment is untyped when both the source and the receptacle are untyped structures.

The words of the source are assigned to the words of the receptacle. The lesser of the number of words of the source and the number of words of the receptacle is the number of words to be assigned. The assignment is effectively performed on a word-by-word basis, beginning with the first word of the source and the first word of the receptacle. Thus, if the source and the receptacle have the same structure, the value of the source is assigned to the receptacle (even if the source and receptacle are items of tables of different table types).

Examples

```

TABLE TABS V 6 8 $
  FIELD H7F H 7 0 31 $
  FIELD I8U I 8 U 5 7 $
  FIELD NOUDONT A 12 S 3 4 14 $
  ITEM-AREA GROSSVAL $
END-TABLE TABS $

SET TABS(2) TO GROSSVAL $

```

The item-area GROSSVAL is assigned to the six words comprising the third item of table TABS.

6.1.1.1.3 Word Assignment

An assignment is a word assignment if the receptacle is a word data unit. The source must be a numeric, Boolean, or character expression. (A source consisting of a word data unit is considered a numeric expression.)

If the source is a numeric expression, the receptacle is considered to be of type I 32 S. The effect of the assignment is that of a simple assignment to a receptacle of fixed-point type.

If the source is a Boolean expression, the receptacle is considered to be of a universal type of length 32. The value of the Boolean expression is extended to the length of the receptacle by appending 0 bits on the left and the generated value is assigned to the receptacle.

If the source is a character expression, the bits of the receptacle are interpreted as a string of four characters. The leftmost characters of the value of the source are assigned to the receptacle.

Examples

```

TRUE EQUALS 1 $
TABLE TAB2 V 5 9 $
END-TABLE TAB2 $
VRBL BOOL B P 1 $
LOC-INDEX XX $

SET TAB2(0,0) TO 28 $

```

The first word of the first item in table TAB2 is assigned the numeric constant 28. The receptacle is an integer type.

/(U) CM2Y-MAN-PGR-M5049-R04C0

```
SET TAB2(8,XX) TO TRUE EQ BOOL $
```

The XXth word of the last item in table TAB2 is assigned the Boolean value 1 (true) since BOOL has the value 1. The receptacle is a Boolean type.

```
SET TAB2(XX,4) TO H($$$$) $
```

The last word (four characters) of the XXth item in table TAB2 is set to four dollar signs. The receptacle is a character type.

6.1.1.1.4 Value Flush Assignment

An assignment is a value flush assignment if the receptacle is not simple and the source is a numeric, Boolean, or character expression.

The value of the source expression is assigned to each word of the receptacle, which is treated as being of universal type.

Examples

```
TABLE TAB3 H 32 256 $  
END-TABLE TAB3 $  
VRBL I5S I 5 S P 15 $
```

```
SET TAB3(18) TO I5S $
```

All 32 words of the nineteenth item in table TAB3 are assigned the value 15.

```
SET TAB3 TO I5S $
```

All 8,192 words of table TAB3 are assigned the value 15.

6.1.1.1.5 Multivalued Assignment

An assignment is a multivalued assignment if the source and receptacle are both tables.

The words of the source are assigned to the words of the receptacle. The lesser of the number of words of the source and the number of words of the receptacle is the number of words to be assigned. The assignment is effectively performed on a word-by-word basis, beginning with the first word of the source and the first word of the receptacle. Table type is ignored in a multivalued assignment.

Examples

TABLE TAB4 V 2 8 \$
 END-TABLE TAB4 \$

TABLE TAB5 H 5 6 \$
 END-TABLE TAB5 \$

SET TAB4 TO TAB5 \$

The first 16 consecutive words of TAB5 are stored in the same sequence in TAB4.

SET TAB5 TO TAB4 \$

All 16 words of TAB4 are stored in the same sequence into the first 16 words of TAB5, irrespective of their differing table types. The last 14 words of TAB5 are unchanged.

Note

Assignment operations in CMS-2Y(7) can be made explicitly by means of the assignment phrase or implicitly through parameter passage in the procedure call phrase, the indexed procedure call phrase, and the function reference. Source operands are converted to the type of the receptacle immediately prior to assignment (see Figure 6-01).

		Source Operand Type					
		A	F	B	C	S	U
Receptacle Operand Type	A	A	A				A
	F	F	F				F
	B			B			B
	C				C		C
	S					S	S
	U	U	U	U	U	U	U

Figure 6-01. Simple Assignment Operation Types

6.1.1.2 Swap Phrase

Syntax

<swap phrase>
 ::= SWAP <receptacle 1>, <receptacle 2>

<receptacle 1>
 ::= <receptacle>

<receptacle 2>
 ::= <receptacle>

Semantics

A swap phrase specifies that two stored values are to be interchanged.

SWAP - A language keyword indicating a swap phrase.

<receptacle 1> - Two receptacles that contain the values to be interchanged.
<receptacle 2>

The two receptacles must be assignment-compatible with each other and must not share memory.

The effect of the execution of a swap phrase is as follows:

- a. The value of the first receptacle is assigned to a third receptacle. This third receptacle and the first have identical attributes; thus, no data is lost or changed in any way as a result of this assignment.
- b. The value of the second receptacle is assigned to the first, according to the semantics of an assignment phrase.
- c. The value of the third receptacle is assigned to the second, according to the semantics of an assignment phrase.

Example

```
VRBL ATYPE A 24 S 16 $  
VRBL FTYPE F $  
SWAP FTYPE, ATYPE $
```

The current value of ATYPE is converted to floating-point type and assigned to FTYPE, while the current value of FTYPE is converted to fixed-point type and assigned to ATYPE.

TABLE LOTS V (H 40) 10 \$
ITEM-AREA SPECLOTS \$
END-TABLE LOTS \$

SWAP LOTS (4), SPECLOTS \$

The 40 characters of item-area SPECLOTS are exchanged with the 40 characters of the fifth item in table LOTS.

6.1.1.3 Branch Phrase

Syntax

<branch phrase>
 ::= GOTO <statement name> [<special condition>]

<special condition>
 ::= KEY1
 ::= KEY2
 ::= KEY3
 ::= STOP
 ::= STOP5
 ::= STOP6
 ::= STOP7

Semantics

A branch phrase specifies the next statement to be executed.

- GOTO - A language keyword indicating a branch phrase, an indexed branch phrase, or an item branch phrase.
- <statement name> - The name of the statement to be executed next.
- <special condition> - Optional. A specification that execution of the branch phrase depends on a special hardware or software condition.

If the optional special condition is present, the execution of the branch phrase depends on operator action.

If the special condition KEY1, KEY2, or KEY3 is present, the corresponding console key is tested and the named statement is executed next if that key is on; if it is off, the statement following the branch phrase is executed next. If the compiler option MONITOR has been specified, the simulated console keys provided as a monitor feature (refer to manual M-5050) are tested. If MONITOR has not been specified, the actual target machine central processing unit (CPU) console keys are tested. (Refer to manual M-5048 for a detailed discussion of the CPU console keys and stop indicator lights.)

The special conditions STOP, STOP5, STOP6, and STOP7 are valid only when the compiler option MONITOR has not been specified and the phrase appears in a system procedure element specified to execute in the executive state. The STOP special condition causes an unconditional CPU 4-stop prior to execution of the named

statement. The STOP5, STOP6, and STOP7 special conditions cause a CPU 5-stop, 6-stop, or 7-stop, respectively, prior to execution of the named statement if the corresponding key is on. In all cases of a CPU stop, the named statement is executed next upon normal CPU restart.

Examples

```
GOTO LABEL $
```

```
      .  
      .  
      .  
LABEL. RETURN $
```

Control is transferred to the statement named LABEL.

6.1.1.4 Indexed Branch Phrase

Syntax

<indexed branch phrase>
 ::= GOTO <label switch name> <switch index> [<invalid specification>] [<special condition>]

<switch index>
 ::= <numeric expression>

<invalid specification>
 ::= INVALID <abnormal branch>

<abnormal branch>
 ::= <statement name>

Semantics

An indexed branch phrase specifies the next statement to be executed, depending on the value of an index expression.

- GOTO - A language keyword indicating a branch phrase, an indexed branch phrase, or an item branch phrase.
- <label switch name> - The name of an indexed label switch that specifies the possible statements to which control will be transferred.
- <switch index> - A numeric expression with a value, in conjunction with the indexed switch declaration, that specifies the statement that will be executed next.
- INVALID - Optional. A language keyword indicating that an abnormal branch is being specified.
- <abnormal branch> - Optional. The name of the statement to be executed next if the value of the index is out of range.
- <special condition> - Optional. A specification that execution of the indexed branch phrase depends on a special hardware or software condition.

If the switch index expression is of integer type, its value is the index value. If it is not of integer type, its value is converted to integer and the converted value is the index value.

In the following text, let k denote the index value and n denote the number of switch points in the declaration of the label switch.

If k is in the interval $[0, n-1]$, the statement named in the k th switch point is executed next.

If k is not in the interval $[0, n-1]$ and the invalid specification is present, the abnormal branch statement is executed next.

If k is not in the interval $[0, n-1]$ and the invalid specification is not present, the effect of executing the indexed branch phrase is undefined.

If the optional special condition is present, the execution of the indexed branch phrase depends on operator action.

If the special condition KEY1, KEY2, or KEY3 is present, the corresponding console key is tested and the selected statement is executed next if that key is on; if it is off, the statement following the indexed branch phrase is executed next. If the compiler option MONITOR has been specified, the simulated console keys provided as a monitor feature (refer to manual M-5050) are tested. If MONITOR has not been specified, the actual target machine CPU console keys are tested. (Refer to manual M-5048 for a detailed discussion of the CPU console keys and stop indicator lights.)

The special conditions STOP, STOP5, STOP6, and STOP7 are valid only when the compiler option MONITOR has not been specified and the phrase appears in a system procedure element specified to execute in the executive state. The STOP special condition causes an unconditional CPU 4-stop prior to execution of the selected statement. The STOP5, STOP6, and STOP7 special conditions cause a CPU 5-stop, 6-stop, or 7-stop respectively, prior to execution of the selected statement if the corresponding key is on. In all cases of a CPU stop, the selected statement is executed next upon normal CPU restart.

If both an invalid specification and a special condition are present, the testing for the invalid condition is performed before the action required by the special condition. Therefore, in this case, if the index value is out of range, the abnormal branch statement will be executed next without pause under any circumstances.

Examples

```
LOC-DD $
VRBL A4U0 A 4 U 0 $
SWITCH SWA $
  SB1 $
  SB2 $
  SB3 $
  SB4 $
END-SWITCH SWA $
END-LOC-DD $

GOTO SWA A4U0 $
```

Program control is transferred to the statement name with a switch point position equaling the value contained in variable A4U0 within the range 0 to 3. If the value exceeds 3, the result of executing this GOTO command is undefined.

```
SB0. GOTO SWA A4U0 INVALID SB5 $
.
.
SB5. SET A4U0 TO 3 $
    GOTO SB0 $
```

The results of this example are the same as in the preceding example when the range of values in variable A4U0 is 0 to 3; however, if the value exceeds 3, control is transferred to the statement named SB5.

Note

The relationship between an invalid specification and a special condition in an indexed branch phrase is not the same as in an item branch phrase.

6.1.1.5 Item Branch Phrase

Syntax

<item branch phrase>
 ::= GOTO <item label switch name> [<invalid specification>]
 [<special condition>]

Semantics

An item branch phrase specifies the next statement to be executed, depending on the value of a switch selector.

- GOTO - A language keyword indicating a branch phrase, an indexed branch phrase, or an item branch phrase.
- <item label switch name> - The name of an item label switch that specifies the possible statements to which control will be transferred, and the switch selector whose value controls the transfer.
- <invalid specification> - Optional. Specification of the statement to be executed next if the value of the switch selector is not one of the switch values.
- <special condition> - Optional. A specification that execution of the branch phrase depends on a special hardware or software condition.

The value of the switch selector corresponding to the named item label switch determines the next statement to be executed. If the value of the switch selector at the time the item branch phrase is executed is equal to one of the switch values, the statement named in the item label switch point containing that switch value is executed next.

If the value of the switch selector at the time the item branch phrase is executed is not equal to any of the switch values and the invalid specification is present, the abnormal branch statement is executed next.

If the value of the switch selector at the time the item branch phrase is executed is not equal to any of the switch values and the invalid specification is not present, the statement following the item branch phrase is executed next.

If the optional special condition is present, the execution of the item branch phrase depends on operator action.

If the special condition KEY1, KEY2, or KEY3 is present, the corresponding console key is tested and the selected statement is executed next if that key is on; if it is off, the statement following the item branch phrase is executed next. If the compiler option MONITOR has been specified, the simulated console keys provided as a monitor feature (refer to manual M-5050) are tested. If MONITOR has not been specified, the actual target machine CPU console keys are tested. (Refer to manual M-5048 for a detailed discussion of the CPU console keys and stop indicator lights.)

The special conditions STOP, STOP5, STOP6, and STOP7 are valid only when the compiler option MONITOR has not been specified and the phrase appears in a system procedure element specified to execute in the executive state. The STOP special condition causes an unconditional CPU 4-stop prior to execution of the selected statement. The STOP5, STOP6, and STOP7 special conditions cause a CPU 5-stop, 6-stop, or 7-stop, respectively, prior to execution of the selected statement if the corresponding key is on. In all cases of a CPU stop, the selected statement is executed next upon normal CPU restart.

If both an invalid specification and a special condition are present, the action required by the special condition is performed before the testing for the invalid condition. Therefore, in this case, if the value of the switch selector is not one of the switch values, it is possible that the abnormal branch statement will not be executed next. (If KEY1, KEY2, or KEY3 is specified and the corresponding console key is on, the statement following the item branch phrase will be executed next.)

Examples

```
VRBL FINISH H 4 $
SWITCH SWOFF(FINISH) $
  H(END),ELEMENT $
  H(STOP),UNCOND $
  H(TERM),DONE $
END-SWITCH SWOFF $

GOTO SWOFF $
  SET FINISH TO H(BOMB) $
  RETURN $
DONE. SET FINISH TO H( ) $
  RETURN $
UNCOND. SET FINISH TO H(****) $
  RETURN $
ELEMENT. SET FINISH TO H(1234) $
```

Program control is transferred to the statement label (in switch SWOFF) which corresponds to the value of variable FINISH. That is, for example, control is transferred to the statement labeled UNCOND if the value of variable FINISH is STOP. If the contents do not equal END, STOP, or TERM, control passes to the next sequential statement: SET FINISH TO H(BOMB) \$.

Note

The relationship between an invalid specification and a special condition in an item branch phrase is not the same as in an indexed branch phrase.

| 6.1.1.6 Procedure Call Phrase

Syntax

| <procedure call phrase>
| ::= <user procedure call phrase>
| ::= <supplied procedure call phrase>

Semantics

| A procedure call phrase specifies the execution of a procedure.

| CMS-2Y(7) supports two classes of procedures: user procedures, which are declared with procedure declarations and defined in procedure blocks, and supplied procedures, which are specified as part of a compiler as a convenience to users.

6.1.1.6.1 User Procedure Call Phrase (Parameter Passage Style)

Syntax

```

<user procedure call phrase>
    ::= <procedure name> [<actual procedure parameters>]

<actual procedure parameters>
    ::= <actual i/o parameters> [EXIT <actual exit parameter>@]

<actual i/o parameters>
    ::= [INPUT <actual input parameter>@]
       [OUTPUT <actual output parameter>@]

<actual output parameter>
    ::= [<receptacle>]

<actual exit parameter>
    ::= <statement name>
    
```

Semantics

A user procedure call phrase specifies the execution of a user-defined procedure. It also optionally specifies the values to be supplied as inputs to the procedure, receptacles to receive the outputs of the procedure, and the names of statements that could be executed immediately after execution of the procedure.

- <procedure name> - The name of the procedure to be executed.
- INPUT - Optional. A language keyword indicating that one or more formal input parameters were declared.
- <actual input parameter> - Optional. An expression whose value will be the value of a formal input parameter at the beginning of execution of the subprogram body.
- OUTPUT - Optional. A language keyword indicating that one or more formal output parameters were declared.

<actual output parameter> - Optional. A receptacle that will receive the value of a formal output parameter at the end of execution of the procedure body.

EXIT - Optional. A language keyword indicating that one or more formal exit parameters were declared.

<actual exit parameter> - The name of a statement to be executed after execution of the procedure.

The execution of a user procedure call phrase comprises the following steps:

- a. The value of each actual input parameter is assigned to the corresponding formal input parameter.
- b. The body of the procedure is executed. The execution of the procedure body is terminated by the execution of a procedure return phrase or the execution of an end-procedure declaration.
- c. If execution of the procedure was terminated by executing an end-procedure declaration or a procedure return phrase without a formal exit parameter, each actual output parameter assumes the value of the corresponding formal output parameter.

The correspondence between formal and actual input parameters and the semantics of omitting an actual input parameter are the same as in a user function reference.

The first actual output parameter corresponds to the first formal output parameter, the second actual output parameter corresponds to the second formal output parameter, etc.

The effect of an actual output parameter assuming the value of the corresponding formal output parameter is the same as if the formal parameter were assigned to the actual. Each formal output parameter must be assignment-compatible with its corresponding actual output parameter.

If an actual output parameter is omitted in a procedure call phrase, the value of the corresponding formal output parameter is

not assumed by any receptacle at the end of execution of the procedure body. Omitting an actual output parameter implies that the value of the corresponding formal output parameter is irrelevant for that procedure call.

The first actual exit parameter corresponds to the first formal exit parameter, the second actual exit parameter corresponds to the second formal exit parameter, etc. Actual exit parameters may not be omitted.

The names of all formal input and output parameters must be known in the scope containing the procedure call phrase.

If execution of a procedure is terminated by executing a procedure return phrase specifying a formal exit parameter, the values of the actual output parameters are undefined, and the next statement to be executed is the statement whose name is the actual exit parameter corresponding to the specified formal exit parameter.

If the procedure being called is declared to have formal input parameters, the keyword INPUT and the appropriate number of commas must appear, even if all of the actual input parameters are omitted. If the procedure being called is declared to have formal output parameters, the keyword OUTPUT and the appropriate number of commas must appear, even if all of the actual output parameters are omitted.

Examples

```
VRBL XDOT A 16 S 0 $
VRBL YDOT A 14 S 0 $
VRBL SPEED A 30 S 10 $
VRBL COURSE I 9 U $
VRBL SPD1 A 32 S 10 $
VRBL CS1 I 9 U $
```

```
PROCEDURE MOTION INPUT XDOT, YDOT OUTPUT SPEED, COURSE $
```

```
END-PROC MOTION $
```

```
MOTION INPUT 0,2.83E3 OUTPUT SPD1,CS1 $
```

The constant 0 is assigned to variable XDOT; the constant 2.83E3 is assigned to variable YDOT. Procedure MOTION is then called and upon return, the values in variables SPEED and COURSE are assigned respectively to variables SPD1 and CS1.


```
MOTION INPUT 5,7 OUTPUT SPEED, COURSE $
MOTION INPUT 5,7 OUTPUT , $
```

These two calls to procedure MOTION produce identical results: 5 and 7 are assigned respectively to variables XDOT and YDOT, and the values in variables SPEED and COURSE do not change status between the end of procedure MOTION and the statement following the call to procedure MOTION.

```
MOTION INPUT , OUTPUT SPD1, CS1 $
MOTION INPUT XDOT, YDOT OUTPUT SPD1, CS1 $
```

In the first call the values of the formal input parameters XDOT and YDOT are unchanged when procedure MOTION receives control. It operates identically to the second call.

```
PROCEDURE ALPHA EXIT KHI, PSI, OMEGA $
  IF XDOT GT 0
  THEN
    BEGIN $
      IF YDOT LT 0
      THEN
        RETURN OMEGA $
      ELSE
        RETURN PSI $
    END $
  ELSE
    RETURN KHI $
  .
  .
  RETURN $
END-PROC ALPHA $
PROCEDURE BETA $
```

```
  .
  .
  ALPHA EXIT RHO, SIGMA, TAU $
  SET SPEED TO 0 $
  RETURN $
RHO.  SET SPEED TO 5 $
      RETURN $
TAU.  SET SPEED TO 10 $
      RETURN $
SIGMA. SET SPEED TO -3 $
      END-PROC BETA $
```

In this example variable SPEED will be set to zero if the return from procedure ALPHA is normal. Otherwise the exit logic will cause control to transfer respectively from RETURN KHI, RETURN

PSI, or RETURN OMEGA to RHO, SIGMA, or TAU, depending on the path taken as a result of the conditional statements.

Implementation Note

When a procedure is defined with a parameter output of A0 and an abnormal exit is defined, the output parameter is destroyed when the normal exit is taken. This is because a Replace Add instruction is done on A7 to implement the exiting path, and this modifies the contents of A0 which had been previously set.

6.1.1.6.2 Supplied Procedure Call Phrase

Syntax

```
<supplied procedure call phrase>  
 ::= VECTORP INPUT <abscissa>, <ordinate> OUTPUT  
    [<new magnitude>], [<new angle>]  
 ::= VECTORHP INPUT <abscissa>, <ordinate> OUTPUT  
    [<new magnitude>], [<new angle>]  
 ::= ROTATEP INPUT <abscissa>, <ordinate>, <rotation> OUTPUT  
    [<new abscissa>], [<new ordinate>]  
 ::= ROTATEHP INPUT <abscissa>, <ordinate>, <rotation> OUTPUT  
    [<new abscissa>], [<new ordinate>]
```

```
<new magnitude>  
 ::= <receptacle>
```

```
<new angle>  
 ::= <receptacle>
```

```
<rotation>  
 ::= <numeric expression>
```

```
<new abscissa>  
 ::= <receptacle>
```

```
<new ordinate>  
 ::= <receptacle>
```

Semantics

The supplied procedure call phrases are used to convert between various plane coordinate systems.

- | | |
|----------|--|
| VECTORP | - A predefined identifier indicating that a conversion from the cartesian coordinate system to the vector (polar) system is to be performed. |
| VECTORHP | - A predefined identifier indicating that a conversion from the cartesian coordinate system to the hyperbolic coordinate system is to be performed. |
| ROTATEP | - A predefined identifier indicating that a conversion from one cartesian coordinate system to another by rotation through a specified angle is to be performed. |
| ROTATEHP | - A predefined identifier indicating that a conversion from one cartesian coordinate |

system to another by a hyperbolic rotation through a specified angle is to be performed.

- <abscissa> - A numeric expression whose value is the x-coordinate of a point in the cartesian plane.
- <ordinate> - A numeric expression whose value is the y-coordinate of a point in the cartesian plane.
- <new magnitude> - Optional. A receptacle to receive the distance between the point (x,y) and the origin (0,0).
- <new angle> - Optional. A receptacle to receive the polar angle of the point (x,y).
- <rotation> - A numeric expression whose value is the angle through which the plane is to be rotated.
- <new abscissa> - Optional. A receptacle to receive the x-coordinate of a point after rotation.
- <new ordinate> - Optional. A receptacle to receive the y-coordinate of a point after rotation.

The execution of a supplied procedure call phrase comprises the following steps:

- a. The value of each actual input parameter is assigned to the corresponding formal input parameter.
- b. The body of the procedure is executed.
- c. The value of each formal output parameter is assigned to the corresponding actual output parameter, if that actual output parameter has been specified.

For VECTORP and VECTORHP, <new angle> represents an angle measured from the positive y-axis to the vector in a clockwise direction.

For the rotation procedures (ROTATEP and ROTATEHP), a positive value of <rotation> denotes a clockwise rotation.

In the cartesian cases (VECTORP and ROTATEP), <new angle> and <rotation> represent angles measured in the BAMS system. In the hyperbolic cases (VECTORHP and ROTATEHP), they represent hyperbolic angles measured on the unit hyperbola.

Every actual input parameter (<abscissa>, <ordinate>, <rotation>) must be present in a supplied procedure call phrase. The omission of an actual output parameter (<new angle>, <new magnitude>, <new abscissa>, <new ordinate>) means that the value is unwanted for that call; the corresponding output value will be lost.

The type of the anonymous formal parameters of these procedures corresponding to <abscissa> and <ordinate> depends on the types of the actual parameter expression. If either actual parameter is of a floating-point type, then the type of the formal parameters is A 32 S 15. If both actual parameters are of fixed-point types, then the formal parameters are of type A 29 S \underline{x} , where \underline{x} depends on the actual parameters. First, the two actual parameters are aligned as for an addition or subtraction operation. If the CMS-2Y scaling rules are in effect (paragraph 5.3.1.3), then \underline{x} is the number of fractional bits of the aligned actual parameters. If the MSCALE scaling rules are in effect (paragraph 5.3.1.6), the value of \underline{x} is determined as follows: after this alignment, let \underline{f} denote the common number of fractional bits and let \underline{m} denote the larger of their magnitude bits values. If $\underline{m} \leq 29$, then $\underline{x} = \underline{f}$. If $\underline{m} > 29$, then $\underline{x} = \underline{f} - (\underline{m} - 29)$ and $\underline{m} = 29$; that is, the values of the actual parameters are shifted right until the larger has 29 magnitude bits and their number of fractional bits is adjusted appropriately.

The type of the anonymous formal parameters corresponding to <new magnitude>, <new abscissa>, and <new ordinate> is A 32 S \underline{x} if the CMS-2Y scaling rules are in effect, or A $\underline{m}+2$ S \underline{x} if the MSCALE scaling rules are in effect, where the values of \underline{m} and \underline{x} are as described above.

The type of the anonymous formal parameters corresponding to <new angle> is A 32 U 32 for VECTORP and ROTATEP and A 32 S 31 for VECTORHP and ROTATEHP. The type of the anonymous formal parameters corresponding to <rotation> is A 32 S 31.

Examples

```

VRBL A32S10  A  32  S  10  $
VRBL A32S11  A  32  S  11  $
VRBL A32S12  A  32  S  12  $
VRBL A32S14  A  32  S  13  $
VRBL A32S15  A  32  S  15  $

```

```

VECTORP INPUT  A32S10,A32S11
          OUTPUT A32S14,A32S15 $

```

In this example values are computed to describe the point with abscissa of A32S10 and ordinate of A32S11 in polar coordinates, with the resulting magnitude stored in A32S14 and the angle stored in A32S15.

```

VECTORHP INPUT  A32S10,A32S11
          OUTPUT A32S14,          $

```

In this example values are computed to describe the point with abscissa of A32S10 and ordinate of A32S11 in hyperbolic coordinates, with the resulting magnitude stored in A32S14 and the hyperbolic angle not stored.

```

ROTATEP INPUT  A32S10,A32S11,A32S12
          OUTPUT          ,A32S15  $

```

In this example values are computed to describe the result of a rotation of the point with abscissa A32S10 and ordinate of A32S11 through an angle of A32S12 BAMS. The new abscissa is not saved and the new ordinate is stored in A32S15.

```

ROTATEHP INPUT  A32S10,A32S11,A32S12
          OUTPUT A32S14,A32S15  $

```

In this example values are computed to describe the result of a hyperbolic rotation of the point with abscissa A32S10 and ordinate of A32S11 through the hyperbolic angle of A32S12. The new abscissa is stored in A32S14 and the new ordinate is stored in A32S15.

6.1.1.7 Indexed Procedure Call Phrase

Syntax

```
<indexed procedure call phrase>  
 ::= <indexed procedure switch name> USING <switch index>  
    [<invalid specification>] [<actual i/o parameters>]
```

Semantics

An indexed procedure call phrase specifies the execution of one of a set of procedures, depending on the value of an index expression.

- <indexed procedure switch name> - The name of an indexed procedure switch that specifies the possible procedures to be executed.
- USING - A language keyword indicating that the switch index expression follows.
- <switch index> - A numeric expression whose value in conjunction with the indexed procedure switch declaration specifies the procedure to be executed.
- <invalid specification> - Optional. Specification of a statement to be executed next if the value of the index is out of range.
- <actual i/o parameters> - Optional. Specification of the actual input parameters and the actual output parameters to be used in the procedure call.

The switch index expression yields an integer index value in the same manner as in an indexed branch phrase.

- | In the following, let k denote the index value and let n denote the number of procedure switch points in the declaration of the indexed procedure switch.
- | If k is in the interval $[0, n-1]$, the procedure named in the k th procedure switch point is called.

If k is not in the interval $[0, n-1]$ and the invalid specification is present, the abnormal branch statement is executed next.

If k is not in the interval $[0, n-1]$ and the invalid specification is not present, the effect of executing the indexed procedure call phrase is undefined.

The sequence of events in the execution of an indexed procedure call phrase is the following:

- a. The switch index expression is evaluated and the index value is obtained.
- b. If the invalid specification is present, the index value is tested and the abnormal branch is taken if the value is out of range.
- c. The procedure corresponding to the index value is called.

Thus, if the abnormal branch is taken, the formal input parameters will not have assumed the values of the actual input parameters. (The assumption of values by the formal input parameters is part of the procedure call.)

The names of all formal parameters must be known in the scope containing the indexed procedure call phrase.

If the declaration of the indexed procedure switch specifies formal input parameters, the keyword INPUT and the appropriate number of commas must appear, even if all the actual input parameters are omitted. If the declaration of the indexed procedure switch specifies formal output parameters, the keyword OUTPUT and the appropriate number of commas must appear, even if all the actual output parameters are omitted.

Examples

```
SYS-INDEX 5 L $
VRBL CARD H 7 $
VRBL ARNG I 13 U $
VRBL IMAGE H 7 $
VRBL ORDER I 13 U $
P-SWITCH JCCARD INPUT CARD OUTPUT ARNG $
  PROC1 $
  PROC2 $
  PROC3 $
END-SWITCH JCCARD $
```


/(U) CM2Y-MAN-PGR-M5049-R04C0

JCCARD USING L INVALID STEP-OUT
INPUT IMAGE OUTPUT ORDER \$

STEP-OUT. RETURN \$

Program control is transferred to the procedure which has a position within the P-SWITCH declaration that equals the value contained in system index L, within the range 0 to 2. If the value exceeds 2, control is transferred to the statement named STEP-OUT. Each of the procedures identified in the switch has variables CARD and ARNG as its respective formal input and output parameters.

JCCARD USING L INPUT OUTPUT \$

Program control is transferred in the same manner as above except that no actual input and output parameters are used, and that if the value of L exceeds 2, the result of executing this statement is undefined.

6.1.1.8 Item Procedure Call Phrase

Syntax

```
<item procedure call phrase>  
  ::= <item procedure switch name> [<invalid specification>]  
    [<actual i/o parameters>]
```

Semantics

An item procedure call phrase specifies the execution of one of a set of procedures, depending on the value of a switch selector.

- <item procedure switch name> - The name of an item procedure switch that specifies the possible procedures to be executed.
- <invalid specification> - Optional. Specification of a statement to be executed next if the value of the switch selector is not one of the switch values.
- <actual i/o parameters> - Optional. Specification of the actual input parameters and actual output parameters to be used in the procedure call.

The value of the switch selector corresponding to the named item procedure switch determines the procedure to be called. If the value of the switch selector at the time the item procedure call phrase is executed is equal to one of the switch values, the procedure named in the item procedure switch point containing that switch value is called. If the value of the switch selector at the time the item procedure call phrase is executed is not equal to any of the switch values, no procedure is called and the values of the actual input parameters are not assigned to the formal input parameters. In this case, if no invalid specification is present the statement following the item procedure call phrase is executed next. If an invalid specification is present, the abnormal branch statement is executed next.

The names of all formal parameters must be known in the scope containing the item procedure call phrase.

If the declaration of the item procedure switch specifies formal input parameters, the keyword INPUT and the appropriate number of commas must appear, even if all the actual input parameters are omitted. If the declaration of the item procedure switch

specifies formal output parameters, the keyword OUTPUT and the appropriate number of commas must appear, even if all the actual output parameters are omitted.

Examples

```
VRBL DISTANCE A 64 S 8 P 0 $
VRBL XDOT A 16 S 0 $
VRBL YDOT A 14 S 0 $
VRBL (XX,YY,ZZ) A 10 S 3 $
VRBL QUADRANT S 'N', 'E', 'W', 'S', 'F' $
(LOCREF) PROCEDURE NORTH INPUT XDOT, YDOT OUTPUT DISTANCE $
(LOCREF) PROCEDURE EAST INPUT XDOT, YDOT OUTPUT DISTANCE $
(LOCREF) PROCEDURE WEST INPUT XDOT, YDOT OUTPUT DISTANCE $
(LOCREF) PROCEDURE SOUTH INPUT XDOT, YDOT OUTPUT DISTANCE $
(LOCREF) PROCEDURE FOLD INPUT XDOT, YDOT OUTPUT DISTANCE $
P-SWITCH COMPASS (QUADRANT) INPUT XDOT,
  YDOT OUTPUT DISTANCE $
  'N', NORTH $
  'E', EAST $
  'W', WEST $
  'S', SOUTH $
  'F', FOLD $
END-SWITCH COMPASS $
.
.
PROCEDURE FULANO $
.
.
COMPASS INVALID GOOF INPUT XX, YY OUTPUT ZZ $
.
.
GOOF. FOLD INPUT XX, YY OUTPUT ZZ $
END-PROC FULANO $
```

In this example, one of five procedures will be called depending on the value of status variable QUADRANT. If the value in the variable is not N, E, W, S, or F, control is transferred to the statement labeled GOOF.

6.1.1.9 Stop Phrase

Syntax

<stop phrase>
 ::= STOP [<stop condition>]

<stop condition>
 ::= KEY1
 ::= KEY2
 ::= KEY3
 ::= STOP5
 ::= STOP6
 ::= STOP7

Semantics

A stop phrase specifies suspension of execution of the CMS-2Y program. The suspension of execution may be made conditional on the setting of console switches.

STOP - A language keyword indicating a stop phrase.

<stop condition> - Optional. A specification that execution of the stop phrase depends on the setting of a console switch.

A stop phrase is valid only when the compiler option MONITOR has not been specified and the phrase appears in a system procedure element specified to execute in the executive state (paragraph 9.3.3).

Execution of a stop phrase consisting of only the keyword STOP results in a target machine CPU 4-stop. (Refer to manual M-5048 for a detailed discussion of the CPU console keys and stop indicator lights.)

If a stop phrase contains one of the stop conditions KEY1, KEY2, or KEY3, the corresponding target machine CPU console key is tested. If it is on, a CPU 4-stop results; if it is off, no stop occurs and the statement following the stop phrase is executed.

If a stop phrase contains one of the stop conditions STOP5, STOP6, or STOP7, the corresponding console key is tested. If it is on, a CPU 5-stop, 6-stop, or 7-stop, respectively, results. If it is off, no stop occurs and the statement following the stop phrase is executed.

In all cases of a CPU stop, the statement following the stop phrase is executed next upon normal CPU restart.

Examples

STOP \$

This statement results in an unconditional program stop. The program will continue if it is restarted from the CPU console.

STOP KEY1 \$

This statement results in a program stop if console key number 1 is on. Program execution will proceed with the next statement if it is restarted from the CPU console.

6.1.1.10 Return Phrase

Syntax

<return phrase>
 ::= <procedure return phrase>
 ::= <function return phrase>

<procedure return phrase>
 ::= RETURN [<formal exit parameter>] [<special condition>]

<function return phrase>
 ::= RETURN (<function value>)

<function value>
 ::= <expression>

Semantics

A return phrase specifies the end of execution of a subprogram body. When used in a function subprogram, it also specifies the value of the function reference.

- | | |
|-------------------------|--|
| RETURN | - A language keyword indicating a return phrase. |
| <formal exit parameter> | - Optional. Specification of a statement in the calling subprogram that is to be executed next. |
| <special condition> | - Optional. A specification that execution of the return phrase depends on a special hardware or software condition. |
| <function value> | - An expression whose value is the value of a function reference. |

A procedure return phrase may only appear in a procedure body. A function return phrase may only appear in a function body.

Execution of a return phrase terminates execution of a subprogram body. After execution of a function return phrase, the function value is made available for the evaluation of the expression in which the function reference that initiated execution of the function body appears. After execution of a procedure return phrase that does not specify a formal exit parameter, the values of the formal output parameters, if any, are assumed by the corresponding actual output parameters and the next statement to be executed is the statement following the procedure call phrase, indexed procedure call phrase, or item procedure call phrase that

initiated execution of the procedure body. After execution of a procedure return phrase that specifies a formal exit parameter, the next statement to be executed is the statement whose name is the corresponding actual exit parameter; the values of the actual output parameters, if any, are undefined in this case.

If the optional special condition is present, the execution of the return phrase depends on operator action.

If the special condition KEY1, KEY2, or KEY3 is present, the return phrase is a conditional return phrase, dependent on the setting of the corresponding console key. The console key is tested and the return phrase is executed as described above if that key is on; if it is off, the statement following the return phrase is executed next. If the compiler option MONITOR has been specified, the simulated console keys provided as a monitor feature (refer to manual M-5050) are tested. If MONITOR has not been specified, the actual target machine CPU console keys are tested. (Refer to manual M-5048 for a detailed discussion of the CPU console keys and stop indicator lights.)

The special conditions STOP, STOP5, STOP6, and STOP7 are valid only when the compiler option MONITOR has not been specified and the phrase appears in a system procedure element specified to execute in the executive state (paragraph 9.3.3). The STOP special condition causes an unconditional CPU 4-stop prior to execution of the return phrase. The STOP5, STOP6, and STOP7 special conditions cause a CPU 5-stop, 6-stop, or 7-stop, respectively, prior to execution of the return phrase if the corresponding key is on. In all cases of a CPU stop, the return phrase is executed as described above upon normal CPU restart.

It is not necessary for a procedure body to contain any procedure return phrases. The end-procedure declaration can serve as a surrogate for a procedure return phrase.

A function body must contain at least one function return phrase. The end-function declaration cannot serve as a surrogate for a function return phrase.

The type of the function value expression must be assignment-compatible (paragraph 6.1.1.1) with the type of the function. The value of a function reference is the value of the function value expression that appears on the return phrase that terminates execution of the function body, converted to the type of the function according to the rules for conversion during assignment.

If a function is declared to have character type, then each function value expression in the function body must either be a constant or have the same length as the declared function type.

A return phrase is not permitted in an executive procedure block.

Examples

```
VRBL (XX,YY,ZZ) A 10 S 3 $  
VRBL PARAM B $  
  
FUNCTION F1(XX,YY,ZZ) A 5 U 0 $  
:  
:  
:  
RETURN (XX+YY+ZZ) $
```

In this RETURN phrase, the expression XX+YY+ZZ will be evaluated and returned to the calling expression.

```
FUNCTION F2 (PARAM) H 4 $  
:  
:  
:  
RETURN (H(ER1A) ) $  
:  
:  
:  
RETURN (H(ER2) ) $
```

Depending on which RETURN phrase is executed, the value "ER1A" or "ER2" will be returned to the calling expression. Since function F2 allows four characters, the value that is returned is "ER2 ", the same as if it were coded as H(ER2)). Note that the space between the double right parentheses in both return statements is required so that the character string is terminated.

6.1.1.11 Exit Phrase

Syntax

```
<exit phrase>  
    ::= EXIT [<block name>]
```

Semantics

An exit phrase specifies the end of execution of a loop block.

EXIT - A language keyword indicating an exit phrase.

<block name> - Optional. The name of the loop block whose execution is to cease.

An exit phrase must appear in a loop block body. If the optional block name is present, it must be the name of a loop block and the exit phrase must appear in the body of that loop.

If the optional block name is not present, execution of the innermost loop block in which the exit phrase appears ceases. If the optional block name is present, execution of the named loop ceases. The next statement to be executed is the statement following the loop block whose execution ceased.

Examples

```
VARY VB1 FROM 0 THRU 10 $  
  SET VB2 TO VB2**2 $  
  IF VB2 GT 100  
  THEN  
    BEGIN $  
      SET VRB1 TO VB2 $  
      EXIT $ 'EXIT THE VARY VB1 LOOP'  
    END $  
END $
```

In this example the loop will execute from 1 to 11 iterations. Should the value of VB2 become greater than 100 then the EXIT phrase will be executed and the loop processing will terminate.

```

VARY VB1 FROM 0 THRU 10 $
  VARY VB2 FROM 0 THRU 10 $
    SET VB3 TO VB3**2 $
    IF VB3 GT 100
      THEN
        EXIT 'EXIT THE VARY VB2 LOOP' $
      END $
    IF VB4 GT 1000
      THEN
        EXIT 'EXIT THE VARY VB1 LOOP' $
      END $
    END $
  END $
END $

```

In this example the inner loop will execute from 1 to 11 iterations for each iteration of the outer loop. The outer loop will execute from 1 to 11 iterations. If VB3 becomes greater than 100 then the inner loop will terminate its execution and the outer loop will continue its iterations. If VB4 becomes greater than 1000 then the second EXIT phrase will be executed and the outer loop will cease its iterations.

```

LOOP1. VARY VB1 FROM 0 THRU 10 $
LOOP2. VARY VB2 FROM 0 THRU 10 $
        SET VB3 TO VB3**2 $
        IF VB3 GT 100
          THEN
            EXIT LOOP1 'EXIT LOOP1 - VARY VB1' $
          END LOOP2 $
        IF VB4 GT 1000
          THEN
            EXIT 'EXIT LOOP1 - VARY VB1' $
          END LOOP1 $

```

This example has the same structure as the previous example with the addition of labels on the VARY statements and labels on the EXIT phrases. Should either EXIT phrase be executed the outer loop will cease its iterations since the outer loop name is the name on the EXIT phrase of the inner loop.

6.1.1.12 Resume Phrase

Syntax

```
<resume phrase>  
 ::= RESUME [<block name>]
```

Semantics

A resume phrase specifies that the next iteration of a loop block or a find statement is to be performed.

RESUME - A language keyword indicating a resume phrase.

<block name> - Optional. The name of the loop block or find statement whose next iteration is to be performed.

Execution of a resume phrase causes the end-of-loop processing (paragraphs 6.1.2.2 and 6.2.2) of the specified loop or find statement to be executed.

The optional block name must be the block name of a loop block or a find statement.

If the block name is omitted, the resume phrase must be in at least one loop body. In this case end-of-loop processing for the innermost loop body in which the phrase appears is being specified.

If the block name is present and is the name of a loop block, the resume phrase must follow the loop block head of that loop. If the block name is present and is the name of a find statement, the resume phrase must follow the find clause of that find statement.

Examples

```
VRBL SEND I 5 U $  
VRBL PLAN I 7 U $  
TABLE TAB1 V MEDIUM 20 $  
  FIELD KIND I 10 U $  
END-TABLE TAB1 $
```


6.1.1.13 Executive Call Phrase

Syntax

<executive call phrase>
 ::= EXEC <executive function> [, <executive input
 parameter>]

<executive function>
 ::= <numeric constant expression>

<executive input parameter>
 ::= <simple expression>

Semantics

An executive call phrase specifies that execution of the CMS-2Y program is to be suspended and control is to be transferred to the target machine's executive program.

EXEC - A language keyword indicating an executive call phrase.

<executive function> - A numeric constant expression that is a parameter of the executive call.

<executive input parameter> - Optional. A second parameter of the executive call.

The value of the executive function expression must be integer in the range [0,65535]; i.e., 16 bits or less.

If the optional executive input parameter is present, the value of the expression is placed in register A0. The value of the parameter must be expressed in 32 bits or less.

Examples

EXEC 15 \$

This executive call phrase produces an enter executive state instruction which includes the value 15 in the lower half-word (i.e., generates an XS 017 instruction).

EXEC 15, CORAD(VRBLX) \$

This example provides the address of VRBLX to the executive program in task register A0.

6.1.1.14 Shift Phrase

Syntax

<shift phrase>
 ::= SHIFT <shift source> <shift type> [-] <shift amount>
 [<shift assign clause>]

<shift source>
 ::= <single-valued data unit>

<shift type>
 ::= CIRC
 ::= ALG
 ::= LOG

<shift amount>
 ::= <numeric expression>

<shift assign clause>
 ::= INTO <receptacle>

Semantics

A shift phrase specifies a machine-dependent shift operation on the bits that make up the value of a simple single-valued data unit.

- SHIFT - A language keyword denoting a shift phrase.
- <shift source> - A single-valued data unit whose bit pattern is to be shifted.
- <shift type> - One of the language keywords CIRC, ALG, or LOG, denoting the type of target machine shift to be performed.
- <shift amount> - A numeric expression whose value specifies the number of bit positions to be shifted.
- INTO - Optional. A language keyword indicating that a receptacle to receive the shifted value follows.
- <receptacle> - Optional. A receptacle to receive the shifted value.

If the optional shift assign clause is not present, the shifted bit pattern is assigned to the single-valued data unit that is the shift source.

The shift type CIRC specifies a circular shift, in which vacated bit positions on one end are filled with the bits that were shifted off the other end. The shift type ALG specifies an algebraic shift, in which vacated bit positions are filled with the sign bit. The shift type LOG specifies a logical shift, in which vacated bit positions are filled with zeros.

If the shift amount expression is of integer type, its value is the number of bit positions to be shifted. If it is not of integer type, its value is converted to integer, and the converted value is the number of bit positions to be shifted. The number of bit positions to be shifted cannot be negative.

If the shift amount expression is preceded by a minus sign, the shift is to the left; otherwise, it is to the right.

The shifted bit pattern has universal type. If the optional shift assign clause is present, the specified receptacle may be of any type and the shifted bit pattern is assigned to the bits of the receptacle without regard to its type.

The shift phrase is highly machine-dependent. That dependency is reflected in the following specifications:

- a. The length of the shift source cannot exceed 64 bits.
- b. If the shift type is CIRC, the length of the shift source must be either 32 or 64 bits.
- c. All left shifts are performed using the target machine circular shift instructions. Left algebraic shift operations can result in filling on the right with magnitude bits that differ from the sign bit. Left logical shift operations can result in filling on the right with nonzero magnitude bits.
- d. If the number of bit positions to be shifted is greater than 64, the result of execution of a shift phrase is undefined.

Examples

```
SHIFT INT LOG -2 $
```

The contents of INT are shifted left logically by two bit positions. The two leftmost bits of INT will be truncated and two zero bits will be added on the right.

SHIFT INT ALG 2 \$

The contents of INT are shifted right algebraically by two bit positions. Two bits on the right will be truncated and two bit positions on the left will be sign-filled.

VRBL MYDATA H 4 P H(ABCD) \$

SHIFT MYDATA CIRC 8 \$

Given the character type variable MYDATA preset as indicated, MYDATA will contain the character string DABC after execution of the shift phrase.

VRBL MYDATA H 4 P H(ABCD) \$

VRBL NEWSPOT H 1 \$

SHIFT MYDATA CIRC -8 INTO NEWSPOT \$

After execution of the shift phrase, MYDATA will still retain the character order ABCD, and NEWSPOT will contain the character B.

6.1.1.15 Open Phrase

Syntax

```
<open phrase>
  ::= OPEN <file name> <i/o capability>

<i/o capability>
  ::= INPUT
  ::= OUTPUT
  ::= SCRATCH
```

Semantics

An open phrase specifies that a file is to be opened for input or output. This prepares it for subsequent I/O operations; an open phrase must be the first operation performed on a user-defined file.

- OPEN - A language keyword identifying an open phrase.
- <file name> - The name of a user-defined file.
- INPUT - A language keyword specifying a file that can be read but not written upon.
- OUTPUT - A language keyword specifying a file that can be written upon but not read.
- SCRATCH - A language keyword specifying a file that can be both read from and written upon.

Opening a magnetic tape file causes the tape to be rewound.

It is an error to attempt to open a file which is already open. To change the I/O capability of an open file, the file must first be closed and then opened with the new capability specified.

Standard files are always open and have an I/O capability appropriate to the particular device: READ has the INPUT capability, PUNCH and PRINT have the OUTPUT capability, and OCM has the SCRATCH capability.

All user-defined files are closed at the beginning of execution of a program.

Examples

OPEN LBR INPUT \$

This statement causes the file with the name LBR to be opened, and specifies its use as input only.

6.1.1.16 Close Phrase

Syntax

```
<close phrase>  
 ::= CLOSE <file name>
```

Semantics

A close phrase is used to close user-defined files. No I/O operations can be performed on closed files except to open them.

CLOSE - A language keyword identifying a close phrase.

<file name> - The name of a user-defined file.

Closing a magnetic tape file causes the tape to be rewound.

Examples

```
CLOSE OUT $
```

This example causes the file named OUT to be closed.

Note

Standard files can not be closed.

Implementation Note

All user-defined files should be closed before program execution is terminated. This is particularly true of output files that, at any moment, usually have data that have been moved from the program data areas to the buffer but have not yet been written to the file itself. Part of the closing operation is the writing of such data, which would be lost if the program execution were terminated with the file open.

6.1.1.17 Endfile Phrase

Syntax

```
<endfile phrase>  
 ::= ENDFILE <file name>
```

Semantics

The endfile phrase specifies that an end-of-file mark is to be written on a magnetic tape file.

ENDFILE - A language keyword identifying an endfile phrase.

<file name> - The name of a user-defined file.

After an end-of-file mark is written, the subfile count of the file is incremented by 1 and its record count is set to 0. The file must have been opened with either the output or scratch capability.

The number of end-of-file marks in a file is limited only by the length of the tape.

Example

```
ENDFILE BAR $
```

An end-of-file mark will be written on the magnetic tape which has a file name of BAR.

6.1.1.18 Define Label Phrase

Syntax

```
<define label phrase>  
 ::= DEFID <file name> <label definition>
```

```
<label definition>  
 ::= STANDARD  
 ::= (<character>&)
```

Semantics

A define label phrase creates a header record on a file.

DEFID - A language keyword identifying a define label phrase.

<file name> - The name of a user-defined file.

STANDARD - A language keyword specifying a standard tape label.

If the STANDARD label is used, the name of the file, padded with trailing blanks, forms the record. Otherwise the specified character string, padded with trailing blanks if necessary, forms the record. All of the characters between the parentheses, including leading and trailing blanks, are part of the record. The maximum length of the character string is 120 characters. As with character constants, a right parenthesis is denoted by two consecutive right parentheses.

Examples

```
DEFID LPR STANDARD $
```

The header "LPR" (the name of the file) will be written on the device referenced in the file declaration LPR.

```
DEFID LPR (INVENTORY SDIEGO 1 JULY 81) $
```

The header "INVENTORY SDIEGO 1 JULY 81" will be written on the device referenced in the file declaration LPR.

6.1.1.19 Check Label Phrase

Syntax

```
<check label phrase>  
 ::= CHECKID <file name> <label definition>
```

Semantics

A check label phrase verifies the header record on a file.

- CHECKID - A language keyword identifying the check label phrase.
- <file name> - The name of a user-defined file.
- <label definition> - Specification of the label to be compared against the label on the file.

The check label phrase causes the verification of the header record on a file. If the label definition is STANDARD, then the file name padded with blanks is expected to be the header record; otherwise the specified character string, padded with trailing blanks as necessary, is expected to be the header record. If the header record does not have the expected content, the message WRONG TAPE MOUNTED followed by a MOUNT TAPE message is displayed to the operator. If the operator chooses to mount another tape and continue execution, the header record on the new tape will be checked.

If a check label phrase is executed, it must be the first operation on the file after it is opened. The file must be opened with the input capability.

Examples

```
OPEN TAB INPUT $  
CHECKID TAB STANDARD $
```

The content of the header record will be compared to the expected label "TAB".

6.1.1.20 File Positioning Phrase

Syntax

```
<file positioning phrase>  
    ::= SET FIL (<file name>) TO <numeric expression>
```

Semantics

A file positioning phrase specifies that a file be positioned at the beginning of a specific subfile.

- | | |
|----------------------|---|
| SET FIL | - A language keyword identifying the file positioning phrase. |
| <file name> | - The name of a user-defined file. |
| TO | - A language keyword separating the file name from the subfile specification. |
| <numeric expression> | - An expression whose value specifies file positioning. |

The value of the numeric expression specifies the subfile. The value must be an integer or the result of the execution is unpredictable. The record count is always set to zero as a result of executing a file positioning phrase.

If the value of the numeric expression is negative or zero, the file is positioned at record 0 of file 0. The effect is equivalent to a rewind, except that if the file is declared with the WITHLBL option, the header record is skipped. However, the special form

```
SET FIL (<name>) TO -0
```

has a conventional meaning: the file is closed.

If the value of the numeric expression is greater than the number of any subfile in the file, the message OUTSIDE TAPE PHYSICAL FILE is output and execution is terminated.

The file positioning phrase can be used with files opened for input, output, or scratch.

Examples

```
FILE MTF3 B 300 R 120 MT13 $  
VRBL FILPOS I 15 U $  
:  
:  
OPEN MTF3 INPUT $  
:  
:  
ALPHA. SET FIL(MTF3) TO FILPOS $  
:  
:  
BETA. SET FIL(MTF3) TO 0 $  
:  
:  
GAMMA. SET FIL(MTF3) TO -0 $
```

Execution of statement ALPHA causes the hardware device MT13 to be positioned to the subfile which corresponds to the value of FILPOS. Execution of statement BETA causes the hardware device MT13 to be set to the beginning, file position 0, record position 0. Execution of statement GAMMA closes the file.

6.1.1.21 Record Positioning Phrase

Syntax

<record positioning phrase>
 ::= SET POS (<file name>) TO <numeric expression>

Semantics

A record positioning phrase specifies that a file be positioned at the beginning of a specific physical record in the subfile within which it is currently positioned.

- | | |
|----------------------|--|
| SET POS | - A language keyword identifying the record positioning phrase. |
| <file name> | - The name of a user-defined file. |
| TO | - A language keyword separating the file name from the record specification. |
| <numeric expression> | - An expression whose value specifies file positioning. |

The value of the numeric expression must be an integer or the result of the execution is undefined. The file count is not changed as a result of executing a record positioning phrase.

If the value of the numeric expression is greater than any record number in the subfile, the message OUTSIDE FILE BOUNDARY is output and execution is terminated.

The record position phrase can be used with files opened for input, output, or scratch.

Examples

```
SET POS(MAGFILE1) TO 3 $
```

The file MAGFILE1 is positioned at the beginning of the fourth record (record number 3) of the current subfile.

```
SET POS(FNAME) TO CURRPOS-1 $
```

The file FNAME is positioned at the beginning of the record with a number that is 1 less than the value of CURRPOS. If CURRPOS contains the number of the record prior to execution, the effect is that of a backspace.

Note

The record positioning phrase should not be used with stream files, because of the lack of any simple relationship between the physical records affected by the phrase and the logical records used in the program.

6.1.1.22 Output Phrase

Syntax

<output phrase>
 ::= OUTPUT <output file name> [<output list>] [<format name>]

<output file name>
 ::= <file name>
 ::= PRINT
 ::= PUNCH
 ::= OCM

<output list>
 ::= <output item>
 ::= (<output item>@)

<output item>
 ::= <i/o data unit>
 ::= <extended subscript data unit>
 ::= <numeric constant expression>
 ::= <character constant>

<i/o data unit>
 ::= <single-valued data unit>
 ::= <multivalued data unit>
 ::= <word data unit>

<extended subscript data unit>
 ::= <extended structured variable data unit>
 ::= <extended table data unit>

<extended structured variable data unit>
 ::= <variable name> (<extended field>@)

<extended field>
 ::= <field name>
 ::= <word specification>

<extended table data unit>
 ::= <table name> (<extended subscript> [, <extended field>@])

<extended subscript>
 ::= <subscript expression>@
 ::= (<subscript expression>@) ... (<subscript expression>@)

Semantics

An output phrase specifies the transfer of data to a file that is open for output or scratch.

- OUTPUT - A language keyword indicating an output phrase.
- <output file name> - The name of a file to receive the transferred data.
- <output list> - Optional. A list of data units or constants to be output to the file.
- <format name> - Optional. A format declaration that controls conversion during execution of the output phrase.
- <extended subscript data unit> - A specification of selective portions of multiword data units.

The file name may be either user-defined or a standard output file.

The data transfer is in the order indicated by the output list, reading from left to right. The output list can be omitted if a format is specified, in which case the data are transmitted from the format itself, usually by character constant format descriptors.

If the optional format name is omitted, the output is unformatted; if it is present, the output is formatted.

If a data unit in an output list specifies a simple single-valued data unit (e.g., a variable, a major index, or a field of a table item), the value of that datum is output. If a table or untyped structured data unit is specified in an unformatted output phrase, the totality of values making up the data unit is output.

6.1.1.22.1 Extended Subscript Data Unit

An extended subscript data unit is a shorthand notation that can be used in input lists and output lists to specify multiple fields or words of a structured variable, multiple consecutive items of a table, or multiple fields or words of multiple consecutive items of a table. This is purely a notational convenience; the effect of writing an extended subscript data unit is identical to the effect of writing a list of the individual data units.

Multiple fields or words of a structured variable are indicated by the name of the variable followed by the names of the fields or word indexes, separated by commas and enclosed in parentheses.

/(U) CM2Y-MAN-PGR-M5049-R04C0

thus, if ITAR is a structured variable at least three words long and FLD1 and FLD2 are the names of two of its fields, writing

ITAR(FLD1, 2, FLD2)

is the same as writing

ITAR(FLD1), ITAR(2), ITAR(FLD2)

A similar notation can be used for fields of an item of a table. The name of the table is followed in parentheses by the item index and a list of field names or word indexes, all separated by commas. Thus if TAB is a table having an item structure similar to ITAR, above, writing

TAB(I-1, FLD1,2, FLD2)

is the same as writing

TAB(I-1, FLD1), TAB(I-1, 2), TAB(I-1, FLD2)

Multiple consecutive items of a table are indicated by the name of the table followed in parentheses by the beginning item index in parentheses, three consecutive periods, and the ending subscript expression in parentheses. Thus

TAB((I+1) ... (I+3))

is the same as writing

TAB(I+1), TAB(I+2), TAB(I+3)

The value of the beginning subscript expression must be less than the value of the ending subscript expression.

For arrays the effect is as if the subscripts were varying from the beginning subscript expression to the ending subscript expression, with the first subscript varying most rapidly, the second subscript varying next most rapidly, etc. Each subscript increases until it reaches its upper limit, after which it begins again at its initial value and the next subscript is incremented. Thus, if ARY is an array with dimensions 3 and 4, writing

ARY ((1,1) ... (2,2))

is the same as writing

ARY(1,1), ARY(2,1), ARY(0,2), ARY(1,2), ARY(2,2)

Multiple fields or words of multiple consecutive items of a table are indicated by a combination of the above notations. The table name is followed by a list enclosed in parentheses, with the entries of the list separated by commas. The first entry in the list is the extended subscript in parentheses. The remaining entries in the list are the field names or word indexes. The effect is as if the fields and words were listed together for each of the indicated items. Thus

```
TAB((I+1) ... (I+3), FLD1,2)
```

is the same as writing

```
TAB(I+1,FLD1), TAB(I+1,2), TAB(I+2,FLD1), TAB(I+2,2),  
TAB(I+3,FLD1), TAB(I+3,2)
```

6.1.1.22.2 The Format Scan

In describing the effect of the format statement in input and output, it is convenient to classify format descriptors as constant descriptors or variable descriptors. The X and T format descriptors, the character constant format descriptor, and the slash are constant descriptors -- their effects are always the same, independent of any data values. The I, O, F, E, A, and L format descriptors, on the other hand, are variable descriptors -- their general effects can be described but their precise effects vary with the value of the data being converted.

For the remainder of this discussion, the list of format items in a format declaration should be considered as a list of non-repeated format declarations; that is, any list which has repeats should be considered as expanded into the equivalent form without repeats. Similarly, an output list should be considered as a simple list of values and an input list as a simple list of data units, expanded into the equivalent form without any extended subscript data units if necessary.

At the beginning of execution of a formatted input phrase or a formatted output phrase, a left-to-right scan of the format descriptors begins. Any initial constant descriptors are processed in order until the first variable descriptor is encountered. This descriptor is used to convert the first entry in the input list or output list. Following this conversion, any succeeding constant descriptors are processed in order until the second variable descriptor is encountered, which then controls the conversion of the second entry in the input list and output list. This procedure -- scanning and processing constant descriptors until a variable descriptor is encountered to control the conversion of the next entry in the list -- continues until either a variable descriptor is encountered and all entries in the input list or

output list have been processed, or until the end of the format list is reached. In the former case execution of the input phrase or output phrase terminates; the remaining format descriptors in the list are not used. Execution also terminates when the end of the format list is reached, provided all entries in the input list or output list have been processed. If unprocessed entries remain, however, the scan of the format list starts again at the left. This rescanning process is used as many times as necessary, until one of the termination conditions described above occurs.

While the format scan is in progress, records of the file are also being processed. This processing can be described by means of a pointer which always points to the left of the next character string to be created on output or converted on input. At the beginning of execution of an input phrase or output phrase this pointer points to the first position (position 0) in the record. An X-type format descriptor causes the pointer to be moved to the right the specified number of positions, and a T-type format descriptor causes the pointer to be moved to the specified position. A character constant format descriptor on output causes the constant to be placed in the record at the position of the pointer and the pointer to be moved to the right of the constant in the record; on input, the movement of the pointer is the same but the constant is not input from the record. A slash causes the physical record to be output or a new record to be input, and the pointer to be positioned at the first position of the next record. The variable format descriptors all specify a width, which is the number of characters in the string to be created or converted. At the end of each of these conversions, the pointer is positioned to the right of the character string.

During the processing of a record, the pointer can never be positioned beyond the end of the record. On output files with variable length records, the last position of the pointer determines the size of the record.

When a data unit or value is matched with a variable format descriptor during a format scan, the properties of the data unit or value are not verified to be valid for the conversion specified by the format descriptor. The conversion is carried out mechanically; it is the responsibility of the programmer to insure that the matchup is valid. If the data unit is a table or untyped structured data unit, an indeterminate number of bits at the beginning of the data unit are used, not the entire data unit.

6.1.1.22.3 Output to the Printer

Printer output differs from all other formatted output because of header and control character capabilities.

Headers, or header lines, are lines which, if present, are automatically printed at the top of each page. The CMS-2 system supports three header lines, a major header and two minor headers, which are printed in the following order: the major header line, a blank line, and the two minor header lines. No headers are present at the beginning of execution of a program. Headers can be established and cancelled as many times as desired during execution of the program.

Control characters are certain characters that may appear in the first position of a record. The first character of a record is never printed, but has the effect of a blank if it is not a control character. The control characters are summarized in the following list:

<u>Control Character</u>	<u>Effect</u>
Blank	Single space and print the line.
0(zero)	Double space and print the line.
-	Triple space and print the line.
1	Eject to the top of a new page, print any headers, and print the new line.
H	Cancel all headers, eject to the top of a new page, and print the line.
A	Cancel all headers, eject to the top of a new page, print the line, and save the line as the major header.
B	Cancel the minor headers, double space, print the line, and save the line as the first minor header.
C	Cancel the second minor header, single space, print the line, and save the line as the second minor header.
Z	Cancel all headers, space one line, and print.

6.1.1.22.4 Record Size with Unformatted Input and Output

If a file specified with nonzero record size is used with an unformatted input phrase, there must be sufficient data in the record to be input to the data units of the input list. Any excess data in the record is lost. The only exception is when the last data unit in the input list is a multiword data unit and there is sufficient data to partly fill it but not enough to completely fill it. In this case the remainder of the data unit is filled with binary zeros if the file type is binary or blanks if the file type is character.

If a file with zero record size is used, each item in an output list creates a new record and each item in an input list uses a new record. On input, the actual record size has the effect described in the previous paragraph.

Examples

```
TABLE TAB H NONE 100 MITAB $
  FIELD FLD1 A 12 S 2 $
  FIELD FLD2 I 6 U $
  ITEM-AREA ITAR $
END-TABLE TAB $
VRBL VBL A 9 S 1 $
FORMAT FMT I1, I6.1, I7.3, I2 $
.
.
ALPHA. OUTPUT FNAME (MITAB, TAB, VBL) $
.
.
BETA. OUTPUT PRINT (0,VBL, ITAR(FLD1), TAB(0,FLD2))
      FMT $
```

The execution of statement ALPHA will cause the value of the major index MITAB of the table TAB, all the values of the table TAB, and the value of the variable VBL to be output to the file FNAME. The execution of statement BETA will cause a zero, the value of the variable VBL, the value of field FLD1 of the item-area ITAR, and the value of field FLD2 of the first item of the table TAB to be printed, with the conversion specified by the format FMT.

Notes

A table or untyped structured data unit should not be specified in a formatted output phrase.

/(U) CM2Y-MAN-PGR-M5049-R04C0

A print line is limited to a maximum of 120 ASCII characters, including control characters.

6.1.1.23 Input Phrase

Syntax

```
<input phrase>
  ::= INPUT <input file name> <input list> [<format name>]

<input file name>
  ::= <file name>
  ::= READ
  ::= OCM

<input list>
  ::= <input item>
  ::= (<input item>@)

<input item>
  ::= <i/o data unit>
  ::= <extended subscript data unit>
```

Semantics

An input phrase specifies the transfer of data from a file that has been opened for input or scratch.

- INPUT - A language keyword identifying the input phrase.
- <input file name> - The name of a file containing the data to be transferred.
- <input list> - A list of data units to which the input data will be transferred.
- <format name> - Optional. A format declaration that controls conversion during execution of the input phrase.

The file name may be either user-defined or be a standard input file.

The data transfer is in the order indicated by the input list, reading from left to right. One effect of this order of transfer is that a value input to a datum can affect a data unit which appears later in the list. (See the example below.)

If the optional format name is omitted, the input is unformatted; if it is present, the input is formatted.

If a data unit in an input list specifies a single-valued data unit (e.g., a variable, a major index, a field of a table item) a

single value is transmitted to that data unit. If a table or untyped structured data unit is specified in an unformatted input phrase, values are transmitted to the entire structure. A table or structured data unit that is not simple should not be specified in a formatted input phrase.

In unformatted input, the properties of a data unit receiving a value must be identical to the properties of the data unit whose value was originally output in order for the result of the input operation to be predictable. For example, the result is unpredictable if the value of a variable or field declared as A 6 S 1 is output and later input into a variable or field declared as A 9 S 3. Similarly, the result is unpredictable if the values of a table are output and later input into a table of identical logical structure if the tables have different compiler packing.

Examples

```
TABLE TAB H NONE 100 MITAB $
  FIELD FLD1 I 6 U $
  FIELD FLD2 I 6 U $
  ITEM-AREA ITAR $
END-TABLE TAB $
VRBL VBL I 7 U $
FORMAT FMT I2, 1X, I7.3, 1X, I2 $
```

```
ALPHA. INPUT FNAME (MITAB, TAB, VBL) $
```

```
BETA. INPUT READ (VBL, ITAR(FLD1), TAB(VBL,FLD2))
      FMT $
```

Execution of statement ALPHA causes the value of the major index MITAB of the table TAB, all the values of the table TAB, and the value of the variable VBL to be input from the file FNAME. The number of values input to TAB will depend on the value input to MITAB. Execution of statement BETA causes the value of the variable VBL, the value of the field FLD1 of the item-area ITAR, and the value of field FLD2 of item VBL of table TAB to be input from the card reader, with the conversion specified by the format FMT. The particular item of TAB whose FLD2 receives a value is determined by the value input to VBL.

6.1.1.24 Encode Phrase

Syntax

```
<encode phrase>  
 ::= ENCODE <pseudo buffer> <output list> <format name>  
  
<pseudo buffer>  
 ::= <data unit>
```

Semantics

An encode phrase specifies the conversion of data internally from the target machine internal form to character string form.

- ENCODE - A language keyword indicating an encode phrase.
- <pseudo buffer> - A data unit that will contain the character string form of the data after conversion.
- <output list> - A list of data units whose values are to be converted.
- <format name> - The name of a format declaration that controls the conversion process.

The execution of an encode phrase is identical to that of an output phrase, except that the converted character strings are simply placed in the pseudo buffer, rather than being placed in an actual buffer and transmitted to an output device.

During execution of an encode phrase the pseudo buffer acts as a single logical record; therefore the virgule (/) format descriptor cannot be used in the format specified in an encode phrase.

Examples

```
VRBL OBTAIN I 9 S P -2 $  
VRBL STEP H 5 P H(CAT) $  
VRBL CAN F P 2.8 $  
FORMAT RMC I3,L2,F6.2 $  
VRBL TAN H 14 P H() $  
  
ENCODE TAN(OBTAIN,STEP,CAN) RMC $
```

/(U) CM2Y-MAN-PGR-M5049-R04C0

Execution of this encode phrase converts the values of OBTAIN, STEP, and CAN to character form and places the characters into variable TAN.

6.1.1.25 Decode Phrase

Syntax

<decode phrase>
 ::= DECODE <pseudo buffer> <input list> <format name>

Semantics

A decode phrase specifies the conversion of data internally from character string form to the target machine internal form.

- DECODE - A language keyword indicating a decode phrase.
- <pseudo buffer> - A data unit that contains the character strings to be converted.
- <input list> - A list of data units that will contain the converted values.
- <format name> - The name of a format declaration that controls the conversion process.

The execution of a decode phrase is identical to that of an input phrase, except that the character strings to be converted are taken from the pseudo buffer, rather than from an actual buffer after being transmitted from an input device.

During execution of a decode phrase the pseudo buffer acts as a single logical record; therefore the virgule (/) format descriptor cannot be used in the format specified in a decode phrase.

Examples

```
VRBL PASTE I 7 U $
VRBL SOOT I 5 U $
VRBL KNOT H 20 $
VRBL JADE I 6 U $
FORMAT HFC A12,I7.3,I2 $
```

```
DECODE KNOT(SOOT,PASTE,JADE)HFC $
```

Execution of the decode phrase converts the characters in variable KNOT to their internal form and places them in SOOT, PASTE, and JADE.

6.1.1.26 Convertin Phrase

Syntax

<convertin phrase>
 ::= CONVERTIN <input buffer> <inputlist> <stringform
 specification>

<input buffer>
 ::= <single-valued data unit>
 ::= <table name>

<stringform specification>
 ::= <stringform name>
 ::= [*] <single-valued data unit>

Semantics

A convertin phrase specifies the conversion of character strings into internal values.

- | | |
|----------------------------|--|
| CONVERTIN | - A language keyword indicating a convertin phrase. |
| <input buffer> | - The name of the data unit containing the character string. |
| <inputlist> | - The specification of a list of receptacles into which the results of the character string conversion will be placed. |
| <stringform specification> | - The specification of a stringform governing the conversion into internal values. |

If the input buffer is a single-valued data unit, it must be of a character type.

If the stringform specification consists of a stringform name, the named stringform is the specified stringform. If the stringform specification consists of a single-valued data unit, that data unit must be of a character type. The value of the data unit at the time the convertin phrase is executed must be a character string that is a valid stringform list, subject to the constraints specified below; the value of the data unit is then the stringform that controls the conversion process. If the stringform specification consists of a single-valued data unit preceded by an asterisk, the data unit must be of an integer type. The

value of the data unit at the time the convertin phrase is executed must be the address of a stringform name; that stringform then controls the conversion process.

There must be at least as many conversion specifiers in the specified stringform list as there are specified receptacles.

Execution of a convertin phrase comprises three steps:

- a. The conversion cursor is set to the first character position (position 0) of the input buffer. If the input buffer is a table name, this is the first character position of the first word of the table. The first receptacle of the specified receptacles list is made the next receptacle to be processed, and the first stringform item of the specified stringform is made the next stringform item to be processed.
- b. Stringform items are processed from left to right until a conversion specifier or the end of the stringform list is encountered. If the end of the stringform list is encountered, execution of the convertin phrase has been completed.
- c. If there is not another receptacle in the specified receptacles list, execution of the convertin phrase has been completed. If there is another receptacle, the current character string is converted as specified by the conversion specifier, the converted value is assigned to the receptacle, and step b is performed next.

Execution of a convertin phrase is undefined if any receptacle specified by the inputlist shares memory with the input buffer, the stringform specification, or the specified stringform.

6.1.1.26.1 Run-Time Stringforms

When the stringform of a convertin phrase or a convertout phrase is the value of a single-valued data unit, that value must be in the form of a valid stringform list, except for the following constraints:

- a. All repeat value, field width, fraction size, exponent size, and position expressions must be decimal integer constants.
- b. Blanks are treated as null characters. (Embedded blanks in constants are permitted.)

- c. The end of the stringform list is denoted by the character "\$".

Implementation Notes

The convertin phrase is not yet implemented.

6.1.1.27 Convertout Phrase

Syntax

<convertout phrase>
 ::= CONVERTOUT <output buffer> <outputlist> <stringform
 specification>

<output buffer>
 ::= [*] <single-valued data unit>
 ::= <table name>

Semantics

A convertout phrase specifies the conversion of internal values into character strings. The converted character strings are placed in the output buffer.

- | | |
|----------------------------|--|
| CONVERTOUT | - A language keyword indicating a convertout phrase. |
| <output buffer> | - The name of the data unit in which the character string is to be built. |
| <outputlist> | - The specification of a list of values to be converted into a character string. |
| <stringform specification> | - The specification of a stringform governing the conversion from internal values. |

If the output buffer is a single-valued data unit, it must be of a character type.

If the stringform specification consists of a stringform name, the named stringform is the specified stringform. If the stringform specification consists of a single-valued data unit, the data unit must be of a character type. The value of the data unit at the time the convertout phrase is executed must be a character string that is a valid stringform list, subject to the constraints specified in the semantics section of the convertin phrase; the value of the data unit is then the stringform that controls the conversion process. If the stringform specification consists of a single-valued data unit preceded by an asterisk, the data unit must be of an integer type. The value of the data unit at the time the convertout phrase is executed must be the address of a stringform name; that stringform then controls the conversion process.

There must be at least as many conversion specifiers in the specified stringform list as there are specified values to be converted.

Execution of a convertout phrase comprises three steps:

- a. The conversion cursor is set to the first character position (position 0) of the output buffer. If the output buffer is a table name this is the first character position of the first word of the table. The first value of the values list is made the next value to be processed, and the first stringform item of the specified stringform is made the next stringform item to be processed.
- b. Stringform items are processed from left to right until a conversion specifier or the end of the stringform list is encountered. If the end of the stringform list is encountered, execution of the convertout phrase has been completed.
- c. If there is not another value to be converted, execution of the convertout phrase has been completed. If there is another value to be converted, it is converted as directed by the conversion specifier, and step b is performed next.

Execution of a convertout phrase is undefined if the output buffer shares memory with any data unit referenced in any expression that specifies a value to be converted, the stringform specification, or the specified stringform.

Implementation Notes

The convertout phrase is not yet implemented.

6.1.1.28 Display Phrase

Syntax

```
<display phrase>
  ::= DISPLAY <display item>@

<display item>
  ::= <data unit> [<preset magnitude>]
  ::= REGS
```

Semantics

A display phrase specifies that the contents of certain data units are to be printed on the system output device.

- DISPLAY - A language keyword indicating a display phrase.
- <data unit> - The name of a data unit to be printed.
- <preset magnitude> - Optional. Specification of nonstandard magnitudes to be assigned to the bits of the data unit for the display print-out.
- REGS - A language keyword specifying a display of both the A and B registers.

The display for each data unit consists of the description of the data unit (a duplication of the data item in the display phrase) and the value(s) of the data unit. If only one value is displayed for a data unit, it is displayed to the right of the data unit description. If more than one value is displayed, the first is displayed to the right of the data unit description and the rest are displayed on the following lines, one to a line. If the display phrase is preceded by a statement name, the statement name is displayed on the line preceding the display of the data unit.

If the data unit is simple, the display of its value is consistent with its type. If it is not simple (which includes multivalued data units), each word of the data unit is displayed as an 11-digit octal number, in the order in which the words of the data unit are allocated in the target machine memory. Each A and B register is also displayed as an 11-digit octal number.

A statement name on a display phrase is used only in the display; it is not valid as the destination of a branch phrase.

Examples

DISPLAY M,X,Y \$

Assuming M is a 4-word table, X is a character type variable, and Y is a floating-point type variable, the printout might appear as follows:

```
M 046732115043
  362341023456
  265123245675
  145676343210
X DOG GONE
Y 0.34244632E+07
```

BETA. DISPLAY TABL(ALPHA,FELD) \$

Assuming FELD is a fixed-point type field, the printout might appear as follows:

```
BETA
TABL(ALPHA,FELD) 432.06

TABLE NAV V 2 1 $
  FIELD SPEED I 10 U 0 13 $
END-TABLE NAV $
```

KNOTS. DISPLAY NAV(0,SPEED) V(40,8) \$

Assume that field SPEED had bit assignments as indicated in Figure 6-02. Execution of the display phrase (with the conversion specification that the eighth bit (from the right) of field SPEED represents 40) would result in an output of:

```
KNOTS
NAV(0,SPEED) 6.25
```

where 6.25 is the sum of the bit values for bits 3 and 5.

Field SPEED

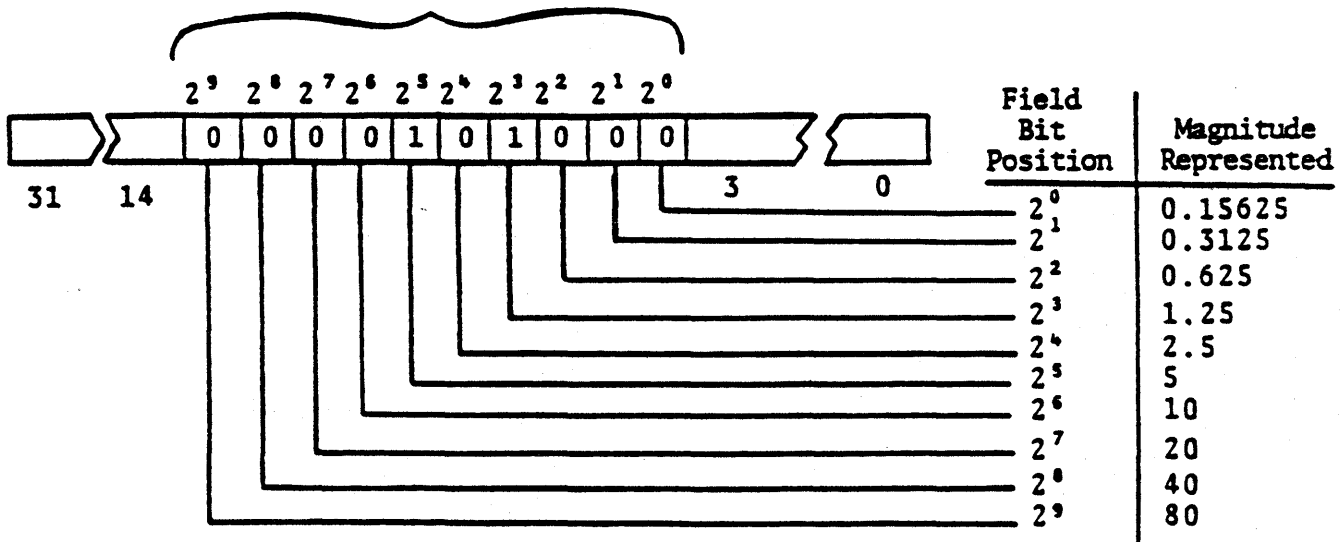


Figure 6-02. Example of Bit Assignments for the Display Phrase

6.1.1.29 Snap Phrase

Syntax

```
<snap phrase>  
 ::= SNAP <data unit> [<preset magnitude>]
```

Semantics

A snap phrase specifies that the contents of a data unit are to be printed on the system output device each time the snap phrase is executed and the value of the data unit is different from its value at the previous execution of the phrase.

- SNAP - A language keyword identifying the snap phrase.
- <data unit> - The name of the data unit to be printed.
- <preset magnitude> - Optional. Specification of nonstandard magnitudes to be assigned to the bits of the data unit for the display print-out.

A display of the contents of the specified data unit is always produced on the first execution of a snap phrase. A display will be produced on subsequent executions only if the contents during that execution differ from the contents at the previous execution.

The form of the display generated by execution of a snap phrase is identical to the display generated by execution of a display phrase.

A statement name on a snap phrase is used only in the display; it is not valid as the destination of a branch phrase.

Examples

```
CHECK. SNAP LENGH $
```

On the first execution and each subsequent execution in which the floating-point type variable LENGH has changed, a display of the following form would be produced:

/(U) CM2Y-MAN-PGR-M5049-R04C0

```
CHECK
LNGTH 0.17496325E-02

VAIF INDEXES(IX) EQ 0
  THEN
END $
```

A sample snap output from executing this vary block follows:

First snap execution:

```
INDEXES      00000000005
              00000000007
              00000000001
              00000000000
              00000000004
              00000000002
```

Fourth snap execution:

```
INDEXES      00000000005
              00000000007
              00000000001
              00000000003
              00000000004
              00000000002
```

Implementation Note

Compilation of a snap phrase causes an area of the target machine memory to be reserved that is the same size as the specified data unit. If the data unit is an adjustable table, the number of items that will be snapped is the compile-time value of the table's ltag, not the load-time value.

6.1.1.30 Trace PhraseSyntax

```
<trace phrase>
  ::= TRACE
```

Semantics

A trace phrase marks the beginning of a sequence of source statements to be traced during execution of the program.

TRACE - A language keyword identifying the trace phrase.

The effect of a trace phrase is on the presentation sequence of the source program, not the execution sequence of the object program. When a trace phrase is encountered during compilation and the trace class has been enabled by a debug enabling declaration, subsequent statements are compiled with additional code to implement tracing to the next end-trace phrase, or to the end of the compilation module in the absence of an end-trace phrase.

When a traced statement is executed, a line is printed on the system output device identifying the statement. The identification is in the form of the name of the most recent statement name (the name of the subprogram if no previous statement name has appeared in the subprogram), followed by a '+', followed by the number of the statement relative to the statement name. The statement corresponding to the statement name has a relative number of zero. The relative number is increased by the compiler each time it encounters a '\$' or the keyword THEN in a simple statement.

Examples

```
      SET XX TO 3 $
      TRACE $
AA1.  SET XX TO XX -2 $
      IF XX LT 0 THEN
          SET YY TO YY+1 THEN
              GOTO BB3 $
          SET YY TO YY+XX $
          GOTO AA1 $
BB3.  PRNTPROC INPUT YY $
      END-TRACE $
```

Execution of the above sequence of statements would cause the following trace printout to be produced:

/(U) CM2Y-MAN-PGR-M5049-R04C0

AA1+0
AA1+1
AA1+4
AA1+5
AA1+0
AA1+1
AA1+2
AA1+3
BB3+0

6.1.1.31 End-Trace Phrase

Syntax

<end-trace phrase>
 ::= END-TRACE

Semantics

An end-trace phrase marks the end of a sequence of source statements to be traced during execution of the program.

END-TRACE - A language keyword identifying the end-trace phrase.

The effect of an end-trace phrase is on the presentation sequence of the source program, not the execution sequence of the object program. When an end-trace phrase is encountered during compilation, subsequent statements will be compiled without the additional code needed to implement tracing, down to the next trace phrase or the end of the compilation module in the absence of a trace phrase.

6.1.1.32 Null Phrase

Syntax

```
<null phrase>  
 ::=
```

Semantics

A null phrase specifies that no action is to be performed.

A null phrase contains no characters.

Examples

```
LOC-INDEX Q $  
IF Q EQ 5 THEN $  
ELSE SET Q TO 0 $
```

In this example, if Q equals 5, no action is taken.

Note

A null phrase is a programmer convenience. It enables certain kinds of stubbing, such as omitting the alternative statement of an else clause, the value block body of a case block, or the statement following a statement label.

Implementation Note

The compiler will output a warning message in many cases when it discovers a null phrase. In particular, such a message will be issued for a null phrase that appears as a part of an if statement, because a misplaced \$ could radically change the meaning of a program, causing severe debugging problems.

6.1.2 Statement Blocks

Syntax

```
<statement block>  
  ::= <begin block>  
  ::= <loop block>  
  ::= <case block>
```

Semantics

A statement block consists of a group of statements that together specify a sequence action of a CMS-2Y(7) program.

A statement block may contain another statement block as one of its component statements, which may itself contain another statement block, etc. The amount of nesting possible depends on the type of blocks being nested. Each block requires a certain number of nesting units. A begin block requires three nesting units, a loop block requires five nesting units, and a case block requires four nesting units. Within any nest, a maximum of 150 nesting units can be used; no more than nine loop blocks may be nested within a loop block.

6.1.2.1 Begin Block

Syntax

<begin block>
 ::= <begin block head> [<begin block body>] <end phrase>

<begin block head>
 ::= BEGIN \$

<begin block body>
 ::= <statement>&

Semantics

A begin block specifies a grouping of statements.

- BEGIN - A language keyword indicating the beginning of a block.
- <begin block body> - Optional. The statements grouped by the begin block.
- <end phrase> - A language construct indicating the end of a block.

Execution of a begin block results in the execution of the statements of the begin block body, according to the usual rules governing execution of any group of statements.

Examples

```
VRBL BOOL B $  
  
IF BOOL  
THEN  
  BEGIN $  
    SET XX TO YY $  
    PROC1 $  
  END $  
ELSE  
  BEGIN $  
    SET XX TO ZZ $  
    PROC2 $  
  END $
```

Each begin block in this example combines two simple statements into two statement alternatives of the conditional statement. The first BEGIN block will be executed if BOOL is true; the second BEGIN block will be executed if BOOL is false.

Note

The begin block does not define a scope in CMS-2Y(7), as does the similar construct in many other programming languages. It is merely a syntactic device to permit groups of statements to appear in constructs that call for a single statement.

6.1.2.2 Loop Block

Syntax

```
<loop block>
  ::= <loop block head> [<loop block body>] <end phrase>

<loop block head>
  ::= VARY [<index clause>@] [<top test clause>] [<bottom test
    clause>] $

<index clause>
  ::= <loop index> [<control clause>]

<loop index>
  ::= <single-valued data unit>

<control clause>
  ::= [<initiation clause>] [<termination clause>]
    [<incrementation clause>]
  ::= [<initiation clause>] [<incrementation clause>]
    [<termination clause>]
  ::= [<termination clause>] [<initiation clause>]
    [<incrementation clause>]
  ::= [<termination clause>] [<incrementation clause>]
    [<initiation clause>]
  ::= [<incrementation clause>] [<initiation clause>]
    [<termination clause>]
  ::= [<incrementation clause>] [<termination clause>]
    [<initiation clause>]

<initiation clause>
  ::= FROM <initial value>

<initial value>
  ::= <numeric expression>
  ::= <status expression>

<termination clause>
  ::= THRU <final value>
  ::= WITHIN <table name>

<final value>
  ::= <numeric expression>
  ::= <status expression>

<incrementation clause>
  ::= BY [-] <change value>

<change value>
  ::= <numeric expression>
```

<top test clause>
 ::= WHILE <top test>

<top test>
 ::= <conditional expression>

<bottom test clause>
 ::= UNTIL <bottom test>

<bottom test>
 ::= <conditional expression>

<loop block body>
 ::= <statement>&

Semantics

A loop block specifies a group of statements that are to be repeatedly executed and, optionally, conditions to terminate the repeated execution.

- | | |
|-----------------|---|
| VARY | - A language keyword indicating the beginning of a loop block. |
| <loop index> | - Optional. A single-valued data unit used as an index during execution of the loop body. |
| FROM | - Optional. A language keyword indicating that an initial value for the loop index follows. |
| <initial value> | - Optional. An expression that specifies the initial value of the loop index. |
| THRU | - Optional. A language keyword indicating that a final value for the loop index follows. |
| <final value> | - Optional. An expression that specifies the final value of the loop index. |
| WITHIN | - Optional. A language keyword indicating that a table name limiting the value of the loop index follows. |
| <table name> | - Optional. The name of a table for which the value of the loop index must be a valid subscript. |

- BY - Optional. A language keyword indicating that an increment value for the loop index follows.
- <change value> - Optional. A numeric expression that specifies the amount by which the loop index changes on each iteration.
- WHILE - Optional. A language keyword indicating that a top-of-loop test follows.
- <top test> - Optional. A conditional expression to be evaluated prior to each execution of the loop body.
- UNTIL - Optional. A language keyword indicating that a bottom-of-loop test follows.
- <bottom test> - Optional. A conditional expression to be evaluated after each execution of the loop body.
- <loop block body> - Optional. The group of statements that are to be repeatedly executed.
- <end phrase> - A language construct indicating the end of a block.

| The loop index must be of a numeric or status type.

The execution of a loop block comprises six steps:

- a. If the loop block head contains one or more index clauses, each loop index is set to its initial value. The determination of that initial value depends on the following conditions:
 - (1) An initiation clause is present. The value of the initial value expression is assigned to the index, according to the rules for assignment of the type of the index.
 - (2) No initiation clause is present and no termination clause using the keyword WITHIN is present. For an index of a numeric type, the initial value is 0. For an index of a status type, the initial value is the first value of the type's status constant list (the value of FIRST for the type).
 - (3) No initiation clause is present, a termination clause using the keyword WITHIN is present, and the type of the

loop index is numeric. If an incrementation clause is present and the keyword BY is followed by a minus sign, the initial value is one less than the number of items in the table specified in the termination clause (in other words, it is the index of the last item in that table). Otherwise the initial value is 0.

- (4) No initiation clause is present, a termination clause using the keyword WITHIN is present, and the type of the loop index is status. The initial value is the first value of the status constant list for the type of the subscript of the table specified in the termination clause.
- (5) All other cases. For a loop index of a numeric type, the initial value is 0. For a loop index of a status type, the initial value is the first value of the type's status constant list (the value of FIRST for the type).
 - b. If the loop block head contains a top test clause, the top test expression is evaluated. If its value is false (0), the loop block execution terminates, and the next statement to be executed is the statement following the loop block.
 - c. The statements of the loop body are executed. Execution of the statements of the loop body may terminate execution of the loop block by causing another statement in the subprogram body to be executed; by the execution of a return phrase; or in some machine-dependent manner (using a direct code block).
 - d. If execution of the loop body results in execution of the end phrase and the loop head contains neither a bottom test clause nor an index clause, step (b) of this sequence is performed next.
 - e. If the loop block head contains one or more index clauses, each loop index is modified. For each numeric loop index, if an incrementation clause is present the change value expression is added to or subtracted from the loop index according to the rules for numeric addition or subtraction (paragraph 5.3.1). The value is subtracted if the keyword BY is followed by a minus sign; otherwise, it is added. If an incrementation clause is not present for a numeric loop index, the value of the index is incremented by 1. For a status type loop index, the current value is replaced by its successor in the type's list of status constants (the value of SUCC for the current value). If the current value is the last value in the type's list of status

constants, the value of the loop index becomes undefined.

For each loop index for which a termination clause has been specified, the value of the index after updating is tested to determine if the termination condition has been satisfied (see below). If any loop index satisfies its termination condition, the loop block execution terminates and the next statement to be executed is the statement following the loop block. If no loop index satisfies its termination condition (a loop index for which no termination clause has been specified has no termination condition to satisfy), and the loop block head contains a bottom test clause, step (f) of this sequence is performed next. Otherwise, step (b) is performed next.

- f. If the loop block head contains a bottom test clause, the bottom test expression is evaluated. If its value is true (1), the loop block execution terminates and the next statement to be executed is the statement following the loop block. If its value is false (0), step (b) of this sequence is performed next.

Steps (d), (e), and (f) of this sequence make up end-of-loop processing.

An incrementation clause is permitted only if the associated loop index is of a numeric type.

The value of the change expression is assumed to always be positive. The effect of execution of a loop block is undefined if the value of the change expression is negative.

If a termination clause contains the keyword THRU and the associated loop index is of a numeric type, the termination condition for the corresponding loop index depends on a comparison of the value of the loop index (after updating) and the value of the final value expression, according to the rules for numeric comparison (paragraph 5.3.2.3). If the incrementation clause specifies an increment or is not present, the termination condition is satisfied if the loop index, after incrementation, is greater than the value of the final value expression. If the incrementation clause specifies a decrement, the termination condition is satisfied if the loop index, after decrementation, is less than the value of the final value expression.

If a termination clause contains the keyword THRU and the associated loop index is of a status type, the termination condition is satisfied if the value of the loop index before it was updated was the value of the final value expression. The final value expression is evaluated each time end-of-loop processing occurs.

If a termination clause contains the keyword WITHIN and the associated loop index is of a numeric type, the termination condition for the corresponding loop index is that its value (after updating) is not a valid index for the table named in the termination clause. If the incrementation clause specifies an increment, termination occurs when the loop index value becomes greater than or equal to the number of items in the table; if a decrement is specified, it occurs when the loop index value becomes negative.

If a termination clause contains the keyword WITHIN and the associated loop index is of a status type, the termination condition is satisfied if the value of the loop index before it was updated was the last value of the status constant list for the type of the subscript of the table specified in the termination clause.

The loop body can contain a branch phrase or indexed branch phrase whose execution causes the end phrase of the loop block to be executed next. Execution of the end phrase results in end-of-loop processing.

Figure 6-03 illustrates the control logic options available with the VARY block.

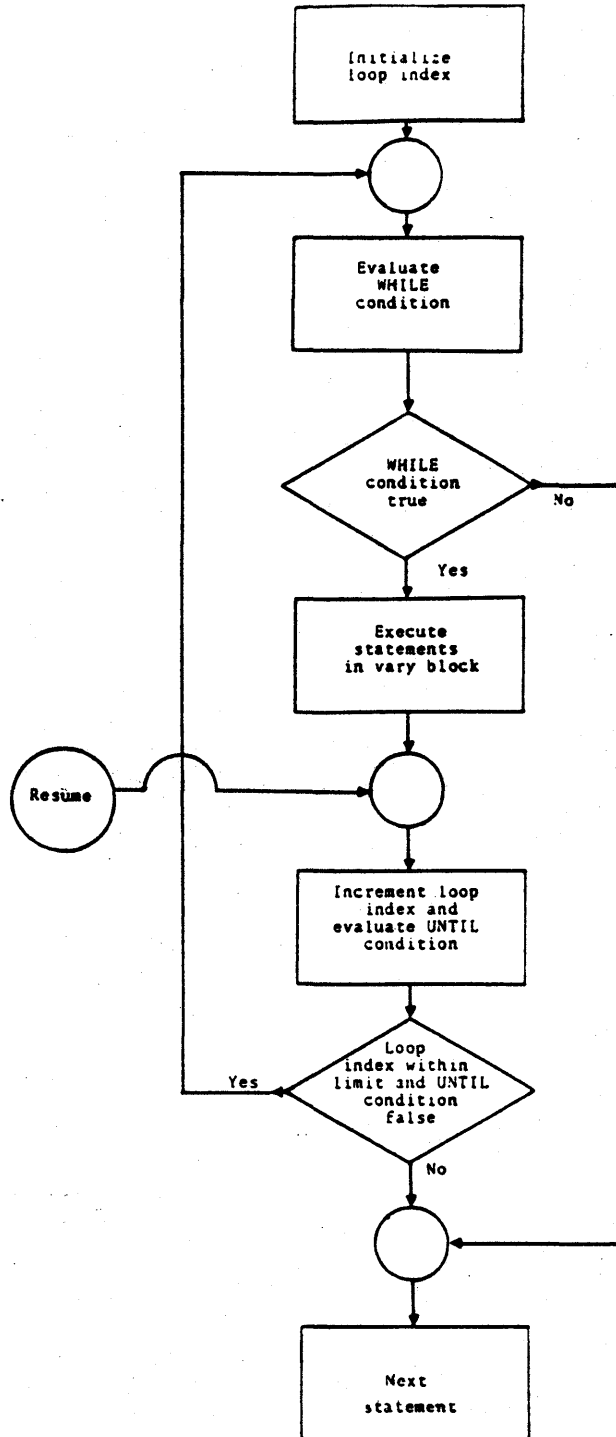


Figure 6-03. Vary Block Control Flow

Examples

```

TABLE PAR H 5 31 $
  FIELD FCX1 A 7 S 4 3 14 $
  FIELD FCX2 A 8 S 4 3 7 $
END-TABLE PAR $
VRBL COUNT I 5 U $
VRBL (XX,YY,ZZ) A 10 S 3 $

VARY COUNT FROM 1 THRU 10 $
SET PAR(COUNT,FCX2) TO 0 $
END $

```

The data unit COUNT is incremented from the value 1 through the value 10 by an implied increment of 1. Field FCX2 of table PAR is cleared for items 1 through 10; items 0 and 11 through 30 remain unchanged.

```

ONE.  VARY XX FROM 31 THRU 0 BY -1
      WHILE COUNT GT 0 $
      SET YY TO COUNT * 4 $
      SET COUNT TO COUNT - 1 $
TWO.  VARY ZZ THRU YY UNTIL ZZ EQ 2 $
      SET PAR(XX,ZZ) TO YY/PAR(XX,FCX1) $
      SET YY TO PAR(XX,FCX2)/2 $
      END TWO $
      END ONE $

```

This example shows VARY block nesting. Block TWO is executed completely after the initialization, and after each subsequent decrement and test of block ONE until variable XX reaches a value less than 0, or until variable COUNT reaches a value of 0. The value of ZZ in block TWO implicitly starts from 0 and is incremented by 1. The block TWO statement will continue to be executed as long as the value of variable ZZ is not 2 or until its value exceeds that of variable YY.

Note

Since the keyword VARY is the only required semantic entry in the loop block head statement, an infinite loop may be created by the simple statement:

```
VARY $
```

The burden of exiting from such a loop lies in the logic of the loop block body.

/(U) CM2Y-MAN-PGR-M5049-R04C0

When a termination clause contains the keyword WITHIN, the number of items in the specified table is determined by the size of the table at load time (if the number of items of the table is specified by a load time constant) and the value of the major index of the table, if any, at the time the termination condition is evaluated.

6.1.2.3 Case Block

Syntax

```

<case block>
    ::= <case block head> <value block>& <end phrase>

<case block head>
    ::= FOR <case selector> [,(<case type>)] [<else clause>] $

<case selector>
    ::= <simple expression>

<case type>
    ::= <simple type>
    ::= <typed structure name>

<else clause>
    ::= ELSE <alternative statement>

<alternative>
    ::= <simple statement>

<value block>
    ::= <value block head> [<value block body>] <end phrase>

<value block head>
    ::= [<statement label>&] BEGIN <case value>@ $

<case value>
    ::= <constant>
    ::= <numeric constant expression>

<value block body>
    ::= <statement>&
    
```

Semantics

A case block specifies a number of blocks of statements of which one is to be executed, depending on the value of an expression.

- | | |
|-----------------|---|
| FOR | - A language keyword indicating the beginning of a case block. |
| <case selector> | - An expression whose value specifies the block of statements to be executed. |
| <case type> | - Optional. Specification of the type of the case selector. |

- ELSE - Optional. A language keyword indicating that an alternative statement follows.
- <alternative statement> - Optional. A statement to be executed if no block of statements is selected.
- BEGIN - A language keyword indicating the beginning of a begin block.
- <case value> - A constant value to be compared to the value of the case selector in determining the block of statements to be executed.
- <value block body> - Optional. A block of statements to be executed if the value of the case selector is one of the case values of the associated value block head.
- <end phrase> - A language construct indicating the end of a block.

When a case block is executed, its case selector expression is evaluated. If the value of the expression is one of the case values of the block, the associated value block body is executed. If the value of the expression is not one of the case values of the block and the optional else clause is present, the alternative statement is executed. If the value of the expression is not one of the case values of the block and the optional else clause is not present, the statement following the case block is executed next.

The optional case type in the case block head can be omitted only if the case selector is a single-valued data unit of a simple type. When the case type is present, the value of the case selector is converted to the case type before it is compared to the case values. The case selector and the specified case type must be assignment-compatible (paragraph 6.1.1.1).

The types of the case values must be such that they could be compared to the case selector expression, after conversion, in a relational expression.

No value may appear as a case value of a single case block more than once.

If execution of the selected value block or the alternative statement does not cause some statement in the subprogram body

not in the case block to be executed and does not cause a return phrase to be executed, the statement following the case block is executed next.

If the statement name of a branch phrase is the name of a value block, execution of the branch phrase has the same effect as selecting the value block during execution of the case block of which it is part.

Examples

```
CASE.  FOR I12S ELSE ERRPROC INPUT I12S $
      BEGIN 2 $
        PROCA $
      END $
      BEGIN 3,5 $
        PROCB $
      END $
      BEGIN 4,6 $
        PROCC $
      END $
    END CASE $
```

This example illustrates how a case block can be used in place of a procedure switch. Procedure PROCA will be called when the data unit I12S has the value 2; PROCB will be called when the value is 3 or 5; PROCC will be called for values 4 or 6; and ERRPROC will be called in all other cases.

```
FOR XI $
  BEGIN 0,7 $
  :
  :
  END $
  BEGIN 4 $
  :
  :
  END $
  BEGIN 1,2,3 $
  :
  :
  END $
END $
```

/(U) CM2Y-MAN-PGR-M5049-R04C0

The appropriate value block will be executed if XI has the value 0,1,2,3,4, or 7. If XI is less than zero, equal to 5 or 6, or greater than 7, no value block will be executed, and the statement following the case block will be executed next.

```
FOR VD(J,H2) $
  BEGIN H(ZA), H(XY) $
  .
  .
  .
  END $
  BEGIN H(AZ), H(YX) $
  .
  .
  .
  END $
  BEGIN H(M ), H( M) $
  .
  .
  .
  END $
END $
```

The 2-character field H2 in the Jth item of table VD will be compared with each of the 2-character constants specified in the value blocks. One of the value blocks will be executed only if the appropriate match is found.

```
FOR STATE3 $
  BEGIN 77 $
  FOR STATE5 ELSE
    FOR SUBSTATE ELSE RETURN $
    BEGIN 2 $
    .
    .
    .
  END $
  BEGIN 7,6,5,4 $
  .
  .
  .
  END $
  END ''SUBSTATE'' $
  BEGIN -1 $
  .
  .
  .
  END $
```

```
BEGIN 0 $
.
.
END $
BEGIN 1 $
.
.
END $
END ''STATE5'' $
VARY WHILE ERCODE NOT 3 $
.
.
END $
END ''77'' $
BEGIN 128 $
.
.
END ''128'' $
END ''STATE3'' $
```

This example illustrates case blocks nested three levels deep in which the second level also includes a vary block. The vary block will be executed whenever STATE3 is 77, ERCODE is not 3, and the RETURN statement is not executed.

6.1.2.4 End Phrase

Syntax

<end phrase>
 ::= END [<block name>]

<block name>
 ::= <statement name>

Semantics

An end phrase denotes the end of a begin block, a loop block, a case block, or a value block.

END - A language keyword indicating the end of a block.

<block name> - Optional. The name of the block being terminated.

The optional block name must appear if the block is named; it may not appear if the block is not named. If the block head has more than one statement label, any of the labels may appear on the end phrase.

The meaning of execution of an end phrase depends on the type of block it terminates:

Begin Block - The statement following the begin block is executed next.

Loop Block - The end-of-loop processing is performed.

Case Block - The statement following the case block is executed next.

Value Block - The statement following the case block of which the value block is part is executed next.

Examples

```
FOR XI $  
  VA. BEGIN 0,7 $  
  .  
  .  
  .  
  END VA $
```

```

VB. BEGIN 5 $
    .
    .
    END VB $
    BEGIN 6 $
    .
    .
    END $
END $

```

This example illustrates required END block names on blocks VA and VB.

```

TYPE FORTYPE A 13 U 3 $
VRBL TYPEFOR A 16 U 6 $

FOR TYPEFOR, (FORTYPE)
  ELSE SET TYPEFOR TO 0 $
  BEGIN O(1.2) $
    SET TYPEFOR TO 12 $
  END $
  BEGIN O(37.5) $
    SET TYPEFOR TO 0 $
  END $
END $ ''FOR''

```

This example illustrates the use of the optional case type, using a type declared in a TYPE declaration. The value contained in the variable TYPEFOR will be converted to A 13 U 3 type before the comparisons are made to the values O(1.2) and O(37.5). If the converted value is not one of these values, the variable TYPEFOR will be set to zero. If a match is made, the variable TYPEFOR will be set to the appropriate value.

6.2 Conditional Statements

Syntax

```
<conditional statement>  
  ::= <if statement>  
  ::= <find statement>
```

Semantics

A conditional statement is a statement whose execution depends on the value of a conditional expression.

6.2.1 If Statement

Syntax

<if statement>
 ::= IF <primary condition> THEN <primary statement> [<elsif clause>]& [<else clause>]

<primary condition>
 ::= <boolean expression>

<primary statement>
 ::= <simple statement>

<elsif clause>
 ::= ELSIF <secondary condition> THEN <secondary statement>

<secondary condition>
 ::= <boolean expression>

<secondary statement>
 ::= <simple statement>

Semantics

An if statement specifies sequences of Boolean conditions and statements, of which one is executed when its associated condition is true.

- | | |
|-----------------------|--|
| IF | - A language keyword indicating an if statement. |
| <primary condition> | - A Boolean expression whose value controls execution of the primary statement. |
| THEN | - A language keyword indicating that a primary or secondary statement follows. |
| <primary statement> | - A simple statement that is executed if the primary condition is true. |
| <ELSIF> | - Optional. A language keyword indicating an elsif clause. |
| <secondary condition> | - Optional. A Boolean expression whose value controls execution of the associated secondary statement. |

<secondary statement> - Optional. A simple statement whose execution is controlled by the associated secondary condition.

<else clause> - Optional. A clause containing an alternative statement to be executed if no primary or secondary condition is true.

When an if statement is executed the sequence of Boolean expressions consisting of the primary condition and any optional secondary conditions is evaluated in order until one is found to be true, at which time the associated primary or secondary statement is executed. If all of the conditions evaluate to false and the optional else clause is present, then the alternative statement of the else clause is executed. If all of the conditions evaluate to false and the optional else clause is not present, then no subsidiary statement of the if statement is executed.

At most one subsidiary statement of the if statement is executed. After a selected statement is executed, no following secondary conditions are evaluated.

Examples

```
VRBL B1 B $
VRBL HOLRTH H 1 $
VRBL (XX,YY,ZZ) I 16 U $

IF B1 THEN RETURN $
```

In this example control is returned to the calling procedure if the value of B1 equals 1 (true).

```
IF HOLRTH NOT H(*) OR XX+YY LT ZZ
  THEN SET B1 TO 0 $
ELSE SET ZZ TO 89 $
```

In this example, variable B1 will be set to zero only if variable HOLRTH does not contain an asterisk or if the sum of XX and YY is less than ZZ. Otherwise, the ELSE statement will be executed.

```
VRBL STAT S 'GOOD', 'BAD DATA', 'BAD HARD', 'BAD PROC' $
FILE FILE1 H 100 R 50 MT3 'BUSY', 'FINISHED',
  'SENTINEL', 'HARDWARE', 'BAD PKG', 'EMPTY', 'NODEVICE' $
```

```
IF FILE1 NOT 'HARDWARE' THEN RETURN S
ELSE SET STAT TO 'BAD HARD' $
```

This is a sample of a conditional i/o expression used in an if statement. If the value of the expression is not 3, the return is executed. Otherwise, STAT is assigned the value 2.

```
IF XX EQ 0
  THEN SET YY TO 0 $
ELSIF XX EQ 500
  THEN SET YY TO 10 $
ELSIF XX GT 99
  THEN SET YY TO 99 $
ELSE SET YY TO -1 $
```

This is an example of an if statement with elseif clauses. YY will be set to 0 only if XX equals 0. YY will be set to 10 only if XX is not 0 and XX is equal to 500. YY will be set to 99 only if XX is not 0 and XX is not 500 and XX is greater than 99. Otherwise YY will be set to -1.

6.2.2 Find Statement

Syntax

```
<find statement>
  ::= <find clause> <action clause> [<else clause>]

<find clause>
  ::= FIND <find condition> [<varying clause>] $

<find condition>
  ::= <find relational expression> [<boolean binary operator>
    <boolean expression>]

<find relational expression>
  ::= <subscripted data unit> <relational operator> <simple
    expression>

<boolean binary operator>
  ::= AND
  ::= OR

<varying clause>
  ::= VARYING <index clause>

<action clause>
  ::= IF DATA FOUND THEN <simple statement>
  ::= IF DATA NOTFOUND THEN <simple statement>
```

Semantics

A find statement specifies a search of a table for an item satisfying a specified condition and one or two statements to be executed, depending on the result of the search.

- | | |
|------------------|--|
| FIND | - A language keyword indicating a find statement. |
| <find condition> | - A Boolean expression specifying a condition on an item of the table that must be met for the search to be satisfied. |
| VARYING | - Optional. A language keyword indicating that an index clause for the table index follows. |
| <index clause> | - Optional. Specification of constraints on the index of the table as the search is performed. |

- IF DATA FOUND THEN - Language keywords indicating that the following statement is to be executed if a table item satisfying the find condition is found.
- IF DATA NOTFOUND THEN - Language keywords indicating that the following statement is to be executed if no table item satisfying the find condition is found.
- <simple statement> - A statement to be executed conditionally, depending on the result of the table search.
- <else clause> - Optional. Specification of an alternative statement to be executed if the result of the table search is such that the simple statement of the action clause is not executed.

A find condition is a special form of a Boolean expression. It must begin with a relational expression, the left comparand of that relational expression must be a subscripted reference, and the first subscript expression of that subscripted reference must consist of a variable name of a numeric or status type. That variable becomes the index of a loop implied by the find clause.

If the optional varying clause is present, it must specify the implied loop index.

The form of the loop implied by a find statement depends on the form of the find clause. In the following, let loop index denote the implied index of the loop, as defined above, and let table denote the table of the subscripted data unit that begins the find relational expression.

1. Find clauses of the form FIND <find condition>, FIND <find condition> VARYING loop index, and FIND <find condition> VARYING loop index WITHIN table imply loops of the form VARY loop index WITHIN table.
2. All other forms of find clauses (which must include a varying clause) imply loops of the form VARY loop index <index clause>.

The form of the loop implied by a find statement must be a valid loop block head.

The semantics of a find condition are the same as those of any Boolean expression.

Execution of a find statement comprises three steps:

- a. The loop index is initialized. The initial value can be explicitly specified in the optional index clause, or it can be implied, as described above.
- b. The find condition is evaluated. If the value of the find condition is 0 (false), step c of this sequence is performed next. If the value of the find condition is 1 (true), the table search is completed and the following action depends on the form of the action clause and the optional else clause.
 - (1) If the action clause contains the keyword FOUND, the simple statement of the action clause is executed, which completes execution of the find statement.
 - (2) If the action clause contains the keyword NOTFOUND and the optional else clause is not present, execution of the find statement is completed without further action.
 - (3) If the action clause contains the keyword NOTFOUND and the optional else clause is present, the alternative statement is executed, which completes execution of the find statement.
- c. End-of-loop processing. The loop index is incremented and the loop termination condition is tested. The type of incrementation and the termination condition can be explicitly specified in the optional index clause, or either one can be implied, as described above. If the termination condition is not satisfied, step b of this sequence is performed next. If the termination condition is satisfied, the following actions depends on the form of the action clause and the optional else clause.
 - (1) If the action clause contains the keyword FOUND and the optional else clause is not present, execution of the find statement is completed without further action.
 - (2) If the action clause contains the keyword FOUND and the optional else clause is present, the alternative statement is executed next, which completes execution of the find statement.

- (3) If the action clause contains the keyword NOTFOUND, the simple statement of the action clause is executed, which completes execution of the find statement.

Examples

```
CASE.  FIND DEX(PC,STATE)EQ BRASS
        VARYING PC THRU 5 $
        IF DATA NOTFOUND THEN GOTO NEXT $
        SET REP TO REP + 1 $
        RESUME CASE $

NEXT.  VARY SEND FROM 1 THRU 3 $
        END NEXT $
```

Table DEX is searched for values equal to BRASS. If no matching values are found, control is transferred to the statement labeled NEXT.

SECTION 7. SUBPROGRAMS

Subprograms are the computational units of a CMS-2Y(7) program. In the body of a subprogram is a sequence of statements that is executed each time the subprogram is invoked. A subprogram may contain data that can be referenced only by the subprogram itself.

There are three types of subprograms: procedures, executive procedures, and functions. Each can receive input values through input parameters and can reference data in any scope that contains its scope of definition. Procedures and functions can return values to the subprogram that invoked them. A procedure can return values through its output parameters. A function always returns only a single value, known as the value of the function. An executive procedure cannot return values to the invoking subprogram; it can communicate with the invoking subprogram only by changing the values of data known in the scope of definition of the executive procedure, a capability that is also available to procedures and functions.

7.1 Subprogram Block

Syntax

```
<subprogram block>  
  ::= <procedure block>  
  ::= <executive procedure block>  
  ::= <function block>
```

Semantics

A subprogram block defines a subprogram.

7.1.1 Procedure Block

Syntax

<procedure block>
 ::= <procedure declaration> <procedure body> <end-procedure
 declaration>

<procedure body>
 ::= <subprogram body>

<end-procedure declaration>
 ::= END-PROC <procedure name> \$

Semantics

A procedure block defines a procedure.

- <procedure declaration> - A declaration of the attributes of the procedure being defined.
- <procedure body> - The statements that are to be executed when the procedure is called, and declarations of data whose scope is the procedure body.
- END-PROC - A language keyword indicating the end of a procedure block.
- <procedure name> - The name of the procedure being defined.

The procedure name that appears on the end-procedure declaration must be the same as the procedure name that appears on the procedure declaration.

Examples

```
VRBL (V1,V2,V3) I 16 U $  
PROCEDURE CHECKIT INPUT V1,V2  
  OUTPUT V3 $  
.  
.  
END-PROC CHECKIT $
```

The PROCEDURE and END-PROC statements mark the bounds of the subprogram body named CHECKIT. Variables V1 and V2 are the formal input parameters; variable V3 is the formal output parameter.

7.1.2 Executive Procedure Block

Syntax

```
<executive procedure block>  
  ::= <executive procedure declaration> <procedure body>  
      <end-procedure declaration>
```

Semantics

An executive procedure block defines an executive procedure.

<executive procedure declaration> - A declaration of the attributes of the executive procedure being defined.

An executive procedure may execute in either the executive state or the task state. The programmer must supply the entrance and exit logic. An executive procedure commonly executes in the task state of the AN/UYK-7 or AN/UYK-43 computer, but is called from the executive state with an interrupt return instruction (refer to manual M-5048).

The compiler does not generate code to save any registers upon entry to an executive procedure. The programmer must supply the return linkage from an executive procedure; return phrases are not permitted in the body of an executive procedure.

The procedure name that appears on the end-procedure declaration must be the same as the procedure name that appears on the executive procedure declaration.

Examples

```
EXEC-PROC EXPROC1 $  
  .  
  .  
  .  
END-PROC EXPROC1 $
```

The limits of an executive procedure named EXPROC1 are defined.

7.1.3 Function Block

Syntax

<function block>
 ::= <function declaration> <function body> <end-function
 declaration>

<function body>
 ::= <subprogram body>

<end-function declaration>
 ::= END-FUNCTION <function name> \$

Semantics

A function block defines a function.

- <function declaration> - A declaration of the attributes of the function being defined.
- <function body> - The statements that are to be executed when the function is referenced, and declarations of data whose scope is the function body.
- END-FUNCTION - A language keyword indicating the end of a function block.
- <function name> - The name of the function being defined.

The function name that appears on the end-function declaration must be the same as the function name that appears on the function declaration.

Examples

```
VRBL ALPHA A 30 S 13 $  
VRBL AZM A 14 S 0 $
```

```
FUNCTION TPOS(AZM) A 12 S 5 $  
  SET ALPHA TO 3+AZM/4 $  
  IF ALPHA GT 0 THEN RETURN (ALPHA) $  
  ELSE RETURN (0) $  
END-FUNCTION TPOS $
```

/(U) CM2Y-MAN-PGR-M5049-R04C0

In this example TPOS is declared as a function with one formal input parameter, AZM, and an output value type of signed fixed-point with six integer bits and five fractional bits. The content of variable ALPHA is converted to A 12 S 5 upon return to the expression that called TPOS. If the value of ALPHA is negative, that value is incremented upon return.

```
FUNCTION BOOL(ALPHA) B $  
  IF ALPHA GT 256*AQM THEN RETURN(1) $  
  ELSE RETURN(0) $  
END-FUNCTION BOOL $
```

This is an example of a Boolean function wherein only the true (1) or false (0) value is returned to the expression which referenced BOOL.

7.2 Subprogram Body

Syntax

```
<subprogram body>  
 ::= [<local index declaration>&] [<subprogram data block>]  
    [<statement>&]
```

Semantics

A subprogram body specifies the statements to be executed when the subprogram is invoked and, optionally, local indexes and subprogram data blocks to be accessed during execution of the subprogram.

- <local index declaration> - Optional. Declarations of local indexes to be accessed during execution of the subprogram.
- <subprogram data block> - Optional. Declarations of data whose scope is the subprogram body.
- <statement> - Optional. A statement to be executed when the subprogram is invoked.

When a subprogram is executed, the first statement to be executed is the first statement of the subprogram body.

A procedure body may contain no statements. The effect of executing such a subprogram is the same as executing a return phrase with no formal exit parameter specified.

An executive procedure body may not contain a return phrase.

A function body must contain a return phrase.

Examples

Section 6 contains examples of the various statement options available.

7.2.1 Subprogram Data Block

Syntax

```
<subprogram data block>  
 ::= <subprogram data declaration> <subprogram data  
      sentence>& <end-subprogram-data declaration>
```

```
<subprogram data declaration>  
 ::= [<data block name>] SUB-DD $
```

```
<subprogram data sentence>  
 ::= <type declaration>  
 ::= <variable declaration>  
 ::= <parameter variable declaration>  
 ::= <table block>  
 ::= <array block>  
 ::= <overlay declaration>  
 ::= <compile-time constant declaration>  
 ::= <ltag declaration>  
 ::= <address declaration>  
 ::= <procedure switch block>  
 ::= <label switch block>  
 ::= <file declaration>  
 ::= <format declaration>  
 ::= <range declaration>  
 ::= <preset value declaration>  
 ::= <inputlist declaration>  
 ::= <outputlist declaration>  
 ::= <stringform declaration>  
 ::= <direct code block>  
 ::= <local index declaration>
```

```
<end-subprogram-data declaration>  
 ::= END-SUB-DD [<data block name>] $
```

Semantics

A subprogram data block contains declarations of data that are to be referenced during execution of the subprogram body that contains the block.

- | | |
|-------------------|---|
| <data block name> | - Optional. The name of the data block being specified. |
| SUB-DD | - A language keyword indicating the beginning of a subprogram data block. |

<subprogram data sentence> - A declaration of a datum to be referenced during the execution of the subprogram.

END-SUB-DD - A language keyword indicating the end of a subprogram data block.

If the optional data block name appears on either the subprogram data declaration or the end-subprogram-data declaration, then the same name must appear on the other.

The scope of entities declared in a subprogram data block is the subprogram body of the subprogram block in which the data block appears. Declarations in a subprogram data block may not contain declaration modifiers.

The scope of the name of a subprogram data block is the subprogram body of the subprogram block in which the data block appears. The name has no function in the execution of a CMS-2Y(7) program.

SECTION 8. SYSTEM ELEMENTS

Syntax

```
<system element>  
  ::= <system data element>  
  ::= <system procedure element>
```

Semantics

System elements are the basic blocks of a CMS-2Y(7) compilation module. There are two types of system elements: system data elements and system procedure elements.

A system data element contains definitions of data whose scope is global.

A system procedure element contains definitions of procedures, functions, and declarations of data whose scope is the system procedure element, unless modified by the (EXTDEF) or (EXTREF) scope modifiers.

8.1 System Data Element

Syntax

```
<system data element>
  ::= [<minor header>] <system data block>

<system data block>
  ::= <system data declaration> [<data sentence> &]
     <end-system-data declaration>

<system data declaration>
  ::= <data block name> SYS-DD [<key specification> @] $

<data block name>
  ::= <name>

<data sentence>
  ::= <type declaration>
  ::= <variable declaration>
  ::= <parameter variable declaration>
  ::= <table block>
  ::= <array block>
  ::= <overlay declaration>
  ::= <compile-time constant declaration>
  ::= <!tag declaration>
  ::= <address declaration>
  ::= <procedure declaration>
  ::= <executive procedure declaration>
  ::= <function declaration>
  ::= <procedure switch block>
  ::= <file declaration>
  ::= <format declaration>
  ::= <range declaration>
  ::= <default type specification>
  ::= <preset value declaration>
  ::= <inputlist declaration>
  ::= <outputlist declaration>
  ::= <stringform declaration>
  ::= <direct code block>

<end-system-data declaration>
  ::= END-SYS-DD <data block name> $
```

Semantics

A system data element consists of a system data block containing declarations of data whose scope is global, optionally preceded by a minor header containing declarations whose scope is the system data block.

- <minor header> - Optional. Declarations that affect the following system data block.
- <data block name> - The name of the data block being declared.
- SYS-DD - A language keyword indicating the beginning of a system data block.
- <key specification> - Optional. A specification of a key and element form to identify elements in ISCM files.
- <data sentence> - Optional. A declaration of a datum that can be referenced during the execution of the CMS-2Y(7) program.
- END-SYS-DD - A language keyword indicating the end of a system data block.

The system data block name that appears on the end-system-data declaration must be the same as the system data block name that appears on the system data declaration.

A procedure declaration, executive procedure declaration, or function declaration within a system data block must include the (EXTREF) scope modifier.

The scope of entities declared in a system data block is the system block of which the system data block is an element.

The name of a system data block has global scope. It has no function in the execution of a CMS-2Y(7) program, but it may be referenced by the loader or librarian.

Examples

```

TESTDD SYS-DD $
  VRBL BOOL B $
  TABLE SMALL H 1 1 $
    FIELD I3U06 I 3 U 0 6 $
  END-TABLE SMALL $
  (EXTREF) PROCEDURE CHECKOUT INPUT BOOL $
END-SYS-DD TESTDD $

```

In this example system data block TESTDD contains only three of the possible data sentences allowed, i.e., a variable declaration, a table block, and a procedure declaration.

8.2 System Procedure Element

Syntax

```
<system procedure element>
    ::= [<minor header>] <system procedure block>

<system procedure block>
    ::= <system procedure declaration> [<system procedure
        sentence>&] <end-system-procedure declaration>

<system procedure declaration>
    ::= <procedure block name> <system procedure type> [<key
        specification>@] $

<procedure block name>
    ::= <name>

<system procedure type>
    ::= SYS-PROC
    ::= SYS-PROC-REN

<system procedure sentence>
    ::= <subprogram block>
    ::= <local data block>
    ::= <automatic data block>

<end-system-procedure declaration>
    ::= END-SYS-PROC <procedure block name> $
```

Semantics

A system procedure element consists of a system procedure block containing definitions of subprograms and declarations of data to be accessed during the execution of those subprograms, optionally preceded by a minor header containing declarations whose scope is the system procedure block.

- | | |
|------------------------|--|
| <minor header> | - Optional. Declarations that affect the following system procedure block. |
| <procedure block name> | - The name of the procedure block being declared. |
| SYS-PROC | - A language keyword indicating the beginning of a system procedure block. |

- SYS-PROC-REN - A language keyword indicating the beginning of a re-entrant system procedure block.
- <system procedure sentence> - Optional. The definition of a subprogram or a block of data declarations.
- END-SYS-PROC - A language keyword indicating the end of a system procedure block or a re-entrant system procedure block.

The procedure block name that appears on the end-system-procedure declaration must be the same as the procedure block name that appears on the system procedure declaration.

A re-entrant system procedure block is compiled in such a manner that multiple invocations of the subprograms of the block may execute simultaneously. Otherwise, the semantics of a system procedure block and a re-entrant system procedure block are the same.

The name of a system procedure block has no function in a CMS-2Y(7) program, but it can be referenced by the loader or librarian.

Examples

```
TESTSP SYS-PROC $
  PROCEDURE SETUP $
  .
  .
  .
  END-PROC SETUP $
  PROCEDURE CLOSEOUT $
  .
  .
  .
  END-PROC CLOSEOUT $
END-SYS-PROC TESTSP $
```

In this example, system procedure TESTSP contains subprogram blocks SETUP and CLOSEOUT.

8.3 Local Data Block

Syntax

<local data block>
 ::= <local data declaration> [<local data sentence> &]
 <end-local-data declaration>

<local data declaration>
 ::= [<data block name>] LOC-DD \$

<local data sentence>
 ::= <data sentence>
 ::= <label switch block>

<end-local-data declaration>
 ::= END-LOC-DD [<data block name>] \$

Semantics

A local data block contains declarations of data that are to be referenced during execution of subprograms defined in the system procedure block that contains the local data block.

- | | |
|-----------------------|---|
| <data block name> | - Optional. The name of the data block being declared. |
| LOC-DD | - A language keyword indicating the beginning of a local data block. |
| <local data sentence> | - Optional. A declaration of a datum to be referenced during the execution of a subprogram of the system procedure element. |
| END-LOC-DD | - A language keyword indicating the end of a local data block. |

The local data block name that appears on the end-local-data declaration must be the same as the local data block name that appears on the local data declaration.

The scope of entities declared in a local data block is the system procedure block in which the data block appears unless the declaration is preceded by the (EXTDEF) or (EXTREF) scope modifier.

A procedure declaration, executive procedure declaration, or function declaration within a local data block must include the (EXTREF) or (LOCDEF) allocation modifier.

The scope of the name of a local data block is the system procedure element in which the data block appears. The name has no function in the execution of a CMS-2Y(7) program, but it may be referenced by the loader.

Examples

```
LOC-DD $  
  VRBL BCD H 7 $  
  SWITCH CHOICE $  
    ANDA1 $  
    ANDA2 $  
    ANDA3 $  
  END-SWITCH CHOICE $  
END-LOC-DD $
```

In this example, variable and switch declarations are the only data sentences in the local data block.

8.4 Automatic Data Declaration

Syntax

```
<automatic data block>  
  ::= <automatic data declaration> [<automatic data  
    sentence>&] <end-automatic-data declaration>
```

```
<automatic data declaration>  
  ::= <data block name> AUTO-DD $
```

```
<automatic data sentence>  
  ::= <type declaration>  
  ::= <variable declaration>  
  ::= <parameter variable declaration>  
  ::= <table block>  
  ::= <array block>  
  ::= <overlay declaration>  
  ::= <compile-time constant declaration>  
  ::= <!tag declaration>  
  ::= <address declaration>  
  ::= <procedure declaration>  
  ::= <executive procedure declaration>  
  ::= <function declaration>  
  ::= <range declaration>  
  ::= <default type specification>
```

```
<end-automatic-data declaration>  
  ::= END-AUTO-DD <data block name> $
```

Semantics

An automatic data block contains declarations of data that are to be referenced during execution of subprograms defined in the re-entrant system procedure block that contains the automatic data block.

- | | |
|---------------------------|--|
| <data block name> | - The name of the data block being declared. |
| AUTO-DD | - A language keyword indicating the beginning of an automatic data block. |
| <automatic data sentence> | - Optional. A declaration of a datum to be referenced during the execution of a subprogram of the re-entrant system procedure element. |

END-AUTO-DD

- A language keyword indicating the end of an automatic data block.

The declarations of all local data that are to change during execution of a subprogram of a re-entrant system procedure must appear in an automatic data block.

Data in an automatic data block cannot be preset in any fashion.

The scope of entities declared in an automatic data block is the re-entrant system procedure block in which the data block appears unless the declaration is preceded by an (EXTDEF) or (EXTREF) scope modifier.

A procedure declaration, executive procedure declaration, or function declaration within a local data block must include the (EXTREF) or (LOCREF) allocation modifier.

The scope of the name of an automatic data block is the re-entrant system procedure element in which the data block appears. The name has no function in the execution of a CMS-2Y(7) program, but it may be referenced by the loader.

The data block name that appears on the end-automatic-data declaration must be the same as the data block name that appears on the automatic data declaration.

Examples

```
SHOW AUTO-DD $
TABLE COMPT V MEDIUM 20 $
  FIELD SAVE I 6 U $
  FIELD GATE I 5 U $
END-TABLE COMPT $
VRBL GAME A 30 S 2 $
END-AUTO-DD SHOW $
```

In this example, SHOW is an automatic data block containing one variable and one table.

8.5 Minor Header

Syntax

```
<minor header>
  ::= <minor header block>&

<minor header block>
  ::= [<header declaration>] [<minor header sentence>&]
     [<end-header declaration>]

<minor header sentence>
  ::= <header sentence>
  ::= <dependent element declaration>

<header sentence>
  ::= <substitution declaration>
  ::= <compile-time constant declaration>
  ::= <ltag declaration>
  ::= <constant mode declaration>
  ::= <default type specification>
  ::= <library declaration>
  ::= <source retrieval declaration>
  ::= <compiler directive>
```

Semantics

A minor header contains declarations and directives that affect the following system data block or system procedure block.

- | | |
|-------------------|---|
| <header name> | - Optional. The name of the header block being declared. |
| HEAD | - Optional. A language keyword indicating the beginning of a header block. |
| <header sentence> | - Optional. A declaration that will affect the following system data block or system procedure block. |
| END-HEAD | - A language keyword indicating the end of a header block. |

The header name that appears on the end-header declaration must be the same as the header name that appears on the header declaration.

The scope of the name of a minor header block is the first system data block or system procedure block that follows the header block. The name has no function in the execution of a CMS-2Y(7) program, but it may be referenced by the librarian.

Examples

```
HDR1 HEAD $
  PI MEANS 3.14159 $
  YTYPE EQUALS 7 $
  YCATEG EQUALS YTYPE*4 $
  YCLASS EQUALS YCATEG/2 + 5 $
  LIBS PROJECT (XYZ123) $
  SEL-SYS (MSTR) $
END-HEAD HDR1 $
```

Minor header HDR1 illustrates possible header sentences allowed in a minor header.

SECTION 9. COMPILATION MODULES

Syntax

<system block>
 ::= <system declaration> <major header> [<system element> &]
 <end-system declaration>

<system declaration>
 ::= <system name> SYSTEM [<key specification> @] \$

<system name>
 ::= <name>

<end-system declaration>
 ::= END-SYSTEM <system name> \$

Semantics

A compilation module -- the smallest unit acceptable to a compiler -- is called a system block in CMS-2Y(7).

- <system name> - The name of the system block being compiled.
- SYSTEM - A language keyword indicating the beginning of a system block.
- <major header> - Declarations that affect the system elements of the system block.
- <system element> - Optional. A system data element or system procedure element that is to be compiled.
- END-SYSTEM - A language keyword indicating the end of a system block.

The system name that appears on the end-system declaration must be the same as the system name that appears on the system declaration. The system name has global scope. It has no function in the execution of a CMS-2Y(7) program, but it may be referenced by the librarian.

/(U) CM2Y-MAN-PGR-M5049-R04C0

Examples

GREAT SYSTEM \$

.

END-SYSTEM GREAT \$

The system block is named GREAT.

9.1 Major Header

Syntax

```
<major header>
 ::= <options declaration>& [<major header sentence>&]
    <end-header declaration>
 ::= <options declaration>& [<major header sentence>&] <major
    header block>

<major header sentence>
 ::= <system index declaration>
 ::= <debug enabling declaration>
 ::= <address counter separation declaration>
 ::= <compool retrieval declaration>
 -::= <header sentence>

<major header block>
 ::= <header declaration> [<major header sentence>&]
    <end-header declaration>

<header declaration>
 ::= [<header name>] HEAD [<key specification>@] $

<header name>
 ::= <name>

<end-header declaration>
 ::= END-HEAD [<header name>] $
```

Semantics

A major header contains declarations that are applicable throughout an entire system block.

- | | |
|-------------------------|--|
| <options declaration> | - A specification of one or more compiler feature options. |
| <header name> | - Optional. The name of the header block being declared. |
| HEAD | - A language keyword indicating the beginning of a header declaration. |
| <major header sentence> | - Optional. A declaration that will affect the system block. |
| END-HEAD | - A language keyword indicating the end of a header block. |

The header name that appears on the end-header declaration must be the same as the header name that appears on the header declaration. CMS-2Y(7) major headers are used to:

- a. Parameterize the system block being compiled.
- b. Specify the legality or illegality of certain source statements.
- c. Provide processing directives that govern the compiler's interpretation of many CMS-2Y(7) operations.
- d. Specify the number and kind of compiler outputs desired.
- e. Activate specialized hardware and software processing features.

A system block must begin with and contain only one major header. The major header is bracketed by the system declaration and by the first end-header declaration. (All header sentences encountered after the first end-header declaration are part of a minor header.)

Any name declared in a major header has global scope.

The name of a major header block also has global scope. It has no function in the execution of a CMS-2Y(7) program, but it may be referenced by the librarian.

Examples

```
GREAT SYSTEM $
  OPTIONS UYK7 $
  OPTIONS OBJECT (SM,CR) $
  SYS-INDEX 3 XX $
END-HEAD $
```

In this example, the programmer has specified the AN/UYK-7 as the target machine, and requested a source and mnemonic listing and an address cross-reference listing. In addition, register 3 has been specified as a system index named XX.

```
GREAT SYSTEM $
  OPTIONS UYK7 $
  OPTIONS OBJECT (SM,CR) $
HDRA HEAD $
  SYS-INDEX 3 XX $
END-HEAD HDRA $
```

This example is functionally identical to the former, except that the system index statement is bracketed in a major header block named HDRA. It is a major header block because its END-HEAD declaration is the first one encountered in the system.

9.2 Options Declarations

Syntax

```
<options declaration>  
 ::= OPTIONS <target machine> [,<option specification>@] $  
 ::= OPTIONS <option specification>@ $
```

```
<target machine>  
 ::= UYK7  
 ::= UYK43
```

```
<option specification>  
 ::= <source specification>  
 ::= <object specification>  
 ::= <listing specification>  
 ::= <message level specification>  
 ::= <monitor specification>  
 ::= <non-real-time specification>  
 ::= <structured specification>  
 ::= <mode variable specification>
```

Semantics

An options declaration specifies one or more compiler options.

OPTIONS	- A language keyword indicating an options declaration.
<target machine>	- Specification of the computer for which the object program is to be generated (AN/UYK-7 or AN/UYK-43).
<option specification>	- Optional. Specification of the hardware and software options for the compile.
<source specification>	- Optional. Specification of the disposition of the compiler's source file output.
<object specification>	- Optional. Specification of the disposition of the compiler's object file output.

- <listing specification> - Optional. Specification of the disposition of the compiler's listing file output.
- <message level specification> - Optional. Specification of the type of error message to be produced by the compiler.
- <monitor specification> - Optional. Specification that enables the compilation of statements that directly or indirectly require access to the CMS-2Y monitor.
- <non-real-time specification> - Optional. Specification that indicates that the program is to be executed in a nonreal-time environment.
- <structured specification> - Optional. Specification that the compiler should issue warnings if nonstructured programming constructs have been used in the source program.
- <mode variable specification> - Optional. Specification that instructs the compiler to create local variable definitions for undefined names appearing in statements where the syntax of the statement permits references to variable names.
- <scaling specification> - Optional. Specification that indicates the CMS-2M scaling rules are to be used in the evaluation of numeric expressions.

The options declaration permits the programmer to select from a list of available features the required software and hardware options. Refer to Figure 9-01 for examples of available options and the results of combining those options. Refer to Appendix D for samples and descriptions of the listing formats resulting from various options requests.

All options declarations must immediately follow the system declaration (i.e., they must be the first declarations of a major header).

Multiple options specifications may appear in a single options declaration, or one options declaration may be used for each options specification. Option specification phrases may be used in any order, with the following restriction: the target machine must be specified on the first options declaration.

Examples

```
OPTIONS UYK7 $
OPTIONS SOURCE(LIST) $
OPTIONS OBJECT(CR) $
OPTIONS UYK7, SOURCE(LIST), OBJECT(CR) $
```

Figure 9-01 shows several OPTIONS permutations.

	AN/UYK-7 Language	Source Error Diagnostics	Object Warning Diagnostics	Source Warning Diagnostics	Source and Allocation Error Diagnostics	Address Cross Reference Listing	Source Cross Reference Listing	Source Cross Reference Sequencing	Object Cross Reference Listing	Listing File Output	Source File Output	Object File Output	Symbol Deck Output	Nonstructured Warning Diagnostics
RESULTS														
DECLARATIONS														
OPTIONS UYK7 \$	X	X	X	X										
OPTIONS UYK7,SOURCE,OBJECT(SA) \$	X	X	X	X	X	X	X							X
OPTIONS UYK7,SOURCE(LIST),OBJECT(CRL,SCRL) \$	X	X	X	X	X	X	X			X	X			
OPTIONS UYK7,SOURCE,OBJECT \$	X	X	X	X	X	X	X							
OPTIONS UYK7,SOURCE(COMN),OBJECT(SCR) \$	X	X	X	X	X	X				X		X	X	
OPTIONS UYK7,SOURCE(CARDS) \$	X	X	X	X						X			X	X
OPTIONS UYK7,LEVEL(1) \$	X	X	X											
OPTIONS UYK7,SOURCE(COMN,LIST) \$	X	X	X	X		X	X				X			
OPTIONS UYK7,SOURCE(CARDS,LIST) \$	X	X	X	X		X								X
OPTIONS UYK7,SOURCE(LIST),OBJECT(CR) \$	X	X	X	X	X	X	X			X	X			
OPTIONS UYK7,SOURCE,OBJECT(COMN,CR) \$	X	X	X	X	X	X	X			X	X		X	
OPTIONS UYK7,OBJECT(SM),LISTING(CLIST) \$	X	X	X	X	X	X		X						X
OPTIONS UYK7,OBJECT(CMP,SM,COBJT) \$	X	X	X	X	X	X		X					X	X
OPTIONS UYK7,SOURCE(LIST),OBJECT(SM) \$	X	X	X	X	X	X	X	X						
OPTIONS UYK7,OBJECT(CRG) \$	X	X	X	X	X	X					X			
OPTIONS UYK7,OBJECT(COMN) \$	X	X	X	X	X	X							X	
OPTIONS UYK7,SOURCE,LISTING(COMN), STRUCTURED,OBJECT(SCRG) \$	X	X	X	X	X	X	X					X		X
OPTIONS UYK7,OBJECT(CARDS) \$	X	X	X	X	X	X								X

Figure 9-01. Some Options Parameter Combinations and Results

9.2.1 Source Specification

Syntax

<source specification>
 ::= SOURCE [(<source parameter> @)]

<source parameter>
 ::= LIST
 ::= CCOMN
 ::= CSRCE
 ::= CARDS

Semantics

A source specification requests list, punched-card, and file output of the source code that has been input to the compiler. Any or all of these outputs may be requested in any order by the programmer.

SOURCE - A language keyword indicating a source specification.

<source parameter> - Optional. Specification of the file on which the source statement listing is to be output.

The keyword SOURCE with no source specification parameters specifies the production of the source statement listing on the hardcopy device.

LIST results in the listing of the source output on the hardcopy device. LIST is the source specification default parameter; it is necessary only if a hardcopy listing is desired in addition to outputs on CCOMN or CSRCE.

CCOMN results in the production of source code output on the CCOMN file for each system element and named header block. The CCOMN file may also contain other element types in addition to source elements.

CSRCE results in production of source code output on the CSRCE file for each system element and named header block.

CARDS results in production of source card images for each system element and named header block in punched card form.

If CCOMN or CSRCE is specified, these file outputs, along with any listings that are produced by the compiler, will contain a 4-digit card sequence number in card columns 5 through 8. The compiler performs source statement sequencing according to the

CMS-2Y library card-sequencing convention. A sequence-numbered source listing from a system block compilation can be used to build file correction decks for compilations using library retrieval of the corresponding source output.

Source-statement sequence numbers begin with 1 at the start of each library element and continue until the end of the system procedure block, system data block, or named header element.

Examples

```
OPTIONS SOURCE(CARDS) $
OPTIONS SOURCE(CCOMN) $
OPTIONS SOURCE(CSRCE) $
OPTIONS SOURCE(LIST) $
```

These options could also be specified as follows:

```
f  OPTIONS SOURCE(CARDS,CCOMN,CSRCE,LIST) $
```

Implementation Note

If the keyword OBJECT is present and if a source-and-mnemonic (SM) listing is not requested, each source statement in the source listing on the hardcopy device will be preceded by the relocatable address of the first instruction generated for that statement.

9.2.2 Object Specification

Syntax

<object specification>
 ::= OBJECT [(<object parameter> @)]

<object parameter>
 ::= CMP [(<compool name>)]
 ::= CR
 ::= CRG
 ::= CRL
 ::= SA
 ::= SADUMP
 ::= SM
 ::= CCOMN
 ::= COBJT
 ::= CARDS
 ::= CNV
 ::= SCR
 ::= SCRG
 ::= SCRL

<compool name>
 ::= <name>

Semantics

An object specification requests the compiler to proceed through the object-generation phase.

OBJECT - A language keyword indicating an object specification.

<object parameter> - Optional. Specification of the types of listings, compool, and object code to be generated.

The object specification may be requested with no object parameters. This will generate a compiler diagnostic listing.

The CMP parameter requests generation of a compool. It may be used only in systems whose system block consists solely of system data elements or headers. If the optional compool name is used, it will become the name of the compool element ISCM file; otherwise, the name of the last system data block will be used for the compool name.

The CR object parameter specifies that both global address and local address cross-reference listings are to be generated. CRG requests a global address cross-reference listing, and CRL

requests a local address cross-reference listing. Address cross-reference listings consist of addressable names, in alphabetical order, that have been defined within a system block.

A local address cross-reference is output following every system element. Each addressable name defined or referenced in the system element is listed, along with the address at which the name is allocated and all the addresses where the name is referenced. References to unallocated names will be listed as eight asterisks (*****). If a local address cross-reference is not specified, the compiler will output at the end of the system block a list of unallocated names detected in each system element.

The global address cross-reference is produced at the end of the system block. For each addressable global name, the global address cross-reference contains the system element in which the name is defined and each system element in which the name is referenced.

SM specifies that a source and mnemonic listing is to be generated. It contains all of the source statements, with each statement followed by the address and the numeric and mnemonic representations of each instruction generated for the statement.

SA specifies that a symbol analysis listing is to be produced. It provides a summary of identifiers declared in each element, grouped according to declarative category. Within each grouping the identifiers are alphabetized and their attributes are listed in short descriptive summaries.

SADUMP specifies that the symbol analysis information is to be output in machine-readable form to a tape file. The information will be in a subfile of the object tape file (COBJT or CCOMN). The format of the file is described in Appendix E.

The SCR object parameter specifies that both global and local source cross-reference listings are to be produced. SCRG requests a global source cross-reference listing, and SCRL requests a local source cross-reference listing. The source cross-references are alphabetical listings of names defined or used. In addition to all addressable names the source cross-reference includes the following names: types, fields, cswitch flags, local index names, system index names, substitution string names, ntag, ltag and rtag names, header names, system element names, local data block names, subprogram data block names, system names, pooling names, and form labels.

When a local source cross-reference is requested, a line number is appended at the end of each text line. These compiler generated numbers are assigned in numerical order, beginning with 1, for the major header and each system element. The listing

consists of all names defined or referenced in the element in alphabetical order, with line numbers for all references to each name. If a reference to a name causes the named entry to be modified, the line number is followed by an asterisk (*). A local source cross-reference listing is produced for the major header and for each system element.

When a global source cross-reference is requested, an alphabetical list of each global name defined in the system block is printed. Associated with each name is a list of all system elements in which the name is referenced. If the value of a named data unit is modified within the system element, an asterisk follows the system element name. The global source cross-reference is output at the end of the system block.

CCOMN specifies that the relocatable binary object code, the output compool, if specified, and the symbol analysis information (SADUMP), if specified, are to be output on the CCOMN file.

COBJT specifies that the relocatable binary object code, the output compool, if specified, and the symbol analysis information (SADUMP), if specified, are to be output on the COBJT file.

CARDS specifies that the relocatable binary object code is to be output in the form of a binary punched card deck.

CNV specifies that conversions between fixed-point and floating-point data formats are to be performed by library routines. If CNV is specified, it must be remembered that the use of run-time conversion routines is less efficient in terms of execution time but may require less memory than in-line generation if numerous conversions are performed. Use of CNV also requires that MONITOR or NONRT be specified in the options declaration. If the target computer is the AN/UYK-43, there are cases in which the inline conversion code is shorter than the code required to call the run-time routine. The in-line code will always be used in these cases.

Examples

```
OPTIONS OBJECT(CCOMN) $
OPTIONS OBJECT(CMP) $
OPTIONS OBJECT(CMP(CPOL)) $
OPTIONS OBJECT(COBT) $
OPTIONS OBJECT(CR) $
OPTIONS OBJECT(CRG) $
OPTIONS OBJECT(CRL) $
OPTIONS OBJECT(SM) $
```

Object parameters may also be concatenated within the same statement, such as:

```
OPTIONS OBJECT(CCOMN,COBJT,CR,SM) $
```

Implementation Note

If a global source cross-reference (SCRG or SCR) is requested and a global address cross-reference (CRG or CR) is requested, only the global source cross-reference will be output.

9.2.3 Listing Specification

Syntax

```
<listing specification>  
    ::= LISTING [( <listing parameter> @ ) ]  
  
<listing parameter>  
    ::= PRINT  
    ::= CCOMN  
    ::= CLIST
```

Semantics

A listing specification designates disposition of the output listings that are produced by the compiler.

LISTING - A language keyword indicating a listing specification.

<listing parameter> - Optional. Specification of the disposition of generated listings.

The compiler output listings may be written on the ISCM files CLIST and CCOMN, or on the standard hardcopy device, or on any combination. The parameters control the source listings, SM listings, SA listings, and cross-reference listings that result from object specifications.

PRINT indicates that compiler outputs are to be printed on the standard hardcopy device. It is the default parameter and is necessary only if printer output is desired in addition to output on CCOMN or CLIST. If only hardcopy output from the standard output device is required, listing specifications are not necessary.

CCOMN indicates that compiler outputs are to be written on the CCOMN file. CCOMN may not be used as a listing parameter if it is also used as an object parameter.

CLIST indicates that compiler outputs are to be written on the CLIST file.

Examples

```
OPTIONS LISTING(CCOMN) $  
OPTIONS LISTING(CLIST) $  
OPTIONS LISTING(PRINT) $
```

These option statements show different listing possibilities that may be requested.

Implementation Note

LISTING(CLIST) and LISTING(CCOMN) will have no effect if OBJECT is not specified.

9.2.4 Message Level Specification

Syntax

```
<message level specification>  
 ::= LEVEL (0)  
 ::= LEVEL (1)
```

Semantics

A message level specification indicates the kind of diagnostic messages that are to be output by the compiler.

LEVEL - A language keyword indicating a message level specification.

The parameter 0 specifies that all error and warning messages are to be listed. This is the default value.

The parameter 1 specifies that only error messages are to be listed (warning messages are not to be output).

Examples

```
OPTIONS LEVEL(0) $  
OPTIONS LEVEL(1) $
```

These examples show the format of the message level specification.

9.2.5 Monitor Specification

Syntax

```
<monitor specification>  
 ::= MONITOR
```

Semantics

The monitor specification enables compilation of statements that directly or indirectly require access to the CMS-2Y monitor (e.g., CMS-2Y(7) input/output statements, debug statements, and certain bit and character modified data unit references).

MONITOR - A language keyword indicating the CMS-2Y(7) monitor specification.

This specification also results in all testing of the special console conditions (e.g., KEY1) to be simulated by the monitor. In addition, the monitor specification implies access to nonreal-time facilities controlled by the NONRT option. This specification should be used only when the object code produced by the compiler is to be executed under monitor control (See manual M-5050, Section 2, for setting of simulated special console conditions.)

Examples

```
OPTIONS MONITOR $
```

This example shows the format of the monitor specification.

9.2.6 Nonreal-Time Specification

Syntax

```
<non-real-time specification>  
 ::= NONRT
```

Semantics

The nonreal-time specification indicates that the program is to be executed in a nonreal-time environment.

NONRT - A language keyword indicating the nonreal-time specification.

This specification enables generation of calls to implicit run-time functions that support exponentiation, BIT/CHAR, CAT, and conversions between fixed-point and floating-point data formats. The monitor specification automatically implies the nonreal-time specification. In the absence of NONRT (or MONITOR) all implicit references to these run-time functions will cause source warning messages and/or object error diagnostics.

Examples

```
OPTIONS NONRT $
```

This example shows the format of the nonreal-time specification.

9.2.7 Structured Specification

Syntax

<structured specification>
 ::= STRUCTURED

Semantics

The structured specification indicates that the source input is to follow CMS-2Y(7) structured programming conventions.

STRUCTURED - A language keyword indicating the structured specification.

The compiler will issue a warning message for each statement that violates these conventions. Nonstructured statements are:

- a. Label switch declarations.
- b. Branch phrases.
- c. Procedure switch call phrases containing an invalid specification.
- d. Procedures with exit parameters.

Examples

OPTIONS STRUCTURED \$

This example shows the format of the structured specification.

9.2.8 Mode Variable Specification

Syntax

```
<mode variable specification>  
 ::= MODEVRBL
```

Semantics

The mode variable specification instructs the compiler to create local variable definitions for undefined names appearing in statements where the syntax of the statements permits references to variable names.

MODEVRBL - A language keyword indicating the mode variable specification.

The implicitly defined variables are given the default type for variables that is in effect at the time the undefined name is encountered.

Examples

```
OPTIONS MODEVRBL $
```

This example shows the format of the mode variable specification.

9.2.9 Scaling Specification

Syntax

```
<scaling specification>  
 ::= MSCALE
```

Semantics

The scaling specification in an options declaration specifies that the CMS-2M scaling rules, as modified for an AN/UYK-7 or AN/UYK-43 computer (paragraph 5.3.1.6), are to be used in the evaluation of numeric expressions.

MSCALE - A language keyword indicating the scaling specification.

If no scaling specification appears, numeric expressions will be evaluated according to the CMS-2Y scaling rules (paragraph 5.3.1).

Examples

```
OPTIONS MSCALE $
```

This example shows the format of the scaling specification.

9.3 Compiler Directives

Syntax

```
<compiler directive>  
 ::= <parameter passage directive>  
 ::= <single precision directive>  
 ::= <executive directive>  
 ::= <spill directive>  
 ::= <pooling directive>
```

Semantics

A compiler directive specifies a detail of the compilation process, but it does not affect either the syntax or semantics of the language. It is similar to a compiler option, but a compiler option affects the entire compilation module, while a compiler directive can affect only a single element.

If a compiler directive appears in a minor header, it affects the system element of which that header is a part. If it appears in the major header, it affects the entire compilation module.

9.3.1 Parameter Passage Directive

Syntax

```
<parameter passage directive>
 ::= PASSAGE-SPEC <passage type> [<subprogram name>@]$
```

```
<passage type>
 ::= DIRECT
 ::= REGISTER[, CALLING ONLY]
```

```
<subprogram name>
 ::= <procedure name>
 ::= <function name>
```

Semantics

A parameter passage directive specifies the type of code sequence to be used in passing values between formal parameters and actual parameters during the invocation of a subprogram.

PASSAGE-SPEC - Language keyword indicating a parameter passage directive.

<passage type> - The type of the directive.

The parameter passage type DIRECT specifies that all code affecting the passing of values is to be generated in the calling subprogram. The passage type REGISTER specifies that the values are to be passed through registers; the calling subprogram contains code to handle the values of the actual parameters of the call, and the called subprogram contains code to handle the values of its formal parameters. The passage type REGISTER, CALLING ONLY is similar to the passage type REGISTER, except that the compiler generates code only on the calling side; no putaway code is generated in the bodies of the named subprograms. Details of these passage types are given below.

Parameter passage directives may only appear in headers. A parameter passage directive that appears in the major header can only list the names of subprograms whose scope is global in the system block being compiled. A parameter passage directive that appears in a minor header can only list the names of subprograms that are declared in the associated element, but the subprogram names can have either global or local scope.

If a function name appears in the list of a parameter passage directive, the passage type refers only to its input parameters.

If the optional list of subprogram names is omitted from a parameter passage directive, a default passage type is being specified. If the directive appears in the major header, the passage type becomes the default type for all subprograms, global or local, in the system block. If the directive appears in a minor header, the passage type becomes the default type for all subprograms declared in the associated element.

Redundant parameter passage directives are permitted.

9.3.1.1 Routine Linkage

The LBU B6 is used to transfer control from a calling subprogram to a called subprogram.

9.3.1.2 Function Value Return

Function values are returned from a function to the calling subprogram as follows:

- a. Values requiring a single word are returned in register A0.
- b. Values requiring a double word are returned in register pair A0-A1.
- c. For character-typed functions whose value is longer than eight characters, an indirect word pointing to a memory location containing the value is returned in register A0.

9.3.1.3 Direct Passage

Direct passage is the default parameter passage mechanism. When a subprogram using the direct passage mechanism is invoked, all of the parameter passage code is generated in the calling subprogram. Before control is transferred to the called subprogram, the values of the actual input parameters are stored into the corresponding formal input parameters. After control is returned from the called subprogram the values of the formal output parameters are stored into the corresponding actual output parameters.

9.3.1.4 Register Passage Algorithm

When register passage is specified for a subprogram, some parameter values might be passed directly. Only if a formal parameter is the name of a variable typed numeric, Boolean, or status, or a CORAD receptacle, is its value eligible for register passage; if the formal parameter is a table name, a system index name, the name of a variable typed character, or the name of an untyped variable, the value is passed directly. Of the eligible values,

only those for which registers are available according to the following algorithm are passed through registers.

The eight accumulators are treated as a sequence: A0, A1, A2, A3, A4, A5, A6, and A7. The formal parameter list is processed from beginning to end and each eligible formal parameter is assigned to the next available register or register pair (depending on whether the value of the formal parameter requires one word or two) in the sequence. If during this assignment process seven registers (A0-A6) have been assigned, register A7 will be assigned to the next formal parameter that requires only one word, even if intervening formal parameters requiring two words have been passed. Parameters that cannot be assigned to a register by this process are passed directly.

9.3.1.5 Register Passage, Calling Only

When register passage on the calling side only is specified, no code is generated in the subprogram body to move the values of the formal input parameters from the registers to memory, to load the values of formal output parameters into registers from memory prior to execution of a return phrase, nor to save the return linkage address.

The code generated for any CMS-2Y(7) statement in the called subprogram assumes that the registers assigned for parameter passage are available. If the statement references a formal input parameter, the value used is the value stored in its assigned memory location, not its assigned register.

9.3.2 Single Precision Directive

Syntax

```
<single precision directive>  
 ::= SINGLE $
```

Semantics

A single precision directive specifies that the compiler is to assume that all fixed-point arithmetic can be performed using the target machine single precision arithmetic instructions. In particular, the product of two fixed-point values (which might be longer than 32 bits and thus require the double precision instructions in some cases) will be assumed to be no longer than 32 bits in all cases.

9.3.3 Executive Directive

Syntax

```
<executive directive>  
 ::= EXECUTIVE $
```

Semantics

An executive directive specifies that the compiler is to generate code that will execute in the target machine executive state. This directive will affect the instructions that contain target machine control memory references.

9.3.4 Spill Directive

Syntax

```
<spill directive>  
 ::= SPILL $
```

Semantics

The spill directive instructs the compiler to provide to the loader, at object output time, every local and subprogram scope identifier within the scope of the directive as an external definition:

SPILL - A language keyword indicating the spill directive.

The directive does not alter the normal scope of identifiers during the compilation process.

If a spill directive appears in a major header, all addressable names in the system block and their associated addresses will be provided to the loader by the compiler. If a spill directive appears in a minor header, all addressable names in the associated system element will be provided to the loader.

Note

The spill directive primarily facilitates patching of resulting relocatable object code by permitting the use of symbolic addresses - such as statement names, procedure names, or data unit names - to specify the locations to be patched.

Implementation Note

Load-time DUPLICATE IDENTIFIER error messages can result from indiscriminate use of the spill directive because all names appear to be global to the loader, regardless of their compile-time scope.

9.3.5 Pooling Directive

Syntax

```
<pooling directive>
  ::= [<compound section name>] <pooling type> [<allocation
      information>] $
```

```
<compound section name>
  ::= <name>
```

```
<pooling type>
  ::= LOCDDPOOL
  ::= TABLEPOOL
  ::= DATAPPOOL
  ::= BASE
```

```
<allocation information>
  ::= [<base register specification>] [<address
      specification>]
```

```
<base register specification>
  ::= ([T], [<register number>])
```

```
<address specification>
  ::= <numeric constant expression>
```

Semantics

A pooling directive specifies that certain parts of the compiled program are to be gathered together in such a manner that they can be processed as a unit by the loader. Optionally, direction can be given to the loader concerning the allocation of those parts.

- | | |
|-------------------------|--|
| <compound section name> | - Optional. The name of the compound section generated as a result of the pooling directive. |
| LOCDDPOOL | - A language keyword indicating that local data blocks are to be pooled. |
| TABLEPOOL | - A language keyword indicating that global tables are to be pooled. |
| DATAPPOOL | - A language keyword indicating that the data of a system data block is to be pooled. |

- BASE - A language keyword indicating that the code of a system procedure block is to be pooled.
- T - Optional. A language keyword indicating that the pooled information is to be referenced transiently.
- <register number> - Optional. Specification of a target machine base register to be used in addressing the pooled information.
- <address specification> - Optional. Specification of an absolute target machine address at which the pooled information is to be loaded.

As a result of a compilation, the compiler produces, as a part of its binary object code file, various loader directives which are used by the CMS-2Y(7) loader during the binding of the final program. Pooling directives enable the specification of parameters, at the source program level, of two of the directives: The compound address section directive (*CS directive) and the address section definition directive (*AC directive). (See manual M-5050 for details on the use of these loader directives.)

The pooling types TABLEPOOL and DATAPPOOL are applicable to system data blocks only. TABLEPOOL specifies that all of the tables allocated within the system data block are to be grouped into an address section. DATAPPOOL specifies that all of the data allocated within the data block are to be grouped into an address section, unless TABLEPOOL has also been specified, in which case DATAPPOOL will only refer to those data that are not tables.

The pooling types LOCDDPOOL and BASE are applicable to system procedure blocks only. LOCDDPOOL specifies that all of the local and subprogram data allocated within the system procedure block are to be grouped into an address section. BASE specifies that the entire system procedure block is to be allocated into an address section, unless LOCDDPOOL has also been specified, in which case BASE will only refer to the code of the subprograms within the system procedure block.

If a directive for a pooling type appears in the major header, it affects the allocation of all of the system elements of the compilation module, except those for which a pooling directive of that type is specified in a minor header.

A pooling directive for any of the four pooling types may appear in any minor header. If the pooling type is inappropriate for the following system element, it is ignored by the compiler.

If the optional compound section name is specified on a pooling directive, that name is placed on the loader compound address section directive (*CS directive) generated by the compiler. If no name is specified, a default name will be used, depending on the pooling type:

<u>Pooling Type</u>	<u>Default Name</u>
BASE	SYSP
DATAPool	SYSDD
LOCDDPool	LOCDD
TABLEPool	TABLE

The default names SYSP and SYSDD are used for an entire system procedure element or system data element, respectively, when no pooling directives have been specified.

A compound section name has global scope during the compilation, therefore, all other global identifiers in the compilation module must be different from it. The same name can be used on more than one pooling directive, however.

If the optional T appears in the allocation information, the pooled information is to be referenced transiently, which means that no fixed target machine base register is assigned to the information. Each time the information is referenced, a base register must be loaded appropriately. The compiler generates the code to do this loading. If the T does not appear, the information is referenced normally.

If the optional base register specification appears in the allocation information, the specified register is used in addressing the pooled information. If the pooled information is addressed normally, the specified register is the first of as many consecutive registers as are needed to address the information. If the information is addressed transiently, it must be no more than can be addressed using only a single base register, which is the specified register.

If no base register is specified, a register is supplied by the loader.

The value of a base register specification must be an integer in the range [0,7].

If the optional address specification appears in the allocation information, it specifies a fixed target machine address at which

/(U) CM2Y-MAN-PGR-M5049-R04C0

the pooled information is to be loaded. The value of the address specification must be an integer in the range [0,262143] if the target computer is the AN/UYK-7, [0,4294967295] if the target computer is the AN/UYK-43.

If no address is specified, the address at which the pooled information is loaded is determined by the loader.

9.4 Address Counter Separation Declaration

Syntax

```
<address counter separation declaration>  
 ::= ACSEPARATION $
```

Semantics

An address counter separation declaration specifies that certain parts of the compiled program are to be gathered together in such a manner that they can be processed as a unit by the loader. Optionally, direction can be given to the loader concerning the allocation of those parts.

ACSEPARATION - A language keyword indicating the address counter separation declaration.

An address counter separation declaration causes the compiled program to be divided into the following parts: subprograms (instructions), data allocated within system data blocks (excluding tables, inputlists, and outputlists), data allocated within local and subprogram data blocks (excluding inputlists and outputlists), data allocated within automatic data blocks, compiler-generated constant data (including compiler-generated indirect words), compiler-generated temporary data, inputlists and outputlists, and variable length tables.

As a result of a compilation the compiler produces, as a part of its binary object code file, various loader directives which are used by the CMS-2Y(7) loader during the binding of the final program. The address counter separation declaration causes names to be supplied to the loader for the compound address section directive (*CS directive) and the address section definition directive (*AC directive). See manual M-5050 for details on the use of these loader directives.

If the optional compound section name is specified on a pooling directive, that name is placed on the loaded compound address section directive (*CS directive) generated by the compiler. If no name is specified, and for parts of the compiled program which do not have an associated pooling directive, a default name will be used:

<u>Default Name</u>	<u>Pooling Type</u>	<u>Program Part</u>
SYSP	BASE	Subprograms (instructions)
SYSDD	DATAPool	Data allocated within system data blocks, excluding inputlists and outputlists
LOCDD	LOCDDPool	Data allocated within local and subprogram data blocks, excluding inputlists and outputlists
AUTODD		Data allocated within automatic data blocks
CONST		Compiler-generated constants
TEMP		Compiler-generated storage locations
IOLIST		Inputlists and outputlists
Name of the variable length table		Variable length tables

The name placed on the address section definition directive (*AC directive) for variable length tables will be the table name. For all other program parts, the name placed on the address section definition directive will be composed of a two-character prefix and a six-character suffix. The prefix will be an A followed by the address section number. The suffix will be the first six characters of the system element name (blank-filled on the right if less than six characters).

Note

Because the system element name is truncated to six characters on the *AC directive, element names must be unique within their first six characters whenever ACSEPARATION is specified.

9.5 Compiler Input and Output Files

The CMS-2Y(7) compiler uses the standard input/output files supported by the monitor for punched-card input and output, and for hardcopy printer listings. The compiler also communicates with other system programs by means of intrasystem communication medium (ISCM) files. In general, within the CMS-2Y(7) compiling system, all permanent files produced by one component for subsequent input to another component are in ISCM form. Since the compiler accepts input files produced by other CMS-2Y(7) components (e.g., the librarian) and produces output files for other components (e.g., the librarian and loader), the compiler uses both input and output ISCM files. The compiler may also use as input an ISCM file that the compiler itself produced previously.

A specialized ISCM file is produced as output from the CMS-2 librarian. This output file, known as a library, is identical to a standard ISCM file, except that it is preceded by a directory (listing the elements contained on the file) and a history block.

9.5.1 ISCM File Elements

ISCM files contain outputs from the compiler in five forms: generated object code, compool elements, source elements, listing elements and symbol analysis elements. The generated object code is suitable for loading by the CMS-2Y loader. A compool element is the compiler's internal representation of the symbol table created for a set of system data elements. A source element contains 80-column source card images. A listing element contains 120-character print line images suitable for printing. A symbol analysis element contains a machine readable form of symbol analysis information. ISCM files may also be used as inputs to the compiler to supply source and/or compool elements.

Each element on an ISCM file generally corresponds to a CMS-2Y(7) major header, minor header, system data block or system procedure block. An ISCM file may contain a number of separate elements, and some ISCM files may include elements of differing forms (e.g., both source and object elements).

Each element on an ISCM file has three identifiable attributes: name, form, and key, except for a symbol analysis element, which has no key.

The element name, which identifies the file element, is provided when the element is added to the ISCM file. The name of an object code, source code, listing or symbol analysis element produced by the compiler is the same as the name of the corresponding system data block, system procedure block, major header, or named minor header. The name of a compool element is the name specified on the CMP object parameter, if any; otherwise, the name of the last system data block in the compool compile is used.

The element key is used to differentiate multiple elements of the same name and form.

9.5.2 Compiler Input ISCM Files

Elements may be retrieved from previously created ISCM files (compiler output files or CMS-2Y libraries) as part of the input to the compiler. Retrieval is effected by first specifying an ISCM file and then specifying either a set of source elements, using source retrieval declarations, or compool elements, using compool retrieval declarations. The source elements contain source statements. A compool element contains the attributes of the declarations in a system block, which can consist of a major header and system data elements.

9.5.2.1 Library Declaration

Syntax

```
<library declaration>  
 ::= LIBS <internal-id> [( <external-id> ) ] $
```

```
<internal-id>  
 ::= <name>
```

```
<external-id>  
 ::= <alphanumeric name>
```

```
<alphanumeric name>  
 ::= <alphanumeric character> &
```

Semantics

A library declaration specifies the name of a library or an ISCM file and its internal and external identifications, from which source and compool elements may be retrieved.

LIBS - A language keyword that indicates a library declaration.

<internal-id> - A name used for internal file system identification.

<external-id> - Optional. A name used for external file identification.

If an external-id is not specified, the internal-id is used for external file identification.

The external-id alphanumeric name must have at least one but no more than eight alphanumeric characters. It may start with a digit.

Examples

```
LIBS CMS2TAPE $  
LIBS XX (987) $  
LIBS COBJT (SVR4) $  
LIBS CCOMN $
```

These examples show the format of the library declaration.

9.5.2.2 Source Retrieval Declaration

Syntax

```
<source retrieval declaration>
 ::= SEL-ELEM <source element name> [( <key> ) ] [ , <dep
   specification> ] $ [<correction block header>]
 ::= SEL-SYS [( <key> ) ] $ [<correction block header>]
 ::= SEL-HEAD <source element name> [( <key> ) ] [ , <dep
   specification> ] $ [<correction block header>]
```

```
<source element name>
 ::= <name>
```

```
<dep specification>
 ::= ALL
 ::= ONLY
 ::= <numeric constant>
```

```
<correction block header>
 ::= CORRECT [NOLIST] $
```

Semantics

A source retrieval declaration specifies the retrieval of source elements from a file for compilation and possible corrections or other modifications.

- | | |
|-----------------------|---|
| SEL-ELEM | - A language keyword indicating that a specified element, and possibly all of its declared dependent elements, is to be retrieved. |
| <source element name> | - The name of the element to be retrieved. |
| <key> | - Optional. The key of the elements to be retrieved. |
| <dep specification> | - Optional. The level of dependent element retrieval. |
| SEL-SYS | - A language keyword indicating that all elements with a specified key in the system are to be retrieved. |
| SEL-HEAD | - A language keyword indicating that a specified element, and possibly all of its declared dependent elements, are to be retrieved. |

- CORRECT - Optional. A language keyword indicating that a correction block follows.
- NOLIST - Optional. A language keyword indicating that the source retrieval function should not produce a listing.

SEL-ELEM and SEL-HEAD statements are functionally identical.

The key is required on a SEL-ELEM or a SEL-HEAD statement only if the named element has a key on the ISCM file. If not required, the key is considered blank. If no key is specified on a SEL-SYS declaration, all source elements, regardless of key, are retrieved.

If the dep specification is:-

- a. ALL. All dependent elements are to be retrieved.
- b. ONLY. No dependent elements are to be retrieved.
- c. <numeric constant>. Dependent elements to the specified depth are retrieved. The value must be an integer in the range [0,255]. If the dep specification is 0, then all dependent elements are to be retrieved (this is equivalent to ALL). If the dep specification is 1, then no dependent elements are to be retrieved (this is equivalent to ONLY). If the numeric constant value has a value of n , where $2 \leq n \leq 255$, that specific element, plus $n-1$ levels of dependencies, are to be retrieved.

If no dep specification is given, ALL is assumed. This parameter is used only in conjunction with CMS-2Y libraries. It is ignored when retrieving from simple ISCM files (e.g., from compiler output files).

Manual M-5050 contains further discussion of the levels of dependency and the order of element retrieval.

SEL-SYS statements can be used in conjunction with SEL-ELEM and SEL-HEAD statements. The order of retrieval is dependent upon the order of the elements on the library. Retrieval of elements specified in one or more consecutive source retrieval declarations commences when one of the following conditions occurs:

- a. The compiler encounters a correction block header.
- b. The compiler encounters a CMS-2Y(7) statement other than a source retrieval declaration or comment phrase.

- c. The number of consecutive source retrieval declarations exceeds 60.

When retrieval is completed for a given set of requests and corrections, the compiler continues by processing the next line of the source program (which could be an additional library declaration or source retrieval declaration).

Source elements can be corrected during the retrieval process. The corrections do not modify the input source file itself, but only the elements as they are passed to the compiler. The name of the element and the card image sequence numbers, as given in the compilation or librarian listing, provide the reference points for making corrections in the form of deletions, insertions, or replacement of card images. Correction blocks must be introduced by a correction block header, which takes the place of the librarian's /CORRECT command. It indicates that one or more of the elements to be retrieved, as directed by preceding source retrieval declarations, are to be corrected. Unlike the librarian's /CORRECT command, a correction block header cannot start in card columns 1 through 10. The correction block header is followed by the correction block which has the same format as librarian control cards, and which is therefore terminated by the librarian's /ENDCOR command. Within a block of corrections, the order of the corrected elements must be that of the elements on the file or library. If the NOLIST parameter is included, none of the corrections is listed as part of the stream of input statements. Manual M-5050 contains more details on the correction capability.

Examples

```
SEL-ELEM PROCA $
```

Element PROCA with no key is to be retrieved from the specified libraries and ISCM files.

```
SEL-ELEM PROCB (SUB) $
```

Element PROCB, keyed SUB, is to be retrieved from the specified libraries and ISCM files. Other elements with the same name (PROCB) and different keys (or no key) will not be retrieved.

```
SEL-HEAD HDR1, ONLY $
```

Header HDR1 with no key is to be retrieved from the specified libraries and ISCM files. (This retrieval statement cannot apply to an ISCM file, since dep specifications are applicable only to library files.) Any dependent elements are not to be retrieved.

```
SEL-SYS (MJR) $
```

/(U) CM2Y-MAN-PGR-M5049-R04C0

Every element keyed MJR is to be retrieved from specified libraries or ISCM files.

9.5.2.3 Compool Retrieval Declaration

Syntax

```
<compool retrieval declaration>  
 ::= <library declaration>& <compool retrieval  
    specification>&
```

```
<compool retrieval specification>  
 ::= SEL-POOL <compool name> [( <key> ) ] $
```

Semantics

A compool retrieval declaration specifies the retrieval of a compool element from an ISCM file.

SEL-POOL - A language keyword indicating a compool retrieval.

<compool name> - The name of the compool element to be retrieved.

A compool retrieval declaration specifies the names of one or more compools and the ISCM files on which they can be found. Any number of compool retrieval declarations can appear in a system block, but they must appear in the major header block prior to the declaration of any name (i.e., they must immediately follow the options declarations).

The libraries specified in a compool retrieval declaration are searched in order of appearance. The compools named in the compool retrieval specifications are retrieved as they are encountered during the search of the libraries, which is not necessarily their order of appearance in the compool retrieval declaration. If more than one compool element of the same name is encountered during the library search, only the first element encountered is retrieved.

Compool elements cannot be corrected during the retrieval process.

The effect of compool retrieval is as if a symbol table containing all of the symbols of all of the compools were present from the beginning of the compilation. In particular, declarations of the same name in two different compools is an error, unless at least one of the declarations is an attribute (EXTREF) declaration (as it would be if the two declarations appeared in different SYS-DDs of the same compilation). The only exception to this is those declarations that appear in the major header (MEANS, ntags, ltags, etc.). This allows one major header, containing system parameters, to be used in compiling all of the compools.

Examples

SEL-POOL POOL(U7) \$

Compool POOL keyed U7 is to be retrieved from the library or ISCM file.

9.5.3 Compiler Output ISCM Files

Output files are requested and their contents specified by three different CMS-2Y(7) constructs:

- a. The options declaration is used to specify output ISCM files and to select elements for these files.
- b. The key specification specifies a key for an element or elements in a file, distinguishing that element from all other elements with the same name and form.
- c. The dependent element declaration specifies which elements are dependent elements of a given element.

9.5.3.1 ISCM File Specification With the Options Declaration

Options specifications are used to select the desired output ISCM files, and to specify the output data forms for the files. Four output files, named CCOMN, CLIST, COBJT and CSRCE, are available. The file names are used as both the internal- and external-ids. Each file and the output elements it may contain are listed below.

<u>File Id</u>	<u>Element Form</u>
CCOMN	Source, Object, Listing, Compool, Symbol Analysis
CLIST	Listing
COBJT	Object, Compool, Symbol Analysis
CSRCE	Source

One CCOMN file can include output elements for more than one system block in a compile. If CCOMN is designated as an output file in any source, object and/or listing options specification within any system block, all of the output data will be written on one CCOMN file in the order in which it is produced by the compiler.

Elements on CCOMN files for each CMS-2Y(7) system block are ordered as follows:

- a. All source elements.
- b. The compool element.
- c. All symbol analysis elements and object elements or all listing elements (but not both).

The range of the CSRCE, COBJT, and CLIST files is confined to a single system block. If one of these files is specified in an options declaration, all of the appropriate elements between the system declaration and end-system declaration will be produced. If CSRCE, COBJT or CLIST files are specified for more than one system block, the output for each system block is written on a separate file. The CSRCE, COBJT, and CLIST files are closed at the end of each system block.

Both CSRCE and CCOMN source parameters result in the output of a source element for all source lines contained within each of the following:

- a. System data block.
- b. System procedure block.
- c. Named major header block.
- d. Named minor header block.

Whenever CLIST or CCOMN are designated by the listing specification, the following listing elements result:

- a. The first element, identified by the system name, contains printline images that include all the information between the system declaration and the end-header declaration of the major header.
- b. The second ISCM element through the last ISCM element are the listing elements corresponding to each system data element or system procedure element. These listing elements include printline images from any minor headers through the local cross-reference. The name of each element is the name of the system element.

When the CMP parameter is specified, the single compool element will precede all object elements on either the CCOMN file or the COBJT file. The ISCM object files and binary object decks produced will contain a CMS-2Y(7) object element for each system data element in the compool compile. However, neither object nor listing elements will be produced for these system data elements in subsequent CMS-2Y(7) compilations that specify retrieval of this compool.

Examples

Refer to paragraph 9.2.1, paragraph 9.2.2, and paragraph 9.2.3 for examples.

9.5.3.2 Key Specification

Syntax

<key specification>
 ::= (<key>) [<element form>]

<key>
 ::= <alphanumeric name>

<element form>
 ::= *S
 ::= *O
 ::= *C
 ::= *L

Semantics

A key specification assigns a key to an element of an ISCM file, allowing differentiation of multiple elements having the same name and form.

<key> - The key to be assigned to an element.

<element form> - Optional. An *S, *O, *C, or *L indicating whether an element is a source, object, compool, or listing element, respectively.

A key can have no more than four alphanumeric characters.

Element keys may be specified in an ISCM file at the time of initial file creation, or by an edit function in the librarian. Although the use of key is optional, its use means that the element must be identified and referenced by both name and key.

More than one key may be specified on any of the applicable declarative statements. The key specifications included in the system declaration apply to all elements of the designated form output for the system block. Key specifications included in the header declaration, system data declaration, and system procedure declaration apply only to outputs associated with that element. If system-declared key specifications and an element-declared key specification designate the same form of output, the element-declared key is used.

If no element form is attached to the key specification, all forms produced are keyed. If no key specification is active for an element form, elements of that form are given a blank key.

/(U) CM2Y-MAN-PGR-M5049-R04C0

- e. System procedure block ELEM2. A source element named ELEM2, with a key of SK, on file CSRCE; an object element named ELEM2, with a key of OK, on file COBJT; a listing element named ELEM2, including minor headers HELEM2A and HELEM2B, with a key of LK, on file CCOMN.

9.5.3.3 Dependent Element Declaration

Syntax

```
<dependent element declaration>  
  ::= DEP <dep element>@ $  
  
<dep element>  
  ::= <element name> [(<key>)]  
  
<element name>  
  ::= <name>
```

Semantics

A dependent element declaration specifies the name of the elements that are dependent on the following system element.

- DEP - A language keyword indicating a dependent element declaration.
- <dep element> - The name and key of the dependent element.
- <element name> - The name of the dependent element.

Any system data block or system procedure block of a system block may have other elements dependent on or subordinate to it. When the system element is retrieved from a library, all of its dependent elements can also be retrieved.

The source element for a named minor header is automatically made a dependent element of its associated system data block or system procedure block. (This procedure ensures that retrieval from a CMS-2Y library of source for a system data block or system procedure block will normally result in automatic retrieval of associated named minor headers.) It is not possible to declare elements as dependent elements of a header element using a dependent element declaration. This function must be performed by the librarian.

The dependent element declaration has no direct effect on compilation of the current CMS-2Y(7) system; the information is used only in the preparation of the output source and object file elements. During library retrieval, whether of source or object elements, dependent elements are retrieved automatically with the selected element, unless otherwise specified by the user.

/(U) CM2Y-MAN-PGR-M5049-R04C0

Examples

```
HD HEAD $
  DEP SPROCB, SPROCC(QRS) $
  END-HEAD HD $
SPROCA SYS-PROC $
.
.
.
END-SYS-PROC SPROCA $
```

In the ISCM source and object elements produced for the system block shown above, source elements SPROCB, SPROCC (with a key of QRS), and HD will be dependent elements of source element SPROCA; object elements SPROCB and SPROCC will be dependent elements of object element SPROCA.

SECTION 10. CONDITIONAL COMPILATION

Syntax

```
<conditional compilation directive>  
 ::= <cswitch header statement>  
 ::= <cswitch terminal statement>  
 ::= <cswitch selection declaration>  
 ::= <cswitch delete declaration>
```

Semantics

Conditional compilation directives define blocks of code that are to be compiled if certain conditions are satisfied, select the blocks that are to be compiled, and direct the form of the compiler listing and other outputs with regard to those blocks.

The conditional compilation directives are extra-language statements and they do not appear in any other syntax productions within this manual.

Conditional compilation directives are never executed. They affect the execution of a CMS-2Y(7) program only by determining which statements are to be compiled.

10.1 Conditional Compilation Brackets

Syntax

```
<cswitch header statement>  
 ::= CSWITCH <cswitch flag> $  
  
<cswitch flag>  
 ::= <name>  
  
<cswitch terminal statement>  
 ::= END-CSWITCH <cswitch flag> $  
 ::= END-CSWITCHS $
```

Semantics

A cswitch header statement and cswitch terminal statement bracket a sequence of source statements, called a conditional compilation block, that is eligible for conditional compilation.

- CSWITCH - A language keyword indicating the beginning of a conditional compilation block.
- <cswitch flag> - A flag whose value during compilation determines if the conditional compilation block is to be compiled.
- END-CSWITCH - A language keyword indicating the end of a conditional compilation block.
- END-CSWITCHS - A language keyword indicating the end of all unended conditional compilation blocks.

Each cswitch header statement must be followed in the CMS-2Y(7) system block by a matching cswitch terminal statement. A cswitch terminal statement containing the keyword END-CSWITCH will match a cswitch header statement only if they contain the same cswitch flag. A cswitch terminal statement consisting of the keyword END-CSWITCHS will match all preceding cswitch header statements.

Cswitch header statements and cswitch terminal statements may appear anywhere in a system block except before the options declarations, in a direct code block, or between a find statement and its action clause.

Any number of statements may appear in a conditional compilation block.

A conditional compilation block must fully contain, or be fully contained in, a system data block, a system procedure block, a

local data block, an automatic data block, or a subprogram data block. A conditional compilation block may be contained in a major header.

Conditional compilation blocks may be nested, to a maximum of 10 levels. They may not overlap.

The effect of a conditional compilation block depends on the sequence of source statements that make up the system block, not the execution sequence of the program. If a cswitch header statement is encountered and its cswitch flag is on, the statements of the conditional compilation block will be compiled in the usual manner. If the cswitch flag is off, the statements will not be compiled, and the effect on program execution will be the same as if the statements of the block did not appear at all.

In a conditional compilation block that is not compiled because its associated cswitch flag is off, the only checking performed by the compiler is for proper bracketing of any nested conditional compilation blocks. No syntax checking is performed on any other statements.

If a conditional compilation block is being compiled because the associated flag is on, any nested conditional compilation block whose associated flag is off will not be compiled. If the block being compiled contains a cswitch declaration that turns the block's associated flag off, the flag will be set off, but the remainder of the block will be compiled as though the flag were on.

If a conditional compilation block is to be ignored because its associated cswitch flag is off, any cswitch declaration that appears in the block will be ignored and any nested conditional compilation block will be ignored, even if its associated cswitch flag is on.

Examples

```
CSWITCH UYK7 $
.
.
.
END-CSWITCH UYK7 $
```

Statements appearing between these header and terminal statements will be compiled only when the cswitch labeled UYK7 has been turned on with a cswitch selection declaration.

10.2 Compilation Selection Directives

Syntax

```
<cswitch selection declaration>  
 ::= CSWITCH-ON <cswitch flag>@ $  
 ::= CSWITCH-OFF <cswitch flag>@ $
```

Semantics

A cswitch selection declaration specifies one or more cswitch flags to be turned on or off.

CSWITCH-ON - A language keyword indicating that the flags in the following list are to be turned on.

CSWITCH-OFF - A language keyword indicating that the flags in the following list are to be turned off.

<cswitch flag> - The name of a flag whose state is being specified by the cswitch selection declaration.

A cswitch selection declaration may appear anywhere in a system block except before the options declarations or in a direct code block.

It is not necessary for all cswitch flags to appear in a cswitch selection declaration. The default setting of a cswitch flag is off; that is, if a cswitch flag appears in a cswitch header statement before it appears in a cswitch selection declaration, the value of the flag is off.

The values of cswitch flags at the end of the major header become default values for the remainder of the system block. The values of the flags can be changed by conditional compilation directives in any system element, but the flags revert to the default values at the end of each system element.

Examples

```
CSWITCH-ON TESTX $
```

The cswitch flagged TESTX is turned on such that all statements between any following header/terminal pairs with the name TESTX will be compiled.

10.3 Cswitch Delete Declaration

Syntax

```
<cswitch delete declaration>  
 ::= CSWITCH-DEL $
```

Semantics

A cswitch delete declaration specifies that any conditional compilation block whose associated flag is off is to be omitted from the compiler listings and from any source file output, along with the corresponding cswitch header and terminal statements.

A cswitch delete declaration may only appear in a header. If it appears in a minor header, its effect is from its point of appearance in the compilation sequence through the following system element. If it appears in a major header, its effect is from its point of appearance through the end of the system block.

APPENDIX A

ERROR AND WARNING MESSAGES

A.1 Source Error and Source Warning Messages

The following error and warning messages are issued as a result of errors detected during the source analysis phase of a compilation. Error messages are preceded by SE (for source error). Warning messages are preceded by SW (for source warning). Error messages are produced when the compiler is unable to take corrective action for a user error.

All CMS-2Y Compiler messages are included here for completeness.

SW	0	NO END-CSWITCH XXXXXXXX A CSWITCH bracket does not have a corresponding END-CSWITCH bracket before the end of the header, element, or data block.
SE	1	IDENTIFIER TOO LONG An attempt to define a name greater than eight characters long.
SE	2	CHARACTER CONSTANT TOO LONG A character constant is greater than 132 characters.
SE	3	RESERVED WORD USED AS ID Illegal use of a reserved word as a name.
SE	4	CHARACTER NOT RECOGNIZED Illegal ASCII input character.
SE	5	USER MUST PACK FIELDS The user defined a field declaration without defining the starting position of the field.
SW	6	NOTE TERMINATED BY \$ Notes were not completed before end of statement.
SE	7	INCORRECT OCTAL CONSTANT The decimal digits 8 or 9 appear in an octal constant.
SE	8	MISPLACED SEL-POOL A definition of a name other than the system name appears prior to SEL-POOL statement, or the SEL-POOL is not in the major header block.

- SE 9 ILLEGAL INTEGER VALUE
A numeric constant value must be an integer, an integer exceeds its maximum, or a negative value was used where a non-negative is required.
- SE 10 NO STATEMENT TERMINATOR
A missing \$ statement terminator.
- SE 11 IDENTIFIER MISSING
A missing name in a data unit declaration.
- SE 12 DUPLICATE IDENTIFIER
An attempt to declare a name previously declared in the same scope.
- SE 13 OUTSIDE TABLE BOUNDS
A subtable is not contained within a table; a field is not contained within an item; a multiword field is in a horizontal table; or presets are not contained within a table.
- SE 14 NO DESCRIPTIVE OPERATOR
A missing descriptive or separator term.
- SE 15 ILLEGAL IN MINOR HEADER
A statement is not allowed in a minor header (must be placed in a major header).
- SE 16 TOO MANY DIMENSIONS
More than seven dimensions are in an array declaration.
- SW 17 COMMA MISSING
A comma is missing in a statement.
- SW 18 OVERLAY PARENT MISMATCH
The total size of the overlay siblings in an overlay declaration exceeds the size of the overlay parent.
- SE 19 DUPLICATE OVERLAY
A data unit appears as an overlay sibling in more than one overlay declaration.
- SE 20 OVERLAY SEQUENCE ERROR
An overlay sibling in an overlay declaration appeared as an overlay parent in a previous or the current overlay declaration.

- SE 21 UNDECLARED IDENTIFIER
A referenced name has not been previously declared.
- SE 22 SCOPE CONFLICT
A local name has been used in a global context or the same name has been declared both local and global in the same element.
- SE 23 STATEMENT NOT RECOGNIZED
A statement is unrecognizable. Possible causes are a misspelled keyword, a valid statement in the wrong contexts or garbled syntax.
- SE 24 ILLEGAL OPTIONS
An illegal term has been specified in an options declaration. If CCOMN has been designated as the output unit for both the LISTING and OBJECT options, only the OBJECT option is honored; the LISTING option on CCOMN is ignored.
- SW 25 PARENTHESIS MISSING
Parenthesis missing within a statement.
- SE 26 ILLEGAL IN ARRAY
A subtable or like-table is declared in an array.
- SE 27 ILLEGAL OVERLAY DATA UNIT
An illegal data unit appears in an overlay.
- SE 28 ILLEGAL OVERLAY PARENT
A specified data unit may not be used as an overlay parent.
- SE 29 DUPLICATE RANGE
More than one range statement for the same variable or field.
- SE 30 PRESET NOT ALLOWED
A data unit preset is not allowed in an automatic data block, in an attribute definition, or for a field in a type declaration or an indirect table declaration.
- SE 31 ILLEGAL HARDWARE NAME
An illegal hardware device is specified in file declaration.
- SE 32 ILLEGAL FORMAT DESCRIPTOR
An illegal conversion descriptor is specified in format statement.

SE 33 MORE THAN 1 LEVEL NESTED
Format descriptors are nested (parenthesized) to more than one level.

| SE 34 UNUSED

SE 35 ILLEGAL SIZE DESCRIPTOR
Illegal data unit size attribute (e.g., character type over 132 characters; too many bits for numeric types) or the starting bit of a field is not 31 and the field crosses a word boundary.

| SW 36 UNUSED

-| SW 37 MONITOR OPTION REQUIRED
The monitor option must be declared for processing of this statement.

| SW 38 NONRT OPTION REQUIRED
Processing of the statement requires the NONRT (or MONITOR) option.

SE 39 SYSTEM LIMIT nn EXCEEDED
One of the following compiler limits denoted by nn has been exceeded. The code nn has the following values:

nn = 1 The constant conversion limit was exceeded; the value of the constant lies outside the limits defined below:

a. Target machine: AN/UYK-7 or AN/UYK-43. Lower limit: 1E-38. Upper limit: 1E 75.

b. Target machine: AN/UYK-20. Lower limit: 1E-78. Upper limit: 1E 75.

c. Target machine: CP-642. Lower limit: -536,870,911. Upper limit: 536,870,911.

nn = 2 The number of nested subexpressions within the condition of an IF statement may not exceed 10.

nn = 3 The number of libraries requested for retrieval may not exceed 10.

nn = 4 The number of operands in a DISPLAY statement has exceeded the compiler

limit. The card column indicator in the error output listing points to the operand which first exceeds the limit. This and following operands should be written as a separate DISPLAY statement. The limit may be calculated as follows:

- a. Allow $3 + n$ words for each operand, where n is the number of words required to contain the operand as a character string.
- b. The sum of step a may not exceed 94.

- nn = 5 The maximum number of exit parameters per procedure declaration is 10.
- nn = 6 The number of format descriptors exceeds 94 or the number of operands of an input/output list for INPUT, OUTPUT, ENCODE or DECODE statements exceeds 94. (For each operand that is a character constant, add the number of words required to contain the constant value.)
- nn = 7 A maximum of seven levels of subscripting and function calls per operand is allowed.
- nn = 8 An item beyond item 255 was specified in a field preset.
- nn = 9 The length of a statement is too long for the compiler to process properly. This may be due to the complexity of an expression or an abundance of embedded notes.
- nn = 10 The maximum number of elements declared dependent of another is 58.
- nn = 11 A VRBL declaration may define no more than 25 names.
- nn = 12 The offset of a sibling overlaid data unit relative to its parent data unit must not exceed 65535 words.
- nn = 13 More than 250 elements.

- nn = 14 Symbol table overflow -- number of global and local names. The compile will be aborted at this point in the source program.
- nn = 15 Compiler-packed table has more than 256 words per item.
- nn = 16 More than 100 nested block units.
- nn = 17 More than 10 nested VARY loop indexes.
- nn = 18 COMMENT statement or notes between FIND and IF DATA is (are) too long.
- nn = 19 COMMENT statement or notes between last THEN clause and ELSE clause is (are) too long.
- nn = 20 through nn = 29 Not used.
- nn = 30 A dependent retrieval level greater than 255 was requested. 255 is assumed.
- nn = 31 A magnitude value greater than 32767 was specified in a magnitude specification. 32767 is assumed.
- nn = 32 Table is greater than 65535 words.
- nn = 33 EQUALS term absolute value is greater than 65535.
- nn = 34 CCOMN specified for both library and output; output ignored.
- nn = 35 The maximum number of input parameters per procedure or function declaration is 25.
- nn = 36 The maximum number of output parameters per procedure declaration is 25.
- nn = 37 More than 10 nested TDEF or CONF operators.
- nn = 38 The maximum length of a single digit string (excluding any radix point) is 132 characters.

nn = 39 The maximum length of a SNAP or DISPLAY item (the item identification as printed, with blanks removed) is 132 characters.

nn = 40 The maximum number of input compools is 127.

- SW 40 CSWITCH NEST EXCEEDED
Nesting of CSWITCH brackets exceeded.
- SE 41 ILLEGAL EXTERNAL MODIFIER
Illegal or misplaced EXTREF, EXTDEF, or LOCREF declaration; or illegal use of * on a direct code label.
- SW 42 END DECLARATION MISSING
A statement that indicates the end of this program element or segment is not present.
- SE 43 HEADER NOT RECOGNIZED
An unrecognizable or illegal statement appearing in a header.
- SW 44 END-HEAD MISSING
No end header declaration at the end of the major or minor header element.
- SW 45 FUNCTION RETURN MISSING
A return phrase is missing from the function.
- SE 46 ILLEGAL EXIT PARAMETER
An illegal name is specified as a formal exit parameter.
- SE 47 COMPOOL REQUEST IGNORED
The requested COMPOOL was not produced due to detection of SYS-PROC statement.
- SE 48 UNUSED
- SE 49 INCOMPATIBLE DATA UNIT
Expression operands do not fit the context required by the operator.
- SW 50 NO DEF CHECK PERFORMED
No validation has been performed between the current declaration and a previous declaration of the same entity.

- SE 51 FILE TYPE MISSING
A type descriptor is missing in a file declaration.
- SW 52 CMS-2 BRACKET MISSING
The CMS-2 statement is missing as a terminator for a direct code block.
- SE 53 VALUE SIGNIFICANCE LOST
The most significant bits have been lost during alignment of a numeric constant used as a variable or field preset or a value block value.
- SW 54 DUPLICATE STATUS CONSTANT
A status constant appears more than once in a status type specification. The name maintains its position in the list both times but the second occurrence is inaccessible.
- SE 55 DUPLICATE ALLOCATION
A name appears on the left of more than one tag declaration.
- SE 56 ILLEGAL ALLOCATION
An attempt has been made to establish EQUALS allocation through a constant (absolute allocation) or illegal EQUALS expression; or a name appeared in a previous tag declaration; or an illegally allocatable name has been declared.
- SE 57 NO LIBRARIES SPECIFIED
A source retrieval or compool retrieval statement is appearing prior to a LIBS statement.
- SE 58 xxxxxxxx NOT RETRIEVED
The requested element, named xxxxxxxx, was not found in any of the declared libraries.
- SW 59 FIELD LIST MISSING
No fields were specified for a compiler-packed table or type.
- SE 60 WRONG PARAMETER COUNT
The number of procedure or function actual parameters is not the same as the declared number of formal parameters.
- SE 61 UNUSED
- SE 62 UNUSED

- SE 63 MUST BE FORMAT NAME
Syntax requires a name to be a format statement reference.
- SW 64 WRONG END NAME
An incorrect name on an END- statement.
- SE 65 SYNTAX ERROR
An erroneous statement syntax or punctuation.
- SE 66 COMPILER PROBLEM, SYNTAX
Syntax of a statement cannot be analyzed by the compiler.
- SW 67 INCORRECT END KEYWORD
The wrong KEYWORD appeared on the END- statement.
- SW 68 NO SYSTEM DECLARATION
A missing system declaration as the first statement of a source input.
- SW 69 NO END-SYSTEM
A missing END-SYSTEM statement.
- SW 70 SYNTAX WARNING
Syntax of a statement is not correct, but the compiler has assumed an interpretation.
- SW 71 OPTIONS STATEMENT MISSING
An options declaration is missing from a major header. Only output will be syntax diagnostics.
- SW 72 PARAMETER PROCESSED AS VRBL
Parameter variables are not allowed in function definitions.
- SE 73 MISPLACED STATEMENT
A misplaced or extraneous END statement has been encountered at a point in the program where all block declarations and their END delimiters have been paired; or a statement has been detected outside its valid limits.
- SE 74 ILLEGAL KEY TYPE
The key type is not legal for this element.
- SE 75 DUPLICATE KEY
The key was previously declared.

- SE 76 ELEMENT KEY GREATER 4 CHARS
A library element key is greater than four characters.
- SW 77 MISPLACED STATEMENT
An options declaration has been detected following other header declarations, or a local index declaration has been misplaced.
- SW 78 VALUE PRECISION LOST
The least significant bits have been lost during alignment of a numeric constant used as a variable or field preset or as a value block value.
- SE 79 ILLEGAL DECREMENT WITHIN
An illegal VARY contains explicit FROM and WITHIN parameters with a negative BY parameter.
- SW 80 32 BIT UNSIGNED DATA UNIT
A variable is 32 bits unsigned (requiring two words).
- SE 81 ILLEGAL FORWARD REF
Forward reference PROCEDURE and FUNCTION calls may not have status constants as input or output parameters.
- SW 82 NOT IMPLEMENTED
This feature is not implemented within the operating system (e.g., word typing on a table declaration).
- SW 83 TRUNCATED TO INTEGER
A scaled value has been truncated to an integer value where an integer is syntactically required.
- SW 84 SADUMP REQUIRES OBJECT TAPE
SADUMP has been specified, but no object tape (COBJT or CCOMN) has been specified.
- SE 85 NESTED MEANS OR EXCHANGE
A referenced MEANS or EXCHANGE name contains another MEANS or EXCHANGE name in its substitution string.
- SW 86 NON-STRUCTURED STATEMENT
The current statement violates CMS-2Y structured programming conventions.
- SW 87 CONSTANT PRECISION LOST
Precision bits of a converted constant in the

decimal range of 1E-24 to 1E-38 or the octal range of 1E-32 to 1E-52 have been lost.

- SE 88 ILLEGAL EQUALS
Illegal operator or operands in an EQUALS expression. A global data unit cannot be allocated to a local data unit.
- SW 89 NO CSWITCH FOR THIS END
An END-CSWITCH or END-CSWITCHS was detected which had no corresponding CSWITCH bracket.
- SW 90 DUPLICATE REGISTER
Microparameter registers were duplicated.
- SE 91 VARY INDEX IS THRU VALUE
A VARY loop index is the same data unit as the THRU clause data unit; hence, an illogical VARY statement.
- SE 92 DEFINITION MISMATCH
Two declarations of the same entity do not have the same attributes.
- SW 93 DUPLICATE SYS-INDEX
A register that has already been declared as a system index is defined as a system index.
- SW 94 IDENTIFIER EXTERNALIZED
A local identifier definition has been made global because of a previous external reference.
- SE 95 STATUS CONSTANT TOO LONG
More than eight characters were specified in a status constant.
- SE 96 UNEXPECTED END OF SOURCE
The end of the source file was detected before an end system declaration was detected.
- SE 97 TYPE NOT SPECIFIED
A FOR-type was not specified for a FOR-expression which requires an explicit type specification.
- SE 98 ERROR LIMIT EXCEEDED
More than 250 syntax errors if options OBJECT was requested, or more than 1000 syntax errors if options SOURCE was requested. The compile is aborted.

- SE 99 DUPLICATE CASE VALUE
The same value was specified for more than one case in the same case block.
- SE 100 VALUE MISSING
A value is not present in the BEGIN statement of a value block.
- SE 101 VALUE BLOCK MISSING
A BEGIN with associated value is not present following either a FOR statement or a value block that is not the last value block of a FOR block.
- SE 102 INCOMPATIBLE TYPE
The type of an operand is not compatible with its associated operator or operand.
- SE 103 MISPLACED VALUE BLOCK
A BEGIN with an associated value is present in a context other than immediately following a FOR statement or another value block.
- SE 104 CONDITIONAL NOT BLOCKED
A conditional statement not enclosed within BEGIN-END brackets is present in a primary, secondary, or alternative statement of another conditional statement.
- SW 105 UNCOMPLETED CONDITIONAL
The compound statement of a conditional statement was not completed at the end of the containing block, procedure, or function.
- SW 106 CONSTANT TRUNCATED
The rightmost characters have been truncated during alignment of a character constant used as a variable or field preset or as a value block value.
- SE 107 ILLEGAL REGISTER
For CMS-2Y(7), a register other than 0 through 7 was specified as a PARAMETER register or pooling declaration register, or a register other than 1 through 5 was specified as a system index register. For CMS-2Y(20), a register other than 0 through 15 was specified as a microparameter, or a register other than 6 through 11 was specified as a system index register. For CMS-2Y(642), a register other than 1 through 5 was specified as a system index register or SDS register.

- SW 108 SYSTEM LIMITATION
The host operating system does not support the requested feature.
- SE 109 MISPLACED IDENTIFIER
An illegal definition of a name for an EVEN, ODD, ORIG, REORIG, or CMS-2 directive.
- SE 110 VIOLATES LANGUAGE SUBSET
The referenced language feature is not included in the language subset being compiled.
- SW 111 RESERVED IN HIGHER LEVEL
The defined name is a reserved primitive in a higher language level of CMS-2.
- SE 112 LIST LIMIT EXCEEDED
The maximum number of items in one of the lists below has been exceeded.

Maximum

List of micro input parameters	16
List of micro output parameters	16
List of file states	7

- SE 113 MISPLACED ALLOCATION
The allocation phrase for a name precedes the definition of the name.
- SE 114 FLOAT NOT ENABLED
A reference to floating-point type or a floating-point constant has been made and the FLOAT option was not enabled.
- SE 115 MISPLACED MACHINE SPEC.
The machine specification is not the first options specification in the CMS-2 system; or a duplicate machine specification was detected.
- SW 116 USER RESERVED REGISTER
A register in the range 6 through 11 has been specified as a microparameter register.
- SE 117 EVEN REGISTER REQUIRED
An odd register was specified for a microparameter type which requires an even register.

SW 118 xxxxxxxx IS UNDEFINED
The LOCREF defined procedure or function xxxxxxxx did not have an allocation declaration in the current system procedure block.

SE 119 LANGUAGE STRUCTURE VIOLATION
END-CSWITCH was found in a different language structure than the CSWITCH bracket.

SE 120 END-CSWITCH MISPLACED
An END-CSWITCH phrase was not encountered within the block containing the conditional compilation block.

SW 121 VIOLATES LANGUAGE SUBSET
The referenced language feature is not included in the language subset being compiled. However, the feature will be correctly processed.

SW 122 NULL STATEMENT
A THEN or ELSE is followed directly by a \$.

SE 123 COMPILER ERROR XXX + YYYY
An internal compiler error; notify CMS-2Y maintenance personnel.

SE 124 ILLEGAL IN EXEC-PROC
Output parameters and exit parameters are illegal in an EXEC-PROC.

SE 125 CORAD PRESET ERROR
An Illegal term in CORAD preset.

SW 126 UNUSED

SE 127 READ-ONLY DATA MODIFICATION
An attempt to assign a value to a data unit in a read-only data block.

SE 128 UNDEFINED LABEL
The referenced label has not been defined.

SE 129 ILLEGAL LABEL
A character which is not alphabetic or a space appears in column 11 of a direct code statement.

SW 130 LABELED ELSE OR ELSIF
A label on an ELSE or ELSIF statement is illegal.

- SE 131 OVERLAY SCOPE CONFLICT
An overlay parent and siblings are defined in different system elements.
- SE 132 UNDEFINED PROCEDURE
A reference to a procedure that has not been declared.
- SE 133 SCALE FACTOR OUT OF RANGE
The value of the scale factor expression in a SCALF predefined function must be in [-127,127].
- SW 134 SUBSCRIPT OUT OF RANGE
The constant subscript expression has exceeded the declared bounds of the tabular data unit.
- SE 135 ILLEGAL IN INDIRECT TABLE
LIKE-TABLE and SUB-TABLE declarations are illegal in an indirect table.
- SE 136 UNUSED
- SE 137 INVALID SDS REGISTER
A register other than 1 through 5 was specified for an SDS register (CMS-2Y(642)).
- SW 138 xxxxxxxx DECLARATION MISSING
A system data block has an SDS register assigned, but it did not appear in any SDS declaration (CMS-2Y(642)).
- SE 139 SDS REG/SYS-INDEX DUPLICATE
A register cannot be used for both a system index and an SDS register (CMS-2Y(642)).
- SE 140 IDENTIFIER NOT A SYS-DD
The name on an SDS declaration is not a SYS-DD name (CMS-2Y(642)).
- SE 141 SYS-DD/SDS REGISTER MISMATCH
The SYS-DD declaration from a compool specifies a different register than on the SDS declaration (CMS-2Y(642)).
- SE 142 OPTION NOT PROCESSED
An option is illegal under the current operating system (because operating systems vary with installation requirements); or a CARDS option was requested for CMS-2Y(642).

- SE 143 CONSTANT EXPRESSION ILLEGAL
The numeric expression has been resolved to a single constant value in a context that requires a non-constant expression.
- SW 144 END NAME MISSING
No name appears on an END-statement when one is required.
- SE 145 LOC-DD TYPE ERROR
Different LOC-DD access types were specified for terms of an EQUALS declaration.
- SW 146 READ-ONLY DATA REF WARNING
A formal input parameter was defined in a data element with read-only access.
- SE 147 MISALIGNED OVERLAY SIBLING
The indicated overlay sibling has not been properly positioned in a target machine word as required for a data unit of its type.
- SE 148 UNRESOLVED TAG xxxxxxxx
The named tag has one or more terms that are not defined in the same element as the tag.
- SE 149 ILLEGAL USE OF PREDEFINED ID
A predefined identifier has not been user-declared in this scope, but the current use is incompatible with its predefined attributes.
- SW 150 ILLEGAL WITH USER-PACKING
An overlay declaration may not appear in a user-packed table or type.
- SE 151 NEGATIVE SUBSCRIPT ILLEGAL
Negative values within a subscript are illegal.
- SE 152 MISSING/ILLEGAL TGT MACHINE
A target machine option specification was omitted, or the target machine specified is illegal for CMS-2Y(Subset 0).
- SW 153 TERMINATE MISSING
A terminate statement is missing after an end-system declaration.

- SW 154 LOAD-VRBL INCOMPATIBILITY
The ltag used as the number of items of this table declaration was declared either signed or with more than 15 magnitude bits. If it was declared signed it has been changed to unsigned; if it was declared with more than 15 magnitude bits it has been changed to I 15 U. If the magnitude of the preset value requires more than 15 bits or if the preset value is negative, the preset value is changed to zero.
- SE 155 TM/COMPOOL MISMATCH xxxxxxxx
The input compool whose name has replaced xxxxxxxx was compiled for a different target machine than the one specified for the current compilation. The compool is bypassed.
- SE 156 COMPOOLS CONFLICT wwwwww and xxxxxxxx:
yyyyyyy (zzzzzzz)
Inconsistent definitions of the data unit yyyyyyy have been found in compools wwwwww and xxxxxxxx. zzzzzzz will appear only if yyyyyyy is the name of a user-packed type and zzzzzzz is a field whose definitions are inconsistent. In general, the definition in the first compool is the one used.
- SE 157 DECLARATION OF IMPLIED LABEL
The data unit being declared has been referenced previously in a context that caused the compiler to assume it was a label. The declaration must precede the reference.
- SE 158 OPTIONS MATHPAC REQUIRED
The indicated feature is available only if the AN/UYK-20 MATHPAC option has been specified.
- SW 159 DUPLICATE LOCAL LOAD-VRBL
The same name has been used for two local ltags. This is acceptable during compilation, but could cause problems at load time.
- SW 160 REGISTER PASSAGE ILLEGAL
A subprogram specified to use the register passage algorithm has a formal parameter declared as a parameter variable. The subprogram will use the direct passage algorithm.

- SE 161 COMPOOL FORMAT ERROR: xxxxxxxx
The specified compool has a format that is incompatible with the current compiler. This usually occurs when attempting to input a compool compiled with an earlier version of the compiler. The compool must be recompiled.
- SE 162 CONFLICTING PASSAGE TYPES
The procedures of a procedure switch do not all have the same passage type.
- SE 163 INVALID SCALF EXPRESSION
The controlled expression of a SCALF function reference does not contain an operation.
- SE 164 ILLEGAL NAME IN THIS CONTEXT
A table cannot be named H, O, or D. A type cannot be named A, B, F, I, P, or S.
- SW 165 TOO MANY FILE STATES
More than seven file states have been specified in a file declaration.
- SE 166 INVALID STATUS EXPRESSION
A constant status expression (involving a combination of SUCC, PRED, FIRST and LAST) has generated an undefined value.
- SE 167 ILLEGAL STRUCTURED TYPE
The name of a structured type has been used in a context requiring a simple type, or the name of a structured type having a multiword field is used in declaring a horizontal table.
- SE 168 ILLEGAL INHERITED FIELD REF
A name in a field overlay declaration or a range declaration is the name of a field inherited from the parent structured type.
- SE 169 ILLEGAL ARITHMETIC OPERATION
An illegal operation has been attempted in a numeric constant expression (e.g., division by zero).
- SE 170 ILLEGAL FILE OPERATION
A file operation is incompatible with the file's attributes (e.g., INPUT for PRINT).

A.2 Library Retrieval Diagnostic Messages

The following messages are issued for errors encountered during library retrieval.

**** CARD TOO LONG

One of the correction cards has too many parameters. Processing of it will continue.

**** END SENTINEL READ

An attempt was made to read an end sentinel card from standard input. Library retrieval continues.

**** ILLEGAL CONSTANT

A nonnumeric character is part of a numeric constant. Processing continues as if the character were a 0.

**** UNUSABLE CORRECTIONS

Some of the corrections to an element are unusable because of incorrect item numbers. These are ignored and processing continues.

**** WRONG TAPE MOUNTED

The wrong tape was mounted. The correct tape is again requested, and processing continues.

**** NOT ENOUGH CORE

There is not enough memory to load an input library's directory. The library is retrieved as an ISCM file.

**** NOT ENOUGH TAPE UNITS

Too many tapes have been specified. Library retrieval is not possible. No retrieval is performed.

**** I/O ERROR Txx

An I/O error was encountered on unit Txx during the retrieval. Processing will continue.

**** MONITOR CONTROL READ

An attempt was made to read a monitor command from standard input. An ENDCOR command is assumed and processing continues.

**** ILLEGAL RETRIEVAL FUNCTION

An illegal retrieval function was attempted. Notify CMS-2Y maintenance personnel.

**** WARNING - ILLEGAL LEVEL REQUEST

A request for multilevel dependent retrieval SEL-SYS was made. No dependent elements will be retrieved, but retrieval will continue.

/(U) CM2Y-MAN-PGR-M5049-R04C0

**** ELEMENT NAME TABLE OVERFLOW

There is no room to add a dependent element to the list of requested elements. Retrieval will continue, but some dependent elements may not be retrieved.

**** ILLEGAL TYPE

An element with an unrecognized type has been requested. Notify CMS-2Y maintenance personnel.

A.3 Object Error and Object Warning Messages

The following error and warning messages appear as a result of errors detected during the object generation phase of a compilation. Error messages are preceded by OE (Object Error). Warning messages are preceded by OW (Object Warning). Error messages are produced when the compiler is unable to take corrective action for a user error. Warning messages are produced when the compiler is able to attempt corrective action for a user error.

All messages within this list are produced by the CMS-2Y compiler.

- | | | |
|----|-----|--|
| OE | 200 | INCOMPATIBLE DATA TYPES
An attempted assignment or comparison of an incompatible data unit type. |
| OE | 201 | ILLEGAL OPERAND REF
An operand reference is illegal in the context used in the statement. |
| OW | 202 | ABS OF UNSIGNED DATA
An absolute value of unsigned data unit was requested. |
| OE | 203 | DIRECT CODE SYNTAX ERROR
An illegal or undefined operand, operator, or separator in a direct code statement. |
| OE | 204 | SYSTEM LIMIT nnEXCEEDED
One of the following compiler limits denoted by nn has been exceeded. The code nn has the following values: |
20. The allocation table for generated labels has overflowed. A maximum of 1000 generated labels per system procedure is allowed. This error may also occur for cases of more than 96 generated labels for a given procedure.
 21. Compiler use and allocation of temporary words have exceeded certain limits which, depending upon the distribution of temporary word usage and the number of procedures, range from 2460 to 3840 temporary words per system procedure.
 22. A maximum of 1536 binary constants can be generated per system procedure.

23. A maximum of 4800 words of Hollerith constants can be generated per system procedure.
 24. A maximum of 4000 indirect words can be generated per system procedure.
 25. A maximum of 65536 words can be generated on any address counter.
- OE 205 REMAINDER NOT AVAILABLE
SAVING remainder was specified in a statement without fixed-point division.
- OE 206 STMT REQUIRES NONRT OPT
A Run-time call will be generated. This requires the NONRT (nonreal-time) option to be present. It is present by default if the MONITOR option is used.
- OE 207 EXTERNAL DEF MISMATCH
An external reference does not match a subsequent external definition.
- OE 208 UNDEFINED IDENTIFIER
A forward reference to an identifier which is not subsequently defined.
- OE 209 SYSTEM ERROR
Notify CMS-2Y system maintenance personnel.
- OE 210 COMPILER ERROR
A compiler or undetected hardware error.
- OE 211 TRANSREF IN P-SWITCH
An illegal transient reference to procedure in a P-SWITCH.
- OE 212 TOO MANY DIGITS
Too many digits were specified in a direct code constant.
- OE 213 NON-NUMERIC CONSTANT
An illegal constant or improper punctuation in a direct code statement.
- OE 214 TOO MANY CHARACTERS
An illegal MEANS or EXCHANGE character substitution in a direct code statement.

- OE 215 ILLEGAL CHARACTER
An illegal ASCII character is appearing in a direct code statement.
- OE 216 UNRESOLVED EQUALS STMT
A reference to an EQUALS tag which is not resolvable at the time of reference.
- OE 217 ILLEGAL FORM STATEMENT
An illegal parameter in a direct code FORM statement or illegal implied FORM format.
- OE 218 FORM LABEL MISSING
A label is missing from a direct code FORM statement.
- OW 219 RIGHT TERM TRUNCATED
Truncation of an operand has occurred.
- OE 220 ILLEGAL SPECIAL COND
An illegal STOP special condition was specified on GOTO or RETURN statement.
- OE 221 COMPILER PROBLEM, SYNTAX
The syntax of the statement cannot be analyzed by the compiler.
- OE 222 PARAMETER TRANSFER ERROR
A statement results in alteration of contents currently held in the PARAMETER register.
- OW 223 K FIELD IGNORED BY UYK7
Issued by direct code on format III instruction words when the k field was coded with a value that was probably meant for the b field. The ultra formats require a k field to be indicated if subsequent fields are coded, even though the field is meaningless in format III instructions. The k field can be indicated by consecutive commas or by using 0 or k0.
- OW 224 ILLEGAL CORAD PRESET
A variable being preset with CORAD has less than 16 bits of magnitude, or the variable appears on the right of an overlay statement and is allocated less than a full word. The preset is processed by the compiler.

- OE 225 ILLEGAL CORAD PRESET
A variable being preset with CORAD must be allocated to the lower half-word of computer memory; it is greater than 16 bits, the allocation must include the entire lower half-word of computer memory. The preset is not processed by the compiler.
- OW 226 NBITS OR NCHARS IS ZERO
The number of bits or characters requested by BIT or CHAR is 0.
- OW 227 ALLOCATION OVERLAY
The preceding data unit has been preset with more words than it contains. The extra presets will be done but will be overlaid by the data unit(s) that follows.
- OW 228 ILLEGAL EXTREFED SIBLING
A previously defined variable has been encountered in an overlay statement. The code generated for earlier references may not work if the overlay causes it to be accessed using an indirect word.
- OW 229 TOO MANY VRBL LNGTH TBLS
Too many variable length tables have been specified for the AC counters available. The variable-length table which receives the warning has been changed to a fixed-length table, with a length equal to the preset length of the LTAG.
- OE 230 B-REG NOT AVAILABLE
B-register is needed and is not available.
- OE 231 INVALID aaa FIELD
The operand field denoted by aaa is invalid. The code aaa may be any of the following:

A, B, AF4, AK, C, E, I, IA, IU, IR, J, K, KJ, L, M, N, OR, OW, P, R, S, SY, U, UIJ, W, XAM, Y, 1/2
- OE 232 OPTIONS UYK-43 REQUIRED
An AN/UYK-43 instruction is specified for an AN/UYK-7 target computer. No instruction is generated.

- OE 233 IMPLICIT FORWARD REF. xxxxxxxx
A forward reference to a procedure/function formal parameter specified by xxxxxxxx has been detected. At the time of the reference xxxxxxxx has not been defined and cannot be referenced properly.
- OE 234 CONST SIGNIFICANCE LOST
The constant did not fit the scaling specified.

A.4 Reference Listings Error Messages

The following are the error messages generated by the CMS-2Y compiler for the various listing outputs.

*****COMPILE ERROR SUMMARY INCOMPLETE -- TOO MANY ELEMENTS*****

Appears at the end of requested object output when number of elements is greater than 143. The major header and all system data designs and system procedures are counted as elements; minor headers are not included in the element count, since they are considered as part of the succeeding system element. Object output is not affected by this message.

*****COMPILER ERROR*****

This message may appear at the end of the requested cross-reference or symbol analysis output. It indicates that an invalid condition was detected during data collection, and was caused by a compiler error.

*****CROSS REFERENCE INCOMPLETE

This message may appear at the end of an address cross-reference listing. It indicates that the tables for collecting reference data overflowed, and no more references for that element were collected.

*****GLOBAL CROSS REFERENCE UNAVAILABLE

This message is output whenever the BITTABLE overflow has been exceeded. As a result, the global SCR cannot be produced.

This message is also output whenever the number of elements per compile is greater than 143. As a result, neither global address nor global source cross-reference can be produced.

The local cross-reference for each element will still be available and printed. The major header and all system data designs and system procedures are counted as elements; minor headers are not included in the element count, since they are considered as part of the succeeding system element.

*****SORT TABLE OVERFLOW ***

This message informs the user that there will be no cross-reference or symbol analysis output because there are too many identifiers for the compiler-alphabetized identifier table.

INSUFFICIENT SYMBOL TABLE FOR COMPLETE CROSS-REFERENCE

This message appears with all header lines of the local source cross-reference when the table for collecting reference data overflowed. It indicates the local source cross-reference is not complete.

REMAINING FIELD PRESETS NOT PRINTED

This message appears in the source mnemonic when the number of presets exceeds the bounds of the preset packing table. The source lines for the remaining presets will be printed after the message.

A.5 Other Errors

A.5.1 Compiler Phase Errors

The CMS-2Y compiler is a multiphase program. If a condition arises which a phase cannot resolve, the user will be notified by source error 123 or by an object error.

A.5.2 Allocation Errors

The following codes may appear on the output listing to flag allocation errors:

- A Allocation error. Reference to an undefined label name or incorrect program allocation.
- E Programmer error.
- C Compiler error. Incorrect instruction generation or undetected hardware error.
- W Allocation warning. A user-allocated operand address is outside of the addressing segment.

B.1 Basic Constructs

B.1.1 Direct Code Characters

The direct code characters are identical with the CMS-2Y(7) characters. The use of the character \$, however, when used in direct code, refers to the current value of the address counter and therefore represents an address. In CMS-2Y(7) the character \$ is a statement terminator.

B.1.2 Delimiters

Direct code delimiters are special characters that are identical to those used in CMS-2Y(7). A delimiter in direct code is used to separate two tokens, indicate the beginning of a direct code comment, specify an operation involving the tokens on either side of the delimiter, or define a direct code literal or direct code character constant.

Tokens in direct code are separated by one or more spaces or by one of the other delimiters. If two tokens are separated by a delimiter other than a space, one or more spaces can be placed on either side of the delimiter. There are, however, some syntax productions where one or more spaces are required. In these productions the space will be explicitly specified.

B.1.3 Names

Names used in direct code are CMS-2Y(7) names with the same requirements and limitations (scope, uniqueness, etc.). Exceptions are text substitution declaration names, compile-time constant declaration names, and system and local index names.

B.1.3.1 Text Substitution Declaration Names

Text substitution declaration names (MEANS or EXCHANGE names) may be used within direct code statements, subject to the following restrictions and interpretations:

- a. MEANS and EXCHANGE substitution types produce identical results within direct code statements. EXCHANGE substitutions do not result in replacement by the simple string, and are interpreted as MEANS substitutions.
- b. It is not legal to include more than one direct code statement within one simple string.

B.1.3.2 Compile-Time Constant Declaration Names

Compile-time constant declaration names can be used within direct code statements, subject to the following restrictions:

- a. Ntag names. The numeric equivalent for an ntag name will be substituted wherever the ntag name is encountered except where the ntag name occupies the operation code field of a direct code instruction.
- b. Rtag names. The numeric equivalent for an rtag name will be substituted only if the rtag name is used as the y constant operand or as the length operand of a buffer control word.

B.1.3.3 Load-Time Variable Declaration

The numeric equivalent for an ltag name will be substituted only if the ltag name is used as the y constant operand.

B.1.3.4 System and Local Index Names

System and local index names are recognized and the register name is substituted wherever the system or local index name is encountered, except where the local index has been assigned to a memory word. A local index is assigned to a memory word only if all allotted registers for system indexes and/or local indexes have been previously assigned. In that case, the local index name is recognized as a direct code addressable name.

B.1.4 Operation Codes

Syntax

<operation code>
 ::= <AN/UYK-7 operation code>
 ::= <AN/UYK-43 operation code>
 ::= <1832 operation code>
 ::= <pseudo operation code>

<AN/UYK-7 operation code>

- ::= AA
- ::= AB
- ::= AEI
- ::= AFC
- ::= AIC
- ::= ALP
- ::= ANA
- ::= ANB
- ::= AOC
- ::= AXC
- ::= BC
- ::= BCW
- ::= BCWE
- ::= BS
- ::= BZ
- ::= C
- ::= CG
- ::= CL
- ::= CM
- ::= CNT
- ::= CXI
- ::= D
- ::= DA
- ::= DAN
- ::= DC
- ::= DJNZ
- ::= DJZ
- ::= DL
- ::= DS
- ::= FA
- ::= FAN
- ::= FANR
- ::= FAR
- ::= FB
- ::= FD
- ::= FDR
- ::= FM
- ::= FMIR
- ::= FMR
- ::= HA

:: = HAI
:: = HALT
:: = HAN
:: = HAND
:: = HC
:: = HCB
:: = HCL
:: = HCM
:: = HCP
:: = HD
:: = HDCP
:: = HDLC
:: = HDRS
:: = HDRZ
:: = HDSF
:: = HK
:: = HLB
:: = HLC
:: = HLCI
:: = HLCT
:: = HM
:: = HNO
:: = HOR
:: = HPI
:: = HRS
:: = HRT
:: = HRZ
:: = HSCI
:: = HSCT
:: = HSF
:: = HSIM
:: = HSTC
:: = HWFI
:: = HXOR
:: = IB
:: = IBS
:: = IBZ
:: = ILTC
:: = IMIR
:: = IO
:: = IPI
:: = IR
:: = ITSF
:: = IW
:: = IWB
:: = IWC
:: = IWCI
:: = IWS
:: = J
:: = JBNZ
:: = JC

::= JE
::= JEP
::= JG
::= JGE
::= JIO
::= JL
::= JLE
::= JLT
::= JN
::= JNE
::= JNF
::= JNW
::= JNZ
::= JOF
::= JOP
::= JP
::= JS
::= JSC
::= JW
::= JZ
::= LA
::= LB
::= LBJ
::= LBMP
::= LCI
::= LCT
::= LDIF
::= LICM
::= LIM
::= LLP
::= LLPN
::= LM
::= LNA
::= LSUM
::= LXB
::= M
::= MP
::= MS
::= NLP
::= NOOP
::= OB
::= OMIR
::= OR
::= PEI
::= RA
::= RALP
::= RAN
::= RD
::= RI
::= RJ
::= RJC

::= RJSC
::= RLP
::= RMS
::= RNLP
::= ROR
::= RP
::= RSC
::= RXOR
::= SA
::= SB
::= SC
::= SCI
::= SCT
::= SDIF
::= SICM
::= SLP
::= SM
::= SNA
::= SSUM
::= SXB
::= SZ
::= TBS
::= TBZ
::= TFB
::= TIB
::= TOB
::= TSF
::= TXB
::= XB
::= XMIR
::= XOR
::= XR
::= XRL
::= XS
::= ZA
::= ZB

<AN/UYK-43 operation code>

::= <AN/UYK-7 operation code>
::= AFCE
::= AICE
::= AOCE
::= ATSF
::= AXCE
::= CB
::= CBN
::= CBR
::= CCT
::= CE
::= CHCL
::= CICB

::= CMPS
::= CRB
::= DSP
::= EECM
::= ESCM
::= ETCM
::= FAC
::= FAS
::= FAT
::= FBE
::= FEX
::= FLN
::= FLTF
::= FLTL
::= FMIE
::= FPA
::= FPD
::= FPM
::= FPS
::= FSA
::= FSC
::= FSD
::= FSM
::= FSS
::= HAEI
::= HCRC
::= HLCA
::= HLTC
::= HPEI
::= HR
::= HSCA
::= HSIM
::= HST1
::= HST2
::= HST3
::= HST4
::= HSTC
::= HV
::= IADB
::= IADBC
::= IADD
::= IADDC
::= IAND
::= IANDC
::= IBE
::= IBSC
::= ICA
::= ICAC
::= ICB
::= ICBC
::= ICID

::= ICII
::= ICPA
::= ICPN
::= ICPU
::= ICT
::= IILM
::= IJC
::= IJD
::= IJE
::= IJGE
::= IJLT
::= IJN
::= IJNE
::= IJNZ
::= IJP
::= IJU
::= IJUI
::= IJZ
::= ILA
::= ILAC
::= ILB
::= ILB0
::= ILB1
::= ILB2
::= ILB3
::= ILBC
::= ILBJ
::= ILOB
::= ILOBC
::= ILRC
::= ILRCC
::= ILS
::= ILSC
::= ILST
::= ILSTC
::= IMIE
::= IOCL
::= IOCR
::= IOCS
::= IOR
::= IORC
::= IPM
::= IRA
::= IRPD
::= IRS
::= ISA
::= ISAC
::= ISB
::= ISBC
::= ISJC
::= ISJCC

::= ISL
::= ISLA
::= ISOB
::= ISOBC
::= ISP
::= ISR
::= ISRA
::= ISS
::= ISSC
::= ISST
::= ISSTC
::= ISTD
::= ISTCC
::= ISTSB
::= IXOR
::= IXORC
::= JBAE
::= JBAN
::= LCM1
::= LCM2
::= LCM3
::= LCM4
::= LCMA
::= LCMP
::= LCMT
::= LCPA
::= LCRA
::= LECM
::= LIBP
::= LIMP
::= LIOAM
::= LRRR
::= LSCM
::= LTCM
::= MOVE
::= OBE
::= OMIE
::= PACK
::= PFCB
::= PFCE
::= PFR
::= PMM
::= PMR
::= PIE
::= POP
::= PUSH
::= RCCR
::= RMMS
::= RMSR
::= RPD
::= SBN

::= SBPC
::= SCM1
::= SCM2
::= SCM3
::= SCM4
::= SCMA
::= SCMP
::= SCMT
::= SCPA
::= SCRA
::= SCSR
::= SDCM
::= SIBP
::= SICB
::= SICP
::= SIMC
::= SIMP
::= SIOAM
::= SIRC
::= SITC
::= SMCC
::= SMSR
::= SRRA
::= STAF
::= STSB
::= TBN
::= TFBE
::= TIBE
::= TOBE
::= TR
::= TSBN
::= TSM
::= TV
::= TXBE
::= UNPK
::= WFBP
::= WFM
::= XBE
::= XMIE

<1832 operation code>

::= HEIB
::= HSIB
::= KBCW
::= KCB
::= KECM
::= KEDB
::= KEFB
::= KEIB
::= KESM
::= KFBB

::= KI
::= KIB
::= KIBF
::= KJ
::= KOB
::= KOBF
::= KRB
::= KRCH
::= KRTC
::= KSB
::= KSBS
::= KSBX
::= KSCM
::= KSSM
::= KTB
::= KTDB
::= KTEI
::= KTSB

<pseudo operation code>

::= ABS
::= BAM
::= BYTE
::= CHAR
::= DO
::= FORM
::= ISFP
::= ISFPS
::= RAD
::= RES

Semantics

Operation codes specify target machine instructions and directives in direct code.

<pseudo operation code> - Nonreserved words that have been assigned a specific meaning in direct code.

Because pseudo operation codes are not reserved words, they can also be used as names if they meet the requirements for names. This means that a name can duplicate an operation code subject to the following restrictions and interpretations:

- a. A text substitution declaration name (MEANS or EXCHANGE name) which duplicates an operation code is always interpreted as a text substitution declaration name, regardless of its location in a direct code statement.

/(U) CM2Y-MAN-PGR-M5049-R04C0

- b. If a valid name (other than a text substitution declaration name) duplicates an operation code, it is interpreted as a name except when it occupies the operation code field. In that case, the name is interpreted as an operation code.
- c. A form name may not duplicate an operation code.

B.1.5 Direct Code Constants

Syntax

```
<direct code constant>  
  ::= <direct code numeric constant>  
  ::= <direct code character constant>
```

Semantics

Although constants in direct code statements serve the same function as constants in CMS-2Y(7) statements, the syntax is different.

Direct code constants can be direct code presets or instruction operands. Subject to certain restrictions, they may be used as terms in direct code expressions.

Direct code numeric constants are fixed-point type or floating-point type. Direct code character constants are character type.

B.1.5.1 Direct Code Numeric Constants

Syntax

```
<direct code numeric constant>
  ::= <direct short numeric constant>
  ::= <direct long numeric constant>
  ::= <uyk-43 floating constant>
  ::= <angular measurement>

<direct short numeric constant>
  ::= <direct code decimal constant>
  ::= <direct code octal constant>
  ::= <ntag name>

<direct code decimal constant>
  ::= <unscaled direct code decimal constant>
  ::= <scaled direct code decimal constant>

<unscaled direct code decimal constant>
  ::= <decimal integer>

<scaled direct code decimal constant>
  ::= <decimal integer> <scale operator> <direct code scale
    factor>
  ::= <uyk-7 floating constant> <scale operator> <direct code
    scale factor>

<scale operator>
  ::= */

<direct code scale factor>
  ::= [<unary numeric operator>] <decimal integer>
  ::= [<unary numeric operator>] <octal integer>

<uyk-7 floating constant>
  ::= <decimal integer> . <decimal integer> [<power of ten
    factor>]

<uyk-43 floating constant>
  ::= ISFP <simple floating constant>
  ::= ISFPS <simple floating constant>

<simple floating constant>
  ::= <decimal integer>[. <decimal integer>]

<power of ten factor>
  ::= * [<unary numeric operator>] <decimal integer>
```

<direct code octal constant>
 ::= <unscaled direct code octal constant>
 ::= <scaled direct code octal constant>

<unscaled direct code octal constant>
 ::= 0<octal integer>

<scaled direct code octal constant>
 ::= 0<octal integer> <scale operator> <direct code scale factor>

<direct long numeric constant>
 ::= <unscaled direct code decimal constant> D
 ::= <unscaled direct code octal constant> D

<angular measurement>
 ::= BAM <basic angle> [, <angle scaling>]
 ::= RAD <basic angle> [, <angle scaling>]

<basic angle>
 ::= 0<octal integer> [. <octal integer>]
 ::= <decimal integer> [. <decimal integer>]

<angle scaling>
 ::= 0<octal integer>
 ::= <decimal integer>

Semantics

Direct code numeric constants may be expressed either as direct code decimal constants or direct code octal constants.

Direct code numeric constants, represented as ones complement binary numbers, can be direct code single-word (short), double-word (long), or floating-point (floating) numeric constants.

- <direct short numeric constant>- Constant that will fit in a single computer word (32 bits).
- <direct code decimal constant>- A numeric constant in base 10 notation in the form of a string of decimal digits.
- <direct code octal constant> - A numeric constant in base 8 notation in the form of a string of octal digits, beginning with a 0.

- <direct long numeric constant>- A constant that will fit in two consecutive computer words (64 bits).
- <uyk-7 floating constant> - A decimal mixed number consisting of an integer portion, a fractional portion, and a decimal point.
- D - A language keyword indicating a double-word numeric constant.
- <uyk-43 floating constant> - Constant in Industry Standard Floating Point.
- <angular measurement> - An angle represented in BAMS or Radians.

A direct short numeric constant can be an ntag name defined in a compile-time constant declaration (EQUALS) or an octal or decimal constant.

Decimal constants cannot have 0 as the first digit. The unary numeric operators (+ and -) must precede the constant in a direct code preset, but are optional when the direct code numeric constant is used in other contexts.

A direct long numeric constant is designated by appending the character D to the unscaled octal or decimal constant. Otherwise the constant is single-word.

Scaled direct code octal constants and scaled direct code decimal constants share one form in common, which can be expressed as $N1*/N2$ where:

N1 - An Unscaled direct code octal or decimal single-word numeric constant.

*/ - Specifies that a scale factor follows.

N2 - An unscaled direct code octal or decimal numeric constant used in aligning N1, in the range of [-127,127].

A scaled direct code decimal constant may have N1 optionally written as a direct floating constant (the floating-point constant cannot be coded with the power of ten factor). The floating-point constant is internally converted to a single precision fixed-point value and aligned to N2.

The optional power of 10 factor for direct code floating-point constants results in a floating-point constant representing the value of the decimal mixed number, multiplied by 10 to the indicated power; that is:

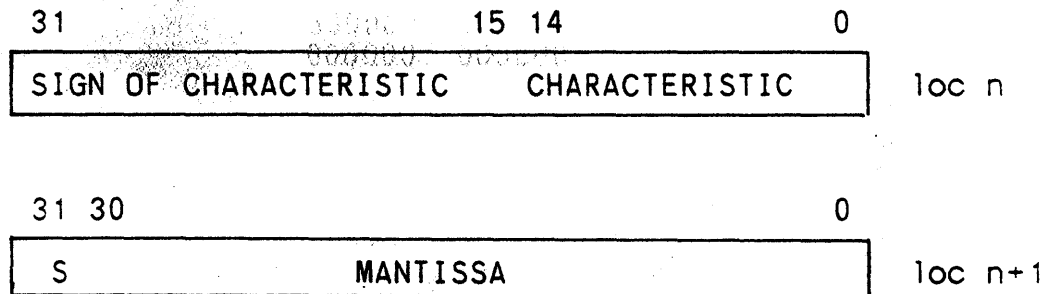
$$75.0^{**+6} = 10^{**6} \text{ times } 75.0$$

$$75.0^{**-6} = (1/10)^{**6} \text{ times } 75.0$$

Scale values are not tested for overflow. Positive scale values cause the contents of a word to be shifted circularly to the left the number of bits equal to the scale value. Negative scale values cause a right shift sign fill equal to the scale value.

For RAD the default scaling is 29. For BAM the default scaling is 32. Both RAD and BAM produce positive results and only accept positive inputs. If the input basic angle is greater than 360 degrees then the result is the input basic angle module 360 degrees.

The 2-word floating-point constants with normalized mantissa can be illustrated as follows:



S is the sign of the mantissa.

Examples

Examples of direct short numeric constants are as follows:

- +512 - Unscaled decimal constant
- +0512 - Unscaled octal constant
- 16 - Unscaled decimal constant
- 016 - Unscaled octal constant
- +TAG1 - Ntag name

Examples of scaled direct code decimal and octal constants are as follows:

+8*/010 - 8 scaled to 10 octal
-077*/9 - Octal 77 scaled 9
-8*/9 - -8 scaled 9
+45.5*/3 - Floating-point 45.5 converted to fixed-point and scaled 3

Examples of direct long numeric constants are as follows:

+1D . doubleword	00000000001
	00000000000
+076543210123D	36543210123
	00000000001

| Examples of uyk-7 floating constants are as follows:

+75.0**+10	000000	000050
	053617	136746
-1.5	000000	000001
	117777	777777
+45.0	000000	000006
	055000	000000

Note

The maximum number of digits allowed in a short numeric constant, positive or negative, is 9. Therefore, the maximum value which can be specified is 999,999,999, even though a 32-bit word is capable of holding a maximum value of 2,147,483,647 (2 to the 31st power, minus 1). Hence a scaling of -30 or -31 guarantees that all significance will be lost, and the word will be flushed with the value of the sign bit.

B.1.5.2 Direct Code Character ConstantsSyntax

```
<direct code character constant>
  ::= +'<character>{'
```

Semantics

A direct code character constant is a string of any valid ASCII characters (except a single prime), bracketed by single primes, and preceded by a +.

Each ASCII character in a direct code character constant is represented by eight bits. A direct code character constant contains from one to 66 ASCII characters. One to four characters are stored into one computer word; five to eight characters are stored into two computer words, etc. Characters are packed right-justified within the generated words with leading binary zeros, as required, to fill the first word.

Examples

```
+ 'ACRE'           - 10120651105
+ 'ACREAGE'        - 00020241522
                   - 10520243505
```


B.1.5.3 Direct Code Literal

Syntax

```
<direct code literal>  
 ::= ([[<unary numeric operator>] <direct code numeric  
      constant>)]  
 ::= (<direct code character constant>)
```

Semantics

A direct code literal is a direct code numeric constant or a direct code character constant that is assigned to a compiler constant table.

Direct code literals are coded by enclosing a direct code numeric constant or a direct code character constant in parentheses. The constant is assigned by the compiler to the appropriate table and the address of the constant is then available for use as an address reference. A maximum of eight characters is allowed when a direct code character constant is used in this context.

| The use of direct code literals allows referencing of constants that would exceed the 16-bit length of the SY instruction field.

Examples

```
LA A0, (56), K3  
LA A1, ('CAT'), K3  
LA A2, (0123), K3  
LA A3, (ntag), K3  
DL A0, (0517632D)
```

Note

When referencing literals, the programmer must assume responsibility for using a proper K designator.

B.2 Direct Code Expressions

Syntax

```
<direct code expression>  
  ::= <direct code numeric constant expression>  
  ::= <direct code address expression>
```

Semantics

A direct code expression can be either a direct code numeric constant expression or a direct code address expression.

Direct code expressions can occur as direct code preset words, or as part of a preset word as in a field of an implied or actual form preset. Direct code expressions can also occur as the Y field of direct code instructions.

B.2.1 Direct Code Numeric Constant Expressions

Syntax

```
<direct code numeric constant expression>  
 ::= <direct short numeric constant> [<numeric operator>  
    <direct short numeric constant> &]
```

Semantics

A direct code numeric constant expression is a series (1 or 2) of single-word numeric constants separated by a numeric operator.

A direct code numeric constant expression evaluates to a single-word numeric constant. Mixed mode constants are not permitted in direct code numeric constant expressions.

Examples

+35*NTAG1

+NTAG2-10

+027/+010

1234 + 5

In these examples, NTAG1 and NTAG2 are both ntag names.

B.2.2 Direct Code Address Expressions

Syntax

```

<direct code address expression>
  ::= <direct code addressable name> [<address offset>]
  ::= [<direct short numeric constant>] + <direct code
      addressable name>
  ::= <direct code literal>
  ::= $ [<address offset>]

```

```

<direct code addressable name>
  ::= <direct code statement name>
  ::= <statement name>
  ::= <function name>
  ::= <table name>
  ::= <procedure name>
  ::= <variable name>
  ::= <switch name>
  ::= $

```

```

<direct code statement name>
  ::= <name>

```

```

<address offset>
  ::= <additive operator> <direct short numeric constant>

```

Semantics

A direct code address expression is an address reference with an optional address offset.

<address offset> - Optional. A single numeric constant that provides the capability of referencing any location in a program.

An address reference can be the character \$, which is the address of the current instruction; a direct code literal, which is the address of a constant assigned to its appropriate constant table; or the address assigned to a direct code addressable name.

The S field of a direct code instruction cannot be specified when there is an addressable name in the Y field.

Examples

TABLE1 + 5 - TABLE1 is a table name

LABEL1 + 7 - LABEL1 is a statement name

\$ -3

('ABCD')

1 + VRBL2 - VRBL2 is a variable name

B.3 Direct Code Statements

Syntax

```
<direct code statement>  
 ::= <direct code comment>  
 ::= <addressable direct code statement> [<direct code  
      comment>]  
 ::= <direct code name>  
 ::= <direct code preset>  
 ::= <direct code directive>
```

```
<direct code comment>  
 ::= . <space> [<character>&]
```

Semantics

All direct code statements begin in column 11 and must be entirely contained in one line image (through column 80).

<direct code comment> - A period followed by a space terminates a direct code statement prior to column 80. Any characters after the space are treated as a comment and are ignored by the compiler.

<direct code name> - A direct code name must start in column 11, consist of one to eight alphanumeric characters, and must not be followed by a period. A space in column 11 indicates the absence of a statement name.

B.3.1 Direct Code Name

Syntax

<direct code name>
 ::= <form name>
 ::= <direct code program statement name>
 ::= <direct code data statement name>

<form name>
 ::= <name>

<direct code program statement name>
 ::= <name>

<direct code data statement name> -
 ::= <name>[<direct code scope modifier>]

<direct code scope modifier>
 ::= *

Semantics

A direct code name can be a direct code program statement name, a direct code data statement name, or a form name.

<direct code program statement name> - A name that can be used as a statement name in direct code address expressions or CMS-2Y(7) statements; its scope is the scope of the name.

<direct code data statement name> - A name which provides a means of referencing addressable direct code statements as data in CMS-2Y(7) statements. The name cannot be used as a direct code program statement name (object of a GOTO phrase).

<direct code scope modifier>

- Optional. An asterisk with no intervening space, which gives the name global scope.

The addressable statement associated with the name and subsequent addressable statements can be referenced as item data units. A direct code data statement name can have global scope either by being in a system data element or by the presence of a direct code scope modifier. Otherwise, the name has local scope or subprogram scope.

The direct code scope modifier may not be used with a name declared in a subprogram data block.

B.3.2 Addressable Direct Code Statement

Syntax

```
<addressable direct code statement>  
 ::= <direct code instruction>  
 ::= <direct code preset>  
 ::= <form preset>
```

Semantics

An addressable direct code statement can be a direct code instruction, a direct code preset, or a form preset.

B.3.2.1 Direct Code Instruction

Syntax

<direct code instruction>
 ::= <operation code> [<operand>@]

<operand>
 ::= <direct code expression>
 ::= <direct code instruction designators>

Semantics

A direct code instruction is a symbolic target machine code instruction that consists of an operation code and its associated operands.

<operation code> - A mnemonic symbol that corresponds to the numeric operation code and the numeric instruction format of the machine code instruction.

<operand> - An expression defining a field of the direct code instruction.

The operands that follow the operation code must be separated by commas, and can further be separated by spaces. Each operation code has a particular format for the operands it requires. This format determines the number of operands, the type of operand, and the order in which the operands follow the operation code. See the instruction repertoire in M-5048 for the format associated with each operation code and the restrictions on each operand. A description of each direct code instruction designator is also contained in M-5048.

For format I instructions, the k-designator can be specified with the special form KX. When this form is used, the compiler will cause the appropriate k-designator value to be used or issue a diagnostic message if there is no appropriate value. If KX is used in a direct code statement, the datum addressed by the statement cannot be forward-referenced.

B.3.3 Direct Code Preset

Syntax

```
<direct code preset>
  ::= <direct code numeric preset>
  ::= <direct code address preset>
  ::= <direct code character preset>
  ::= <implied form>

<direct code numeric preset>
  ::= <single-word numeric preset>
  ::= <double-word numeric preset>

<single-word numeric preset>
  ::= <additive operator> <direct code numeric constant
    expression>

<double-word numeric preset>
  ::= <unary numeric operator><direct long numeric constant>

<direct code address preset>
  ::= + <direct code address expression>

<direct code character preset>
  ::= + <direct code character constant>

<implied form>
  ::= +<direct code numeric preset>@[<direct code address
    expression>]
```

Semantics

A direct code preset is an addressable direct code statement used to assign a numeric value, an address, or a character constant to specific memory locations.

- <single-word numeric preset> - A numeric value that is assigned to the memory location indicated by the current value of the address counter.
- <double-word numeric preset> - A double-word numeric value that is assigned to the two sequential memory locations beginning at the current value of the address counter.

- <direct code address preset> - An address associated with the address expression that is assigned to the memory location indicated by the current value of the address counter.
- <direct code character preset> - A character constant that is packed into sequential memory locations, beginning at the current value of the address counter.
- <implied form> - A form preset which allows for the preset of more than one subfield in one direct code statement.

An address preset or a character preset must be preceded by a plus sign; it has no purpose other than to indicate a preset. A numeric preset must be preceded either by a plus sign or a minus sign. In addition to indicating a preset, the plus or minus sign is also interpreted as an algebraic sign associated with the numeric preset. Note, however, that if the numeric preset is a direct code numeric constant expression, the algebraic sign is associated with only the first term of the expression and not with the entire expression.

In the implied form preset, subfields are separated by commas. The number of subfields must be a divisor of 32 and cannot exceed 16. If a direct code address expression is included as a preset, it must be declared as the last preset because it occupies the lower half word of contiguous bits. Hence, if the statement contains one subfield, the signed subfield value is right-justified in the generated word (32 bits). If the statement contains two subfields, two equal-length signed subfields, 16 bits apiece, are generated with the values right-justified within a generated word. A 4-subfield preset will divide the word into four 8-bit fields. An 8-subfield preset will divide the word into eight 4-bit fields. The first subfield must be signed. Successive subfields may optionally be signed. The absence of a sign implies a positive value. If the variants of this implicit equal subdivision of data words are required, the capabilities of the form directive can be used to derive the desired format.

Examples

```
VRBL CATA A 10 S 0 $
.
.
DIRECT $
.
.
-5, 2, 6, 0, 7, -4, 1, 3
+ 127, 64, -8, -127
+ 16, CATA
```

The first line of presets divides the target machine's word into eight signed 4-bit subfields. The second line of presets divides the word into four signed 8-bit subfields. The third line of presets divides the word into one signed 16-bit subfield in the upper half-word and one unsigned 16-bit address subfield--for CATA--in the lower half-word.

Note

If the user specifies a numeric value which overflows the number of bits allocated for the preset, significance will be truncated and no error diagnostics will be issued. Hence, for example, in the case of 16 2-bit fields, a 0 and -3 are 00 binary, a 1 and -2 are 01 binary, and a 2 and -1 are 10 binary. Overflow considerations for direct code numeric presets are the responsibility of the user.

B.3.4 Direct Code Directives

Syntax

```
<direct code directive>  
  ::= <abs directive>  
  ::= <byte directive>  
  ::= <char directive>  
  ::= <do directive>  
  ::= <form directive>  
  ::= <res directive>
```

Semantics

Direct code directives are statements utilizing the psuedo operation codes of the direct code language.

B.3.4.1 Abs Directive

Syntax

```
<abs directive>  
 ::= [<direct code name>] ABS <direct code name>
```

Semantics

An abs directive is used to request a translation of a compile-time address counter value into a run-time absolute address.

. ABS - A language keyword indicating an abs directive.

Examples

```
ABS CAT
```

The above statement causes the SY value of CAT to be translated into its corresponding 18-bit run-time address. The upper 14 bits of the generated word contain zeros. One practical use for such a directive is to create a value used to load a base register.

B.3.4.2 Byte Directive

Syntax

<byte directive>
 ::= BYTE <b1>[,<b2>]

<b1>
 ::= <decimal integer>

<b2>
 ::= <decimal integer>

Semantics

The byte directive redefines the embedded character size and the number of characters placed in an object word for direct code character strings occurring subsequently within the same CMS-2Y(7) element.

BYTE - A language keyword indicating a byte directive.

<b1> - The number of characters to be packed into an object word.

<b2> - Optional. The number of bits in each byte.

B2 cannot exceed 16 bits. If b2 is omitted from the byte directive statement, the size of the byte field is eight bits.

The product $b1 * b2$ must be less than or equal to 32.

Examples

```
BYTE 6,4  
BYTE 3,10
```

The first directive allows four bits per byte, even though another bit per byte is possible. The second directive allows 10 bits per byte.

B.3.4.3 Char Directive

Syntax

```
<char directive>
    ::= CHAR <parameter constant> @

<parameter constant>
    ::= <C1>, <V1>

<C1>
    ::= <direct code numeric constant>
    ::= <direct code character constant>

<V1>
    ::= <direct code numeric constant>
    ::= <direct code character constant>
```

Semantics

The char directive redefines the 8-bit imbedded character set used for generation of characters coded between apostrophes (character strings).

CHAR - A language keyword indicating a char directive.

<C1> - Defines an octal code (000 thru 137) which is to be redefined.

<V1> - Designates the value to which C1 is redefined.

In the absence of a preceding byte directive an 8-bit character set is assumed. For all character string generation following a char directive, the redefined character code is used until another char directive is encountered, or until a CMS-2 bracket is encountered.

Examples

```
CHAR 0101, 1, 'B', 2, 'C', 3, 0104, 064
+'ABCD'
```

The constant +'ABCD' produces the octal word 000402 001464.

B.3.4.4 Do Directive

Syntax

<do directive>
 ::= [<direct code name>] DO <direct code numeric constant>,
 <direct code constant>

Semantics

The do directive causes a direct code constant to be generated a stipulated number of times.

- | | |
|--------------------------------|---|
| DO | - A language keyword indicating a do directive. |
| <direct code numeric constant> | - Stipulates the number of times that the direct code constant is to be repeated. |
| <direct code constant> | - The constant to be repeated. |

If a direct code name is specified, it applies to the first word of generated data.

Examples

DO 6, 29127

This directive causes six consecutive words to be assigned the value 29127 (70707 octal).

B.3.4.5 Form Directive

Syntax

<form directive>
 ::= <form label> FORM <direct code numeric constant> @

<form label>
 ::= <name>

Semantics

The form directive describes a special word format with fields defined within the word.

- FORM - A language keyword specifying a form directive.
- <form label> - A name used in place of a direct code operation to describe the format of a computer word.
- <direct code numeric constant> - Specifies the width of a field in bits.

The total number of bits specified must be less than or equal to 64 and the number of fields is limited to 16. Each field must be less than or equal to 32 bits in length.

Examples

```
STRINGX FORM 6,3,3,9,6  
          STRINGX 44, 3, 4, 129, -8
```

Each value of STRINGX occupies the number of bits indicated in the respective position within the form directive. For example, the value 44 occupies bits 31 through 26. All of the numeric constants shown produce a word of the value 26161007340 octal.

Note

If fewer fields are coded than the number listed in the form directive, they are left-justified. Coded fields in excess of the specified number result in an object error diagnostic message.

B.3.4.6 Res Directive

Syntax

<res directive>
 ::= RES <direct code numeric constant>

Semantics

The res directive is used to reserve a specified number of sequential memory locations, beginning at the current value of the address counter.

RES - A language keyword specifying a res directive.

<direct code numeric constant> - A positive number specifying the number of words to be reserved in the range [0,65536].

Examples

```
VX EQUALS 64 $
```

```
RES 4096  
RES VX  
RES 0
```

In the first reservation 10,000 octal cells are allocated. In the second reservation 100 octal cells are allocated. In the third reservation no cells are allocated.

B.3.5 Form Preset

Syntax

```
<form preset>
  ::= <form label> <direct code numeric constant>@ [<direct
        code name>]
```

Semantics

The form preset is an addressable direct code statement, used to assign a value to specified memory locations using the fields defined in a form directive.

- <form label> - The name of a form directive describing the fields within a word to be preset.
- <direct code numeric constant> - A value to be preset in the corresponding field.
- <direct code name> - Optional. Specifies an address to be preset in the last field of the word.

Only constants are accepted in form preset reference sub-fields, with the exception of a name appearing in the last sub-field where that sub-field size is defined as 16 bits or greater. When a value appearing in a form preset reference sub-field requires more bits than were defined in the form preset declaration, the leftmost bits of the value will be truncated when packing the resulting constant.

Examples

```
UF1  FORM  6,3,3,3,1,3,13
      .
      .
LOAD  UF1  10,5,3,6,1,2,643
```

The form statement defines the fields of a target machine format 1 instruction. The statement at LOAD is equivalent to

```
LOAD  LA,A5,*643,K3,B6,S2
```

APPENDIX C

TARGET MACHINE INTERFACES

C.1 Compiled Forms of Inputlist and Outputlist Declarations

Each inputlist item and outputlist item is compiled into one or two control words followed by a sequence of code. In general, execution of the sequence of code will cause a value to be loaded into register A0 or register pair A0-A1 (outputlist), or to be stored from that register or register pair (inputlist). Details of the code sequence vary, depending on the first control word, and are explained below.

The phrase access the value is used to mean either load the value into the register(s) or store it from the register(s). The code sequence used to access the value is called the accessing sequence.

The code sequence can use register B7 as a live register. It is the responsibility of the conversion routines to save and restore that register.

The code sequence consists of three subsequences. If the inputlist item or outputlist item is multivalued (a table or an untyped item-area), the sequence begins with an initializing subsequence. This subsequence prepares for multiple executions of the remainder of the sequence, in order to access the multiple values; it must be executed once. Next (or first, if there is no initializing subsequence) is the prologue subsequence. This subsequence must be executed before the conversion routine accesses each value. The epilogue subsequence is last. It must be executed after the conversion routine accesses each value. Thus, for example, if an outputlist item is a numeric typed table, the initializing subsequence might contain the code to prepare for picking up the first item of the table, the prologue subsequence might contain the code to place the value of an item of the table into register A0, and the epilogue code might contain the code to prepare to pick up the next item of the table.

In most cases, the code sequence will not contain compare or branch instructions; the conversion routines must execute the subsequences in these cases by using the target machine XR command or some equivalent technique. In certain low-usage cases the prologue or epilogue subsequence (but never the initializing subsequence) might require compare or branch instructions. In these cases the subsequences will be in the form of a procedure which can be executed by use of an LBJ B6 instruction. The two

/(U) CM2Y-MAN-PGR-M5049-R04C0

types of subsequence will be indicated by flag settings in the control words. The two types of code generation are used in order to generate more compact code in the most common cases.

For multivalued items, it is the responsibility of the conversion routine to keep track of the number of items processed and the beginning of the prologue subsequence.

C.1.1 Control Words

The first control word has two formats. The most common format is:

```

31 27 26 22 21          15 14          8 7          0
*****
* TYPE *FLAGS * NPROLINS * NEPILINS * AUXVAL *
*****

```

When the value of TYPE is zero, the format is:

```

31 27 26 22 21          16 15          0
*****
*TYPE=0*FLAGS * 0 * ADR *
*****

```

The second control word, when present, is of the following format:

```

31          23 22          16 15          0
*****
* 0 * NINITINS * NVALS *
*****

```

The field FLAGS consists of five 1-bit flags (Boolean variables). They are:

- Bit 26 - LITEMFL 1 for the last item of the list; 0 for all others.
- Bit 25 - MULTVFL 1 for multivalued items (tables, untyped item-areas); 0 for all others. (The second control word appears only if MULTVFL = 1.)
- Bit 24 - PTHUNKFL 1 if the prologue subsequence must be executed as a subprogram (using an LBJ B6 instruction); 0 if it must be otherwise executed.
- Bit 23 - ETHUNKFL 1 if the epilogue subsequence must be executed as a subprogram (using an LBJ B6 instruction); 0 if it must be otherwise executed.
- Bit 22 - Unused

The attributes and meaning of the field AUXVAL depend on the value of TYPE. The attributes and meanings of the other fields of the control words are:

ADR	I 16 U	The sy-address of an inputlist or outputlist (appears only if TYPE = 0).
NINITINS	I 7 U	The number of target machine words occupied by the following initializing subsequence (present only if MULTVFL = 1).
NEPILINS	I 7 U	The number of target machine words occupied by the following epilogue subsequence.
NPROLINS	I 7 U	The number of target machine words occupied by the following prologue subsequence.
NVALS	I 16 U	The number of values in a multivalued item (present only if MULTVFL = 1). The value will be 0 for tables whose table subscript declaration is an ltag name, or which have a major index. In these cases, the number of values will be in register A0 as an unsigned integer value after execution of the initializing subsequence.
TYPE	I 5 U	An indicator of the general attributes of the inputlist or outputlist item.

The values of TYPE, its interpretation, and the corresponding attributes of AUXVAL are:

<u>TYPE</u>	<u>Meaning</u>	<u>Auxiliary Value</u>
0	Inputlist/outputlist	Not applicable
1	Indirect inputlist/outputlist	Zero (unused)
2	Single precision fixed-point, unsigned	I 8 S - scaling
3	Single precision fixed-point, signed	I 8 S - scaling
4	Double precision fixed-point, unsigned	I 8 S - scaling
5	Double precision fixed-point, signed	I 8 S - scaling
6	UYK-7 Floating-point	Zero (unused)
7	Character	I 8 U - number of characters (for the meaning of 0, see below)

8	Boolean	Zero (unused)
9	Single precision untyped	Zero (unused)
10	Double precision untyped	Zero (unused)
11	Status	I 8 U - number of values mod 256
12	UYK-43 single-precision floating-point	
13	UYK-43 double-precision floating-point	

The meaning of the accessing sequence depends only on the value of TYPE. The various meanings are:

- TYPE = 0 - (Special) No sequence follows.
- TYPE = 1 - The address of the inputlist or outputlist is loaded into register A0.
- TYPE = 2,3,8,9,12 - The value will be loaded into register A0 (outputlist) or stored from register A0 (inputlist).
- TYPE = 4,5,6,10,13 - The value will be loaded into register pair A0-A1 (outputlist) or stored from register pair A0-A1 (inputlist).
- TYPE = 7 - For an outputlist, an IWC1 that points to the first character of the value will be loaded into register A0 when the prologue subsequence is executed. For an inputlist, an IWC1 that points to the first character of the receptacle will be loaded into register A0 when the prologue subsequence is executed. The epilogue subsequence will be present only for multivalued items, and will then consist only of code to increment the loop index.

If the number of characters cannot be determined at compilation time, AUXVAL will be set to 0, and the number of characters will be in register A1 as an unsigned integer value after execution of the prologue subsequence.

TYPE = 11 -

The value will be loaded into register A0 (outputlist) or stored from register A0 (inputlist), the address of a table of character equivalents of the values will be loaded into register A1, and the number of values will be loaded into register A2. The table consists of one 8-character entry for each value, and is in ascending order by value. (The value in AUXVAL is present only for compatibility with previous forms of inputlist and outputlist.)

C.2 Compiled Form of Stringforms

Stringforms are compiled into sequences of bytes (quarterwords). These sequences are made up of subsequences, each of which consists of a 1-byte control value followed by a sequence of bytes whose length and meaning depend on the control value. These subsequences are defined by:

<u>Control Value (Decimal)</u>	<u>Meaning</u>	<u>Following Sequence</u>
0	Start of stringform	None
1	Left parenthesis	None
2	Right parenthesis	None
3	Repeat value	1-byte repeat value
4	Decimal without exponent (D)	1-byte field width, 1-byte fraction size
5	Decimal with exponent (D)	1-byte field width, 1-byte fraction size, 1-byte exponent size
6	Decimal integer (I)	1-byte field width
7	Binary (B)	1-byte field width
8	Octal (O)	1-byte field width
9	Hexadecimal (X)	1-byte field width
10	Character (C)	1-byte field width
11	Status (E)	1-byte field width
12	unused	
13	Character constant	1-byte constant length, the value of the constant
14	Blanker (Z)	1-byte blank count
15	Absolute tab (T)	1-byte column number
16	Tab right (T+)	1-byte tab count
17	Tab left (T-)	1-byte tab count
18-34	unused	
35	Repeat value	2-byte repeat value
36	Decimal without exponent (D)	2-byte field width, 1-byte fraction size
37	Decimal with exponent (D)	2-byte field width, 1-byte fraction size, 1-byte exponent size
38	Decimal integer (I)	2-byte field width
39	Binary (B)	2-byte field width
40	Octal (O)	2-byte field width
41	Hexadecimal	2-byte field width
42	Character (C)	2-byte field width
43	Status (E)	2-byte field width
44	unused	
45	Character constant	2-byte constant length, the value of the constant
46	Blanker (Z)	2-byte blank count

/(U) CM2Y-MAN-PGR-M5049-R04C0

47	Absolute tab (T)	2-byte column number
48	Tab right (T+)	2-byte tab count
49	Tab left (T-)	2-byte tab count
50-254	unused	
255	End of stringform	None

APPENDIX D
LISTING FORMATS

Appendix D consists of samples and descriptions of the following listing formats generated by the CMS-2Y(7) compiler as a result of various source and object requests:

- Compiler Source Listing
- Source Listing
- Compiler Diagnostic Listing
- Symbol Analysis Listing
- Source Mnemonic Listing
- Local Address Cross Reference
- Local Source Cross Reference
- Global Address Cross Reference
- Global Source Cross Reference
- Compile Summary
- SHARE/7 System Summary
- Batch System Summary

D.1 Source Listings

This section describes the various source listings which may be requested in CMS-2Y(7) compilations. Unless otherwise specified, all numerics in this section hold decimal values. Sample figures indicate possible outcomes and placement of data on the page.

All source specifications result in the compiler proceeding through the source analysis phase. Thus, source diagnostics are generated, which are described in Appendix A. Refer to paragraph 9.2.1 for a more detailed description of source specification declarations.

D.1.1 Compiler Source Listing

See Figure D-01 for an example of a compiler source listing.

The compiler source listing is output from the syntax analyzer phase of the compiler and provides a record of input to the compiler.

The major page header contains the system specification: CMS-2Y(7). It also contains the revision number, the date, the page number, and the title: Compiler Source Listing.

The first minor page header is set off by asterisks on either side. It contains the element type (System, SYS-DD, SYS-PROC), the element name, and the element number.

The second minor page header is defined below. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- CARD ID - Corresponds to card columns 1-10, which identify the card name and statement number. (4-13)
- SOURCE STATEMENT - Corresponds to card columns 11-80, which contain the source statements. (14-83)
- ERROR CONDITION - Description of an error condition, immediately following the statement containing the condition, and including the abbreviation SE (source error) and/or SW (source warning), error number and error statement. (85-119)
- ^ - Caret indicates the compiler's best attempt to specify the location of the error. (14-83)

D.1.2 Source Listing

See Figure D-02 for an example of a source listing.

Output by the generator phase of the compiler, the source listing provides a record of input to the compiler.

The major page header contains the element number, the system or element name, and the element type (System, SYS-DD or SYS-PROC). It also contains the revision number, the date, the page number, and the title: Source Listing.

The minor page header is defined below. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- ERROR - Number indicating errors found in the adjacent source statement. See Appendix A for a list of the errors and meanings associated with these numbers. (1-5)
- AC - Address counter (in octal) for current location of instruction or data unit. (10-11)
- S - Indication (in octal) of the number of 20,000 (octal) word multiples (incremented under the key word LOCATION) occurring before the current line. Loosely referred to as the base register. (12)
- LOCATION - Current location (in octal) of instruction or data unit, relative to (i.e., offset from) the start of the address counter. (14-21)
- LABEL - Eight-character identifier for the adjacent source statement, or the first eight characters of the input statement. (36-43)
- STATEMENT - The remainder of the CMS-2 input statement. (44-105)
- CID - Card identification. Corresponds to card columns 1-4, which are filled by the programmer, usually to identify program subdivisions. (109-112)
- SID - Statement identification. Corresponds to card columns 5-8, which are filled by the programmer with statement sequence numbers. (114-117)

CR - Correction identification. The remaining two numbers or characters of the card and statement ID. (118-119)

When the SCR option is requested, the key words CID, SID and CR are replaced by the key words SCR and LIB. The following description is also applicable to the source mnemonic listing with the SCR option requested.

SCR - The compiler-generated source cross reference number used to identify each source statement and referenced in the local source cross reference. (107-112)

LIB - Identical to the SID and CR characters on the source listing option. (114-119)

D.2 Object Listing

This section describes the various object listings which may be requested in CMS-2Y(7) compilations. Unless otherwise specified, all numerics in this section hold decimal values. Sample figures indicate possible outcomes and placement of data on the page. The symbol [OR] is used to indicate the choices within each column.

All object specifications result in the compiler proceeding through the object-generation phase. Thus, requested listings will include object diagnostics. These diagnostics are described in Appendix A. Refer to paragraph 9.2.2 for a more detailed description of object specification declarations.

Object specifications result in the requested output being generated for the major header and each system element in the order in which they are compiled. Global cross-reference listings are generated for the entire system block and appear after the end-system statement in the output listing. Refer to Figure 9-01 for output results with various combinations of options parameters.

D.2.1 Compiler Diagnostic Listing

See Figure D-03 for an example of a compiler diagnostic listing.

The major page header information contains the system specification: CMS-2Y(7). It also contains the revision number, the date, the page number, and the title: Compiler Diagnostic Listing.

There may be more than one minor header to a page. The minor header is set off by asterisks on either side. It contains the element type (System, SYS-DD, or SYS-PROC), the element name and the element number.

The second page header is defined below. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- | | |
|--------------------------|--|
| CARD IMAGE | - The image on the card. Columns 1-10 identify both the card name and statement number; the remaining columns contain the actual statement. (3-98) |
| ^ | - Caret indicates the compiler's best attempt to specify the location of the error. (29-98) |
| (Error Type) = | - SOURCE ERROR or SOURCE WARNING refers to errors found during the source analysis phase of a compilation (see page A-1); OBJECT ERROR or OBJECT WARNING refers to errors found during the object phase of a compilation (see page A-20). (3-17) |
| (Error Number) | - Number corresponding to description of error (see pages A-2 through A-17 and pages A-20 through A-23). (20-22) |
| (Error Description Text) | - Error message describing the error in a text form. (27-52) |
| COL. NO. | - The column number pointed to by the CARET. May be N/A for "not applicable". (57-70) |

D:2.2 Symbol Analysis Listing (SA)

See Figure D-04 for an example of a symbol analysis listing.

The major page header information contains the element number, the element name, and the element type (SYS-DD or SYS-PROC). It also contains the date, the page number, and the title: Symbol Analysis.

Symbols for each system element are grouped along with group-specific information in the following categories and are divided by a line of asterisks. The name of each category begins in column 20 of the listing. Each group lists the data alphabetically by symbol name.

The values in parentheses following each definition indicate the columns on the listing where the information appears. Some columns contain optional data and will remain blank if not used.

1 2 3 4 5 6 7 8 9 0 1 1
 123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

ELEMENT ddd lcccccc SYS-1111 SYMBOL ANALYSIS mm/dd/yy PAGE ddddd

PROCEDURES - FUNCTIONS

NAME	TP	PASS	REN	INPUT PARAMETERS	OUTPUT PARAMETERS	EXITS
lcccccc	1	1111	111	lcccccc lcccccc lcccccc	lcccccc lcccccc lcccccc	lcccccc lcccccc

LOCAL INDEXES

NAME	REGISTER	PROCEDURE
lcccccc	dd	lcccccc

D-15

1 2 3 4 5 6 7 8 9 0 1 1
 123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

LEGEND	
d = DECIMAL DIGIT	m = MONTH
o = OCTAL DIGIT	d = DAY
c = ALPHANUMERIC CHARACTER	y = YEAR
1 = ALPHA CHARACTER	

Figure D-04. An Example of a Symbol Analysis Listing. (Page 4 of 4)

(U) CM2Y-MAN-PGR-M5049-R04C0

D.2.2.1 Files

For a description of the file declaration, see paragraph 4.23. Symbol analysis column headings are defined as follows:

- NAME - File name. (1-8)
- | TYPE - File type. (12-15)
 - H = Character
 - B = Internal format
- | MODE - File structure. (18-21)
 - R = Rigid record length
 - V = Variable record length
 - S = Stream organization
- HRDWR - Specifies the hardware device containing the file. (24-28)
 - READ = Card reader
 - PNCH = Card punch
 - PRNT = Printer
 - MT## = User defined device
 - OCM = Operator's Communication Medium
- MX.RCD - The length of the buffer associated with the file. (31-36)
- SIZE
- MX.NO. - The maximum number of records permitted in any subfile of the file. (39-44)
- RCDS
- NO. - The number of status constants used in testing I/O conditions. (48-53)
- STATES
- | PROCEDURE- The name of the parent procedure or function in whose subprogram data block (SUB-DD) the file is defined. (55-63)

D.2.2.2 Formats

For a description of the format declaration, see paragraph 4.24. Symbol analysis column headings are defined as follows:

- NAME - Format name. (1-8)
- EXT - External Specification. (12-14)
 - D = EXTDEF (defined in this element)
 - R = EXTREF (referenced in this element)The name has global scope in both cases.
- PROCEDURE - The name of the parent procedure or function in whose subprogram data block (SUB-DD) the format is defined. (23-31)

D.2.2.3 Types

For a description of the type declaration, see paragraph 4.4. Symbol analysis column headings are defined as follows:

- NAME - Type name. (1-8)
- PACK - Indicates NONE, MEDIUM or DENSE. (12-15)
- WDS/
ITEM - The number of words per item as indicated in the type declaration. (20-24)
- EXT - External specification. (27-29)
 - D = EXTDEF
 - R = EXTREF
- FIELD
NAME - Alphabetical list of fields associated with the type. (31-38)
- TP - Type of item-typed type, or field. (41-42)
 - A = Fixed point H = Hollerith
 - F = Floating point I = Integer
 - B = Boolean
- SN - Sign of item-typed type, or field. (46-47)
 - U = Unsigned
 - S = Signed
- START
BIT - The bit position of the leftmost bit of the field. (52-56)
- WORD
LOC - The number of the word in which the leftmost bit of the field is allocated. (61-64)
- FR
BT - The number of fractional bits of a fixed point type or field. (The radix is implied and therefore is not assigned a bit.) (66-69)
- NO. BITS
OR CHARS - The number of bits for numeric typed type or field; the number of characters for Hollerith typed type or field. (75-82)
- PROCEDURE - The name of the parent procedure or function in whose subprogram data block (SUB-DD) the type or field is defined. (88-96)

D.2.2.4 Tables

For a description of the table declaration, see paragraphs 4.8 and 4.9. Symbol analysis column headings are defined as follows:

NAME	- Table, like-table, subtable or item-area name. (1-8)
ASSOC NAME	- Parent table name for like-table, subtable or item-area. (10-17)
TP	- The parent table or associated table type. (25-26) V = Vertical L = Like-table H = Horizontal S = Subtable A = Array I = Item-area
PACK NDIM	- Indicates NONE, MEDM, or DENS if the table is compiler-packed; shows number of dimensions for an array. (28-31)
WDS/ITEM	- The number of words per item as declared in the parent table. (33-37)
NO.ITEMS DIM.SIZE	- Number of items for tables (can be a constant or an ntag, or ltag name). For arrays, indicates the dimension size (number of subscript declarations). (39-46)
ADD MOD	- Addressing mode. (49-51) DIR = Direct IND = Indirect
VLT AC	- The value (in octal) of the variable length table address counter. (53-55)
INDEX NAME	- Name of parent table's major index. (57-64)
EXT	- External specification. (65-67) D = EXTDEF R = EXTREF
START ITEM	- First item of the associated subtable. (69-73)
FIELD NAME	- Alphabetical list of fields associated with the table. (75-82)

- TP - Field type. (83-84)
A = Fixed point S = Status
F = Floating point H = Hollerith
B = Boolean I = Integer
- SN - Sign of numeric table or field. (86-87)
U = Unsigned
S = Signed
- START BIT - The bit position of the leftmost bit of the field. (89-92)
- WORD LOC - The number of the word in which the leftmost bit of the field is allocated. (94-96)
- FR BT - The number of fractional bits of a fixed point field. (The radix is implied and therefore is not assigned a bit.) (98-101)
- NO. BITS OR CHARS - The number of bits for numeric type field, number of characters for Hollerith type field, or number of states for status type field. (103-110)
- PROCEDURE - The name of the parent procedure or function in whose subprogram data block (SUB-DD) the table is defined. (111-118)

D.2.2.5 Switches

For a description of the switch declaration, see paragraphs 4.21 and 4.22. Symbol analysis column headings are defined as follows:

NAME	- Switch name. (1-8)
TP	- Type of switch. (11-13) S = Single or double index switch ITS = Item switch P = Single or double index procedure switch ITP = Item procedure switch
NO. PTS	- Number of switch points. (17-19)
COMPARED VARIABLE	- Name of variable compared with item switch constant value. (24-31)
EXT	- External specification. (34-36) D = EXTDEF R = EXTREF
INPUT PARAMETERS	- List of names of formal input parameters associated with the procedures in the procedure switch list. (40-71)
OUTPUT PARAMETERS	- List of names of formal output parameters associated with the procedures in the procedure switch list. (76-109)
PROCEDURE	- The name of the parent procedure or function in whose subprogram data block (SUB-DD) the switch is defined. (111-119)

D.2.2.6 Variables

For a description of the variable declaration, see paragraph 4.6. Symbol analysis column headings are defined as follows:

- | | | |
|-----------------------------|--|--|
| NAME | - Variable name. (1-8) | |
| TP | - Variable type. (10-11)
I = Integer
A = Fixed point
F = Floating point | B = Boolean
S = Status
H = Hollerith |
| SN | - Sign of numeric variable. (15-16)
U = Unsigned
S = Signed | |
| START
BIT | - Leftmost (starting bit) of the variable.
(Note: When the variable is signed, this would be the sign bit.) (21-25) | |
| NO.CHRS,BITS
OR ST.CONST | - Number of characters if Hollerith; number of bits if numeric or Boolean; number of states if status type variable. (30-40) | |
| FR
BT | - Number of fractional bits for scaled variable. (42-45) | |
| EXT | - External specification. (49-51)
D = EXTDEF
R = EXTREF
M = MODEVRBL | |
| TYPE OR
TABLE | - The name of the parent type or table associated with this variable. (55-12) | |
| PROCEDURE | - The name of the parent procedure or function in whose subprogram data block (SUB-DD) the variable is defined. (24-32) | |

D.2.2.7 Inputlists/Outputlists

For a description of the inputlist and outputlist declarations, see paragraphs 4.26 and 4.27. Symbol analysis column headings are defined as follows:

- NAME - Inputlist or outputlist name. (1-8)
- TP - Type of declaration. (11-13)
 - IN = Inputlist
 - OUT = Outputlist
- EXT - External specification. (17-19)
 - D = EXTDEF
 - R = EXTREF
- PROCEDURE - The name of the parent procedure or function in whose subprogram data block (SUB-DD) the inputlist or outputlist is defined. (24-32)

D.2.2.8 Stringforms

For a description of the stringform declaration, see paragraph 4.25. Symbol analysis column headings are defined as follows:

NAME - Stringform name. (1-8)

EXT - External specification. (13-15)
D = EXTDEF
R = EXTREF

PROCEDURE - The name of the parent procedure or function in whose subprogram data design (SUB-DD) the stringform is defined. (24-32)

D.2.2.9 Procedures and Functions

For a description of the procedure and function declarations, see paragraphs 4.18, 4.19, and 4.20. Also, for information on the parameter passage directive, see paragraphs 5.2 and 6.1.1.6. Symbol analysis column headings are defined as follows:

NAME	- Procedure or function name. (1-8)
TP	- Type of declaration. (11-12) E = EXEC-PROC P = Procedure F = Function
PASS	- The parameter passage specification. (15-18) CALL = Register - calling only REG = Register only DIR = Direct passage
REN	- Indicates that this block of code was defined in a SYS-PROC-REN. (20-22)
INPUT PARAMETERS	- List of names of formal input parameters associated with the procedure, EXEC-PROC, or function. (24-55)
OUTPUT PARAMETERS	- List of names of formal output parameters associated with the procedure. (60-91)
EXITS	- List of names of formal exit parameters associated with the procedure. (96-115)

D.2.2.10 Local Indexes

| For a description of the local index declaration, see paragraph 4.17. Symbol analysis column headings are defined as follows:

- | NAME - Subprogram local index name. (1-8)
- REGISTER - The number of the AN/UYK-7 or AN/UYK-43 hardware index register corresponding to the local index. (12-19)
- | PROCEDURE - The name of the procedure or function in whose subprogram data block (SUB-DD) the local index is defined. (24-32)

D.2.3 Source Mnemonic Listing

See Figure D-05 for an example of a source mnemonic listing.

The source mnemonic listing provides a record of input to the compiler. In most cases, each source line is followed by its mnemonic representation.

The major page header contains the element number, the system or element name, and the element type (System, SYS-DD, SYS-PROC). It also contains the revision number, the date, the page number, and the title: Source Mnemonic Listing.

The minor page header is defined below. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- ERR - The number indicating errors found in the adjacent source statement. See Appendix A for a list of the errors and meanings associated with these numbers. (1-3)
- AC - Address counter (in octal) for current location of instruction or data unit. (5-6)
- S - Indication (in octal) of the number of 20,000 (octal) word multiples (incremented under the key word LOC) occurring before the current line. Loosely referred to as the base register. (8)
- LOC - Current location (in octal) of instruction or data unit, relative to (i.e., offset from) the start of the address counter. (10-14)
- FUNC - The first half of the data or instruction containing the operation. (16-21)
- S - Indication (in octal) of the number of 20,000 (octal) word multiples (incremented under the key word LOC) occurring before the operand location. Loosely referred to as the base register. (23)
- LOC - Relative memory location (in octal) of the operand of the instruction. Also appearing under this key word is the K-designator, indicating to what portion of a word the variable is assigned. (24-29)
- AC - Address counter (in octal) of the operand on the instruction. (31-32)

- X - Declaration (scope) modifier (see pages 4-2 through 4-4). (34)
 - R = Operand is global, defined in another element, and referenced in this element (EXTREF).
 - D = Operand is global, and defined in this element (EXTDEF).
 - T = Operand is defined in another element that cannot be assigned a permanent base register (TRANSREF).
- LABEL - Eight-character identifier for the adjacent source statement, or the first eight characters of the input statement. It can be user-coded (CMS-2) or compiler-generated (source mnemonic). The latter is indented one space. (36-43)
- STATEMENT - The remainder of the CMS-2 input or generated source mnemonic statement. The latter is indented several spaces. (44-105)
- CID - Card identification. Corresponds to card columns 1-4, which are filled by the programmer, usually to identify program sub-divisions. (109-112)
- SID - Statement identification. Corresponds to card columns 5-8, which are filled by the programmer with statement sequence numbers. (114-117)
- CR - Correction identification. The remaining two numbers or characters of the card and statement ID. (118-119)

When the SCR option is requested, the key words CID, SID and CR are replaced by the key words SCR and LIB. The following description is also applicable to the source listing with the SCR option requested.

- SCR - The compiler-generated source cross reference number used to identify each source statement and referenced in the local source cross reference. (107-112)
- LIB - Identical to the SID and CR characters on the source mnemonic listing option. (114-119)

D.2.4 Local Address Cross Reference Listing (CR,CRL)

See Figure D-06 for an example of a local address cross reference.

The major page header information contains the system element number, the element name, and the element type (SYS-DD or SYS-PROC). It also contains the date, the page number, and the title: Local Cross Reference.

Names defined or used within the specified system element are listed alphabetically, along with the address where allocated and addresses of all local references.

The column headings for local cross reference are defined in the following paragraphs. The values in parentheses following each definition indicate the columns on the listing where the information appears.

AC - The address counter value (in octal) associated with the allocation of the data unit listed in the LABEL column. (1-2)

S - The base register value (in octal) associated with the allocation of the data unit listed in the LABEL column. (4)

LOC - The location (in octal) allocated to the data unit listed in the LABEL column. A location of all sevens denotes an allocation error. (6-10)

Note: The AC S LOC combination comprises the referenceable address referred to here as the allocated or defining address in the source or source mnemonic listing outputs. For a further description, see paragraphs D.1.2 and D.2.3.

If the name listed under the LABEL column is not allocated locally, the defining address will be zeros and an R or T will appear in the EXT column.

LABEL - This column contains an alphabetical list of programmer-assigned names which have been defined or used in the current system element designated in the page header. Compiler-generated instruction words are listed at the end. Asterisks in this column indicate an allocation error in referencing an identifier. (12-19)

EXT - External specification
D = EXTDEF (defined in this element)
R = EXTREF (referenced in this element)
T = TRANSREF (defined in another system element which cannot be assigned a permanent base register)

A D, R, or T indicates the name has global scope. (21-23)

REFERENCES - A list of octal addresses within the current system element designated in the page header, that shows the location of each reference to the name in the label column. The word NONE appears if there were no references to the name. The references are given in order of occurrence in the same format (AC S LOC) as the defining address of the label. When the number of references exceeds six, the remainder are printed on succeeding lines. (28-117)

There are some instances which will cause the local address cross reference listing to be incomplete, in which case the appropriate error message will appear and cross referencing for the current element will terminate. For further description of the cross reference error messages, see paragraph A.4.

D.2.5 Local Source Cross Reference Listing (SCR, SCRL)

See Figure D-07 for an example of a local source cross reference.

The major page header information contains the system element number, the element name, and the element type (SYS-DD or SYS-PROC). It also contains the date, the page number, and the title: Local Source Cross Reference.

Names defined or used within the specified system element are listed alphabetically with a source statement location of the definition and all local references included for each.

The column headings for local source cross reference are defined in the following paragraphs. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- SYMBOL** - Any programmer-assigned name which has been defined or used in the current system element designated in the page header. Asterisks in this column indicate an invalid pointer was encountered during cross reference compilation. Fields and local index names are indented two spaces under the table, subtable, like-table, item-area, or procedure in which they are defined or used. (1-8)
- TYPE** - The type assigned to the name in the symbol column. In addition to all addressable local names, this list may contain the following names: fields, cswitch flags, local index names, system index names, means and exchange string names, ntag, ltag and rtag names, header names, system element names, local data block names, system names, pooling names, and form labels. If the compiler encounters an invalid class or an undefined identifier, a line of eight asterisks will appear in the type column. If option MODEVRBL is requested, the type MODEVRBL will appear for implicitly defined names and undefined identifiers. (12-23)
- DEFINED** - Numeric values appearing in this column refer to the compiler-generated statement number assigned to the source text line on which the symbol is defined. If the name is not defined locally (i.e., within the current system element), one of the following will appear: (25-35)

1 2 3 4 5 6 7 8 9 0 1 1														
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789														
ELEMENT	ddd	lcccccc	SYS-1111	LOCAL SOURCE CROSS REFERENCE								mm/dd/yy	PAGE	dddd
SYMBOL	TYPE	DEFINED	REFERENCES											
lcccccc	1111111111	dddd	dddd dddd	dddd	dddd	dddd	dddd	dddd	dddd	ddd	dddd	dddd	dddd	
[OR]	[OR]	[OR]	[OR] [OR]	[OR]	[OR]	[OR]	[OR]	[OR]	[OR]	[OR]	[OR]	[OR]	[OR]	
*****	*****	HEADER	dddd* dddd*	dddd*	dddd*	dddd*	dddd*	dddd*	dddd*	dddd*	dddd*	dddd*	dddd*	
		[OR]	[OR]											
		EXTRNL	NONE											
		[OR]												
		lcccccc(C)												
		[OR]												
		DUPDEF												

D-34

1 2 3 4 5 6 7 8 9 0 1 1

123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

LEGEND	
d = DECIMAL DIGIT	m = MONTH
o = OCTAL DIGIT	d = DAY
c = ALPHANUMERIC CHARACTER	y = YEAR
l = ALPHA CHARACTER	

Figure D-07. An Example of a Local Source Cross Reference Listing.

- HEADER - Defined in the system's major header.
- EXTRNL - Defined in another system element within this compile. (Note: A global source cross reference listing can provide the name of the defining element.)
- (Poolname) (C) - The name of the compool in which this symbol was defined. The (C) indicates this symbol was defined in a compool.
- DUPDEF - Symbol name is a duplicate identifier.

Note: Compiler-generated statement numbers are assigned in consecutive numeric order, beginning with one, for the major header and each system element and appear in the SCR column of source and source mnemonic output listings, along with the associated program text line. These statement numbers are also used in the local source cross reference for listing references to the symbol.

REFERENCES - A list of the compiler-generated source cross reference line numbers (see note from preceding paragraph) corresponding to the source statements in which the symbol name has been referenced. If there is more than one reference per source line, each reference will be indicated. A reference followed by an asterisk indicates the value of the referenced symbol has been modified. If there are more than ten references to a symbol, the remaining references are printed on succeeding lines. If there are no references to the symbol name in the current system element, the word NONE will appear. (35-115)

There are some instances (for example, an undersized symbol table) which will cause the local source cross reference listing to be incomplete, in which case the appropriate error message will appear and source cross referencing for the current element will terminate. For further description of the cross reference error messages, see paragraph A.4.

D.2.6 Global Address Cross Reference (CR, CRG)

See Figure D-08 for an example of a global address cross reference.

The page header information contains the system name, the date, the page number, and the title: Global Cross Reference.

For each addressable global name, the defining system element and each system element which contains a reference to the name is listed.

The column headings for global cross reference are defined in the following paragraphs. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- EXT - This column will remain blank if the item in the LABEL column is defined within the compile-time system. It will contain an R if the symbol definition is external to the compile-time system. (7-9)
- LABEL - An alphabetical listing of addressable global names. (12-19)
- DEFINED IN - The name of the system element (Header, SYS-PROC name, or SYS-DD name) in which the label is allocated. If allocation is within a compool element, the compool name is given with (C) to the right. (23-33)
- REFERENCED BY - An alphabetical list of the elements which reference the label name. If there are more than seven elements listed, the remaining are printed on succeeding lines. If there are no references to the label name, the word NONE appears. (36-115)

The global address cross reference will not be generated when a global source cross reference is concurrently requested. In addition, certain error conditions could cause an incomplete cross reference and generation of an error message. For further description of the cross reference error messages, see paragraph A.4.

D.2.7 Global Source Cross Reference (SCR, SCRG)

| See Figure D-09 for an example of a global source cross reference.

| The major page header information contains the system name, the date, the page number, and the title: Global Source Cross Reference.

Global names defined or used in the system block are listed alphabetically, indicating also the name of the defining system element. In addition to all addressable global names, this list may contain the following names: fields, cswitch flags, system index names, means and exchange string names, ntag, ltag and rtag names, header names, system element names, local data block names, system names, and pooling names. Associated with each name is a list of system elements which reference the name. The system elements which modify the name are indicated by a trailing asterisk.

| The column headings for global source cross reference are defined in the following paragraphs. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- EXT - This column will remain blank if the item in the LABEL column is defined within the compile-time system. It will contain an R if the symbol is defined in another compile-time system. (7-9)
- LABEL - An alphabetical list of qualifying global names defined or used within the system block. Fields are indented and listed alphabetically under the associated tables, subtables, like-tables, or item-areas in which they are defined or used. (12-19)
- DEFINED IN - The system element name (or compool name followed by a (C)) which contains the definition of the item in the label column. If the definition occurred in the header then HEADER appears. EXTRNL is printed when the name is EXTREFed. If there is an allocation error, the label will not be included in the cross reference. Note: allocation errors are included in the source, source mnemonic, and compile summary outputs. (23-33)

REFERENCED BY - A list of the system element names in which the name has been referenced. A reference followed by an asterisk indicates the value associated with the name has been modified. If there are more than six references to a name, the remaining references are printed on succeeding lines. Referencing element names appear in order of occurrence within the system block. The system name which will appear in the LABEL column will show a reference at END-SYSTEM. If there are no references to the name, the word NONE will appear. (36-104)

- If an error is encountered during processing, it may result in either incomplete or no global source cross reference output. For further description of the cross reference error messages, see paragraph A.4.

D.2.8 Compile Summary

See Figure D-10 for an example of a compile summary.

The major page header contains the element type (System), the system name, the date, the page number, and the title: Compile Summary.

The minor page headers are defined below. The values in parentheses following each definition indicate the columns on the listing where the information appears.

- ELEMENT NUMBER - Element number to which the summary applies. (2-15)
- ELEMENT NAME - Name of the element to which the summary applies. (22-33)
- SYNTAX/OBJECT ERRORS - The number of syntax and/or object errors flagged in a specified element. (40-59)
- ALLOCATION ERRORS - The number of allocation errors flagged in a specified element. (66-82)
- TOTAL ERRORS - The amount of syntax/object plus allocation errors in a specified element. (92-103)

D.3 System ListingsD.3.1 SHARE/7 System Summary

See Figure D-11 for an example of a SHARE/7 system summary.

The major page header contains the date and page number.

The minor page headers are defined below. The values in parentheses following each definition indicate the columns on the listings where the information appears.

- SYMBOL TABLE USAGE - Description of symbol table usage, set off by asterisks, listing the number of words available, the number of words used and the percent of available words used. (1-119)
- COMPILER REVISION FILE - The compiler used for this particular compile. (1-50)
- MONIT/CODIR FILE - Name of file containing the compiler director and monitor interface. (54-98)
- JOB START TIME - Time the job run started: hhmm:ss. (1-50)
- JOB END TIME - Time the job run ended: hhmm:ss (54-98)
- ELAPSED REAL TIME - Amount of real time required for the job: hhmm:ss (1-50)
- ELAPSED COMPUTE TIME - Amount of computer time required for the job: hhmm:ss (54-98)
- SYSTEM ID - Revision number, compiler, date: mm/dd/yy. (1-50)

1 2 3 4 5 6 7 8 9 0 1 1
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

mm/dd/yy PAGE dddd

SYMBOL TABLE USAGE - dddd WORDS AVAILABLE dddd WORDS USED ddd PERCENT USED

COMPILER REVISION FILE: lcccccccccc(lcccccccccc) MONIT/CODIR FILE: lcccccccccc(lcccccccccc)

JOB START TIME: hhmm:ss JOB END TIME: hhmm:ss

ELAPSED REAL TIME: hh HRS, mm MINS, ss SECS ELAPSED COMPUTE TIME: hh HRS, mm MINS, ss SECS

SYSTEM ID: REV dd - ccc11 mm/dd/yy

D-44

1 2 3 4 5 6 7 8 9 0 1 1
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

LEGEND

LABEL = HIGH-LEVEL (CMS-2) LABEL.
Label = LOW-LEVEL (DIRECT CODE) LABEL;
HIGH-LEVEL (CMS-2) LABEL PLACED BY
COMPILER IN RELATION TO DIRECT CODE
OR SOURCE MNEMONIC STATEMENTS.
d = DECIMAL DIGIT : h = HOUR : m = MONTH
o = OCTAL DIGIT : m = MINUTE : d = DAY
c = ALPHANUMERIC CHARACTER : s = SECOND : y = YEAR
l = ALPHA CHARACTER

Figure D-11. An Example of a SHARE/7 System Summary.

D.3.2 Batch System Summary

See Figure D-12 for an example of a batch system summary.

The major page header contains the date and page number.

The key word output is defined below:

SYMBOL TABLE USAGE - Description of symbol table usage, set off by asterisks, listing the number of words available, the number of words used, and the percent of available words used. (1-119)

APPENDIX E

FORMAT OF THE SYMBOL ANALYSIS DUMP

When `OPTIONS OBJECT(SADUMP)` is specified in the batch environment, symbol analysis information is output to the object file for the major header and each element of the compilation. The symbol analysis information for each of these is a separate subfile of the object file. These subfiles are distinguished by their header records. The order of subfiles within the object file is not specified. There is no separate subfile for minor headers: minor header information is contained in the subfile for the associated element.

In the Share/7 environment, this information is obtained by the CMS-2Y program's command `SADUMP(<filename>)`. The symbol analysis subfiles are created in the specified file. This file contains only symbol analysis information.

The header record is 30 AN/UYK-7 words long. Only the first four words are used at present. They contain

Word 0	*Abb
Words 1-2	Element Name
Word 3	Target Computer Designation:
	1 => UYK-7
	2 => UYK-43
	3 => reserved
	4 => reserved
	5 => reserved
	6 => reserved
	7 => not used
	8 => not used
	9 => reserved
	10 => reserved

The remainder of the subfile is in ISCM format, containing 30-word ISCM items. Each ISCM item contains one or more SA records. An ISCM item might be only partly filled with SA records, in which case end-of-item is marked. There is no explicit end-of-item mark if the item is filled with SA records. SA records have various lengths; the lengths are implicit in the contents of the records.

In the descriptions of SA records given below, certain order relations are specified. In a subfile corresponding to the major header or a system data block, these are the only order relations that are meaningful. In a subfile corresponding to a system procedure there is one more order relation: All SA records for data local to the system procedure (declared in local data blocks or

automatic data blocks) are output before the first subprogram (procedure or function) SA record; SA records immediately following a subprogram record are for data local to that subprogram (declared in a subprogram data block).

The first three words of all SA records are identical. The name of the entity is in words 1 and 2. Word 0 is a control word that contains a number of fields:

<u>Field</u>	<u>Type</u>	<u>Position</u>	<u>Meaning</u>
SACLAS	I 5 U	31-27	See below
GLOBFL	B	26	1 => Name has global scope
ALLOCFL	B	25	1 => Entity is allocated to memory
AC	I 5 U	20-16	AC number for entity
OFFSET	I 16 U	15-0	Offset from beginning of AC for entity.
ELTTYP	I 2 U	26-25	Element type: 0 => System data block 1 => System procedure 2 => Major header
LPOOLFL	B	24	1 => Local data pooling in effect
REN	B	21	1 => Reentrant system procedure
MAJXFL	B	24	1 => Major index is specified
VLTFLL	B	23	1 => Variable-length table
PSGTYP	I 2 U	24-23	Passage type: 0 => Direct passage 1 => Register passage 2 => Register, calling only
PACKING	I 3 U	24-22	Type packing: 0 => NONE 1 => MEDIUM 2 => DENSE 5 => Words per item specified 7 => Item-typed
PARFL	B	26	1 => Parameter variable
REGNO	I 4 U	3-0	Register number
DATFL	B	24	1 => Data label, 0 => Code label

DCFL	B	23	1 => Direct code label
RADIX	I 8 S	7-0	Radix
STWORD	I 8 U	7-0	Starting word number

(A data label (see DATFL) is a name defined in direct code within a data block.)

Only field SACLAS is present in all records. SACLAS = 0 denotes end-of-item for ISCM items that are partly filled. The other values of SACLAS and the associated fields for each are:

1. Element type: ELTTYP, LPOOLFL, REN
2. Type: GLOBFL, ALLOCFL (= 0), PACKING
3. Field: STWORD
4. Table or Array: AC, OFFSET, GLOBFL, ALLOCFL, MAJXFL, VLTFI
5. Item-area: AC, OFFSET, GLOBFL, ALLOCFL
6. Like-table: AC, OFFSET, GLOBFL, ALLOCFL, MAJXFL, VLTFI
7. Subtable: AC, OFFSET, GLOBFL, ALLOCFL, MAJXFL, VLTFI
8. Variable: AC, OFFSET, GLOBFL, ALLOCFL
9. Function: AC, OFFSET, GLOBFL, ALLOCFL, PSGTYP
10. Procedure: AC, OFFSET, GLOBFL, ALLOCFL, PSGTYP
11. Executive Procedure: AC, OFFSET, GLOBFL, ALLOCFL, PSGTYP
12. Formal Input Parameter: PARFL, REGNO
13. Formal Output Parameter: PARFL, REGNO
14. Formal Exit Parameter: None
15. Index: GLOBFL, REGNO
16. Switch: GLOBFL, ALLOCFL, AC, OFFSET
17. Label: GLOBFL, ALLOCFL, AC, OFFSET, DATFL, DCFL
18. Ntag: RADIX

19. Load-Time Constant: None

20-31. Unused

Several of the classes make use of a common type control word. Its fields are:

<u>Field</u>	<u>Type</u>	<u>Position</u>	<u>Meaning</u>
TYPE	I 4 U	31-28	Data type (see below)
NBITS	I 7 U	27-21	Number of bits
NCHARS	I 8 U	27-20	Number of characters
NVALS	I 15 U	27-13	Number of status values
NFRCBITS	I 8 S	20-13	Number of fractional bits
STBIT	I 5 U	12-8	Starting bit number
ASIZE	I 8 U	7-0	Allocated size, in characters for character data, in bits for all other types.

Only fields TYPE and ASIZE are present in all cases. The meaning of the values of TYPE and the other associated fields for those values are:

- 1 - I-type, unsigned: NBITS, STBIT
- 2 - I-type, signed: NBITS, STBIT
- 3 - A-type, unsigned: NBITS, STBIT, NFRCBITS
- 4 - A-type, signed: NBITS, STBIT, NFRCBITS
- 5 - Boolean: NBITS, STBIT
- 6 - Character: NCHARS, STBIT
- 8 - UYK-7 floating-point: NBITS (= 64), STBIT (= 31)
- 9 - UYK-43 single-precision floating point: NBITS (= 32), STBIT (= 31)
- 10 - UYK-43 double-precision floating point: NBITS (= 64), STBIT (= 31)
- 15 - Status: NVALS, STBIT

The amount of additional information depends on the type of entity:

Element Type (SACLAS = 1) - Records are three or five words long. Words 3-4 are present only if LPOOLFL is on, in which case they contain the pooling name. If no name is on the pooling directive, the default name is given.

Type (SACLAS = 2) - Records are three words long unless the value of PACKING is 7, in which case records are four words long and word 3 contains a type control word.

Type records are immediately followed by the type's field records, if any, then by all tables and structured variables that are of the type. Structured variables are output as item-areas.

Type records appear in the SADUMP file solely to assist in specifying the attributes of the program's other data. As a result, they have several unusual properties: Not all declared types are output; a purely simple type that is not used to declare a table will not be output. A type might appear in the SADUMP file more than once (e.g., if it is used to declare tables in two different scopes.) Finally, some types that appear in the SADUMP file have not been declared by the user; they have been generated by the compiler to simplify the structure of the SADUMP file. These types (called anonymous types) have names that begin with the character '@'.

Fields (SACLAS = 3) - Records are four words long. Word 3 contains a type control word. STWORD is meaningful.

Tables and Arrays (SACLAS = 4) - Records are a variable number of words long. Words 3 and 4 are always present:

<u>Field</u>	<u>Type</u>	<u>Word</u>	<u>Position</u>	<u>Meaning</u>
HFLG	B	3	31	1 => Horizontal table
INDFLG	B	3	30	1 => Indirect table
NDIMS	I 3 U	3	2-0	Number of dimensions
WDSITM	I 16 U	4	31-16	Number of words per item
NITEMS	I 16 U	4	15-0	Number of items

Following word 4 is variable information in the following order: dimensions (arrays only), and major index (if MAJXFL = 1). If any of this variable information is not applicable it does not appear and the following information, if any, is moved up in the record.

For an array of rank greater than 1, the dimension information consists of one word for each dimension, with the size of that dimension in bits 15-0, typed I 16 U. For a table with VLTFLL = 1, the dimension information consists of two words containing the name of the ltag.

The name of the major index occupies two words if MAJXFL = 1.

Note that no distinction is made between a vertical table and an array of rank 1.

Table records are immediately followed by all associated like-table, subtable, and item-area records. No order is guaranteed for these associated records.

Item-areas (SACLAS = 5) - Records are three words long.

Item-area records are used for both classical item-areas and structured variables, which have the same functionality. If it is desired to distinguish between them, the record for classical item-areas will follow the record of its parent table without any intervening type record while the record for a structured variable will follow the record of its type without any intervening table record.

Like-tables (SACLAS = 6) and subtables (SACLAS = 7) - Records are a variable number of words long. Words 0-2 are followed either by one word containing the dimension (if VLTFLL = 0) or by two words containing the name of the ltag (if VLTFLL = 1). This information is optionally followed by two words containing the major index name (if MAJXFL = 1). Notice that OFFSET, WDSITM, and HFLG can be used to calculate the starting item number of a subtable.

Variables (SACLAS = 8) - Records are four words long. Word 3 contains a type control word.

If a variable is declared using the name of a simple, non-structured, type, it will appear in the SADUMP file as a variable, rather than as an item-area following its type record.

Function (SACLAS = 9) - Records are four words long. Word 3 contains a type control word for the type of the function. STBIT and STWORD are not used.

Function records are immediately followed by all associated formal input parameter records in their declaration order. These are then followed by records for all data local to the function (local-indexes and data declared in a subprogram data block).

Procedures (SACLAS = 10) and Executive Procedures (SACLAS = 11) - Records are three words long.

Procedure records are immediately followed by all associated formal input parameter, formal output parameter, and exit parameter records in that order. Within each class the records appear in their declaration order. These are then followed by records for all data local to the procedure (local-indexes and data declared in a subprogram data block).

Formal Input Parameters (SACLAS = 12), Formal Output Parameters (SACLAS = 13), and Exits (SACLAS = 14) - Records are three words long.

Indexes (SACLAS = 15) - Records are three words long. GLOBFL is on for a system index, off for a local index. For a local index, REGNO = 0 implies that no register was available and the index has been allocated to memory.

Switches (SACLAS = 16) - Records are four words long for indexed switches, six words long for item switches. Word 3 contains the following:

<u>Field</u>	<u>Type</u>	<u>Position</u>	<u>Meaning</u>
NSWPTS	I 8 U	7-0	Number of switch points
ITMFLG	B	31	1 => Item switch
PSWFLG	B	30	1 => Procedure switch

For item switches (ITMFLG = 1), words 4 and 5 contain the name of the switch variable.

The records for procedure switches are immediately followed by all associated formal input and output parameter records, in declaration order.

Labels (SACLAS = 17) - Records are three words long.

Ntags (SACLAS = 18) - Records are five words long. Words 0-2 are followed by the signed value of the ntag, which is 64 bits. The value of RADIX is the position of the implied radix point. Note that this position can lie outside the bits of the value.

Load-Time Constants (SACLAS = 19) - Records are six words long. Word 3 contains a type control word. Word 3 is followed by the signed compile-time value of the ltag, which is 64 bits.

INDEX OF SYNTAX SYMBOLS

<u>Symbol</u>	<u>Page No.</u>
<decimal integer>	B-17
<C1>	B-38, B-38*
<V1>	B-38, B-38*
<1832 operation code>	B-5, B-12*
<AN/UYK-43 operation code>	B-5, B-8*
<AN/UYK-7 operation code>	B-5, B-5*, B-8
<BAM>	B-17
<ISFP>	B-16
<RAD>	B-17
<abnormal branch>	6-22, 6-22*
<abs directive>	B-35, B-36*
<abs function reference>	5-17, 5-18*
<abscissa>	5-22, 5-22, 5-22*, 6-34, 6-34, 6-34
<action clause>	6-116, 6-116*
<actual exit parameter>	6-29, 6-29*
<actual input parameter>	5-15, 5-15*, 6-29
<actual i/o parameters>	6-29, 6-29*, 6-38, 6-41
<actual output parameter>	6-29, 6-29*
<actual procedure parameters>	6-29, 6-29*
<additive operator>	4-68, 4-68*, 4-68, 4-68, B-25, B-32
<address counter separation declaration>	9-3, 9-35*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<address declaration>	4-75*, 7-8, 8-2, 8-8
<address offset>	B-25, B-25, B-25*
<address specification>	9-31, 9-31*
<addressable direct code statement>	B-27, B-30*
<addressable name>	4-20, 4-20*, 4-60, 4-68, 4-68, 4-68, 4-68
<addressable unit>	5-19, 5-19*
<allocatable name>	4-75, 4-75*
<allocation information>	9-31, 9-31*
<allocation modifier>	4-2, 4-2*
<alphanumeric character>	3-7, 3-7*, 9-39
<alphanumeric name>	9-39, 9-39*, 9-48
<alternative>	6-105*
<alternative statement>	6-105
<angle>	5-22, 5-22, 5-22*
<angle scaling>	B-17, B-17, B-17*
<angular measurement>	B-16, B-17*
<array block>	4-56*, 7-8, 8-2, 8-8
<array declaration>	4-56, 4-56*
<array information>	4-56, 4-56*
<assignment phrase>	6-4, 6-5*
<atag expression>	4-68, 4-68*, 4-68, 4-68, 4-75
<automatic data block>	8-4, 8-8*
<automatic data declaration>	8-8, 8-8*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<automatic data sentence>	8-8, 8-8*
<b1>	B-37, B-37*
<b2>	B-37, B-37*
<base register specification>	9-31, 9-31*
<basic angle>	B-17, B-17, B-17*
<begin block>	6-93, 6-94*
<begin block body>	6-94, 6-94*
<begin block head>	6-94, 6-94*
<bit length>	4-8, 4-8, 4-8*
<bit modified data unit>	5-9, 5-10*, 5-49
<bit string complement function reference>	5-33, 5-33*
<bit string difference function reference>	5-33, 5-33*
<bit string expression>	5-48, 5-79*, 5-79, 5-79, 5-79
<bit string factor>	5-79, 5-79, 5-79*
<bit string function reference>	5-21, 5-33*
<bit string length>	5-10, 5-10*, 6-5
<bit string operand 1>	5-33, 5-33, 5-33, 5-33, 5-33*
<bit string operand 2>	5-33, 5-33, 5-33, 5-33*
<bit string primary>	5-79, 5-79, 5-79*
<bit string product function reference>	5-33, 5-33*
<bit string receptacle>	6-5, 6-5*
<bit string start>	5-10, 5-10*, 6-5

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<bit string sum function reference>	5-33, 5-33*
<bit string term>	5-79, 5-79, 5-79, 5-79*, 5-79
<block name>	6-48, 6-50, 6-110, 6-110*
<boolean binary operator>	6-116, 6-116*
<boolean comparand>	5-69, 5-69, 5-69*
<boolean constant>	3-9, 3-16*, 4-20, 5-64, 5-69
<boolean expression>	5-48, 5-64*, 5-64, 5-64, 5-79, 6-113, 6-113, 6-116
<boolean factor>	5-64, 5-64, 5-64*
<boolean primary>	5-64, 5-64, 5-64*
<boolean relational expression>	5-64, 5-69*
<boolean term>	5-64, 5-64, 5-64*, 5-64
<boolean type specification>	4-7, 4-12*
<bottom test>	6-97, 6-97*
<bottom test clause>	6-96, 6-97*
<branch phrase>	6-4, 6-20*
<byte directive>	B-35, B-37*
<case block>	6-93, 6-105*
<case block head>	6-105, 6-105*
<case selector>	6-105, 6-105*
<case type>	6-105, 6-105*
<case value>	6-105, 6-105*
<change value>	6-96, 6-96*
<char directive>	B-35, B-38*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<character>	3-2*, 3-6, 6-60, B-21, B-27
<character comparand>	5-71, 5-71, 5-71*
<character constant>	3-9, 3-17*, 4-20, 4-60, 4-108, 4-116, 5-77, 6-66
<character expression>	5-48, 5-71, 5-77*, 5-77, 5-77, 5-79
<character length>	4-13, 4-13*
<character modified data unit>	5-9, 5-12*, 5-77
<character primary>	5-77, 5-77, 5-77*
<character relational expression>	5-64, 5-71*
<character string>	3-6*, 3-17, 3-18
<character string length>	5-12, 5-12*, 6-5
<character string receptacle>	6-5, 6-5*
<character string start>	5-12, 5-12*, 6-5
<character type specification>	4-7, 4-13*
<check label phrase>	6-4, 6-61*
<checkable reference>	5-74, 5-74*
<close phrase>	6-4, 6-58*
<cms-2 phrase>	3-19, 3-19*
<cnt function reference>	5-21, 5-44*
<comment>	2-2, 2-2*, 2-4
<comment character>	2-2, 2-2*
<comment statement>	2-2*
<compile-time constant declaration>	4-68*, 7-8, 8-2, 8-8, 8-10
<compiler directive>	8-10, 9-24*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<compool name>	9-12, 9-12*, 9-44
<compool retrieval declaration>	9-3, 9-44*
<compool retrieval specification>	9-44, 9-44*
<compound section name>	9-31, 9-31*
<conditional compilation directive>	10-1*
<conditional expression>	5-64, 5-74*, 6-97, 6-97
<conditional i/o expression>	5-64, 5-76*
<conditional statement>	6-1, 6-112*
<conf function reference>	5-21, 5-38*
<constant>	3-9*, 4-91, 6-105
<constant mode>	4-5, 4-5*
<constant mode declaration>	4-5*, 8-10
<control clause>	6-96, 6-96*
<controlled expression>	5-36, 5-36*
<conversion source>	5-38, 5-38*
<convertin phrase>	6-4, 6-79*
<convertout phrase>	6-4, 6-82*
<corad function reference>	5-17, 5-19*
<core address receptacle>	4-79, 4-125, 6-5, 6-5*
<correction block header>	9-40, 9-40, 9-40, 9-40*
<cswitch delete declaration>	10-1, 10-5*
<cswitch flag>	10-2, 10-2*, 10-2, 10-4, 10-4
<cswitch header statement>	10-1, 10-2*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<switch selection declaration>	10-1, 10-4*
<switch terminal statement>	10-1, 10-2*
<data block name>	7-8, 7-8, 8-2, 8-2*, 8-2, 8-6, 8-6, 8-8, 8-8
<data sentence>	8-2, 8-2*, 8-6
<data unit>	4-125, 5-2*, 6-76, 6-84, 6-87
<debug enabling declaration>	4-129*, 9-3
<debug parameter>	4-129, 4-129*
<decimal constant>	3-10, 3-11*
<decimal digit>	3-11, 3-11*
<decimal exponent>	3-11, 3-11*
<decimal integer>	3-11, 3-11, 3-11, 3-11, 3-11, 3-11*, B-16, B-16, B-16, B-16, B-16, B-16, B-16, B-16, B-17, B-17, B-37, B-37
<decimal mantissa>	3-11, 3-11*
<decimal number>	3-11, 3-11, 3-11, 3-11*
<declaration modifier>	4-2*, 4-79, 4-82, 4-84
<decode phrase>	6-4, 6-78*
<default type specification>	4-20*, 8-2, 8-8, 8-10
<define label phrase>	6-4, 6-60*
<delimiter>	2-2, 3-2, 3-5*, 3-6
<dep element>	9-51, 9-51*
<dep specification>	9-40, 9-40, 9-40*
<dependent element declaration>	8-10, 9-51*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<digit>	2-2, 3-2, 3-4*, 3-6, 3-7, 3-11
<direct code>	3-19, 3-19*
<direct code address expression>	B-23, B-25*, B-32, B-32
<direct code address preset>	B-32, B-32*
<direct code addressable name>	B-25, B-25, B-25*
<direct code block>	3-19*, 7-8, 8-2
<direct code character constant>	B-15, B-21*, B-22, B-32, B-38, B-38
<direct code character preset>	B-32, B-32*
<direct code comment>	B-27, B-27, B-27*
<direct code constant>	B-15*, B-39
<direct code data statement name>	B-28, B-28*
<direct code decimal constant>	B-16, B-16*
<direct code directive>	B-27, B-35*
<direct code expression>	B-23*, B-31
<direct code head>	3-19, 3-19*
<direct code instruction>	B-30, B-31*
<direct code instruction designators>	B-31
<direct code literal>	B-22*, B-25
<direct code name>	B-27, B-28*, B-36, B-36, B-39, B-42
<direct code numeric constant>	B-15, B-16*, B-22, B-38, B-38, B-39, B-40, B-41, B-42
<direct code numeric constant expression>	B-23, B-24*, B-32

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<direct code numeric preset>	B-32, B-32*, B-32
<direct code octal constant>	B-16, B-17*
<direct code phrase>	3-19, 3-19*, 6-4
<direct code preset>	B-27, B-30, B-32*
<direct code program statement name>	B-28, B-28*
<direct code scale factor>	B-16, B-16, B-16*, B-17
<direct code scope modifier>	B-28, B-28*
<direct code statement>	3-19, B-27*
<direct code statement name>	B-25, B-25*
<direct long numeric constant>	B-16, B-17*, B-32
<direct short numeric constant>	B-16, B-16*, B-24, B-24, B-25, B-25
<direction>	4-116, 4-116*
<display item>	6-84, 6-84*
<display phrase>	6-4, 6-84*
<do directive>	B-35, B-39*
<double label switch block>	4-86, 4-89*
<double label switch declaration>	4-89, 4-89*
<double procedure switch block>	4-93, 4-96*
<double procedure switch declaration>	4-96, 4-96*
<double procedure switch point>	4-96, 4-96*
<double switch point>	4-89, 4-89*
<double-word numeric preset>	B-32, B-32*
<element form>	9-48, 9-48*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<element name>	9-51, 9-51*
<else clause>	6-105, 6-105*, 6-113, 6-116
<elsif clause>	6-113, 6-113*
<encode phrase>	6-4, 6-76*
<end double procedure switch declaration>	4-96, 4-96*
<end double switch declaration>	4-89, 4-89*
<end phrase>	6-94, 6-96, 6-105, 6-105, 6-110*
<end-automatic-data declaration>	8-8, 8-8*
<end-function declaration>	7-5, 7-5*
<end-header declaration>	8-10, 9-3, 9-3, 9-3*
<end-local-data declaration>	8-6, 8-6*
<end-procedure declaration>	7-3, 7-3*, 7-4
<end-procedure-switch declaration>	4-94, 4-94*, 4-98
<end-subprogram-data declaration>	7-8, 7-8*
<end-switch declaration>	4-87, 4-87*, 4-91
<end-system declaration>	9-1, 9-1*
<end-system-data declaration>	8-2, 8-2*
<end-system-procedure declaration>	8-4, 8-4*
<end-table declaration>	4-32, 4-33*, 4-56
<end-trace phrase>	6-4, 6-91*
<endfile phrase>	6-4, 6-59*
<executive call phrase>	6-4, 6-52*
<executive directive>	9-24, 9-29*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<executive function>	6-52, 6-52*
<executive input parameter>	6-52, 6-52*
<executive procedure block>	7-2, 7-4*
<executive procedure declaration>	4-82*, 7-4, 8-2, 8-8
<exit phrase>	6-4, 6-48*
<exponent size>	4-116, 4-116*
<expression>	4-127, 5-15, 5-48*, 6-5, 6-45
<extended field>	6-66, 6-66*, 6-66
<extended structured variable data unit>	6-66, 6-66*
<extended subscript>	6-66, 6-66*
<extended subscript data unit>	6-66, 6-66*, 6-74
<extended table data unit>	6-66, 6-66*
<external-id>	9-39, 9-39*
<field declaration>	4-15, 4-39*
<field name>	4-39, 4-39*, 4-44, 4-44, 4-131, 5-3, 5-3, 5-19, 6-5, 6-66
<field overlay declaration>	4-15, 4-44*
<field overlay parent>	4-44, 4-44*
<field overlay sibling>	4-44, 4-44*
<field width>	4-108, 4-108, 4-108, 4-108, 4-108*, 4-116, 4-116, 4-116, 4-116, 4-116, 4-116, 4-116, 4-116
<file function reference>	5-21, 5-45*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<file declaration>	4-101*, 7-8, 8-2
<file name>	4-75, 4-101, 4-101, 4-101*, 5-45, 5-46, 5-47, 5-76, 6-56, 6-58, 6-59, 6-60, 6-61, 6-62, 6-64, 6-66, 6-74
<file positioning phrase>	6-4, 6-62*
<file specification>	4-101, 4-101, 4-101*
<file status>	4-101, 4-101, 4-102*
<file status operator>	5-76, 5-76*
<file structure>	4-101, 4-101*
<file type>	4-101, 4-101*
<final value>	6-96, 6-96*
<final value function reference>	5-28, 5-32*
<find clause>	6-116, 6-116*
<find condition>	6-116, 6-116*
<find relational expression>	6-116, 6-116*
<find statement>	6-112, 6-116*
<fixed-point arithmetic function reference>	5-21, 5-25*
<fixed-point type specification>	4-8, 4-8*
<floating-point arithmetic function reference>	5-21, 5-22*
<floating-point attribute>	4-8, 4-8*
<floating-point type specification>	4-8, 4-8*
<form directive>	B-35, B-40*
<form label>	B-40, B-40*, B-42
<form name>	B-28, B-28*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<form preset>	B-30, B-42*
<formal exit parameter>	4-79, 4-79*, 6-45
<formal input parameter>	4-79, 4-79*, 4-82, 4-84
<formal i/o parameters>	4-79, 4-79*, 4-94, 4-98
<formal output parameter>	4-79, 4-79*
<formal procedure parameters>	4-79, 4-79*
<format declaration>	4-108*, 7-8, 8-2
<format descriptor>	4-108, 4-108*
<format item>	4-108, 4-108*, 4-108, 4-108
<format list>	4-108, 4-108*, 4-108
<format name>	4-75, 4-108, 4-108*, 6-66, 6-74, 6-76, 6-78
<format positioner>	4-108, 4-108*
<format specification>	4-108, 4-108, 4-108, 4-108, 4-108*
<fraction size>	4-108, 4-108*, 4-116
<fractional bits>	4-8, 4-8*, 4-60
<function block>	7-2, 7-5*
<function body>	7-5, 7-5*
<function declaration>	4-84*, 7-5, 8-2, 8-8
<function name>	4-20, 4-84, 4-84*, 5-15, 5-19, 7-5, 9-25, B-25
<function reference>	5-14*, 5-49, 5-64, 5-69, 5-77, 5-78
<function return phrase>	6-45, 6-45*
<function type>	4-84, 4-84*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<function value>	6-45, 6-45*
<header declaration>	8-10, 9-3, 9-3*
<header name>	9-3, 9-3*, 9-3
<header sentence>	8-10, 8-10*, 9-3
<i/o capability>	6-56, 6-56*
<i/o data unit>	6-66, 6-66*, 6-74
<if statement>	6-112, 6-113*
<imperative phrase>	6-3, 6-4*
<implied form>	B-32, B-32*
<incrementation clause>	6-96, 6-96, 6-96, 6-96, 6-96, 6-96, 6-96*
<index clause>	6-96, 6-96*, 6-116
<indexed branch phrase>	6-4, 6-22*
<indexed label switch block>	4-86, 4-87*
<indexed procedure call phrase>	6-4, 6-38*
<indexed procedure switch block>	4-93, 4-94*
<indexed procedure switch declaration>	4-94, 4-94*
<indexed procedure switch name>	4-20, 4-94, 4-94*, 4-94, 4-94, 4-96, 4-96, 4-96, 4-96, 4-96, 4-96, 6-38
<indexed procedure switch point>	4-94, 4-94*
<initial value>	6-96, 6-96*
<initial value function reference>	5-28, 5-31*
<initiation clause>	6-96, 6-96, 6-96, 6-96, 6-96, 6-96, 6-96*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<input buffer>	6-79, 6-79*
<input file name>	6-74, 6-74*
<input item>	6-74, 6-74, 6-74*
<input list>	6-74, 6-74*, 6-78
<input phrase>	6-4, 6-74*
<input receptacle>	4-125, 4-125*
<inputlist>	4-125, 4-125*, 6-79
<inputlist declaration>	4-125*, 7-8, 8-2
<inputlist item>	4-125, 4-125*
<inputlist name>	4-20, 4-75, 4-125, 4-125*, 4-125, 5-19
<installation hardware name>	4-102, 4-102*
<integer type>	4-72, 4-72*
<integer type specification>	4-8, 4-8*, 4-72
<internal-id>	9-39, 9-39*
<intrinsic function reference>	5-14, 5-17*
<invalid specification>	6-22, 6-22*, 6-25, 6-38, 6-41
<item allocation>	4-32, 4-32*, 4-56
<item branch phrase>	6-4, 6-25*
<item label switch block>	4-86, 4-91*
<item label switch declaration>	4-91, 4-91*
<item label switch name>	4-20, 4-91, 4-91*, 6-25
<item label switch point>	4-91, 4-91*
<item procedure call phrase>	6-4, 6-41*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<item procedure switch block>	4-93, 4-98*
<item procedure switch declaration>	4-98, 4-98*
<item procedure switch name>	4-20, 4-98, 4-98*, 6-41
<item procedure switch point>	4-98, 4-98*
<item replicator>	4-108, 4-108, 4-108, 4-108*, 4-116, 4-116, 4-116
<item-area declaration>	4-33, 4-54*, 4-56
<key>	9-40, 9-40, 9-40, 9-44, 9-48, 9-48*, 9-51
<key specification>	8-2, 8-4, 9-1, 9-3, 9-48*
<label definition>	6-60, 6-60*, 6-61
<label switch block>	4-86*, 7-8, 8-6
<label switch declaration>	4-87, 4-87*
<label switch name>	4-20, 4-87, 4-87*, 4-87, 4-89, 4-89, 4-89, 4-89, 6-22
<label switch point>	4-87, 4-87*
<length function reference>	5-21, 5-47*
<letter>	2-2, 3-2, 3-3*, 3-6, 3-7, 3-7
<library declaration>	8-10, 9-39*, 9-44
<like-table declaration>	4-32, 4-47*
<listing parameter>	9-16, 9-16*
<listing specification>	9-6, 9-16*
<loadvrbl form>	4-72, 4-72*
<local data block>	8-4, 8-6*
<local data declaration>	8-6, 8-6*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<local data sentence>	8-6, 8-6*
<local index declaration>	4-78*, 7-7, 7-8
<local index name>	4-78, 4-78*, 5-3
<loop block>	6-93, 6-96*
<loop block body>	6-96, 6-97*
<loop block head>	6-96, 6-96*
<loop index>	6-96, 6-96*
<ltag declaration>	4-72*, 7-8, 8-2, 8-8, 8-8, 8-10
<ltag list>	4-72, 4-72*
<ltag name>	4-20, 4-32, 4-72, 4-72, 4-72*, 4-72, 5-49
<magnitude>	5-22, 5-22, 5-22*
<magnitude bit>	4-20, 4-20*
<magnitude value>	4-20, 4-20*
<major header>	9-1, 9-3*
<major header block>	9-3, 9-3*
<major header sentence>	9-3, 9-3, 9-3*, 9-3
<major index>	4-32, 4-32*, 4-47, 4-49
<maximum value>	4-131, 4-131*
<message level specification>	9-6, 9-18*
<minimum value>	4-131, 4-131*
<minor header>	8-2, 8-4, 8-10*
<minor header block>	8-10, 8-10*
<minor header sentence>	8-10, 8-10*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<mode variable specification>	9-6, 9-22*
<modified data unit>	5-2, 5-9*
<monitor specification>	9-6, 9-19*
<multivalued data unit>	5-2, 5-6*, 5-19, 6-5, 6-5, 6-66
<name>	3-7*, 4-15, 4-15, 4-15, 4-16, 4-24, 4-32, 4-32, 4-39, 4-66, 4-68, 4-68, 4-72, 4-77, 4-78, 4-79, 4-79, 4-84, 4-87, 4-91, 4-94, 4-98, 4-101, 4-102, 4-108, 4-116, 4-125, 4-127, 6-1, 8-2, 8-4, 9-1, 9-3, 9-12, 9-31, 9-39, 9-40, 9-51, 10-2, B-25, B-28, B-28, B-28, B-40
<new abscissa>	6-34, 6-34, 6-34, 6-34*
<new angle>	6-34, 6-34, 6-34*
<new magnitude>	6-34, 6-34, 6-34*
<new ordinate>	6-34, 6-34, 6-34, 6-34*
<nitems form>	4-72, 4-72*
<non-real-time specification>	9-6, 9-20*
<nonstandard file declaration>	4-101, 4-101*
<nonstandard hardware name>	4-101, 4-101*
<note>	2-4*
<ntag declaration>	4-68, 4-68*
<ntag expression>	4-68, 4-68*, 4-68, 4-68, 4-68, 4-72
<ntag name>	4-68, 4-68*, 4-68, 5-49, 5-62, 5-63, 5-64, 5-69, B-16
<ntag primary>	4-68, 4-68, 4-68*, 4-68,

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
	4-68
<null phrase>	6-4, 6-92*
<number of items>	4-32, 4-32*
<number of words>	4-15, 4-15*, 4-32
<numeric comparand>	5-67, 5-67, 5-67*
<numeric constant>	3-9, 3-10*, 4-68, 4-108, 4-108, 4-108, 5-49, 5-62, 5-63, 9-40
<numeric constant expression>	4-8, 4-8, 4-13, 4-15, 4-20, 4-20, 4-20, 4-29, 4-32, 4-39, 4-39, 4-39, 4-44, 4-49, 4-56, 4-63, 4-72, 4-91, 4-101, 4-101, 4-131, 4-131, 5-36, 5-62*, 5-62, 5-62, 5-62, 5-62, 6-52, 6-66, 6-105, 9-31
<numeric constant factor>	5-62, 5-62, 5-62, 5-62*, 5-62
<numeric constant primary>	5-62, 5-62, 5-62*
<numeric constant term>	5-62, 5-62, 5-62*, 5-62, 5-62
<numeric constant value>	4-60, 4-108, 4-116, 5-49, 5-63*
<numeric expression>	5-3, 5-7, 5-10, 5-10, 5-12, 5-12, 5-18, 5-22, 5-22, 5-22, 5-22, 5-22, 5-22, 5-22, 5-22, 5-22, 5-25, 5-25, 5-25, 5-25, 5-25, 5-25, 5-36, 5-38, 5-42, 5-48, 5-49*, 5-49, 5-49, 5-49, 5-67, 5-79, 6-22, 6-34, 6-53, 6-62, 6-64, 6-96, 6-96, 6-96
<numeric factor>	5-49, 5-49, 5-49, 5-49*, 5-49

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<numeric operator>	4-68, 4-68*, 4-68, 4-68, 4-68, B-24
<numeric primary>	5-49, 5-49, 5-49*
<numeric relational expression>	5-64, 5-67*
<numeric term>	5-49, 5-49, 5-49, 5-49*, 5-49, 5-49
<numeric type specification>	4-7, 4-8*, 5-38, 5-40
<object parameter>	9-12, 9-12*
<object specification>	9-6, 9-12*
<octal constant>	3-10, 3-14*
<octal digit>	3-14, 3-14*
<octal exponent>	3-14, 3-14*
<octal integer>	3-14, 3-14, 3-14, 3-14, 3-14, 3-14*, B-16, B-17, B-17, B-17, B-17
<octal mantissa>	3-14, 3-14*
<octal number>	3-14, 3-14, 3-14*
<open phrase>	6-4, 6-56*
<operand>	B-31, B-31*
<operation code>	B-5*, B-31
<option specification>	9-6, 9-6, 9-6*
<options declaration>	9-3, 9-3, 9-6*
<ordinate>	5-22, 5-22, 5-22*, 6-34, 6-34, 6-34
<output buffer>	6-82, 6-82*
<output file name>	6-66, 6-66*
<output item>	6-66, 6-66, 6-66*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<output list>	6-66, 6-66*, 6-76
<output phrase>	6-4, 6-66*
<outputlist>	4-127, 4-127*, 6-82
<outputlist declaration>	4-127*, 7-8, 8-2
<outputlist item>	4-127, 4-127*
<outputlist name>	4-20, 4-75, 4-127, 4-127*, 4-127, 5-19
<overflow phrase>	6-5, 6-5*
<overlay declaration>	4-63*, 7-8, 8-2, 8-8
<overlay parent>	4-63, 4-63*
<overlay sibling>	4-63, 4-63*
<parameter constant>	B-38, B-38*
<parameter passage directive>	9-24, 9-25*
<parameter variable declaration>	4-29*, 7-8, 8-2, 8-8
<parent receptacle>	6-5, 6-5, 6-5*
<parent unit>	5-10, 5-10*, 5-12
<parity test>	5-74, 5-74*
<passage type>	9-25, 9-25*
<pooling directive>	9-24, 9-31*
<pooling type>	9-31, 9-31*
<pos function reference>	5-21, 5-46*
<position>	4-108, 4-108*, 4-116
<power of ten factor>	B-16, B-16*
<pre-settable name>	4-60, 4-60*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<predecessor function reference>	5-28, 5-30*
<predefined function reference>	5-14, 5-21*
<preset entry>	4-60, 4-60*
<preset item>	4-39, 4-39*
<preset magnitude>	4-20, 4-20*, 6-84, 6-87
<preset semi-entry>	4-60, 4-60, 4-60*
<preset value>	4-20, 4-20*, 4-24, 4-29, 4-39, 4-39
<preset value declaration>	4-60*, 7-8, 8-2
<primary condition>	6-113, 6-113*
<primary statement>	6-113, 6-113*
<procedure block>	7-2, 7-3*
<procedure block name>	8-4, 8-4*, 8-4
<procedure body>	7-3, 7-3*, 7-4
<procedure call phrase>	6-4, 6-28*
<procedure declaration>	4-79*, 7-3, 8-2, 8-8
<procedure name>	4-20, 4-79, 4-79*, 4-82, 4-94, 4-96, 4-96, 4-98, 5-19, 6-29, 7-3, 9-25, B-25
<procedure return phrase>	6-45, 6-45*
<procedure switch block>	4-93*, 7-8, 8-2
<procedure switch name>	4-20, 4-20*, 4-75, 5-19
<pseudo buffer>	6-76, 6-76*, 6-78
<pseudo operation code>	B-5, B-13*
<range declaration>	4-15, 4-131*, 7-8, 8-2, 8-8
<ranged name>	4-131, 4-131*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<receptacle>	6-5, 6-5*, 6-18, 6-18, 6-29, 6-34, 6-34, 6-34, 6-34, 6-53
<receptacle 1>	6-18, 6-18*
<receptacle 2>	6-18, 6-18*
<record limit>	4-101, 4-101*
<record positioning phrase>	6-4, 6-64*
<record size>	4-101, 4-101*
<redefinition source>	5-40, 5-40*
<register number>	4-29, 4-29*, 4-77, 9-31
<relational expression>	5-64, 5-64*
<relational operator>	5-67, 5-67*, 5-69, 5-71, 5-73, 6-116
<rem function reference>	5-21, 5-42*
<remainder phrase>	6-5, 6-5*
<remainder receptacle>	6-5, 6-5*
<remaindering expression>	5-42, 5-42*
<repeat value>	4-39, 4-39*
<res directive>	B-35, B-41*
<resume phrase>	6-4, 6-50*
<return phrase>	6-4, 6-45*
<rotation>	6-34, 6-34*
<rtag declaration>	4-68, 4-68*
<rtag expression>	4-68, 4-68*, 4-68, 4-68, 4-68
<rtag name>	4-68, 4-68*, 4-68, 4-68, 4-68, 4-68, 5-49

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<scale factor>	5-36, 5-36*
<scale operator>	B-16, B-16, B-16*, B-17
<scaled direct code decimal constant>	B-16, B-16*
<scaled direct code octal constant>	B-17, B-17*
<scalf function reference>	5-21, 5-36*
<scaling specification>	9-6, 9-23*
<scaling specifier>	5-49, 5-49, 5-49, 5-49, 5-49*, 6-5, 6-5
<scope modifier>	4-2, 4-2*, 4-24, 4-29, 4-32, 4-47, 4-49, 4-54, 4-56, 4-94, 4-96, 4-98, 4-101, 4-101, 4-108, 4-116, 4-125, 4-127
<secondary condition>	6-113, 6-113*
<secondary statement>	6-113, 6-113*
<shift amount>	6-53, 6-53*
<shift assign clause>	6-53, 6-53*
<shift phrase>	6-4, 6-53*
<shift source>	6-53, 6-53*
<shift type>	6-53, 6-53*
<sign specification>	4-8, 4-8, 4-8*
<simple character>	3-6, 3-6*
<simple expression>	5-33, 5-33, 5-40, 5-44, 5-48, 5-48*, 6-52, 6-105, 6-116
<simple floating constant>	B-16, B-16, B-16*
<simple phrase>	6-1, 6-1, 6-3*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<simple statement>	6-1, 6-1*, 6-105, 6-113, 6-113, 6-116, 6-116
<simple string>	3-6*, 4-66
<simple type>	4-16, 4-16*, 4-20, 4-20, 4-24, 4-39, 5-38, 5-40, 6-105
<simple type declaration>	4-15, 4-15*
<simple type name>	4-15, 4-15*, 4-16, 4-32, 4-72
<simple type specification>	4-7*, 4-15, 4-16
<single precision directive>	9-24, 9-28*
<single-valued data unit>	4-125, 4-127, 4-127, 5-2, 5-3*, 5-10, 5-49, 5-64, 5-69, 5-74, 5-77, 5-78, 5-82, 6-5, 6-5, 6-53, 6-66, 6-79, 6-79, 6-82, 6-96
<single-word numeric preset>	B-32, B-32*
<snap phrase>	6-4, 6-87*
<source>	6-5, 6-5*
<source element name>	9-40, 9-40, 9-40*
<source parameter>	9-10, 9-10*
<source retrieval declaration>	8-10, 9-40*
<source specification>	9-6, 9-10*
<space>	2-2, 3-2, 3-2*, 3-6, B-27
<special character>	2-2, 3-2, 3-2*
<special condition>	6-20, 6-20*, 6-22, 6-25, 6-45
<spill directive>	9-24, 9-30*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<standard file declaration>	4-101, 4-101*
<standard hardware name>	4-101, 4-101*
<starting bit>	4-39, 4-39*
<starting item>	4-49, 4-49*
<starting word>	4-39, 4-39*
<statement>	6-1*, 6-94, 6-97, 6-105, 7-7
<statement block>	6-3, 6-93*
<statement label>	6-1, 6-1, 6-1, 6-1*, 6-105
<statement name>	4-20, 4-87, 4-89, 4-89, 4-91, 5-19, 6-1, 6-1*, 6-5, 6-20, 6-22, 6-29, 6-110, B-25
<status comparand>	5-73, 5-73, 5-73*
<status constant>	3-9, 3-18*, 4-14, 4-20, 4-49, 4-102, 5-76, 5-78
<status expression>	5-3, 5-29, 5-30, 5-48, 5-73, 5-78*, 5-79, 6-96, 6-96
<status operation function reference>	5-21, 5-28*
<status relational expression>	5-64, 5-73*
<status type>	4-32, 4-32*, 4-56, 5-31, 5-32
<status type name>	4-32, 4-32*
<status type specification>	4-7, 4-14*, 4-32
<stop condition>	6-43, 6-43*
<stop phrase>	6-4, 6-43*
<string name>	4-66, 4-66*
<stringform declaration>	4-116*, 7-8, 8-2

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<stringform descriptor>	4-116, 4-116*
<stringform item>	4-116, 4-116*
<stringform list>	4-116, 4-116*, 4-116
<stringform name>	4-20, 4-75, 4-116, 4-116*, 5-19, 6-79
<stringform positioner>	4-116, 4-116*
<stringform specification>	6-79, 6-79*, 6-82
<structure allocation>	4-15, 4-15*
<structure information>	4-15, 4-15*, 4-15, 4-32, 4-56
<structured expression>	5-48, 5-82*
<structured specification>	9-6, 9-21*
<structured type>	4-24, 4-24*
<structured type declaration>	4-15, 4-15*
<structured type end>	4-15, 4-15*, 4-15
<subprogram block>	7-2*, 8-4
<subprogram body>	7-3, 7-5, 7-7*
<subprogram data block>	7-7, 7-8*
<subprogram data declaration>	7-8, 7-8*
<subprogram data sentence>	7-8, 7-8*
<subprogram name>	9-25, 9-25*
<subscript declaration>	4-56, 4-56*
<subscript expression>	5-3, 5-3*, 5-7, 6-66, 6-66, 6-66
<subscripted data unit>	5-3, 5-3*, 5-19, 5-74, 6-5, 6-116

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<substitution declaration>	4-66*, 8-10
<substitution type>	4-66, 4-66*
<subtable declaration>	4-33, 4-49*
<successor function reference>	5-28, 5-29*
<supplied procedure call phrase>	6-28, 6-34*
<swap phrase>	6-4, 6-18*
<switch index>	6-22, 6-22*, 6-38
<switch name>	4-20, 4-20*, 4-75, 5-19, B-25
<switch selector>	4-91, 4-91*, 4-98
<switch value>	4-91, 4-91*, 4-98
<system block>	9-1*
<system data block>	8-2, 8-2*
<system data declaration>	8-2, 8-2*
<system data element>	8-1, 8-2*
<system declaration>	9-1, 9-1*
<system element>	8-1*, 9-1
<system index declaration>	4-77*, 9-3
<system index name>	4-77, 4-77*, 4-79, 4-79, 5-3
<system index specification>	4-77, 4-77*
<system name>	9-1, 9-1*, 9-1
<system procedure block>	8-4, 8-4*
<system procedure declaration>	8-4, 8-4*
<system procedure element>	8-1, 8-4*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<system procedure sentence>	8-4, 8-4*
<system procedure type>	8-4, 8-4*
<table block>	4-32*, 7-8, 8-2, 8-8
<table declaration>	4-32, 4-32*
<table information>	4-32, 4-32*
<table name>	4-20, 4-32, 4-32*, 4-33, 4-47, 4-49, 4-56, 4-60, 4-63, 4-63, 4-75, 4-79, 4-79, 4-127, 5-3, 5-6, 5-7, 6-5, 6-66, 6-79, 6-82, 6-96, B-25
<table subscript declaration>	4-32, 4-32*, 4-47, 4-49
<table type>	4-32, 4-32*
<target conversion type>	5-38, 5-38*
<target machine>	9-6, 9-6*
<target redefinition type>	5-40, 5-40*
<tdef function reference>	5-21, 5-40*
<termination clause>	6-96, 6-96, 6-96, 6-96, 6-96, 6-96, 6-96*
<terminator>	3-2, 3-2*
<top test>	6-97, 6-97*
<top test clause>	6-96, 6-97*
<trace phrase>	6-4, 6-89*
<type>	4-24, 4-24*, 4-29, 4-32, 4-84
<type declaration>	4-15*, 7-8, 8-2, 8-8
<typed structure>	4-16, 4-16*
<typed structure declaration>	4-15, 4-15*

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<typed structure head>	4-15, 4-16*
<typed structure name>	4-16, 4-16*, 4-16, 4-24, 5-38, 5-40, 6-105
<unary numeric operator>	3-11, 3-11*, 3-14, 4-68, 4-68, 5-49, 5-62, B-16, B-16, B-16, B-22, B-32
<unscaled direct code decimal constant>	B-16, B-16*, B-17
<unscaled direct code octal constant>	B-17, B-17*, B-17
<untyped structure declaration>	4-15, 4-15*
<untyped structure head>	4-15, 4-15*
<untyped structure name>	4-15, 4-15*, 4-15, 4-24
<user function reference>	5-14, 5-15*
<user procedure call phrase>	6-28, 6-29*
<uyk-43 floating constant>	B-16, B-16*
<uyk-7 floating constant>	B-16, B-16, B-16*
<validity test>	5-74, 5-74*
<value block>	6-105, 6-105*
<value block body>	6-105, 6-105*
<value block head>	6-105, 6-105*
<variable declaration>	4-24*, 7-8, 8-2, 8-8
<variable list>	4-24, 4-24*, 4-29
<variable name>	4-20, 4-24, 4-24, 4-24*, 4-54, 4-60, 4-63, 4-63, 4-75, 4-79, 4-79, 4-91, 4-131, 5-3, 5-7, 5-19, 6-5, 6-66, B-25

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
<varying clause>	6-116, 6-116*
<word data unit>	5-2, 5-7*, 5-10, 5-19, 5-49, 5-64, 5-69, 5-71, 5-74, 6-5, 6-66
<word specification>	5-7, 5-7, 5-7*, 6-66
0<octal integer>	B-17
<ISFPS>	B-16
A	4-8, 4-56, 4-108
ABS	5-18, B-36
ACOS	5-22
ACOS2	5-22
ACSEPARATION	9-35
ALG	6-53
ALL	9-40
ALOG	5-22
AND	5-64, 5-79, 6-116
ANDF	5-33
ASIN	5-22
ASIN2	5-22
ATAN	5-22
ATAN2	5-22
AUTO-DD	8-8
B	4-12, 4-101, 4-116
BAMS	5-25
BASE	9-31

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
BEGIN	6-94, 6-105
BIT	5-10, 6-5
BY	6-96
BYTE	B-37
C	4-116, 9-48
CALLING ONLY	9-25
CARDS	9-10, 9-12
CAT	5-77
CCOMN	9-10, 9-12, 9-16
CHAR	5-12, 6-5, B-38
CHECKID	6-61
CIRC	6-53
CLIST	9-16
CLOSE	6-58
CMODE	4-5
CMP	9-12
CMS-2	3-19
CNT	5-44
CNV	9-12
COBJT	9-12
COMMENT	2-2
COMP	5-64, 5-79
COMPF	5-33
CONF	5-38

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
CONVERTIN	6-79
CONVERTOUT	6-82
CORAD	4-20, 4-60, 5-19, 6-5
CORRECT	9-40
COS	5-22
CR	9-12
CRG	9-12
CRL	9-12
CSRCE	9-10
CSWITCH	10-2
CSWITCH-DEL	10-5
CSWITCH-OFF	10-4
CSWITCH-ON	10-4
D	3-11, 3-11, 4-5, 4-8, 4-116, B-17, B-17
DATA	4-60, 6-116, 6-116
DATAPPOOL	9-31
DEBUG	4-129
DECODE	6-78
DEFID	6-60
DELETE	4-129
DENSE	4-15, 4-32
DEP	9-51
DIRECT	3-19, 9-25
DISPLAY	4-129, 6-84

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
DO	B-39
E	3-11, 3-14, 4-108, 4-116
ELSE	6-105
ELSIF	6-113
ENCODE	6-76
END	6-110
END-AUTO-DD	8-8
END-CSWITCH	10-2
END-CSWITCHS	10-2
END-FUNCTION	7-5
END-HEAD	9-3
END-LOC-DD	8-6
END-P-SW	4-94, 4-96
END-PROC	7-3
END-SUB-DD	7-8
END-SWITCH	4-87, 4-89, 4-94, 4-96
END-SYS-DD	8-2
END-SYS-PROC	8-4
END-SYSTEM	9-1
END-TABLE	4-33
END-TRACE	6-91
END-TYPE	4-15
ENDFILE	6-59
EQ	5-67, 5-76

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
EQUALS	4-68, 4-68, 4-72, 4-75
EVENP	5-74
EXCHANGE	4-66
EXEC	6-52
EXEC-PROC	4-82
EXECUTIVE	9-29
EXIT	4-79, 6-29, 6-48
EXP	5-22
EXTDEF	4-2, 4-15, 4-15, 4-15
EXTREF	4-2, 4-2
F	4-8, 4-108
FIELD	4-39
FIL	5-45, 6-62
FILE	4-101, 4-101
FIND	6-116
FIRST	5-31
FOR	6-105
FORM	B-40
FORMAT	4-108
FOUND	6-116
FROM	6-96
FUNCTION	4-84
GOTO	6-20, 6-22, 6-25
GT	5-67

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
GTEQ	5-67
H	3-17, 4-13, 4-32, 4-101
HEAD	9-3
I	4-8, 4-108, 4-116
ICOS	5-25
IEXP	5-25
IF	6-113, 6-116, 6-116
INDIRECT	4-32, 4-56
INPUT	4-79, 4-82, 6-29, 6-56, 6-74
INPUTLIST	4-125
INTO	6-53
INVALID	5-74, 6-22
ISIN	5-25
ITEM-AREA	4-54
KEY1	6-20, 6-43
KEY2	6-20, 6-43
KEY3	6-20, 6-43
L	4-108, 9-48
LAST	5-32
LENGTH	5-47
LEVEL	9-18, 9-18
LIBS	9-39
LIKE-TABLE	4-47
LIST	9-10

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
LISTING	9-16
LN	5-25
LOAD-VRBL	4-72
LOC-DD	8-6
LOC-INDEX	4-78
LOCDDPOOL	9-31
LOCREF	4-2
LOG	6-53
LT	5-67
LTEQ	5-67
MEANS	4-66
MEDIUM	4-15, 4-32
MODE FIELD	4-20
MODE VRBL	4-20
MODEVRBL	9-22
MONITOR	9-19
MSCALE	9-23
MT1	4-101
MT10	4-102
MT11	4-102
MT12	4-102
MT13	4-102
MT14	4-102
MT15	4-102

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
MT16	4-102
MT2	4-101
MT3	4-101
MT4	4-101
MT5	4-102
MT6	4-102
MT7	4-102
MT8	4-102
MT9	4-102
NITEMS	4-72
NOLIST	9-40
NONE	4-15, 4-32
NONRT	9-20
NOT	5-67, 5-76
NOTFOUND	6-116
O	3-14, 4-5, 4-108, 4-116, 9-48
OBJECT	9-12
OCM	4-101, 6-66, 6-74
ODDP	5-74
ONLY	9-40
OPEN	6-56
OPTIONS	9-6, 9-6
OR	5-64, 5-79, 6-116

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
ORF	5-33
OUTPUT	4-79, 6-29, 6-34, 6-34, 6-34, 6-34, 6-56, 6-66
OUTPUTLIST	4-127
OVERFLOW	6-5
OVERLAY	4-44, 4-63
P	4-20, 4-24, 4-29, 4-39, 4-72, 4-94, 4-96
P-SWITCH	4-94, 4-96, 4-98
PARAMETER	4-29
PASSAGE-SPEC	9-25
POS	5-46, 6-64
PPTP	4-102
PPTR	4-102
PRED	5-30
PRINT	4-101, 6-66, 9-16
PROCEDURE	4-79
PTRACE	4-129
PUNCH	4-101, 6-66
R	4-8, 4-101
RAD	5-25
RANGE	4-129, 4-131
READ	4-101, 6-74
REGISTER	9-25
REGS	6-84

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
REM	5-42
RES	B-41
RESUME	6-50
RETURN	6-45, 6-45
ROTATEHP INPUT	6-34
ROTATEP INPUT	6-34
S	4-8, 4-8, 4-14, 4-87, 4-89, 4-101, 9-48
SA	9-12
SADUMP	9-12
SAVING	6-5
SCALF	5-36
SCR	9-12
SCRATCH	6-56
SCRG	9-12
SCRL	9-12
SEL-ELEM	9-40
SEL-HEAD	9-40
SEL-POOL	9-44
SEL-SYS	9-40
SET	6-5, 6-62, 6-64
SHIFT	6-53
SIN	5-22
SINGLE	9-28
SM	9-12

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
SNAP	4-129, 6-87
SOURCE	9-10
SPILL	9-30
STANDARD	6-60
STOP	6-20, 6-43
STOP5	6-20, 6-43
STOP6	6-20, 6-43
STOP7	6-20, 6-43
STRINGFORM	4-116
STRUCTURED	9-21
SUB-DD	7-8
SUB-TABLE	4-49
SUCC	5-29
SWAP	6-18
SWITCH	4-87, 4-89, 4-91
SYS-DD	8-2
SYS-INDEX	4-77
SYS-PRCC	8-4
SYS-PRCC-REN	8-4
SYSTEM	9-1
T	4-8, 4-108, 4-116, 9-31
TABLE	4-32, 4-56
TABLEPCC	9-31
TDEF	5-40

* Non-terminal symbol defined on this page.

<u>Symbol</u>	<u>Page No.</u>
THEN	6-1, 6-113, 6-113, 6-116, 6-116
THRU	6-96
TO	6-5, 6-62, 6-64
TRACE	4-129, 6-89
TRANSREF	4-2, 4-2
TYPE	4-15, 4-15, 4-16
U	4-8
UNTIL	6-97
USING	6-38
UYK43	9-6
UYK7	9-6
V	4-20, 4-32, 4-101
VALID	5-74
VARY	6-96
VARYING	6-116
VECTORHP INPUT	6-34
VECTORP INPUT	6-34
VRBL	4-24
WHILE	6-97
WITHIN	6-96
WITHLBL	4-101
X	4-108, 4-116
XOR	5-79

* Non-terminal symbol defined on this page.

	<u>Symbol</u>	<u>Page No.</u>
XORF		5-33
Z		4-116

* Non-terminal symbol defined on this page.

