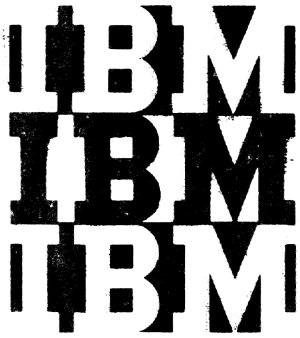


File No. S360-25
Form Y28-2019-1

Program Logic

IBM System/360 Time Sharing System

FORTRAN IV Compiler

This publication describes the internal logic of the IBM System/360 Time Sharing System (TSS/360) FORTRAN IV compiler.

This program logic manual is directed to the IBM customer engineer who is responsible for program maintenance. It can be used to locate specific areas of the program, and it enables the reader to relate these areas to the corresponding program listings. Program logic information is not necessary for program operation and use.

Second Edition (January 1970)

This is a major revision of, and makes obsolete, Form Y28-2019-0 and Technical Newsletters Y28-3057, Y28-3068, Y28-3082, Y28-3087, Y28-3091, and Y28-3097.

Changes on the actual pages are indicated as follows: A bullet (•) next to a page number indicates that the page has been substantially revised and should be reviewed in its entirety. A bullet next to the caption of an illustration indicates substantial revision of the illustration. A vertical bar in the left margin shows the location of a specific change; such revision bars are usually not shown on a page having a bullet next to the page number.

This edition is current with Version 6, Modification 0, of IBM System/360 Time Sharing System (TSS/360) and will remain in effect for all subsequent versions or modifications of TSS/360. Significant changes or additions to this publication will be provided in new editions or Technical Newsletters. Before using this publication in connection with the operation of IBM systems, refer to the latest edition of IBM System/360 Time Sharing System: Addendum, Form C28-2043, for the editions of publications that are applicable and current.

Specifications contained herein are subject to change from time to time. Any such change will be reported in subsequent revisions or Technical Newsletters.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments appears at the back of this publication. Address any additional comments concerning the contents of this publication to IBM Corporation, Time Sharing System/360 Programming Publications, Department 643, Neighborhood Road, Kingston, N. Y. 12401

© Copyright International Business Machines Corporation 1967, 1970

This publication describes the internal logic of the FORTRAN IV compiler.

Section 1 introduces the compiler's structure, briefly explaining the primary functions of each major division and describing the interrelationships of these divisions.

Sections 2 through 7 describe the six major divisions; they explain the logic required to implement the basic functions and objectives and provide a frame of reference for the program listings. Common data, such as tables and work areas, are discussed only to the extent required to understand the logic of the major divisions. Flowcharts compatible with the level of coverage are also provided, as are nesting charts which show the linkages among the subroutines that compose a major division; they show the called and calling relationship among the subroutines. In support of the nesting charts are decision tables that show the calling relationship among the subroutines and indicate the conditions under which subordinate subroutines are called.

All flowcharts for the routines are in Section 8, grouped in the same order as the routines are presented in the text.

The appendixes contain additional reference material.

PREREQUISITE READING

Understanding the material contained in this manual requires knowledge of the information contained in the following manuals:

- IBM System/360 Time Sharing System: IBM FORTRAN IV, Form C28-2007
- IBM System/360 Time Sharing System: Concepts and Facilities, Form C28-2003
- IBM System/360 Time Sharing System: System Logic Summary, Form Y28-2009

Manuals recommended for a fuller understanding of this manual are:

- IBM System/360 Time Sharing System: Command System User's Guide, Form C28-2001
- IBM System/360 Time Sharing System: Linkage Editor, Form C28-2005
- IBM System/360 Time Sharing System: Assembler Language, Form C28-2000
- IBM System/360 Time Sharing System: Assembler User Macro Instructions, Form C28-2004
- IBM System/360 Time Sharing System: FORTRAN Programmer's Guide, Form C28-2025

SECTION 1: INTRODUCTION 1
Object Program Modules 1
Subprogram Calls in OPM Text 1
Object Program Documentation 2
Compiler Interfaces 2
Interface With LPC 2
Interface With Virtual Storage Allocation 2
Interface With Data Management 2
Compiler/Service Routines Interface . 2
Organization of the Compiler 3
Compiler Executive Routine 3
Phase 1 3
Phase 2 4
Phase 3 6
Phase 4 7
Phase 5 7

SECTION 2: EXECUTIVE 8
Introduction 8
General Information 8
Macro Instruction Usage 8
Linkage Conventions 8
Register Notation and Conventions . . 9
Storage Map 9
Brief Routine Description 9
Use of the Phase Controller PSECT (EXCOM) by Other Exec Routines 11
Service External Interface 11
Source Statement Preparation 12
Phase and Interphase File Controller: The Compiler Work Areas and Intercom . . 12
Compiler Edit Lines 17
Compiler Diagnostic Information 17
Miscellaneous 19
Routine Descriptions 20
CEKTA -- Phase Controller (PHC) . . 20
CEKTC -- Get Next Source Statement (GNSS) 24
CEKTD -- Process Terminal Modifications (MOD) 26
CEKTE -- Receive Diagnostic Message (RDM) 28
CEKTF -- Constant Filers (CONFIL) . 28
CEKTH -- Master Input/Output (MIO) . 32
FORTRAN to GETLINE Call 32
GETLINE Entry 32
Line Number to GETLINE 32
Line Number From GETLINE 32
Length of Line 33
Source Line 33
Altered Line Table 33
GETLINE to FORTRAN Return 33
FORTRAN to PUTDIAG Call 33
PUTDIAG Entry 33
PUTDIAG to FORTRAN Return 33
Operation 33
CEKTI Analyze Console Source Line (ANALYZ) 35
CEKTJ -- Inspect a Console Character (INSCON) 35
CEKTK -- Move a Line to the List Data Set (LDMOVE) 35
CEKTL -- Build a List Data Set Buffer (BUILD) 36
CEKTM -- Flush a List Data Set Buffer (FLUSH) 36
CEKTQ -- Compiler File Dump (COMDUMP) 36
CEKTS -- Compiler Line Dump (LINDUMP) 36

SECTION 3: PHASE 1 39
Introduction 39
Program Representation File (PRF) . . 40
Begin Program Entry 41
Subprogram Entry 41
Alternate Entry 41
Label Definition Entry 41
Equation Entry 41
GO TO Entry 41
Assigned GO TO Entry 41
Computed GO TO Entry 41
ASSIGN Entry 41
Arithmetic IF Entry 42
Logical IF Entry 42
CALL Entry 42
Argument Definition Entry 42
RETURN Entry 42
Begin Loop Entry 42
End Loop Entry 42
CONTINUE Entry 42
READ, READ Without Unit, and READ With NAMELIST Entries 42
WRITE and WRITE With NAMELIST Entries 42
PRINT and PUNCH Entries 43
Input/Output List Representation in the PRF Entry 43
End List Entry 43
END FILE, REWIND, and BACKSPACE Entries 43
STOP Entry 43
PAUSE Entry 43
End Program Entry 43
Expression File 43
Subscript Expressions 43
Storage Specification Tables 43
Dimension Table 44
Namelist Table 44
Storage Class Table 44
Format Processing 44
Alphameric Constants 44
Data Processing 44
Cross Reference Index List 45
Phase 1 Routines, Functional Description 45
Pass 1 Statement Processors 45
Pass 2 Statement Processors 45
Expression Processing and Translation 45
Source Extraction and Conversion . . 45
Loop Processing Service Routines . . 45

I/O Statement Processor Service Routines 45
Initial Value Processing Service Routines 45
Miscellaneous Service Routines . . . 45
Routine Descriptions 46
CEKAD -- Phase 1 Main Loop (PH1M) . 55
CEKAK -- Assignment Statement Processor (EQUA) 55
CEKAM -- EXTERNAL Statement Processor (EXTE) 56
CEKAQ -- GO TO Statement Processor (GOTO) 56
CEKAR -- IF Statement Processor (IF) 56
CEKAS -- Type Statements Processor (TYPE) 56
CEKAT -- CONTINUE Statement Processor (CONT) 56
CEKAU -- DIMENSION Statement Processor (DIMN) 57
CEKAV -- COMMON Statement Processor (COMM) 57
CEKAY -- EQUIVALENCE Statement Processor (EQUI) 57
CEKAZ -- DO Statement Processor (DO) 57
CEKBC -- ASSIGN Statement Processor (ASSI) 57
CEKBD -- File Control Statement Processor (FCON) 58
CEKBE -- Input/Output Statement Processor (RWIO) 58
CEKBF -- FORMAT Statement Processor (FORM) 58
CEKBG -- PAUSE, STOP, RETURN Statement Processor (PSR) 59
CEKBH -- NAMELIST Statement Processor (NAML) 60
CEKBI -- BLOCK DATA Statement Processor (BLDA) 60
CEKBM -- DATA Statement Processor (DATA) 60
CEKBN -- IMPLICIT Statement Processor (IMPL) 60
CEKBR -- Blank Statement Processor (BLNK) 60
CEKBS -- Subprogram Entry Statements Processor (SUBE) 61
CEKBU -- CALL Statement Processor (CALL) 61
CEKAL -- END Statement Processor (END) 61
CEKAW -- Declaration Statements, Pass 2 (DCL2) 61
CEKAX -- Executable Statements, Pass 2 (EXEC2) 61
CEKBJ -- BLOCK DATA Statement, Pass 2 (BLDA2) 62
CEKBP -- IMPLICIT Statements, Pass 2 (IMPL2) 62
CEKBT -- Subprogram Entry Statements, Pass 2 (SUBE2) 62
CEKBV -- CALL Statement, Pass 2 (CALL2) 62
CEKBZ -- Statement Function Definition, Pass 2 (STFN2) 63
CEKAG -- Subscript Processor (SUBS) 63
CEKAI -- Expression Processor (EXPR) 6
H to S Class 65
S to H Class 65
S to S Class 65
H to H Class 65
Function/Subroutine Call 67
CEKAN -- Conversion Subroutine (CNVRT) 68
CEKBK -- Statement Function Definition (SFDEF) 69
CEKBL -- Statement Function Expansion (SFEXP) 70
CEKBX -- Function Classifier (FNCLS) 70
CEKBY -- Library Function Selector (LIBN) 70
CEKCB -- Constant Arithmetic Subroutine (ARITH) 71
CEKCG -- Term Processor (TRMPRO) . . 71
CEKCR -- Actual Argument Service Routine (AARG) 71
CEKCS -- Constant Arithmetic Interrupt (CHKINT) 72
CEKAB -- Extract Source Character (ESC) 72
CEKAE -- Assemble Components (ACOMP) 73
CEKCH -- File Real Constant (FLRC) . 74
CEKCI -- Insert Variable in Symbol Table (IVST) 74
CEKCN -- Decimal to Binary Integer Conversion (ICNV) 74
CEKCP -- Decimal to Floating Binary Conversion (FCNV) 74
CEKCQ -- File Integer Constant (FLIC) 74
CEKBA -- Begin Loop Processor (BGNLP) 75
CEKBB -- End Loop Processor (ENDLP) 75
CEKCJ -- Check Limits (CKLIM) . . . 76
CEKCK -- Clear Limits (CLLIM) . . . 76
CEKBW -- I/O List Processor (IOLST) 76
CEKCD -- Format Label Processor for I/O Statements (FLABL) 76
CEKCE -- Read Transfer Processor for I/O Statements (RTRAN) 76
CEKCF -- FORMAT or NAMELIST Name Processor (FNAME) 76
CEKAH -- Initial Value Data Specification Processor (IDATA) . . 77
CEKCL -- Initial Value Processor (IVAL) 77
CEKAF -- Array Dimension Specification Processor (ARDIM) . . 77
CEKCC -- Label String Processor (LBSTR) 77
CEKAC -- Statement of Identification (SID) 78
CEKAJ -- Statement Label Processor (LABL) 78
CEKBQ -- Fallthrough Determination (FALTH) 79
CEKCA -- Diagnostic Message Generator (ERR) 80
SECTION 4: PHASE 2 81
Introduction 81

Routine Descriptions 81
CEKJA -- (PHASE2) 81
CEKJC -- Storage Assignments for Variables (VSCAN) 81
CEKJB -- Process Label References and Definitions (FSCAN) 88
Label Processing 89
Flow Processing 89
DO Loop Processing 89
CEKJD -- Label Reference Processor (RTN1) 90
CEKJE -- Label Reference Processor (LAB) 90
CEKJF -- Statement Label Reference Inspection (ISP) 91
CEKJG -- Format Reference Inspection (FORMAT) 91
CEKJH -- Diagnostic Message Generator (DX) 91
SECTION 5: PHASE 3 92
Introduction 92
Memory Reference Processing 96
Common Expressions 96
Removing Expressions From Loops . . . 98
Optimizing Subscript Computation . . . 98
Loop Variable Expressions 100
Global Register Assignment 101
Routine Descriptions 101
CEKKS -- Phase 3 Storage (PSECT) . .106
Phase 3 Loop Tables 106
Global Register List (GIRL) (Linked, Permanent) 106
Link Pointers 106
Operand Pushdown 107
Triad Table Entry 107
Polish Insertion Entries 108
Compute and Removal Item Table . . .108
Hash Table for Compute and Removal Table (CRT) and Triad Entries (HCRT) 109
Formal Argument Adcon Table 109
CEKKR -- Phase 3 Master Control Routine 109
CEKKU -- PRF Processing Routine . .110
CEKKC -- End Loop PRF Entry Routine 113
CEKKV -- Begin Loop 1 PRF Processor 113
CEKKW -- Begin Loop 2 PRF Processor 114
File EF and Point (FEFP) Subroutine 115
Delete the Undefined Level (DUNL) Subroutine 115
CEKKE -- Expression Scan Routine . .115
CEKLF -- Copy and Edit an Expression 116
CEKKF -- Pushdown Primitive Operand Routine 118
"AT" Operator Insertion 118
Subscripted Variable Processing . .118
Loop Parameter Processing 119
CEKKG -- Variable Compute Point and Removal Level Routine 119
CEKKL -- Operand List Expression Formation Routine 119
CEKKH -- Triad File Manipulation Routine 120
CEKKP -- Search and Insert Triads .120
CEKKN -- Canonical Form Routine . .121
CEKKI -- Expression Removal and Commonality Determination Routine .121
CEKKJ -- Check Commonality 122
CEKKK -- Establish Common Expression Routine 123
CEKLA -- Label Common Expressions .123
CEKLE -- File CRT Entries 123
CEKLD -- Expunge a Removable Subexpression 124
CEKKM -- Subscript Expression Revision Routine 124
CEKKA -- Acquire Entry from Compute and Removal Table 125
CEKKB -- Polish Expression Generation Routine 125
CEKKO -- Save Popularity Counts for Register Assignment 126
CEKL3 -- File Constant and Covering Adcon 126
CEKLI -- Loop Test-Expression Generator 127

SECTION 6: PHASE 4 129
Introduction 129
Phase 4 Processing 129
Expression Generation 130
Expression Storage 132
Common-Expression Storage 132
Register Storage 133
General Register Selection 134
Storage Reference Processing 135
DO Loop Processing 136
Routine Descriptions 139
CEKNX -- Phase 4 Master Control (PHAS4) 153
CEKOD -- Entry Point Processor (ENT) 153
CEKNU -- Referenced Label PF Entry Processor (LABEL) 153
CEKMJ -- Equation PF Entry Processor (EQUAT) 155
CEKNK -- Arithmetic IF PF Entry Processor (AIF) 155
CEKNL -- Logical IF PF Entry Processor (LIF) 155
CEKNT -- GO TO PF Entry Processor (GOTO) 155
CEKNS -- Assign PF Entry Processor (ASSGN) 155
CEKNQ -- Assigned GO TO PF Entry Processor (AGO) 156
CEKNR -- Computed GO TO PF Entry Processor (CGO) 156
CEKOL -- CALL Statement Processor (CALL) 156
CEKOE -- RETURN Processor (RTRN) . .156
CEKNM -- Begin Loop 1 PF Entry Processor (BL1) 156
CEKNN -- Begin Loop 2 PF Entry Processor (BL2) 157
CEKNO -- Begin Loop 3 PF Entry Processor (BL3) 157
CEKNP -- End Loop PF Entry Processor (ENDLP) 157
CEKOH -- I/O Statement PF Entry Processor (RD) 158
CEKOI -- I/O List Element PF Entry Processor (OLIST) 158
CEKOJ -- End List PF Entry Processor (NDLST) 158

CEKOK -- STOP and PAUSE Statement PF Entry Processor (STOP) 159
CEKNW -- Arithmetic Expression Generator (AGEN) 159
CEKML -- Expression Tree Builder (TRBLD) 159
CEKNE -- Weight Subroutine (WGHT) .161
CEKOB -- Common Expression Usage Count (CSX) 162
CEKMC -- Real Plus Generator (RPLUS) 162
CEKMB -- Real Multiply Generator (RMUL) 162
CEKMA -- Real Divide Generator (RDIV) 163
CEKMF -- Integer Plus Generator (IPLUS) 163
CEKME -- Integer Multiply Generator (IMPLY) 164
CEKMD -- Integer Divide Generator (IDVDE) 164
CEKOV -- Add by Load Address (LADDR) 164
CEKMG -- Complex Plus Generator (CPLUS) 165
CEKOF -- Complex Multiply Generator (CMUL) 165
CEKOG -- Complex Divide Generator (CDIV) 166
CEKMH -- Relational Expression Generator (RLTNL) 166
CEKMI -- Logical Expression Generator (ANDOR) 167
CEKMU -- Maximum Operator Generator (MAX) 167
CEKMK -- External Function Generator (FUNC) 168
CEKNJ -- Comma Operator Processing Subroutine (COMMA) 169
CEKOM -- Open Function Control Routine (DCOM) 170
CEKOT -- Open Function Processing Routine (OPEN1) 170
CEKOU -- Open function Processing Routine (OPEN2) 171
CEKOX -- Open Function Processing Routine (OPEN3) 171
CEKOY -- Open Function Processing Routine (OPEN4) 171
CEKOZ -- Open Function Processing Routine (OPEN5) 171
CEKOM2 -- Open Function Processing Routine (OPEN6)* 171
CEKMV -- Memory Access Routine (MEMAC) 172
CEKOP -- Load Covering Adcon Routine (COVER) 172
CEKMZ -- Local Branch Generator (SADDR) 172
CEKNV -- Labeled Branch Generator (LBL) 173
CEKOS -- Operand Fetch Complement/Store Routine (FETCH) . .173
CEKND -- Select Operand Routine (SELOP) 173
CEKNF -- Select Position for Operation (SLPOS) 175
CEKOW -- Select One Operand in a Register (SLONE) 175
CEKNB -- Determine Availability of Register for Multiplication (SELGM) 176
CEKNA -- General Register Availability for Integer Divide (SELGD) 177
CEKOC -- Operand Status Routine (KEY) 177
CEKOR -- Single Operand Locating Routine (KEY1) 177
CEKMR -- Search General Registers (FNDAR) 178
CEKMS -- Search Floating Registers (FNDFR) 178
CEKMW -- Operand Processing Routine (OPND) 178
CEKMY -- Result-Register Operand Processing Subroutine (RSLT) 179
CEKNG -- Select Single General Register (SELSR) 180
CEKNH -- Select Even/Odd General Register Pair (SELDR) 181
CEKMQ -- Select Floating Register (SELFR) 181
CEKMM -- Make Initial Assignment to General Register (ASAR) 181
CEKMN -- Make Synonym Assignment to General Register (ASARS) 182
CEKMO -- Make Initial Assignment to Floating-Point Register (ASFR) . . .182
CEKMP -- Make Synonym Assignment to Floating Register (ASFRS) 182
CEKMT -- Find Temporary Storage (FNDWS) 183
CEKMX -- Release Temporary Storage (RLSWS) 183
CEKON -- Register Storage Clear Routine (FLUSH) 183
CEKNI -- Code File Output Subroutine (INSOT) 184
CEKOQ -- Edit for Code File (EDIT) .184

SECTION 7: PHASE 5 186
Introduction 186
Object Program Module (OPM) 187
Program Module Dictionary (PMD) . . .187
PMD Heading 187
Control Section Dictionary (CSD) . .189
Internal Symbol Dictionary (ISD) . . .194
Heading 194
Section Name Table 194
Statement Number Table 194
Symbol Table 194
Routine Descriptions 195
CEKSA -- FORTRAN Compiler Output Generator (PHASE5) 195
CEKSB -- Object Program Module Builder (BUILD) 198
CEKSC -- Common Control Section Generator (CMSEC) 199
CEKSF -- Code Control Section Generator (COSEC) 200
CEKSG -- PSECT Builder (PRSEC) . . .203
CEKSD -- Preset Data Processor (SPECS) 206
CEKSH -- Internal Symbol Dictionary Generator (ASSIST) 206

CEKSI -- Object Program Documentation (EDIT) 207
CEKSJ -- Symbol Table Sort (SYMSRT) 211
CEKSE -- Output Page Heading (PHEAD) 212
CEKSL -- Constant Conversion (CONCV) 212
CEKSK -- Cross Reference List Routine (CRFSRT) 212

SECTION 8: FLOWCHARTS 214

APPENDIX A: INTERPHASE TABLE AND FILE FORMATS 631
Program Representation File (PRF) . . .631
Storage Specification Tables 636
Preset Data Tables 639
Entry Formats 639
Storage Class Table (STCLTB) 641
Program File (PF) Formats Output by Phase 3 641
Field Identifiers 642
Entry Formats 642
Code File Format 645
Symbol Table 645
General Format 645
Specific Descriptive Part Formats of Intrinsic and Library Functions .646
Constant Format 648
Label Format 649
Address Constant Format 649
Intercom Table 650

APPENDIX B: TSS/360 LINKAGE CONVENTIONS 657
Introduction 657
Conventions For Type I Linkages (Standard) 657
Register Conventions 657
Save Area 657
Parameter List, Type I Linkage . . .658
Type I Linkage, Return and Entry Linkage and Return Code 658
Restricted Linkage Conventions 658
Scope and Applicability of Restricted Linkage 658
Register Usage and Assignment in Restricted Linkage 658
Macro Instruction Support 659
INVOKE Macro 659
STORE Macro 660
RESUME Macro 660

APPENDIX C: FORTRAN INTERNAL MACRO INSTRUCTION USAGE 662

APPENDIX D: LIST OF MAJOR TABLES REFERENCED BY FORTRAN ROUTINES 664

APPENDIX E: MODULE DICTIONARY 667

APPENDIX F: LINKAGE EDITED COMPILER ROUTINES LISTED BY CODED LABELS (MODULE NAMES) 679

INDEX 684

ILLUSTRATIONS

FIGURES

Figure 1. FORTRAN IV Compiler External References 3
Figure 2. Compiler Component Organization 4
Figure 3. Compiler Information Flow . 5
Figure 4. Compiler Interfaces 12
Figure 5. Source-Statement-Preparation Modules 13
Figure 6. Summary of Phase and Interphase File Control Activities . . 14
Figure 7. Symbol Table Storage Layout 15
Figure 8. Process Compiler Edit Line Function 16
Figure 9. Compiler Diagnostic Features 19
Figure 10. Testing for Diagnostic Input and Processing Diagnostic Information Lines 20
Figure 11. Processing Diagnostic Information Following Return From Each Phase 21
Figure 12. Processing of Unexpected Interruptions During Compilation 22
Figure 13. Phase 1 Interface 39
Figure 14. Phase 1 Storage 40
Figure 15. Phase 1 Nesting Chart . . . 47
Figure 16. Symbol Table Save Area . . . 55
Figure 17. Component Storage Area . . 73
Figure 18. Phase 2 Nesting Chart . . . 82
Figure 19. Phase 2 General Flow . . . 84
Figure 20. Sort Table Entry 84
Figure 21. Variable List, Group Connection List, and Group Table Entries 85
Figure 22. Phase 3 Nesting Chart . . .102
Figure 23. Phase 3 Storage Map 106
Figure 24. Expression Tree 131
Figure 25. Name Table 133
Figure 26. MRM Table 133
Figure 27. MRMFR Table 134
Figure 28. Loop Table 137
Figure 29. Phase 4 Nesting Chart . . .140
Figure 30. Phase 4 Master Control . .154
Figure 31. I/O Initialization Parameter List 158
Figure 32. I/O Initialization Control Bytes 158
Figure 33. Stack Table Entry 160
Figure 34. INSOT Input Parameters . . .184
Figure 35. Format of PMD Entry 188
Figure 36. FORTRAN Internal Symbol Dictionary 194
Figure 37. Phase 5 Nesting Chart . . .196
Figure 38. Phase 5 General Flow 199
Figure 39. Output Listing Format (Part 1 of 2) 209
Figure 40. CEKTD, Compiler Exec Process Terminal Modifications (Part 1 of 4) 651
Figure 41. Alphabetically Sorted Listing of Intercom Items, With Displacements (Part 1 of 2) 655

Table 1. Executive Storage Map 10
Table 2. Work Area A Storage Layout . 15
Table 3. Work Area B Storage Layout . 15
Table 4. Work Area C Storage Layout . 15
Table 5. Preparation of Constant Receiving Area by CONFIL 29
Table 6. Constant Chain Anchors and Table Bases 30
Table 7. CONFIL Storage Assignment No-Hole Branch Table TFNOHO 30
Table 8. CONFIL Storage Assignment Hole Availability Table 31
Table 9. CONFIL Storage Assignment Byte Alignment Branch Table TFBAL . . . 32
Table 10. Phase 1 Decision Table (Part 1 of 8) 48
Table 11. Encoding of FORMAT Symbols . 58
Table 12. Translation of Format Codes . 59
Table 13. Operator Precedence 66
Table 14. EXPF Entries (Real Base) . . 69
Table 15. Library Function Names . . . 71
Table 16. Assemble Components Character Table 74
Table 17. Assemble Components Decision Table 75
Table 18. Assignment/Nonassignment - Character Table 78
Table 19. Assignment/Nonassignment Precedence Table 78
Table 20. Nonassignment Type Statement Identification 79
Table 21. Statement ID Numbers 79
Table 22. Phase 2 Decision Table . . . 83
Table 23. Phase 3 Decision Table (Part 1 of 4) 102
Table 24. Phase 4 Decision Table (Part 1 of 12) 141
Table 25. Operand Conversion Function Decision Table 160
Table 26. Complex Division Left Operand Conversion Function Decision Table 161
Table 27. Operand Types Processed by CMUL 165
Table 28. Operand Types Processed by CDIV 166
Table 29. Phase 5 Decision Table (Part 1 of 3) 196

CHARTS

Chart AA. Executive Overall Flow -- CEKUA (Page 1 of 2) 216
Chart AB. Phase Controller (PHC) -- CEKTA (Page 1 of 5) 218
Chart AC. Get Next Source Statement (GNSS) -- CEKTC (Page 1 of 4) 223
Chart AD. Process Terminal Modifications (MOD) -- CEKTD (Page 1 of 2) 227
Chart AE. Receive Diagnostic Message (RDM) -- CEKTE 229
Chart AF. Constant Filers (CONFIL) -- CEKTF (Page 1 of 8) 230
Chart AG. Master Input/Output Routine (MIO) -- CEKTH (Page 1 of 2) 238
Chart AH. Analyze Console Source Line (ANALYZ) -- CEKTI (Page 1 of 2) 240
Chart AI. Inspect a Console Character (INSCON) -- CEKTJ 242
Chart AJ. Move a Line to a List Data Set (LDMOVE) -- CEKTK 243
Chart AK. Build a List Data Set Buffer (BUILD) -- CEKTL 244
Chart AL. Flush a List Data Set Buffer (FLUSH) -- CEKTM 245
Chart AM. Phase 1 Main Loop (PH1M) -- CEKAD (Page 1 of 4) 246
Chart AN. Assignment Statement Processor (EQUA) -- CEKAK 250
Chart AO. EXTERNAL Statement Processor (EXTE) -- CEKAM 251
Chart AP. GO TO Statement Processor (GOTO) -- CEKAQ (Page 1 of 4) 252
Chart AQ. IF Statement Processor (IF) -- CEKAR (Page 1 of 2) 256
Chart AR. Type Statements Processor (TYPE) -- CEKAS (Page 1 of 4) 258
Chart AS. DIMENSION Statement Processor (DIMN) -- CEKAU 262
Chart AT. COMMON Statement Processor (COMM) -- CEKAV (Page 1 of 3) 263
Chart AU. EQUIVALENCE Statement Processor (EQUI) -- CEKAY (Page 1 of 2) 266
Chart AV. DO Statement Processor (DO) -- CEKAZ 268
Chart AW. ASSIGN Statement Processor (ASSI) -- CEKBC (Page 1 of 2) 269
Chart AX. File Control Statement Processor (FCON) -- CEKBD (Page 1 of 2) 271
Chart AY. Input/Output Statement Processor (RWIO) -- CEKBE (Page 1 of 6) 273
Chart AZ. FORMAT Statement Processor (FORM) -- CEKBF (Page 1 of 6) 279
Chart BA. PAUSE, STOP, RETURN Statement Processor (PSR) -- CEKBG (Page 1 of 4) 285
Chart BB. NAMELIST Statement Processor (NAML) -- CEKBH (Page 1 of 2) 289
Chart BC. BLOCK DATA Statement Processor (BLDA) -- CEKBI 291
Chart BD. DATA Statement Processor (DATA) -- CEKBM 292
Chart BE. IMPLICIT Statement Processor (IMPL) -- CEKBN (Page 1 of 3) 293
Chart BF. Subprogram Entry Statement Processor (SUBE) -- CEKBS (Page 1 of 5) 296
Chart BG. END Statement Processor (END) -- CEKAL 301
Chart BH. Executable Statements, Pass 2 (EXEC2) -- CEKAX 302
Chart BI. Subprogram Entry Statements, Pass 2 (SUBE2) -- CEKBT . .303
Chart BJ. CALL Statement, Pass 2 (CALL2) -- CEKBV 304
Chart BK. Subscript Processor (SUBS) -- CEKAG (Page 1 of 6) 305
Chart BL. Expression Processor (EXPR) -- CEKAI (Page 1 of 13) 311
Chart BM. Conversion Subroutine (CNVRT) -- CEKAN (Page 1 of 3) 324
Chart BN. Statement Function Definition (SFDEF) -- CEKBK 327
Chart BO. Statement Function Expansion (SFEXP) -- CEKBL (Page 1 of 2) 328
Chart BP. Function Classifier (FNCLS) -- CEKBX 330
Chart BQ. Library Function Selector (LIBN) -- CEKBY 331
Chart BR. Constant Arithmetic Subroutine (ARITH) -- CEKCB 332
Chart BS. Term Processor (TEMPRO) -- CEKCG 333
Chart BT. Actual Argument Service Routine (AARG) -- CEKCR 334
Chart BU. Constant Arithmetic Interrupt (CHKINT) -- CEKCS 335
Chart BV. Extract Source Character (ESC) -- CEKAB 336
Chart BW. Assemble Components (ACOMP) -- CEKAE (Page 1 of 8) 337
Chart BX. File Real Constant (FLRC) -- CEKCH 345
Chart BY. Insert Variable in Symbol Table (IVST) -- CEKCI 346
Chart BZ. Decimal to Binary Integer Conversion (ICNV) -- CEKCN 347
Chart CA. Decimal to Floating Binary Conversion (FCNV) -- CEKCP 348
Chart CB. Begin Loop Processor (BGNLP) -- CEKBA (Page 1 of 2) 349
Chart CC. End Loop Processor (ENDLP) -- CEKBB 351
Chart CD. Check Limits (CKLIM) -- CEKCJ 352
Chart CE. I/O List Processor (IOLST) -- CEKBW (Page 1 of 4) 353
Chart CF. Format Label Processor for I/O Statements (FLABL) -- CEKCD 357
Chart CG. Read Transfer Processor for I/O Statements (RTRAN) -- CEKCE 358
Chart CH. FORMAT or NAMELIST Processor (FNAME) -- CEKCF 359

Chart CI. Initial Value Data Specification Processor (IDATA) -- CEKAH (Page 1 of 2) 360
Chart CJ. Initial Value Processor (IVAL) -- CEKCL (Page 1 of 6) 362
Chart CK. Array Dimension Specification Processor (ARDIM) -- CEKAF (Page 1 of 2) 368
Chart CL. Label String Processor (LBSTR) -- CEKCC 370
Chart CM. Statement of Identification (SID) -- CEKAC 371
Chart CN. Statement Label Processor (LABL) -- CEKAJ (Page 1 of 2) 372
Chart CO. Fallthrough Determination (FALTH) -- CEKBQ 374
Chart CP. Diagnostic Message Generator (ERR) -- CEKCA (Page 1 of 4) .375
Chart CQ. Memory Assignments for Variables (VSCAN) -- CEKJC (Page 1 of 11) 379
Chart CR. Process Label References and Definitions (FSCAN) -- CEKJB (Page 1 of 8) 390
Chart CS. Label Reference Processor (RTN1) -- CEKJD 398
Chart CT. Label Reference Processor (LAB) -- CEKJE (Page 1 of 2) 399
Chart CU. Diagnostic Message Generator (DX) -- CEKJH (Page 1 of 2) .401
Chart CV. Phase 3 Master Control Routine -- CEKKR (Page 1 of 4) 403
Chart CW. PRF Processing Routine -- CEKKU (Page 1 of 12) 407
Chart CX. End Loop PRF Entry Routine -- CEKKC (Page 1 of 2) 419
Chart CY. Begin Loop 1 PRF Processor -- CEKKV (Page 1 of 3) 421
Chart CZ. Begin Loop 2 PRF Processor -- CEKKW (Page 1 of 3) 424
Chart DA. Expression Scan Routine -- CEKKE (Page 1 of 3) 427
Chart DB. Copy and Edit an Expression -- CEKLF (Page 1 of 5) 430
Chart DC. Push Primitive Operand Routine -- CEKKF (Page 1 of 2) 435
Chart DD. Variable Compute Point and Remove Level Routine -- CEKKG (Page 1 of 2) 437
Chart DE. Operand List Expression Formation Routine -- CEKKL 439
Chart DF. Triad File Manipulation Routine -- CEKKH (Page 1 of 2) 440
Chart DG. Search and Insert Triads -- CEKKP 442
Chart DH. Canonical Form Routine -- CEKKN (Page 1 of 2) 443
Chart DI. Expression Removal and Commonality Determination Routine -- CEKKI (Page 1 of 5) 445
Chart DJ. Establish Common Expression -- CEKKK 450
Chart DK. Check Commonality -- CEKKJ .451
Chart DL. Label Common Expressions -- CEKLA 452
Chart DM. File CRT Entries -- CEKLE . .453
Chart DN. Expunge a Removable Expression -- CEKLD 454
Chart DO. Subscript Expression Revision Routine -- CEKKM (Page 1 of 3) 455
Chart DP. Acquire Entry From Compute and Removal Table -- CEKKA 458
Chart DQ. Polish Expression Generation Routine -- CEKKB (Page 1 of 5) 459
Chart DR. Save Popularity Counts for Register Assignment -- CEKKO 464
Chart DS. File Constant and Covering Adcon -- CEKLB 465
Chart DT. Loop Test-Expression Generator -- CEKLI (Page 1 of 6) 466
Chart DU. Entry Point Processor (ENT) -- CEKOD (Page 1 of 2) 472
Chart DV. Referenced Label PF Entry Processor (LABEL) -- CEKNU 474
Chart DW. Equation PF Entry Processor (EQUAT) -- CEKMJ 475
Chart DX. Arithmetic IF PF Entry Processor (AIF) -- CEKNK (Page 1 of 5) .476
Chart DY. Logical IF PF Entry Processor (LIF) -- CEKNL 481
Chart DZ. ASSIGN PF Entry Processor (ASSGN) -- CEKNS 482
Chart EA. Assigned GO TO PF Entry Processor (AGO) -- CEKNQ 483
Chart EB. Computed GO TO PF Entry Processor (CGO) -- CEKNR 484
Chart EC. CALL Statement Processor (CALL) -- CEKOL 485
Chart ED. RETURN Processor (RTRN) -- CEKOE (Page 1 of 3) 486
Chart EE. Begin Loop 1 PF Entry Processor (BL1) -- CEKNM 489
Chart EF. Begin Loop 2 PF Entry Processor (BL2) -- CEKNN (Page 1 of 9) .490
Chart EG. Begin Loop 3 PF Entry Processor (BL3) -- CEKNO (Page 1 of 3) .499
Chart EH. End Loop PF Entry Processor (ENDLP) -- CEKNP (Page 1 of 6) 502
Chart EI. I/O Statement PF Entry Processor (RD) -- CEKOH 508
Chart EJ. I/O List Element PF Entry Processor (ILIST) -- CEKOI (Page 1 of 2) 509
Chart EK. End List PF Entry Processor (NDLST) -- CEKOJ 511
Chart EL. STOP and PAUSE Statement PF Entry Processor (STOP) -- CEKOK 512
Chart EM. Arithmetic Expression Generator (AGEN) -- CEKNW (Page 1 of 2) 513
Chart EN. Expression Tree Builder (TRBLD) -- CEKML (Page 1 of 2) 515
Chart EO. Weight Subroutine (WGHT) -- CEKNE 517
Chart EP. Common Expression Usage Count (CSX) -- CEKOB (Page 1 of 3) . . .518
Chart EQ. Real Plus Generator (RPLUS) -- CEKMC (Page 1 of 3) 521
Chart ER. Real Multiply Generator (RMUL) -- CEKMB (Page 1 of 2) 524
Chart ES. Real Divide Generator (RDIV) -- CEKMA (Page 1 of 2) 526
Chart ET. Integer Plus Generator (IPLUS) -- CEKMF 528
Chart EU. Integer Multiply Generator (IMPLY) -- CEKME (Page 1 of 3) 529

Chart EV. Integer Divide Generator (IDVDE) -- CEKMD 532
Chart EW. Add by Load Address (LADDR) -- CEKOV (Page 1 of 2) 533
Chart EX. Complex Plus Generator (CPLUS) -- CEKMG (Page 1 of 3) 535
Chart EY. Complex Multiply Generator (CMUL) -- CEKOF (Page 1 of 2) 538
Chart EZ. Complex Divide Generator (CDIV) -- CEKOG (Page 1 of 2) 540
Chart FA. Relational Expression
Chart FB. Logical Expression Generator (ANDOR)-CEKMI 542A
Generator (RLTNL) -- CEKMH 542
Chart FC. Maximum Operator Generator (MAX) -- CEKMU 543
Chart FD. External Function Generator (FUNC) -- CEKMK (Page 1 of 3) 544
Chart FE. Comma Operator Processing Subroutine (COMMA) -- CEKNJ (Page 1 of 3) 547
Chart FF. Open Function Control Routine (DCOM) -- CEKOM 550
Chart FG. Open Function Processing Routine (OPEN1) -- CEKOT (Page 1 of 6) .551
Chart FH. Open Function Processing Routine (OPEN2) -- CEKOU (Page 1 of 3) .557
Chart FI. Open Function Processing Routine (OPEN3) -- CEKOX (Page 1 of 3) .560
Chart FJ. Open Function Processing Routine (OPEN4) -- CEKOY (Page 1 of 4) .563
Chart FK. Open Function Processing Routine (OPEN5) -- CEKOZ (Page 1 of 6) .567
Chart FL. Open Function Processing Routine (OPEN6) -- CEKOM2 (Page 1 of 2) 573
Chart FM. Memory Access Routine (MEMAC) -- CEKMV (Page 1 of 2) 575
Chart FN. Local Branch Generator (SADDR) -- CEKMZ 577
Chart FO. Labeled Branch Generator (LBL) -- CEKNV 578
Chart FP. Operand Fetch Complement/Store Routine (FETCH) -- CEKOS 579
Chart FQ. Select Operand Routine (SELOP) -- CEKND (Page 1 of 2) 580
Chart FR. Select Position for Operand (SLPOS) -- CEKNF (Page 1 of 2) 582
Chart FS. Select One Operand in a Register (SLONE) -- CEKOW (Page 1 of 4) 584
Chart FT. Determine Availability of Register for Multiplication (SELGM) -- CEKNB (Page 1 of 2) 588
Chart FU. General Register Availability for Integer Divide (SELGD) -- CEKNA 590
Chart FV. Operand Status Routine (KEY) -- CEKOC 591
Chart FW. Single Operand Locating Routine (KEY1) -- CEKOR 592
Chart FX. Search General Registers (FNDAR) -- CEKMR 593
Chart FY. Search Floating Registers (FINDFR) -- CEKMS 594
Chart FZ. Operand Processing Routine (OPND) -- CEKNW (Page 1 of 2) 595
Chart GA. Result-Register Operand Processing Subroutine (RSLT) -- CEKMY .597
Chart GB. Select Single General Register (SELSR) -- CEKNG (Page 1 of 2) 598
Chart GC. Select Even/Odd General Register Pair (SELDR) -- CEKNH (Page 1 of 2) 600
Chart GD. Select Floating Register (SELFR) -- CEKMQ (Page 1 of 3) 602
Chart GE. Make Initial Assignment to General Register (ASAR) -- CEKMM 605
Chart GF. Make Synonym Assignment to General Register (ASARS) -- CEKMN . . .606
Chart GG. Make Synonym Assignment to Floating Register (ASFRS) -- CEKMP . . .607
Chart GH. Find Temporary Storage (FNDWS) -- CEKMT 608
Chart GI. Release Temporary Storage (RLSWS) -- CEKMX 609
Chart GJ. Register Memory Clear Routine (FLUSH) -- CEKON 610
Chart GK. Code File Output Subroutine (INSOT) -- CEKNI 611
Chart GL. Object Program Module Builder (BUILD) -- CEKSB 612
Chart GM. Common Control Section Generator (CMSEC) -- CEKSC 613
Chart GN. Code Control Section Generator (COSEC) -- CEKSF (Page 1 of 3) 614
Chart GO. PSECT Builder (PRSEC) -- CEKSG 617
Chart GP. Present Data Processor (SPECS) -- CEKSD 618
Chart GQ. Internal Symbol Dictionary Generator (ASSIST) -- CEKSH 619
Chart GR. Object Program Documentation (EDIT) -- CEKSI (Page 1 of 3) 620
Chart GS. Symbol Table Sort (SYMSRT) -- CEKSJ 623
Chart GT. Constant Conversion (CONCV) -- CEKSL 624
Chart GU. Cross Reference List Routine (CRFSRT) -- CEKSK (Page 1 of 5) 625

The TSS/360 FORTRAN IV compiler is implemented in accordance with the conventions and requirements for systems programs in the TSS/360 environment. It is relocatable, reenterable, closed, nonprivileged, and nonresident.

The compiler organization and information flow are designed particularly for processing in the time-sharing environment. Wherever possible, to reduce the "page-turning" load on TSS/360, the intermediate data is organized and processed serially, in file form, rather than in a form requiring random access. The presence of the entire file in virtual storage ensures fast access to its contents; repeated references to the same virtual storage page, inherent in serial processing, reduces the number of pages needed in rapid succession.

While primarily a conventional batch-processor, the compiler contains special features making it especially suitable for conversational, terminal-oriented operation. The compiler syntax analysis performs statement-by-statement error checking of the source program as it is input through the Language Processor Control program (LPC). Diagnostic messages are returned to the user's terminal via LPC, and each appears at the terminal following the listing of the statement in which the error was detected. LPC gives the user the opportunity to correct the error, whether it be in the last statement processed, or in some earlier statement. Then LPC informs the compiler of whether a change was made and if so, which lines are affected. If only the last statement was changed, the compiler "forgets" the effect of the last statement and begins compilation with the statement replacing it. Otherwise, the compiler, under direction of LPC, restarts compilation from the beginning of the source program module. In this manner the most common errors, those local to the last statement processed, may be corrected with minimum loss of time.

After the END statement has been processed by the first phase, the compiler's second phase may detect errors of a more global nature (undefined statement labels, illegal DO-loop flow, etc.). The resulting error messages are passed to LPC, but now LPC does not allow the user to supply correction lines. When the compiler's second phase is complete, LPC gives the user the opportunity to correct errors or to go on. If errors are corrected, the compiler will recompile from the beginning of the stored source data set, and another conversation is possible. Otherwise, compilation proceeds to termination through the remaining compiler phases.

Detailed information concerning the conversation between terminal user and compiler is included in the description of the Compiler Executive routine (Exec), which interfaces with LPC.

OBJECT PROGRAM MODULES

The compiler produces an object program module (OPM) consisting of a program module dictionary (PMD), an optional internal symbol dictionary (ISD), text (the binary instructions and constants), and a list of external names.

The PMD contains heading information, used to identify the module, and a control section dictionary (CSD) for each control section occurring in the module. The CSD specifies which text entries require loader address computations or satisfaction of external references or references to other control sections. A complete specification of the PMD format is given in Section 7.

The ISD is a table of source language symbols (not subprogram references), the attributes associated with those symbols, and the control section and relative location within control section assigned to each. The ISD information is used by the Program Control System (PCS) to relate the user symbols with the definitions in the OPM. A complete specification of the ISD format is given in Section 7.

SUBPROGRAM CALLS IN OPM TEXT

The text does not contain the machine instructions that actually perform the input/output of data; nor does it contain the machine instructions to perform the more involved mathematical calculations such as those for finding the square root or the logarithm. The text also does not contain the machine instructions that actually perform such services as handling sense lights, overflows, underflows, exceptions, dumps, and the STOP, PAUSE, and CALL EXIT statements. The set of binary instructions produced by a compilation contains code for calls to library subprograms to perform these functions.

These subprograms are all permanently stored in SYSLIB, and consist of:

- FORTRAN I/O library subprograms. FORTRAN I/O source statements (READ, WRITE, BACKSPACE, ENDFILE, REWIND, PRINT or PUNCH) cause the compiler to insert, in the object code, calls to the appropriate FORTRAN I/O Library subprograms. Other FORTRAN I/O subprograms are used to execute the CALL DUMP, CALL PDUMP, CALL EXIT, STOP and PAUSE statements. Note: There are several service subprograms (STOP, PAUSE, CALL DUMP, CALL PDUMP, CALL EXIT) in the FORTRAN I/O group which do not, strictly speaking, perform I/O. These subprograms, however, were included in the FORTRAN I/O group because they use the FORTRAN data conversion routines. These subprograms are described under "Service Subprograms" in FORTRAN IV Library Subprograms.

- Mathematical Subprograms. These subprograms are used for the more complicated mathematical procedures. They are used to perform the explicitly referenced functions (for example, the sine function in X=SIN(Y) as well as to do the more involved computations for mathematical statements which do not explicitly reference a function (for example, the exponentiation in the statement X=Y**I). See FORTRAN IV Library Subprograms for information on these subprograms.

- The Service Subprograms that handle exceptions, pseudo-sense lights, overflows, underflows, and divide checks. For information on these, see FORTRAN IV Library Subprograms.

OBJECT PROGRAM DOCUMENTATION

In accordance with user-specified or defaulted options, the compiler produces the following documentation:

- A listing of the source program.

- An object program storage map giving the storage layout of the object program.

- A list of source program symbols and their storage equipments.

- A cross-reference listing relating symbols and statement numbers to the source line numbers of the statements in which they were referenced or defined.

- A listing of the object module in a representation very nearly in a form that might have been produced by the assembler.

Phase 5 of the compiler either places this information in the list data set, which is stowed by LPC, or writes it on SYSOUT.

COMPILER INTERFACES

All interface with LPC and other external routines is in the Compiler Executive routine (Exec).

INTERFACE WITH LPC

The Compiler Executive routine may be called by LPC at either of two points and may itself call LPC at either of two points (see Figure 1).

The two compiler entries are called INITIAL and CONTINUE. LPC calls the INITIAL entry to pass the user options to the compiler and to initiate the first stage of the compilation (Phases 1 and 2). LPC calls CONTINUE to complete the compilation after the first stage is finished. The compiler return from CONTINUE informs LPC of the size of the OPM's elements, so that LPC can dispose of them.

The compiler calls LPC at either of two places during the first stage (before the compiler returns to LPC from the INITIAL call). The first, GETLINE, is used to obtain a source line. The second, PUTDIAG, is used to pass a source error diagnostic message to LPC. PUTDIAG may also be used after the first stage.

INTERFACE WITH VIRTUAL STORAGE ALLOCATION

The compiler obtains virtual storage for the symbol table and other interphase files via GETMAIN; to release the storage, it uses FREEMAIN. (See Appendix A for a description of the interphase files, including the symbol table.)

INTERFACE WITH DATA MANAGEMENT

The compiler maintains the list data set by means of the virtual access method (VAM). The compiler issues OPEN, SETL, PUT, and CLOSE macro instructions to produce this data set.

COMPILER/SERVICE ROUTINES INTERFACE

The compiler "time-stamps" (includes the relative calendar time in) each object program module (OPM) control section that it produces. It also includes the date and

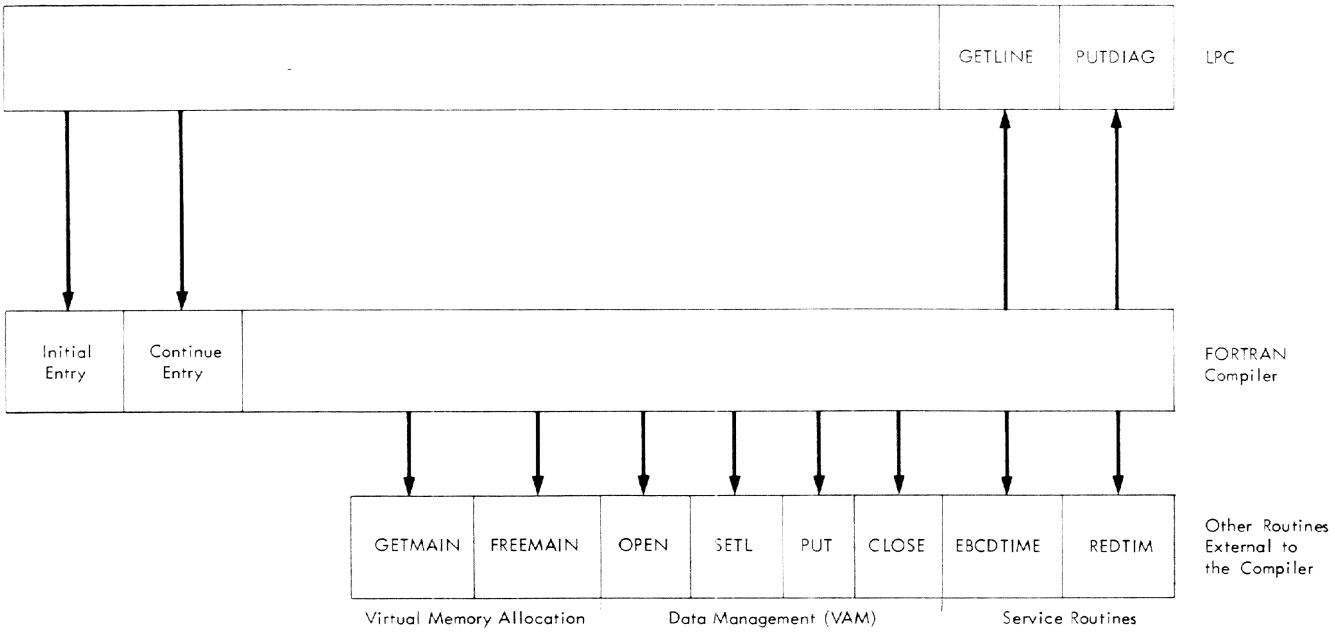


Figure 1. FORTRAN IV Compiler External References

time as identification on each sheet of listing that it produces. To do this, the compiler calls two service routines: REDTIM and EBCDTIME. REDTIM returns the time (in milliseconds elapsed since March 1, 1900), which is used to time-stamp the control sections and as input to EBCDTIME, which edits it into the EBCDIC representation of time of day for inclusion in the listing.

ORGANIZATION OF THE COMPILER

The compiler has six major components: a multifunction compiler executive and five compiler phases. The major functions of each component are summarized here; later sections describe each component in detail.

COMPILER EXECUTIVE ROUTINE

Compiler Executive (Exec) has six functions:

1. Interface with the compiler's environment.
2. Prepare the source statements for processing by Phase 1.
3. Control and order the operation of the phases (see Figure 2).
4. Prepare edited lines for output.
5. Provide compiler diagnostic information.
6. Provide miscellaneous services.

During a compilation, various tables and lists are constructed to contain the results of the operation of each phase and to serve as input to the next phase (see Figure 3).

PHASE 1

Phase 1 performs the source program syntactic analysis, detection and diagnosis of errors, and translation of the source program into a multitabular representation. Each identifier or constant is given an entry in the symbol table (format is shown in Appendix A). Initial values from DATA and type statements, dimension information for arrays, NAMELIST information, and alphameric constants are stored in the preset data table (Appendix A). Information concerning references to, and definitions of, symbols and statement numbers is stored in the cross reference table. Information collected from COMMON and EQUIVALENCE statements is stored in the storage specification list.

The most significant processing, from the point of view of later optimization and code generation, concerns the treatment of executable statements, statement numbers, and arithmetic expressions.

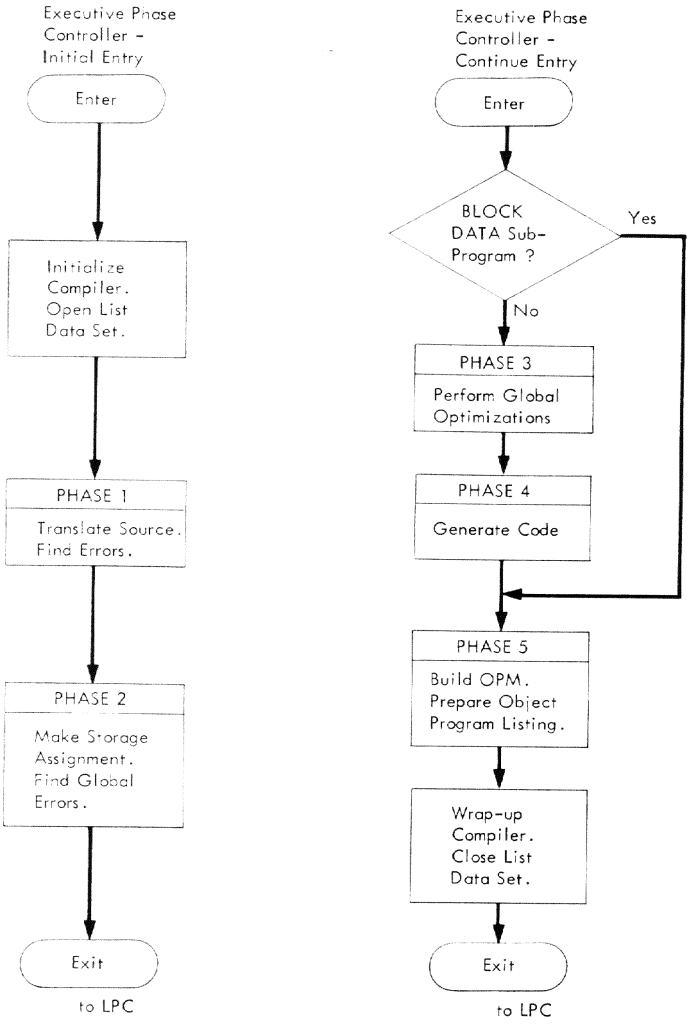


Figure 2. Compiler Component Organization

Each executable statement and statement number is placed in the program representation file (PRF) which, when scanned in the order it was formed, is a skeletal outline-representation of the source program. In addition to the fields that distinguish the items from each other, the PRF entries contain pointers to the appropriate expression representation file (ERF) entries (see below), to symbol table entries for variables, constants, and statement numbers, and to other PRF entries as appropriate to the individual type of entry. Detailed formats of the PRF and ERF are in Appendix A.

Each expression is placed in the expression representation file (ERF) in tabular form. The ERF form of the expression is a parenthesis-free notation in which, reading from left to right, each operand occurs in the order in which it occurred in the original expression; each operation follows its associated operand pair. The form is referred to as "right-hand Polish," or simply "Polish." See "CEKAI -- Expression Processor (EXPR)," in Section 3.

Each of the operator items includes information about its type and a code to indicate which operation is represented. Each variable or constant item includes information about its type and a symbol table pointer. This pointer is the means of reaching the associated symbol table entry and serves to associate the item with other items representing the same variable or constant while distinguishing it from other items.

The detailed description of Phase 1 is in Section 3.

PHASE 2

Phase 2 has five functions:

1. Make storage assignments in the OPM to all variables that are not formal arguments of a subprogram.
2. Detect and diagnose illegal flow in DO nests.
3. Indicate that the DO-loop index variable requires materialization (must be maintained in its storage cell) in a loop that contains an exit.
4. Detect and diagnose references to undefined statement numbers (labels).
5. Determine definition points (points at which a value may be changed) of COMMON variables and subprogram arguments.

COMMON variables are assigned storage in the order dictated by their appearance in the source program, in their appropriate COMMON blocks, and are given as much space as indicated by their individual DIMENSION/type combinations.

Non-COMMON variables that do not appear in EQUIVALENCE statements are assigned storage such that all scalars appear first, followed by all one- then two-dimensional arrays, etc. For any given dimensionality, variables of the same type appear together; those requiring less storage precede those requiring more. In this way, a maximum of address-constant sharing is obtained in the object program.

The relative relationships of storage assignments of variables appearing in EQUIVALENCE statements is determined, and these variables are assigned storage within the appropriate COMMON block, or at the end of the non-COMMON group, as required.

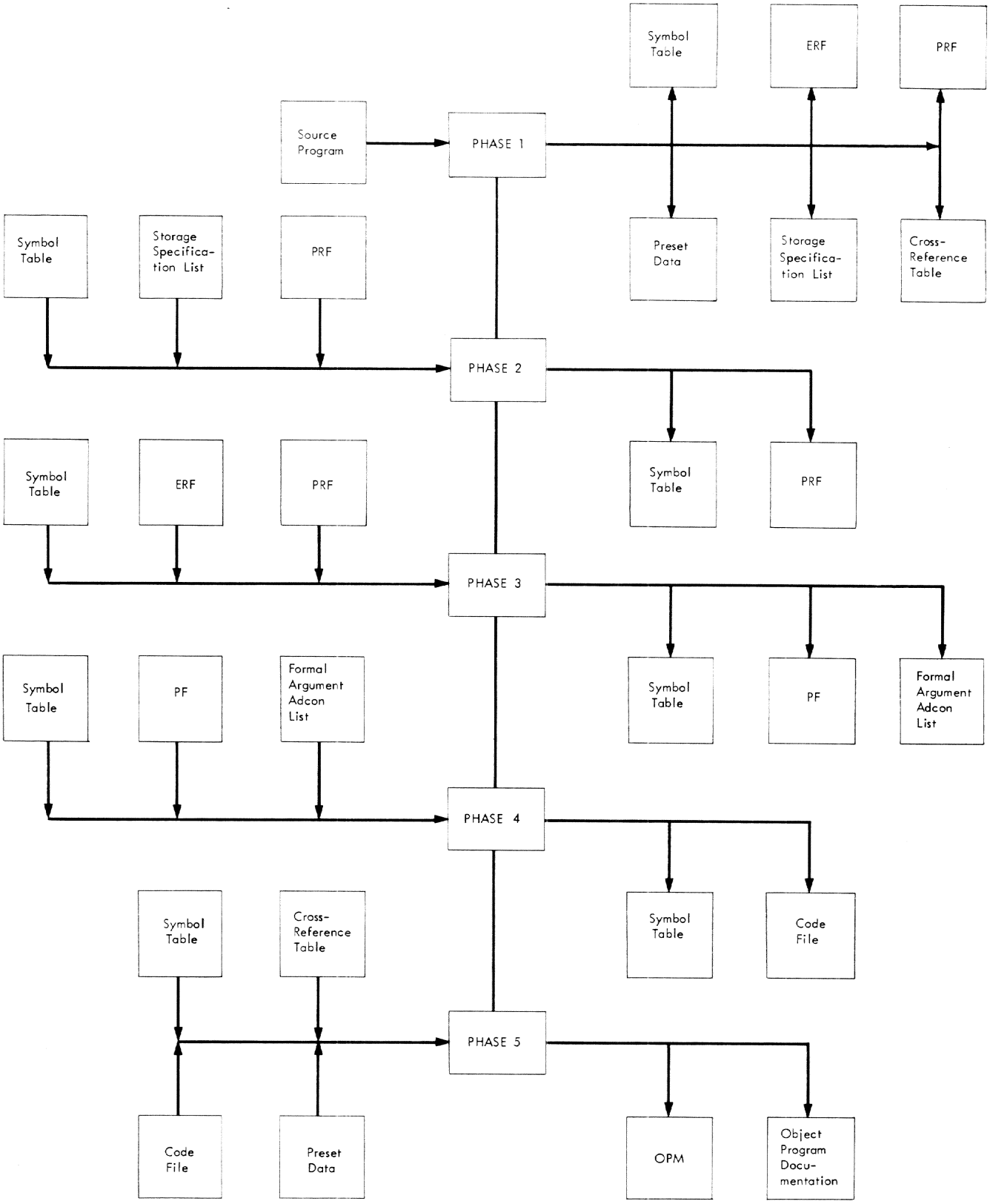


Figure 3. Compiler Information Flow

Variables that do not appear in COMMON statements but appear in EQUIVALENCE statements in conjunction with COMMON variables are flagged as appearing in COMMON.

After a storage assignment is made, its assignment (SLOC) within storage class (STCL) is recorded in the symbol table. Non-COMMON variables are assigned storage class 6, blank COMMON storage class 9, and labeled COMMON storage class 10 to as high as 127 in the order of first appearance of the corresponding labeled blocks in the source program.

In the OPM storage, classes 3 through 8 will be accumulated by Phase 5 and become the module's PSECT in the object program. These classes include alphameric constants, address constants, NAMELISTs and parameter lists, non-COMMON variables, global (unreleasable) temporary storage, and local temporary storage, in that order. The COMMON blocks (storage classes greater than 8) become individual control sections in the OPM where the block name becomes the control section name. Such control sections are combined with control sections of like name from other modules, before execution (during linkage editing or loading).

Information concerning the remaining functions of Phase 2 is in Section 4.

PHASE 3

Phase 3 performs the global optimizations to be done in the code generated by Phase 4 and establishes address coverage for all quantities referred to from storage.

Phase 3 determines which arithmetic expressions can be computed only once and then saved for later uses. It also determines the range of statements over which expressions are not redefined by the definition of one or more of their components. If the occurrence of an expression in that range is contained in one or more DO loops which are also entirely contained in that range, Phase 3 determines the outermost such loop outside which such an expression may be computed, and moves the expression to the front of that DO loop. Only the evaluation process is removed from the loop; any statement number and/or store process is retained in its original position. The moved expression is linked to a place reserved for that purpose in the program-representation-file entries corresponding to the beginnings of DO loops.

In the statements

```
1    A = B+C
2    Y = A+B
3    A = A*2
4    Z = A+B
5    X = B+C
```

the occurrences of expression B+C in statements 1 and 5 are determined to be common because neither B nor C has an intervening definition. The expression identification corresponding to the plus operator will be changed from "operator" to "common expression" (CSX). A CSX has the properties of the original operator (e.g., here the plus operator code is retained), with the additional property that it represents a "named" expression. The CSX item contains a field reserved for the expression name (this name is actually a monotonically increased number), that is identical only for identical expressions. In statements 2 and 4 above, the expression A+B is not a CSX because of the intervening definition of A in statement number 3. Both plus operators retain their "operator" identity; neither becomes a "named" expression.

Consider the statements

```
     DO 1 I = 1, 10
     A = B+C
     Y = E+F
1    F = A
```

Because there are definitions of neither B nor C within the DO loop, the expression B+C is given a "name", and the named expression is linked to the beginning of the DO statement, so that Phase 4 generates the expression once, outside the loop. The occurrence of the expression inside the loop is replaced by a "residue item" (see ERF description in Appendix A) that has the same "name" as the removed expression. Note that expression E+F is neither named nor removed because of the definition of F in statement 1.

Phase 3 creates two new operators, both arising only from subscripts. The first is called a base/index split operator or "? operator"; its right operand is a residue (computed outside a DO loop), and its left operand is an expression that is local to the DO loop. Phase 4 places one quantity in a base register and the other in an index register when generating a storage reference to the subscripted quantity.

The second operator is called the recursive operator or "! operator"; its right operand is the initial value of a subscript (induction variable dependent) constituent that is to be computed recursively over a DO loop, and its left operand is the "step

expression", a quantity to be added to the recursive expression after each pass through the loop. (The induction variable is the variable referenced in the DO statement of the loop. In the DO statement shown above, I is the induction variable.)

Phase 3 merges the ERF and PRF with some modification to form the PF (see Appendix A). This file is the primary output of Phase 3.

Detailed information about the functions of Phase 3 is in Section 5.

PHASE 4

Phase 4 performs the code generation function. Its input consists primarily of the PF and symbol table, and its output is the code file which represents, completed machine instructions and additional editing information.

Phase 4 performs a scan of the PF. Processing is triggered by the various PF items and by the expressions they may reference. A set of tables is maintained that reflects the contents of the various general and floating registers at any time. When the generation of an expression is required, the register tables are searched, and if any constituent operand of the expression is in a register, it is generally used from that register, rather than from storage. Partial results are stored in temporary storage only when a register is needed for some other purpose and there is no better choice of register than the one containing the partial result, or when the partial result is a CSX that has later uses and the operation about to be performed will change the value of the register containing the common expression.

Phase 4 is a collection of PF entry processing routines, arithmetic generators tailored to the various operators and expression types; and service routines to maintain register storage, partial result storage, and CSX storage, to select and assign registers, to determine when operands are no longer needed, to assign and release temporary storage, etc. The detailed description of Phase 4 (in Section 6) indicates the relationships among these routines and provides a much more comprehensive description of the operation of this phase.

PHASE 5

Phase 5 collects the information from the various compiler-generated storage classes and forms a code/numeric-constant-sharable CSECT, a PSECT, and as many COMMON CSECTs as there are declared COMMON blocks. This information, and information (obtained from the Symbol Table) making up the optional ISD, constitutes the object program module.

Optionally, Phase 5 also produces an assembler-like listing of the object program code obtained from the Code File, a storage map, and a cross reference listing indicating the various source-program identifiers and the lines in which they were referenced or defined. The user's selection of these options is passed from LPC to the Compiler Executive and thence through the INTERCOM table to Phase 5.

Section 7 contains the detailed description of Phase 5.

Note: Routine descriptions in Sections 2 through 7 occasionally refer to registers as "P1", "P2", "P3", etc. Such register notations are taken directly from the listing, where they appear in EQU instructions and other instructions.

SECTION 2: EXECUTIVE

INTRODUCTION

The compiler executive (Exec) contains all routines in the compiler that either provide an interface between the compiler and the environment in which it resides or provide a service needed by more than one compiler phase (Chart AA). Functions performed by the Exec routines fall logically into the following categories:

1. Service external interfaces.
2. Prepare source statements.
3. Control the compiler phases and interphase files.
4. Process compiler edit lines.
5. Provide compiler diagnostic information.
6. Provide miscellaneous services.

This discussion of the Exec is divided into seven sections: an initial section entitled "General Information," followed by sections dealing individually with the above six categories.

GENERAL INFORMATION

This section contains general information of value to understanding the computer executive. Topics discussed are:

1. Macro instruction usage.
2. Linkage conventions.
3. Register notation and conventions.
4. Storage map.
5. Brief routine description.
6. Use of the phase controller PSECT by other Exec routines.

MACRO INSTRUCTION USAGE

The Exec routines, like all compiler routines, make heavy use of macro instructions: both "user" macro instructions (such as CALL, SAVE, RETURN -- those described in Assembler User Macro Instructions) and "system" macro instructions (those used only by the exec). User macro instructions are not discussed here. The term macro instructions as used in this discussion means "system macro instruction."

Appendix C lists a brief summary of all exec macro instructions; the following paragraphs group them by function.

1. Macro instructions concerned with the compiler diagnostics features: CEKT0, CEKTG, CEKV3 and CEKV5.
2. Macro instructions written to provide module PSECTs and DSECTs: CEKU7, CEKT8, and CEKT9.
3. The CEKVU macro instruction contains all VAM macros.
4. All uses of the GETMAIN and FREEMAIN macro instructions are contained in the macro instructions CEKVC and CEKV9, respectively.
5. The macro instruction CEKVA issues the system macro instructions EBCDTIME and REDTIM.
6. The CEKU9 macro instruction simplifies the processing associated with output of a message describing a source statement error detected by any Exec modules.
7. The macro instructions CEKTX, CEKV7, and CEKV8 define all V-R con pairs and issue the CSECT and PSECT lines, for all Exec modules.

Use of all other Exec macro instructions is obvious upon inspection of Appendix C and the assembly listings.

LINKAGE CONVENTIONS

All linkages by the compiler are Type I. (See Appendix B, "TSS/360 Linkage Conventions"). The linking mechanism is either by means of the CALL, SAVE, and RETURN macro instructions or by the INVOKE, STORE, and RESUME macro instructions; there are no hand-coded linkages. All exec routines, linked to by other exec or compiler routines through a CALL macro instruction, set return codes in general register 15 before returning. These codes are:

Code	Description
0	Normal return.
4	If a phase suspects a system error, it returns to the phase controller with a code of 4. No phase currently issues this return code.
8	The "Compiler cannot continue -- Abort" code. Table overflow is the usual cause. The phase will return to the phase controller with a return code of 8, causing the phase controller to make a "FORTRAN cannot continue" return to LPC.
16	A compiler restart is to be executed (see comment below). Programs called by the phase controller are to return with this code if, upon calling an exec subroutine, a code of 16 was returned by the subroutine.

A return code of 12 is treated identically to a code of 8. Return code of 16 is expected only during Phase 1 processing; in all other places it is treated as a code of 8. A return code from a compiler module greater than 16 is never expected, is not tested for, and will produce unpredictable results.

REGISTER NOTATION AND CONVENTIONS

The TSS/360 register notation standards (see also Appendix B) describe a division of registers into parameter registers (P1 through P6), volatile registers (V1 and V2), nonvolatile registers (N1 through N5), and linkage registers (L1 through L3). This standard is followed in all Exec modules, with the minor exception that absolute register notation is used where it should be made clear that other registers may not be used. Examples are registers 0 and 1 in the ED instruction and registers loaded by the system macro instructions and macro processors.

Use of all registers is summarized in the prologue contained at the beginning of each assembly listing.

The CEKSZ macro instruction issues all EQUs for general and floating registers.

STORAGE MAP

Table 1 shows the approximate size of each control section in the Exec and the GETMAIN areas used for interphase files. The manner in which the compiler modules are link-edited will, of course, dictate the order in which modules are loaded and the storage required.

BRIEF ROUTINE DESCRIPTION

The routines in the Executive are described briefly below. The description includes the documentation module name (five characters, in parentheses, with the letters CEKT as the first four characters), preceded by the name generally used throughout the Executive documentation. The type of linkage to the routine is described, and a note is given describing conditions if the routine is an assembly module (is assembled separately from all other modules), as well as a documentation module. (A documentation module may or may not represent a separate assembly.)

1. Phase Controller -- PHC (CEKTA, documentation and assembly module).

   The Phase Controller is a Type I linkage subroutine and is the interface between the (LPC) and the five compiler phases. All LPC calls enter PHC, and the phases may be called only by PHC. PHC initializes the work area and communication module as required for each phase, furnishing addresses of tables, pointers in these tables, etc. PHC prepares all parameters for return from the LPC to FORTRAN calls.

   PHC does not call the LPC entries GETLINE or PUTDIAG, nor does it operate on the list data set in any way. These operations are all performed by the master input/output module (see below).

2. Get Next Source Statement -- GNSS (CEKTC, documentation and assembly module).

   This Type I linkage subroutine obtains complete source statements for Phase 1 of the compiler. The source statements are composed of lines furnished GNSS by the LPC GETLINE entry. Facility is included for conversational modification of statements already received. GNSS uses restricted linkages internally.

3. Process Terminal Modification -- MOD (CEKTD, documentation and assembly module).

   MOD is a Type I linkage subroutine whose purpose is to assist GNSS in the formation of source statements when conversational corrections have been made to the source statement. It accomplishes this by analyzing the relation between the line number of a line to be corrected (or inserted) and the line numbers of statements already received by the compiler.

Table 1. Executive Storage Map

Module	Code	PSECT
Phase Controller (CEKTA), (PSECT for CEKTA is the Work Area and Communication Module, CEKTB)	(16,000 bytes)	Save Area (76 bytes)
		Inter-Exec Communication and work area and Intercom (13,000 bytes)
		Symbol Table* (20 pages -- 81,920 bytes)
		Work Area A* (60 pages)
		Work Area B* (60 pages)
		Work Area C*,** (32 pages)
Get-Next-Source Statement (CEKTC)	(1600 bytes)	Save Area, misc. (128 bytes)
Process Terminal Modifications (CEKTD)	(2048 bytes)	Save Area, misc. (128 bytes)
Receive Diagnostic Message (CEKTE)	(300 bytes)	Save Area, misc. (128 bytes)
Constant Filers (CEKTF)	(4096 bytes)	Save Area, misc. (128 bytes)
Master Input/Output (CEKTH)	(4096 bytes)	(600 bytes)

* These areas are obtained using GETMAIN.
** See Table 4 for the allocation of Work Area C for the Output Module.

4. Receive Diagnostic Message -- RDM (CEKTE, documentation and assembly module).

   Any module in the compiler (including Exec modules) that adds a diagnostic message to the user's output does so through RDM. The message may go to the list data set, the conversational console, or both. The LPC entry PUTDIAG is used for console messages.

5. Constant Filers -- CONFIL (CEKTF, documentation and assembly module).

   Several of the compiler phases must add information concerning numeric, address, and label constants to the symbol table. The filing of these constants is performed for the phases by CONFIL, through a Type I linkage. CONFIL also includes an entry which creates numbers used to mark points in the code for the phases and then files these numbers as label constants.

6. Master Input/Output -- MIO (CEKTH, documentation and assembly module).

   All input/output operations are controlled by MIO. These operations include:

   a. Calling GETLINE for source lines

   b. Calling PUTDIAG for diagnostic message output to the conversational console

   c. Opening, closing, and adding source and diagnostic messages to the list data set

MIO contains six Type I linkage entries and uses restricted linkages internally.

7. Analyze Console Source Line -- ANALYZ (CEKTI, documentation module).

   This restricted linkage subroutine is invoked by GNSS to determine where the statement number and first text character are in a console line, and how many text characters are included in the line. ANALYZ is assembled into GNSS.

8. Inspect a Console Character -- INSCON (CEKTJ, documentation module).

   This restricted linkage subroutine is invoked by ANALYZ to determine if a console character is a tab, numeric, blank, or other character. INSCON is assembled into GNSS.

9. Move a Line to the List Data Set -- LDMOVE (CEKTK, documentation module).

   LDMOVE is a restricted linkage subroutine, invoked by MIO to move a line from a buffer to the list data set. LDMOVE counts lines in the current page and, when required, restores the page and adds a page heading. LDMOVE is assembled into MIO.

10. Build the List Data Set Buffer -- BUILD (CEKTL, documentation module).

    BUILD is a restricted linkage subroutine, invoked by MIO to move a line to either a list data set buffer or the list data set. The buffer will be emptied when its capacity is exceeded, or when information contained will not be replaced due to conversational corrections. BUILD is assembled into MIO.

11. Flush the List Data Set Buffer -- FLUSH (CEKTM, documentation module).

    This restricted linkage subroutine is invoked by MIO, to move all lines in a list data set buffer to the list data set. FLUSH is also invoked by GNSS, through the BFLUSH entry to MIO. FLUSH is assembled into MIO.

12. Compiler Dump -- COMDUMP (CEKTQ, documentation and assembly module).

    This Type I linkage module is called by the Phase Controller when a file is to be dumped in hexadecimal. Such dumps are produced only when the compiler is in the diagnostic mode.

13. Dump Line Preparation and Output -- LINDUMP (CEKTS, documentation and assembly module).

    LINDUMP is called in diagnostic-mode processing only, using a Type I linkage. LINDUMP prepares one line of information and adds it to the list data set.

USE OF THE PHASE CONTROLLER PSECT (EXCOM) BY OTHER EXEC ROUTINES

The first two pages of the Phase Controller PSECT contain information required by other routines in the Exec. A definition of this PSECT is supplied to all Exec routines by including a DSECT for the Phase Controller PSECT. Cover for this DSECT is obtained by loading the address of the Phase Controller PSECT from a word in intercom (Exec modules are always passed the location of intercom when called). The term 'excom' (Exec communication region) is used by Exec routines to refer to the Phase Controller's PSECT.

SERVICE EXTERNAL INTERFACE

The compiler's external interfaces are:

1. Entrances from Language Processor Control (LPC)
2. Calls on LPC to get a source line or produce a diagnostic line
3. Macro instructions to get and free main storage
4. Macro instructions to operate on a VAM data set
5. Macro instructions to obtain the time at which the compilation is beginning

Figure 4 shows the above interfaces. For each interface, the Exec routine involved is identified. Note that all calls on LPC are centralized in the Master Input/Output (MIO) routine as are all calls on VAM (except one, the call by LINDUMP, which is issued only if the compiler is in the diagnostic mode, as discussed in the section "Compiler Diagnostic Information").

Routines concerned with external interfaces are:

1. Phase Controller (PHC,CEKTA).
2. Master Input/Output (MIO,CEKTH).
3. Dump Line Preparation and Output (LINDUMP,CEKTS).

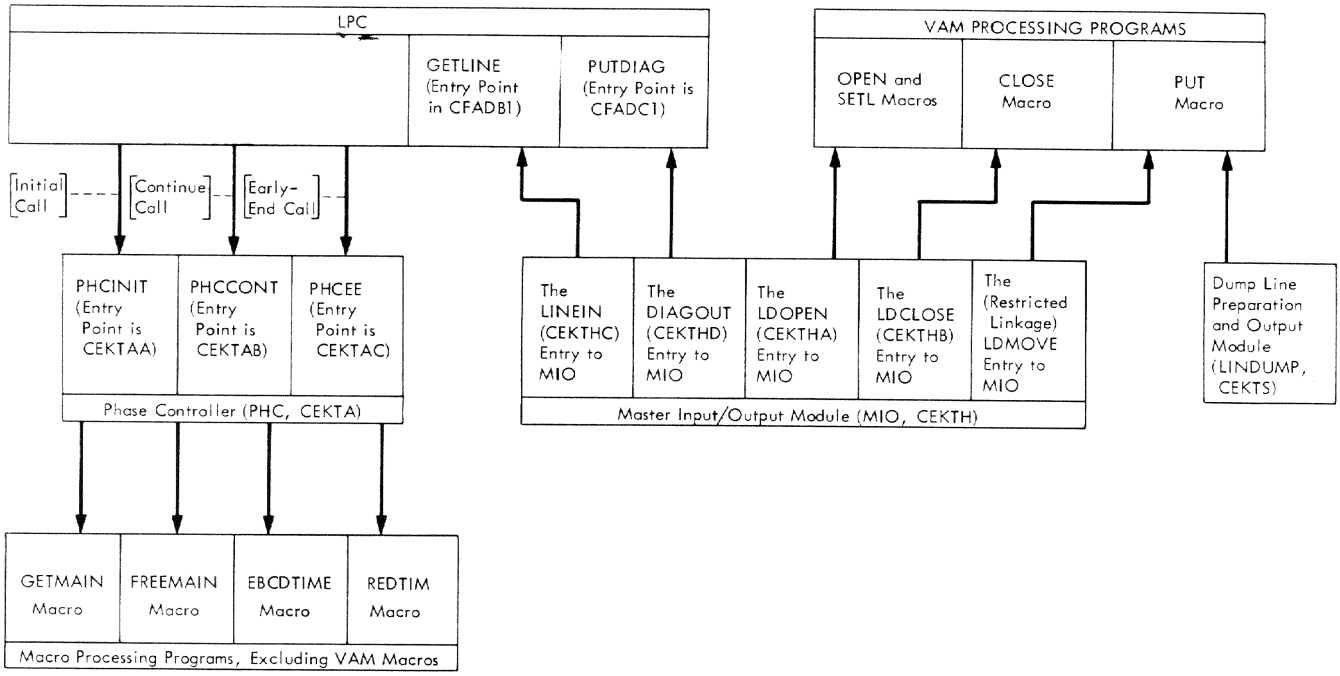


Figure 4. Compiler Interfaces

Details concerning activities of the Executive routines that use and prepare information passed across the interfaces are given under "Routine Descriptions," in this section.

SOURCE STATEMENT PREPARATION

The purpose of the routines described in this section is to prepare complete FORTRAN source statements for processing by Phase 1 of the compiler. This preparation is accomplished by obtaining lines through the services of the LPC entry GETLINE, combining these lines as appropriate (continuation lines may exist), and informing Phase 1 of the location of a complete source statement, and the statement label (if any). If the compiler is in conversational mode, the terminal user may request that changes be made to a line (or lines) previously sent to the compiler. In such an event, the Process Terminal Modifications routine (MOD) determines if the correction was such that the entire program must be recompiled, or if the preceding or current statement is to be ignored or modified and compilation continued. Two routines participate in the preparation of source statements: Get Next Source Statement (GNSS, CEKTC) and Process Terminal Modification Lines (MOD,CEKTD). These routines have no other functions.

Figure 5 illustrates the general relationship between the source-statement-preparation routines and other routines in the compiler.

PHASE AND INTERPHASE FILE CONTROLLER: THE COMPILER WORK AREAS AND INTERCOM

The Phase Controller (PHC,CEKTA) performs the functions of calling the five compiler phases. Associated with each call on a phase are a number of miscellaneous operations concerning the files used by the phases as their medium of information exchange; these operations are also performed by PHC. The phase control operation is a simple one and consists principally of calling each phase in its turn, checking the return code to see if the following

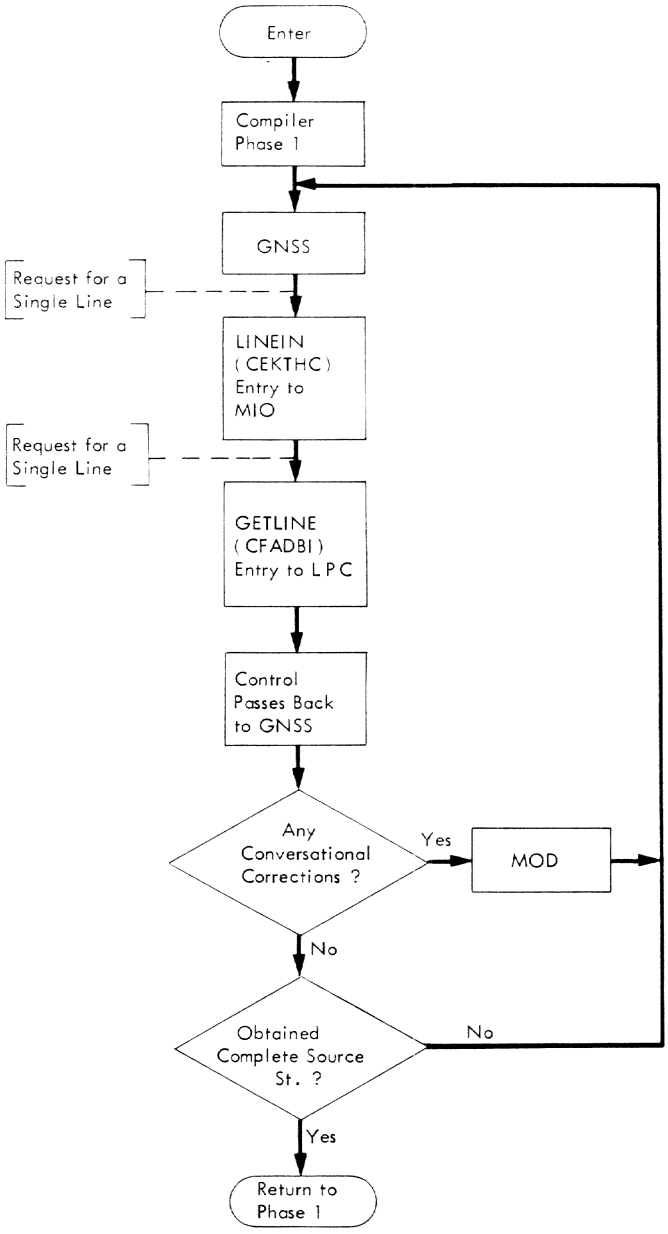


Figure 5. Source-Statement-Preparation Modules

phase should be called, and returning parameters to LPC following the calls on Phases 2 and 5.

Figure 6 summarizes the order and conditions of calls on the phases and shows the GETMAIN and FREEMAIN activities. In Figure 6 and in other figures below the abbreviations for interphase files are used. These abbreviations are:

CF	Code file
CRL	Cross reference list
EF(or ERF)	Expression (representation) file
ENL	External name list
ISD	Internal symbol dictionary
OPM	Output module (text)
PF	Program file
PMD	Program module dictionary
PRF	Program representation file
SPL	Storage specification list
Work areas	See below

Note that all file descriptions given below are for the purpose of summarizing the obtaining, use, and freeing of storage. For detailed descriptions of the contents of all files, see Appendix A.

The term "work area" is used to refer to an area in virtual storage that is logically reused; that is, one phase uses the area, and PHC then makes an area of the same size available to the next phase, etc. Tables in this work area are cleared out when they are no longer needed. The number of pages obtained for each work area is determined by a constant assembled into the PHC PSECT; this number was also given in the storage map description.

Tables 2, 3, and 4 give miscellaneous information concerning the three work areas. Figure 7 shows the symbol table storage layout.

Probably the most important interphase file in the compiler is the file referred to as intercom. A detailed description of the contents of intercom is given in Appendix A; general information on use of this area follows. The intercom area contains 512 bytes. All information required by the Executive and any phase, or to be passed between phases (excluding large lists, files, etc.), is passed by means of the intercom area. Intercom is not obtained by a GETMAIN, but is assembled into the phase controller PSECT. The sequence of intercom use is as follows:

1. The phase controller initializes intercom as required before each call on a phase and makes the location of intercom known to the phase via the calling sequence.

2. The phases move the 512 bytes to intercom from the phase controller to an area within the phase. The phases modify this area during their operation. If a phase calls an executive routine, it furnishes the executive routine with the location (in the phase) of intercom, so that intercom may be updated by the executive routine called.

3. Before returning to the phase controller, the phase moves the up-to-date intercom from the area within the phase back to its original area in the phase controller.

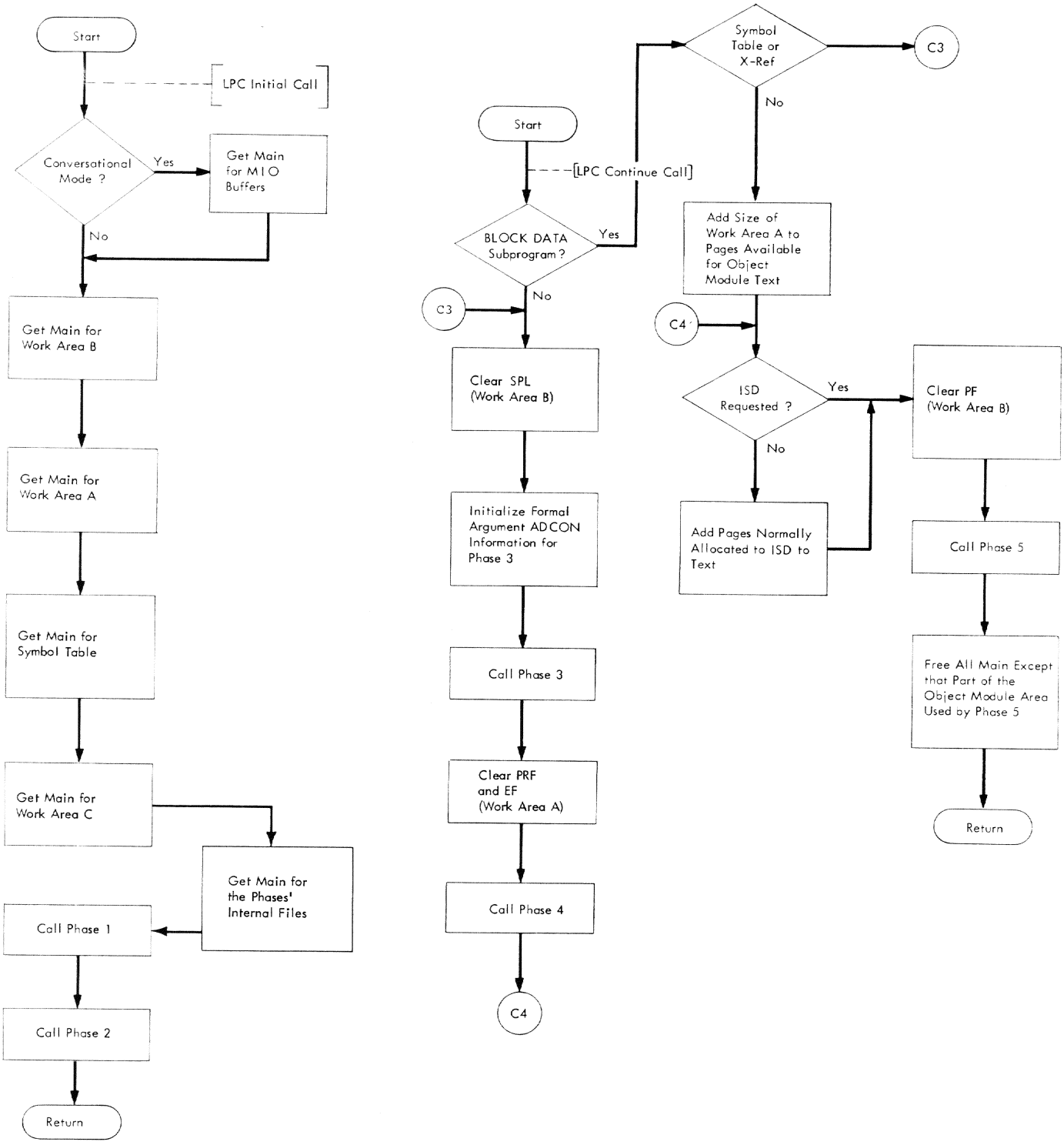


Figure 6. Summary of Phase and Interphase File Control Activities

Table 2. Work Area A Storage Layout

Name	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
TABLO[1,3]	EF[6] (Base and 2-byte top in Intercom)[4]	EF	EF	CF (Base, Top and Upper Limit in Intercom)	CF
TBAHI[2]	(Base and 2-byte top in Intercom)[4,5] PRF	(Not used by Phase 2) PRF	PRF		

NOTES:

1. TBALO is the CF Base and the initial CF Top. See Intercom TECFB, TECFT.
2. TBAHI is the CF Upper Limit. See Intercom TECFU.
3. Direction of increasing addresses is from the top to the bottom of the table.
4. The EF and PRF bases are identical, and are located approximately midway between TBALO and TBAHI. See Intercom TEEFB, TEPRFT, TEEFT, TEPRFT.
5. The address of the first word filed in the PRF is in TEWAAH in Intercom.
6. The EF is also referred to as the ERF.

Table 3. Work Area B Storage Layout

Name	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
TBBLO TBBM[1]	Storage Specification List (Base, Top and Limit in Intercom)	Storage Specification List		PF	External Name List (Base and Top in Intercom)
TBBM[2] TBBM[3]			PF (Base and 2-byte Top and Upper Limit 3,9 in Intercom)		PMD (Base and Top in Intercom)
TBBM[4] TBBM[5]					OPM (Base and Top in Intercom)
TBBM[6] TBBHI					ISD (Base and Top in Intercom)

NOTES:

1. TBBLO is the SPL and PMD Base. It is also the initial SPL and PMD Top. See Intercom TESPLB, TEPMDB, TESPLT.
2. TBBHI is the SPL and ISD Upper Limit. See Intercom TESPLU.
3. Computed by PHC.
4. If required, PHC will GETMAIN rather than use Work Area B.
5. Must Start on a Page Boundary.
6. Not needed if no ISD is requested by the problem programmer.
7. Direction of increasing addresses is from the top to the bottom of the table.
8. The allocation of Work Area B to the four Phase 5 Areas is:
   PMD -- 12 pages ENL -- 2 pages
   OPM -- 80 pages ISD -- 20 pages
9. The PF top is initially set to TBBLO. The PF upper limit is TBBHI.

Table 4. Work Area C Storage Layout

Name	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
TBCLO[1,5] TBCM	Preset Data (Base and Top in Intercom)	Preset Data (Not Used)	Preset Data (Not Used)	Preset Data (Not Used)	Preset Data
TBCM[2] TBCM[3]			Formal Argument Adcons (Base and Top in Intercom)[3]		
TBCM[4] TBCHI	(Base and Top in Intercom) Cross-Reference List[4,6]	(Not Used) CRL	(Not Used) CRL	(Not Used) CRL	CRL

NOTES:

1. TBCLO is the Preset Data Base and (initially) Top. See Intercom TEPSDB, TEPSDT.
2. TBCHI is the CRL Base and (initially) Top. See Intercom TECRLB, TECRLT.
3. The Formal Argument Adcon Base and Top are set by PHC prior to entering Phase 3. See Intercom TEFAAB, TEFAAT.
4. This area is not required if the cross-reference-list option is not chosen by the problem programmer.
5. Direction of increasing addresses is from the top to the bottom of the table.
6. Must start on a double-word boundary.

Symbolic Name	Description
TBSIF	Symbol Tables entries for the 49 Intrinsic Functions (assembled in)
TBSLO	Symbol Table Low -- first item filled will have the first word of its descriptive part put here
Descriptive Part Entries, next available word referenced with TEDES T[1] ↓ ID, FLAGS, ETC SLOC, STCL ↑ VALUE LINK, DPP Name Part entries, last used word referenced with TENAMT[2]	Direction of : Increasing Addresses ↓ / Descriptive Part Filing ↓ / Name Part Filing ↑
TBSHI	Symbol Table High -- first item filed will have the first word of its name part put here

1 The address of the first "ID, FLAGS, ETC" word filed is in TEDESB in Intercom.
2 The address of the first "LINK/DPP" word filed is in TENAMB in Intercom.

Figure 7. Symbol Table Storage Layout

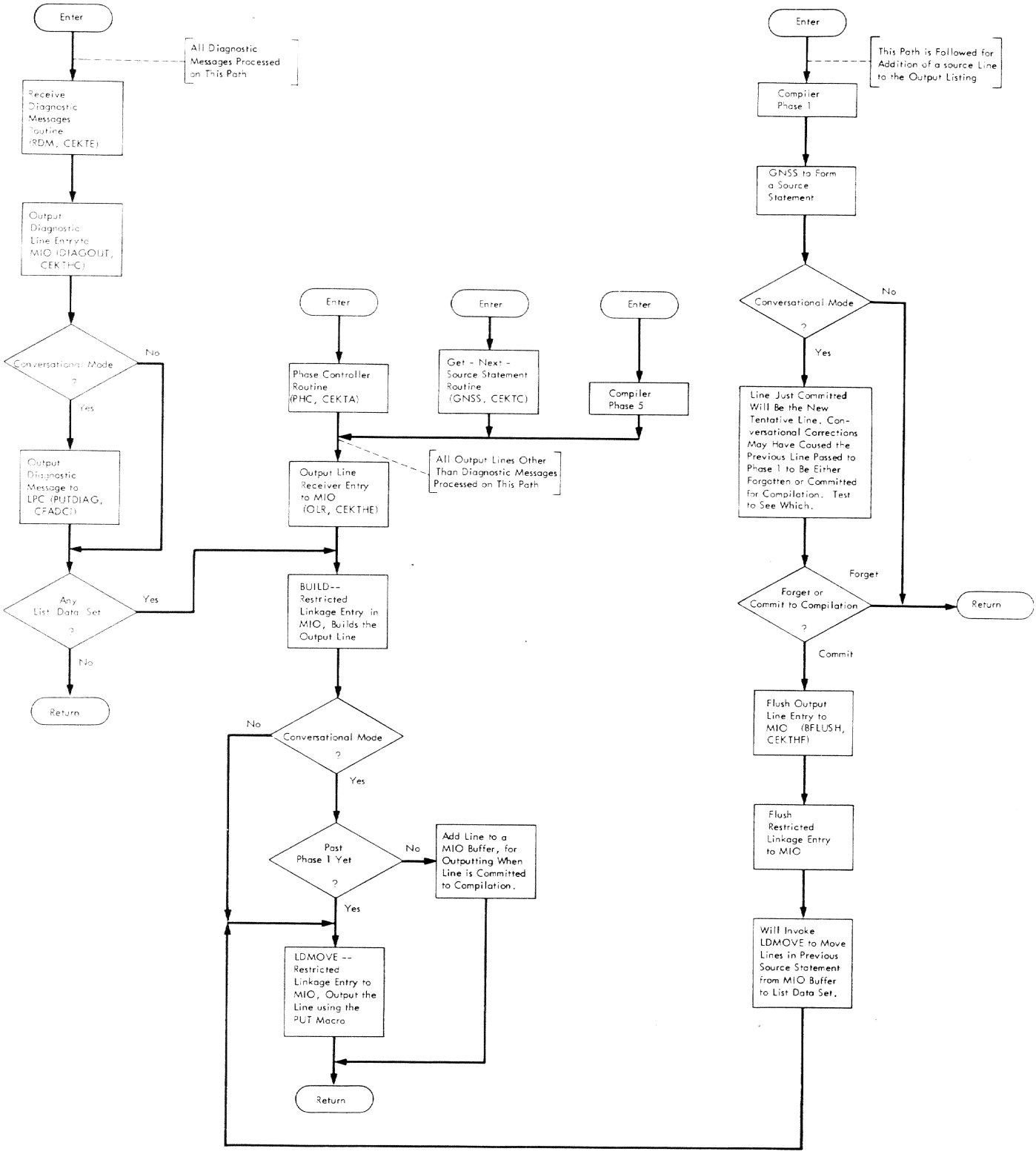


Figure 8. Process Compiler Edit Line Function

COMPILER EDIT LINES

The compiler produces two types of output: edited lines, to be transmitted to the terminal, list data set, or both; and the object module, constituting the compiled program ready for loading and execution. Output of the first type is prepared in the following places in the compiler: RDM (diagnostic messages from phases), PHC (heading lines and warning diagnostics associated with diagnostic mode processing), GNSS (SOURCE lines and associated diagnostic messages), CONFIL (file overflow diagnostics), and OLR (Phase 5 edit lines). Figure 8, "Process Compiler Edit Line Function" describes the path of compiler edit lines in more detail. Output of the second type is prepared completely by the compiler Phase 5 and is passed to LPC at the phase controller's return to the LPC continue call. Preparation of this output is described in Section 7 "Phase 5"; the manner of returning the information to LPC was described earlier in "Phase and Interphase File Controller."

The use of the MIO buffers deserves special mention. If the compilation is conversational, the phase controller (using GETMAIN) obtains two pages for the MIO buffers. These buffers will contain the source lines in the source statement currently being formed by GNSS and in the statement previously passed to Phase 1, but not yet committed to compilation. When a statement is committed to compilation, the associated source lines are added to the user's listing (if any). Following return from Phase 1 to the phase controller, these two pages are released.

COMPILER DIAGNOSTIC INFORMATION

The compiler contains built-in facilities for diagnosing compilation problems. These facilities consist principally of the ability to request hexadecimal dumps of interphase files and phase PSECTs following return from each phase. These dumps may not be directed to the terminal; they are issued to the list data set by use of a PUT macro. If the user has requested no edit options, there will be no list data set. In this case the diagnostic mode may not be entered. If the user attempts to do so the message "DIAGNOSTIC MODE NOT ALLOWED AS NO EDIT OPTIONS SELECTED" will be printed at the terminal.

It is also possible to alter the size of the main storage obtained prior to compilation, in order to measure the effect of large, unused pages of virtual storage, and to exercise the file-overflow tests of the compiler. These features are all contained within macros in the Exec routines and, thus, may be removed from the compiler by modification of these macro instructions. (See the General Information section for a description of all macros.) The diagnostic features do not affect the reenterable characteristic of the compiler. It is nearly impossible for a user to inadvertently request diagnostic output from the compiler, as information not normally available to the user must be known to produce such output. If the diagnostic mode is entered, the warning message "COMPILER IS IN THE DIAGNOSTIC MODE" will be produced at the terminal (if in conversation) and on the list data set. Figure 9 describes the diagnostic features.

The procedure for requesting diagnostic information is:

1. Load the Phase Controller (PHC), and a new PSECT for PHC, with the PCS statement:

```
LOAD      CEKTAR
```

2. Set the PHC PSECT byte TEDIAG to 'Y' (diagnostic mode allowed) with the PCS statement:

```
SET       CEKTAR.(X'11C5') = 'Y'
```

3. The first source line supplied to the compiler must be:

Col. 1 - 6	Col. 7	Col. 17 - end
(blank)	DIAGNOSTIC	Anything

4. The two lines following the DIAGNOSTIC line contain dump and other request information. The content of the second and third lines is described below.

Diagnostic Line 2:

Column	Name [1]	Description
1	TDPHAZ (1)	If 0, compilation will terminate prior to calling Phase 1.[2]
2	TDPHAZ (2)	If 0, compilation will terminate prior to calling Phase 2.[2]
3	TDPHAZ (3)	If 0, compilation will terminate prior to calling Phase 3.[2]
4	TDPHAZ (4)	If 0, compilation will terminate prior to calling Phase 4.[2]
5	TDPHAZ (5)	If 0, compilation will terminate prior to calling Phase 5.[2]
10	TDLOG	If Y, a message will be written when each phase is called, and when return is made from each phase.
31-40	TDBUG1	Requests file edits upon return from a Phase, as follows (the numbers 1 through 10 correspond to columns 31-40 for Phase 1, 41-50 for Phase 2, etc). No. — Description 1 — If Y, edit Intercom; edit the EF and PRF, after Phases 1 through 3; the CF after Phase 4 and 5. 2 — If Y, edit the Symbol Table. 3 — If Y, edit Storage Class Table. 4 — If Y, edit the phase's PSECT and the contents of the internal file used by the phase (Phase 5 does not use this file). If blank or 0 produce no PSECT edit. If X, edit the (3 pages) PHC PSECT. 5 — Edit the SPL and PF in Phases 1 through 4, the PMD, OPM, ENL and ISD if in Phase 5. 6 — Edit the Preset Data, Formal Argument Adcons, and the Cross Reference List. 7 — If Y, do not give file edits for RC = 0 return from each phase (will only give edits if RC≠0 or an unexpected interruption occurs). 8-10 Not used.
41-50		Same as 31-40, but inspected after return from Phase 2.
51-60		Same as 31-40, but inspected after return from Phase 3.
61-70		Same as 31-40, but inspected after return from Phase 4.
71-80		Same as 31-40, but inspected after return from Phase 5.

1. Label of field in PHC PSECT in which the value punch in the corresponding columns is stored.
2. If the column is blank or any character other than 0 (zero), compilation will not be terminated.

Diagnostic Line 3: Allows the user to alter the number of lines to be obtained by PHC in its GETMAINs. The relation between columns in line 3 and the files for which main storage is obtained is given below. If the four columns associated with a file are blank, the number of pages obtained will be the number assembled into the PHC PSECT.

Column	PHC PSECT Name	File
15-18	TDAPAG	Pages in Work Area A
19-22	TDBPAG	Pages in Work Area B
23-26	TDCPAG	Pages in Work Area C
27-30	TDPPAG	Pages in PMD
31-34	TDOPAG	Pages in OPM
35-38	TDEPAG	Pages in ENL
39-42	TDIPAG	Pages in ISD
43-46	TDMIOP	Pages in M10 Buffers
47-50	TDSYMP	Pages in Symbol Table

Note that, following the processing of the three diagnostic information lines, the source lines are read as in a normal compilation.

Figure 10 summarizes the determination of diagnostic mode and the initialization performed if in this mode. Figure 11 summarizes the processing performed at each phase call. Figure 12 shows compiler action if in the diagnostic mode and an unexpected interruption occurs.

MISCELLANEOUS

The miscellaneous modules are those used by any routine in the compiler that causes a diagnostic message to be added to the user's terminal and/or listing output, and files information concerning a numeric, label, or address constant in the symbol table.

The Receive Diagnostic Message (RDM, CEKTE) routine is passed information describing a diagnostic message to be produced. RDM will send this message to the terminal if the user is in conversational mode and will add the message to the user's listing if he has requested any list output. Figure 8 shows the general relationship between RDM and other modules in the compiler.

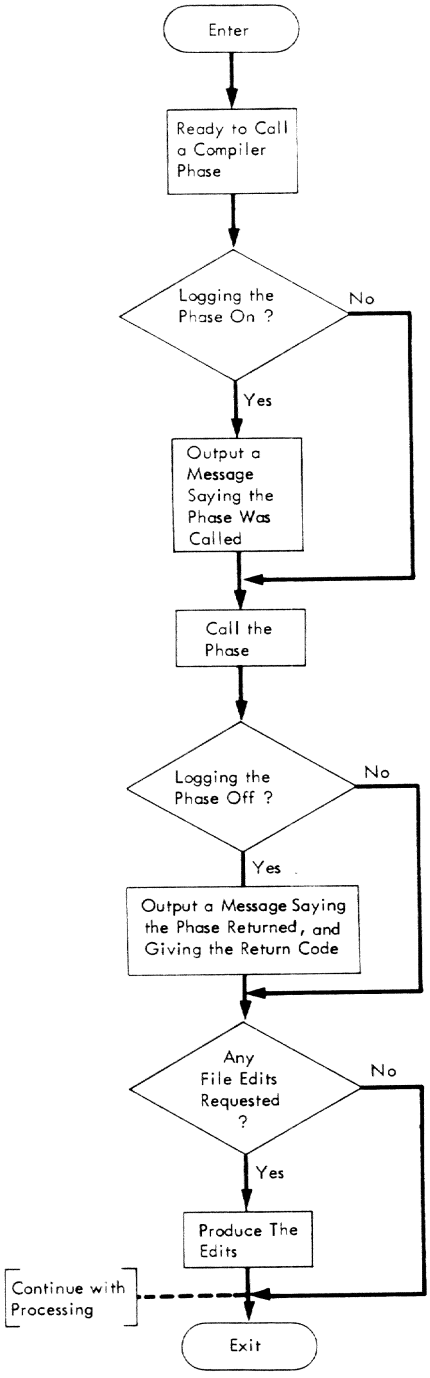


Figure 9. Compiler Diagnostic Features

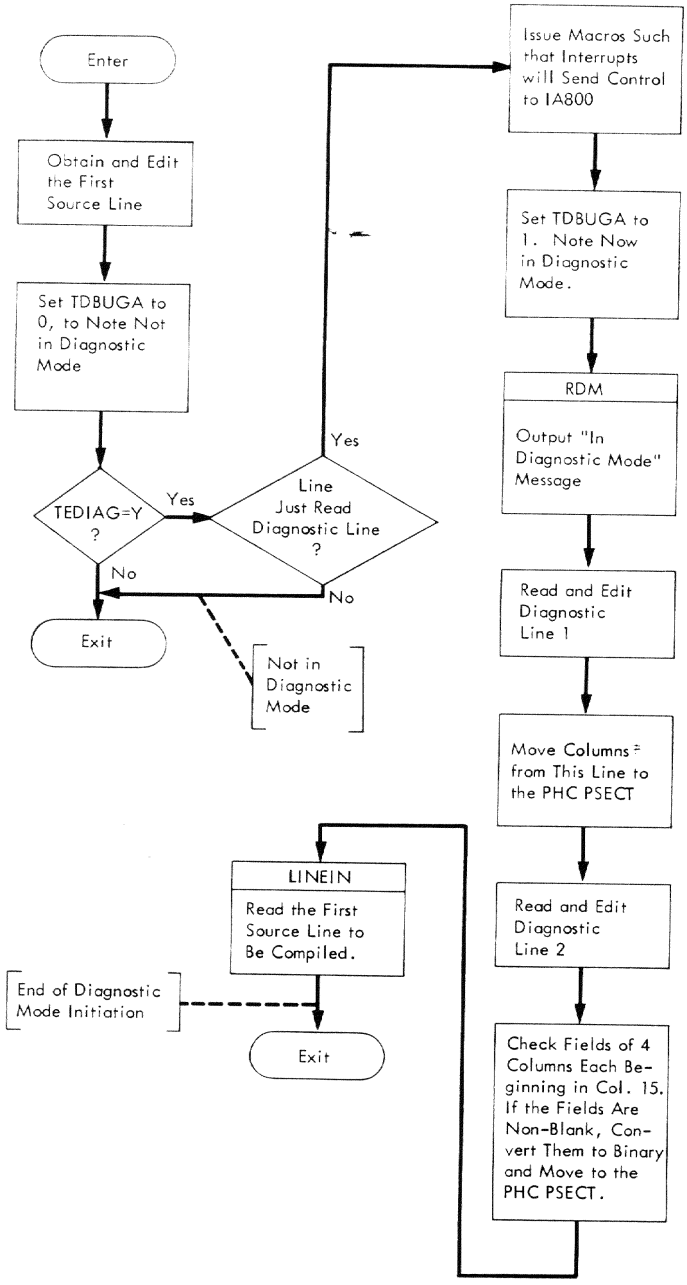


Figure 10. Testing for Diagnostic Input and Processing Diagnostic Information Lines

The Constant Filers (CONFIL, CEKTF) routine is called when information concerning a constant is to be filed in the symbol table for use by a later compiler phase.

ROUTINE DESCRIPTIONS

Exec routines bear mnemonic titles as well as coded labels. The 5-character coded labels begin with the letters CEKT; the fifth letter identifies a specific module. Various entry points to a routine are identified by a sixth letter added to the coded label; for example, the coded label for the Master Input/Output routine is CEKTH, and there are entry points CEKTHA, CEKTHB, etc.

There are no hardware configuration requirements for any of the Exec routines. All these routines are reenterable, nonresident, nonprivileged, and closed.

CEKTA -- Phase Controller (PHC)

The Phase Controller is the interface between the outside world and the compiler phases. It receives the LPC to FORTRAN initial, continue, and early-end calls, prepares for the compiling run, calls the phases as subroutines, and returns to LPC when compilation is terminated (successfully or unsuccessfully). See Chart AB.

ENTRIES: The Phase Controller has three external entry points:

LPC to FORTRAN Initial (ENTRY name is CEKTAA)

Register 1 contains the address of the parameter list.

Register 13 contains the address of the LPC save area to be used by FORTRAN.

Register 14 contains the return address.

Register 15 contains the V-type Adcon for the FORTRAN initial entry initialization routine (i.e., the entry point address).

```
[Symbol] CALL FORTRAN initial entry-symbol
                    (15)

                    [,(module name - addr,
                    batch/conversational
                    indicator - addr, F option
                    table - addr,
                    list data set
                    DCB - addr)]
```

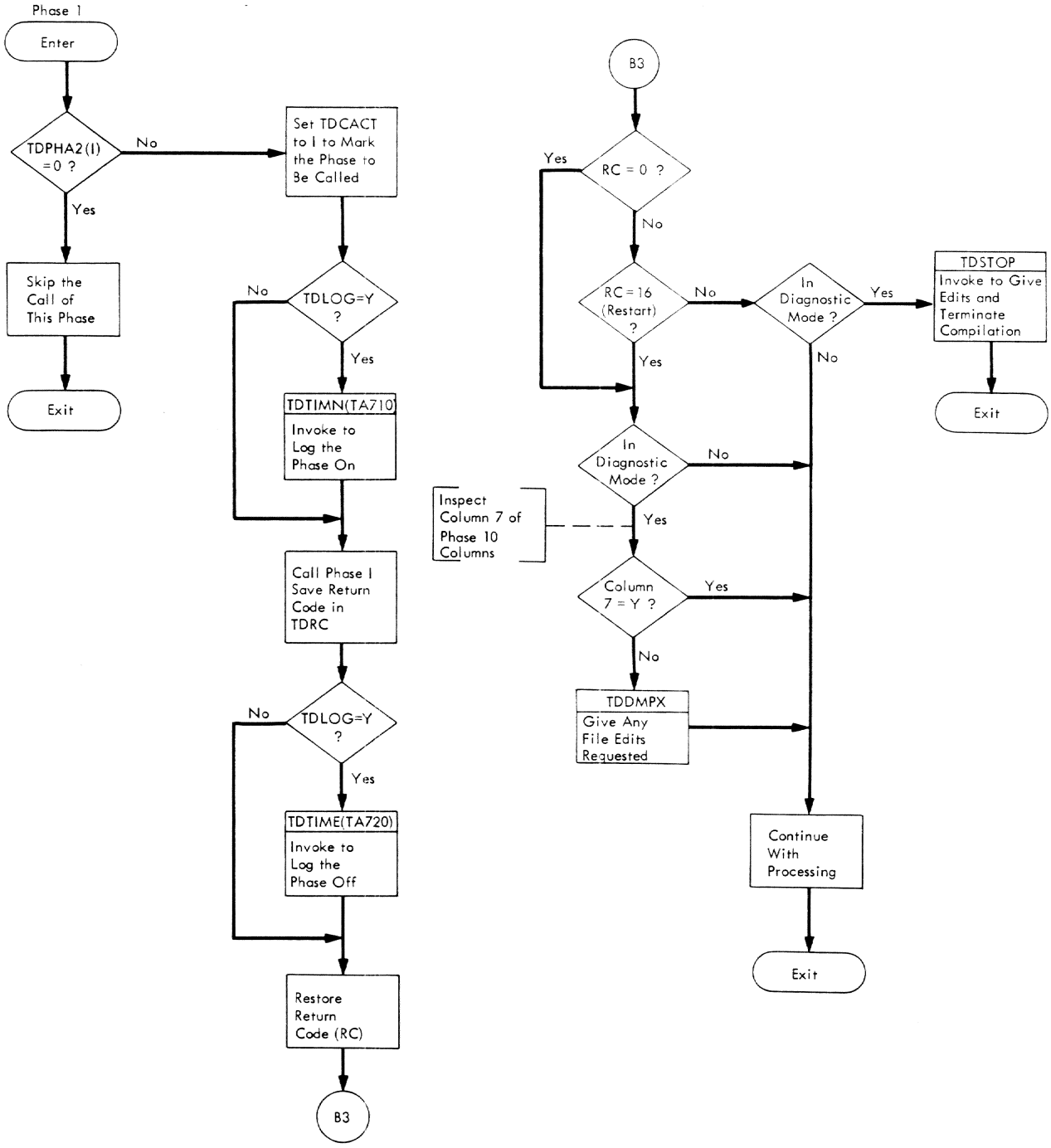


Figure 11. Processing Diagnostic Information Following Return From Each Phase

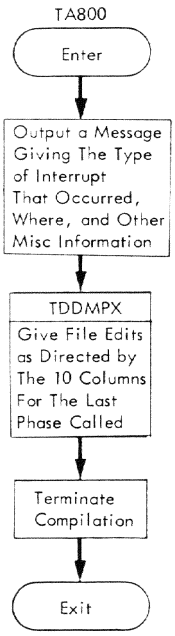


Figure 12. Processing of Unexpected Interruptions During Compilation

FORTRAN Initial Entry
Entry point name for FORTRAN initial entry.

List Data Set Name
Specifies the address of the module name.

Batch/Conversational Indicator
Specifies the address of a 1-byte field which contains 00000000 (for batch) or 00000001 (for conversational).

F Option Table
Specifies the address of an 8-byte option table, where each byte may be Y, N, or other. The default chosen by FORTRAN is shown in parentheses:

byte 1	ISD option (produce) (N)
byte 2	source listing option (Y)
byte 3	object listing option (N)
byte 4	cross reference listing option (N)
byte 5	symbol table edit option (N)
byte 6	storage map option (N)
byte 7	BCD option (N)
byte 8	public CSECT attribute (N)
byte 9	List Data Set option (N)

List Data Set DCB
Specifies the address of the data control block for the list data set.

LPC to FORTRAN Continue
(ENTRY name is CEKTAB)

```
[Symbol] CALL FORTRAN continue entry -
         symbol
                  (15)

              [, (list data set exists
              indicator - addr,
              length of PMD - addr, PMD -
              addr, length of TXT - addr,
              TXT - addr, length of ISD -
              addr, ISD - addr, external
              name list - addr)]
```

FORTRAN Continue Entry
Entry point name for FORTRAN continue entry.

List Data Set Exists Indicator
Specifies the address of a 1-byte field which will contain 00000000 (for no list exists) or 00000001 (for list exists) when FORTRAN returns to LPC normally.

Length of PMD
Specifies the address of a pointer to a 1-word field that contains a binary number when FORTRAN returns to LPC normally. This will be a count of the number of bytes in the PMD which FORTRAN is giving to LPC.

PMD
Specifies the address of a pointer to the area which will contain the PMD when FORTRAN returns to LPC normally.

Length of TXT
Specifies the address of a pointer to a 1-word field that will contain a binary number when FORTRAN returns to LPC normally. This will be a count of the number of bytes in the TXT which FORTRAN is passing to LPC.

TXT
Specifies the address of a pointer to the area which will contain the TXT when FORTRAN returns to LPC normally.

Length of ISD
Specifies the address of a pointer to a 1-word field that will contain a binary number when FORTRAN returns to LPC normally. This will be a count of the number of bytes in the TXT which FORTRAN is passing to LPC.

ISD
Specifies the address of a pointer to the area which will contain the ISD when FORTRAN returns to LPC normally.

External Name List
Specifies the address of a pointer to the area which contains the external name list when FORTRAN returns to LPC normally. (Each external name in the list is an 8-byte field.)

LPC to FORTRAN Early End Call
(ENTRY name is CEKTAC)

Register 1 contains the address of the parameter list, if any.

Register 13 contains the address of the LPC save area to be used by FORTRAN.

Register 14 contains the return address.

Register 15 contains the V-type Adcon for FORTRAN early end (i.e., the entry point address).

```
[Symbol] CALL FORTRAN early-end
          entry-symbol
                    (15)

              [,(list exists indicator -
          addr)]
```

FORTRAN Early-End Entry
Entry point name for FORTRAN early-end entry.

List Exists Indicator
Specifies the address of a 1-byte field which will contain 00000000 (for no list exists) or 00000001 (for list exists) when FORTRAN returns to LPC normally.

ROUTINES CALLED: The phase controller calls the five compiler phases, MIO, and CONFIL.

The calling sequence from the phase controller to these programs is a standard call with the address within PHC of the intercom area in the parameter list.

EXITS: The phase controller has three exits which correspond to the three entries.

Return Linkage to LPC from FORTRAN Initial

```
[Symbol]    RETURN
```

Upon return from FORTRAN initial, register 15 contains a code which LPC interprets as follows:

Code	Type of Return
0	Normal. LPC will query user before continuing if in conversational mode. LPC will call FORTRAN continue if in nonconversational mode.
4	FORTRAN cannot continue. LPC will query user, if in conversational mode, but will not allow him to continue unless he first modifies the source data set. LPC will call FORTRAN's early-end routine in both conversational and nonconversational modes.
8	Abnormal end LPC will not query user and will not call FORTRAN again. FORTRAN never issues this return.

Return Linkage to LPC from FORTRAN Continue

```
[Symbol]    RETURN
```

Upon this return, register 15 contains a code which LPC interprets as follows:

Code	Type of Return	Parameter Information Filled In
0	No errors	List exists indicator Length of PMD PMD-length of TXT TXT-length of ISD ISD-external name list
4	Minor errors (chance of a correct source program still quite high)	Same as for code = 0
8	Major errors (all or parts of source statements were omitted)	Same as for code = 0
12	No object module (probably table overflow within the compiler)	List exists indicator
16	Any highly abnormal condition -- partial object module may have been produced. FORTRAN never issues this return.	Indeterminate

Return Linkage to LPC from FORTRAN Early-End

```
[Symbol]    RETURN
```

Upon return from FORTRAN's early-end routine, register 15 contains a code which is interpreted by LPC as follows:

Code	Type of Return	Parameter Information Filled In
0	Normal	List exists indicator
4	A normal condition	Indeterminate

OPERATION: At the initial call, the Phase Controller fetches the compiler parameters from the LPC parameter list, obtains storage for the work areas using GETMAIN, and initializes the excom and intercom regions of the Phase Controller's PSECT. (The information furnished by LPC is described above under "entries." Items initialized in excom and intercom are described in the description of these regions.) The items initialized include compiler options; module, main entry point, and deck identification names; list data set items (this and all interfacing between Exec and Data Management is performed by the Master Input/Output module MIO); and various flags, switches, and pointers.

Next, initialization for other executive modules and Phase 1 is done. This is also the point at which control may be returned by Exec if conversational corrections to the source program were extensive enough to require a restart of compilation. After initialization, Phase 1 is called.

Following appropriate initialization, Phase 2 is called, followed by the return to LPC from FORTRAN Initial. LPC now enters the Phase Controller at the FORTRAN-continue entry if the user continues. If the program is a BLOCK DATA program, Phase 5 is entered. Otherwise, Phases 3 and 4 are successively initialized and entered.

Phase 5 is then called, produces its output (using MIO's OLR subroutine for list data set lines), and returns to the Phase Controller.

Following terminal processing, such as data set closing, preparing return parameters for LPC, and freeing main storage, the compilation is ended by a return to LPC.

Return parameters to LPC were described under "Exits." These parameters are prepared prior to returning to the initial call and the continue call. At return to the initial call, the parameter may specify:

1. Normal return. This return will be made even if serious program errors occurred; terminal users will, of course, have been advised of any such errors.

2. FORTRAN wants to abort the compilation due to table overflow or some other condition that makes continuing the compilation inadvisable without modifying the source program.

3. An abnormal condition -- FORTRAN may not be called again.

At return to the continue call, the return parameter may specify:

1. No errors in the source program.

2. Minor errors.

3. Major errors (source lines probably truncated or omitted).

4. Table overflow.

5. An abnormal condition.

The early-end entry to the Phase Controller may be reached from LPC if the compilation cannot continue or if the user does not want to continue. Reasons for inability of the compiler to continue include:

1. Source errors so serious that following phases cannot reasonably operate.

2. Storage overflow in a compiler table.

3. An abnormal condition.

In all these cases, appropriate error messages are given. Terminal processing, such as closing data sets and freeing main storage, is then performed and return is made to LPC.

CEKTC -- Get Next Source Statement (GNSS)

GNSS obtains a source statement from the input data set, using the services of LPC, and presents it to Phase 1 for processing. Modifications to lines already received are taken into account in determining what source statement is fetched. See Chart AC.

RESTRICTIONS: Several assumptions underlie the processing done by GNSS: A line in card form is assumed to be in the traditional FORTRAN format; a C in column 1 means a comment; a nonblank or nonzero character in column 6 means a continuation; the statement number is in columns 1-5; and the body of the line is in columns 7-72. Columns 73-80 may contain card identification, etc., and will be contained in the source deck edit on the list data set.

The required format for keyboard input is described in the documentation of the ANALYZ subroutine (module CEKTI), given later in this chapter.

An END statement is a line whose body consists only of the letters E, N, and D. Embedded blanks are allowed. A line meeting this description that is in fact continued (END FILE, for example, with FILE in a second line) will be treated as an END statement, and further FORTRAN statements will be ignored.

ENTRIES: The only entry point (CEKTCA) to GNSS is standard subroutine call from Phase 1. Phase 1 obtains the required V-con/R-con pair from intercom. The parameter list furnished GNSS by Phase 1 contains one address -- the location within Phase 1 where the phase moved intercom when called by Phase Controller. This address is required by GNSS, as GNSS will change items in intercom.

ROUTINES CALLED: GNSS calls the MIO subroutine LINEIN for source lines. LINEIN places all information required by GNSS in excom. The GETLINE, CALL, and RETURN statements are described in the MIO documentation.

GNSS calls the executive subroutine MOD when GETLINE specifies the terminal user has requested that lines be altered. The MOD R-con and V-con are in excom. The linkage to MOD is standard; the parameter list is as follows:

```
LIST  DC  A(PINCOM)   Location of intercom
                      in Phase 1.
```

MOD places a number in register 15 designating the action to be taken by GNSS:

Code	Action
0	Obtain a new statement.
4, 8, 12	Not expected -- GNSS will return to caller if these codes are present.
16	Restart required. Return to caller with a return code of 16.
20	The current statement must be reformed completely.
24	MOD got the first line of the new statement and left information concerning this line in excom.
28	MOD met end-of-data-set in reading a line.

Using a standard call, GNSS calls the executive subroutine RDM when an error message is to be issued. The RDM V-con/R-con pair is in intercom. The linkage is standard with the following parameter list:

```
LIST  DC  A(PINCOM)   Location of intercom
                      in Phase 1.
      .               Specify the message
      .                (see the RDM docu-
      .                mentation).
```

GNSS calls the OLR entry to subroutine MIO when a line is to be added to the source listing. The MIO V-con/R-con pair is in excom. The linkage to MIO is standard. The parameter list is as follows:

```
LIST  DC  A(PINCOM)   Location of intercom
                       in Phase 1
      DC  A(TEXT)     Location of output
                       line
      DC  A(LENGTH)   Line length
      DC  A(LINENO)   Line number, PL4
                       format
```

GNSS calls the MIO subroutine BFLUSH, to flush a buffer by adding its contents to the list data set. The linkage is standard. The parameter list contains the address of intercom in Phase 1 and the flushing parameter.

EXITS: The only exit from GNSS is a standard return to Phase 1, with the return code in register 15 set as described in the executive module.

OPERATION: The primary responsibility of GNSS is to set the necessary information for Phase 1 to process a statement. This information consists of the line number and statement number fields in the intercom area and the text character string in the area indicated by the intercom text pointer.

Certain internal flags and switches control the flow of GNSS:

TDTERM - This excom flag is raised (set to 1) before the first line of a statement is obtained and lowered (set to 0) until the terminal line of a statement is detected.

TDFORM - This excom switch indicates the form (C for card, K for keyboard) of the preceding line of a statement. It is set by the first line of a statement and reset when a statement started in keyboard form switches to card form.

TDOVER - This excom flag is raised when a single statement runs over 1320 characters. Subsequent lines of such a statement are not passed to Phase 1.

TEMEC - This intercom item, maximum error code, will be set to 8 if input lines are ignored due to an error detected by GNSS.

Using the LINEIN entry of MIO, GNSS calls the LPC subroutine GETLINE repeatedly for source lines until a complete statement has been assembled. In addition to assembling a source statement, GNSS sets the intercom items: the line number of the first line (TESLNO) and the statement number (TESTNO). GNSS also detects END statements (and sets the intercom item TEEND to mark this) and sets the excom indicators

TDU, TDPU, TDAPU, TDPUF, and TDAPUF for use by the process terminal modifications subroutine (MOD).

Card and keyboard lines are processed differently, due to their different continuation conventions and formats. The processing of the first line of a statement is also different from the processing of a continuation line. For the latter, the initial character (TDLE) and the length (TDNUMC) of the body of the line text must be found. For an initial line, this is done only after the line number and statement number have been placed in intercom. As the input lines are received from GETLINE, they are added to the internal files area TCTEX1. When a complete source statement has been formed in this area, the end-of-statement character X'FF' is added to the statement in this area, and the intercom item TEVSTB is set to the address of TCTEX1 for the use of Phase 1.

Calls on GETLINE in conversational mode may result in the terminal user's requesting that one or more lines be altered. If so, subroutine MOD is called to determine appropriate action. MOD may raise the FORGET flag to inform Phase 1 that the statement currently held in a tentative status by Phase 1 should be removed from all tables, as it will be replaced. In this case, GNSS will call GETLINE again to obtain a source statement for Phase 1, etc. If the tentative statement is to be accepted, MOD will direct GNSS to ask for new lines without having raised the FORGET flag. MOD may also direct GNSS to return to the Phase Controller requesting a restart. This occurs when the terminal user wants to alter a line that Phase 1 cannot "forget" -- a line permanently added to the Phase 1 tables. In such a case, the next call on GETLINE will again request the first source statement of the program.

When GNSS is initially entered in conversational compilation, the current values of TDBOLD and TDBNEW are interchanged. After the exchange, TDBOLD contains the address of the buffer filled with source lines, and corresponding diagnostic messages, for the statement obtained on the previous GNSS call. TDBNEW is the buffer address for the buffer to be filled on the current call.

At exit from GNSS in conversational operation, the FORGET flag is checked. If its value is zero, the statement is not to be "forgotten" so the statement in the buffer whose index is TDBOLD is to be added to the list data set. If the FORGET flag is 1, the line is to be forgotten and is not added to the list data set.

If GNSS encounters an end-of-data-set return code from GETLINE, the user omitted the END card from his source code. GNSS creates an END statement and returns to Phase 1 normally.

As an example of GNSS operation, consider the case where input is card only. A card is obtained, via LINEIN. If the line is a comment line, it is added to a list data set buffer via OLR; LINEIN is then called for another line. If the card is not a comment line, the status of the TDTERM flag is inspected. This flag is set to one upon entry to GNSS so that the line is written via OLR. Inspection for a continuation line is made. If the card is not a continuation card, several operations are performed, then inspection for an END card is made. If the card is an END card, two flags are raised. The text is then moved to the Phase 1 buffer. The TDTERM flag is then tested again. If down, LINEIN is called for a new line, and the chart is reentered. If TDTERM is up, the EOS (end-of-source-statement) sequence is entered, at which point the end-of-statement character is added and return made to Phase 1.

Certain tests made by the code are not shown in the flowchart, due to their repetitive nature:

1. Before OLR is called to add a line to a list data set buffer, the TESLO flag is checked. If TESLO is not equal to Y, no source listing was requested and the call is not made.

2. All executive subroutines called by GNSS return with a return code of 0, 4, 8, or 16, as described earlier in the General Information section. GNSS tests this code, and if it is nonzero, return is made to Phase 1 at once, with the return code unchanged.

Continuation inconsistencies (a continuation card line received as the initial line of a statement, or a noncontinuation card line received after a keyboard line indicating continuation) produce diagnostic messages and cause the line in question to be ignored. If a statement contains too many characters, a diagnostic message is produced and trailing lines of the statement are ignored.

CEKTD -- Process Terminal Modifications (MOD)

MOD is called by GNSS when GNSS is informed by LPC that a modified line has been entered from the terminal.

MOD will determine the effect of this request upon the obtaining of a source statement by GNSS. MOD may:

1. Direct GNSS to replace part or all of the statement currently being formed for Phase 1.

2. Raise the FORGET flag to inform Phase 1 that the statement currently held by it in tentative status should be removed from the Phase 1 tables (in this case, GNSS will obtain a replacement).

3. Direct Phase 1 to return to the Phase Controller requesting a restart of the entire compilation.

See Chart AD.

ENTRIES: MOD has a single entry point (CEKTDA) and is currently reached only by GNSS, via a standard call. The parameter list contains only the address of intercom within the phase calling GNSS.

ROUTINES CALLED: MOD calls the LINEIN entry of MIO when an altered line must be inspected to see if it is a continuation line.

EXITS: MOD returns to the calling program using a standard RETURN. A return code is set in register 15 by MOD, as follows:

Code	Description
0	A new statement is to be obtained from LINEIN using the current value at TDU (see "operation", below) as the line number following which a line is desired.
4	Suspected system error.
8	Compiler error.
12	Suspected system error.
16	The compiler must restart.
20	The current statement will be reobtained.
24	A line obtained by MOD is to be used by GNSS.
28	MOD met the end-of-data-set (also referred to as "EOS" and "EODS") in obtaining a line.

OPERATION: MOD uses the following excom items to determine its response:

1. TDU. When GNSS is called, TDU equals TDPU and also equals the line number of the last line of the statement passed to Phase 1 on the previous GNSS call. During GNSS operations, TDU is the line number of the last line received from GETLINE (the excom item TDLINF, referred to in the flowchart as "line number"). When returning to Phase 1, TDU will equal the line number of the last line of the statement passed to Phase 1.

2. TDPU. When GNSS is called, TDPU equals the line number of the last line of the statement passed to Phase 1 on the previous call. When GNSS is finished, TDPU is changed to the line number of the last line now being passed to Phase 1.

3. TDAPU. When GNSS is called, TDAPU equals the line number of the last line of the statement prior to that last passed to Phase 1. As only the last passed statement can be forgotten, the statement containing TDAPU is committed for processing.

   When GNSS returns to Phase 1, a new statement is being furnished. Both TDU and TDPU contain the line number of the last line of the statement now being passed. TDAPU is set to the line number of the last line of the statement previously passed. The FORGET flag directs Phase 1 to keep or discard this previously passed statement.

   Note: TDPU is never less than TDU; TDAPU is never less than TDPU.

4. TDPUF, the format of line TDPU.

5. TADPUF, the format of line TDAPU.

6. TDLINO, the line number of the altered line -- the line to replace a line in or be inserted into the source program.

As an example of MOD operation, consider the case where the line number altered is less than or equal to a line number already committed by Phase 1 (TDLINO ≤ TDAPU). A restart must occur, and MOD sets a return code accordingly for GNSS.

For a second example, assume the order is (in sequence of increasing line numbers):

```
TDAPU = TDLINO
TDU   = TDPU
```

Another possibility is one in which the order of increasing statement numbers is:

```
TDAPU
TDLINO
TDPU
TDU
```

If line TDAPU was in card form (TDAPUF = C) the new line could extend TDPU. The new line is inspected. If it is in card format

and a continue line, a restart is required, as TDAPU is committed. If TDAPU was in keyboard form, or if the new line is not in card format and a continue line, TDLINO cannot modify TDAPU, so no restart is required. The next line to be requested by GNSS will be the line following TDAPU. MOD thus sets TDU to TDAPU and directs GNSS to obtain a new line. The insertion of a line preceding TDPU means that TDPU -- currently held in a tentative status by Phase 1 -- must be "forgotten;" therefore, MOD raises the FORGET flag.

CEKTE -- Receive Diagnostic Message (RDM)

RDM accepts a diagnostic message in the form of a list of pointers to pieces of text, assembles the pieces into a line image, adds the message to the source listing, and -- in conversational operation -- sends it to the terminal. See Chart AE.

ENTRIES: RDM has one entry point (CEKTEA), the V-con and R-con for which are found in intercom. RDM is reached only via a standard call. The parameter list is described below with an example containing strings of length 12 and 37 characters to be combined into a message by RDM:

```
LIST   DC   A(PINCOM)    The location within
                         the current active
                         phase of intercom
       DC   A(L1)        String 1 length
       DC   A(T1)        String 1 text
       DC   A(L2)        String 2 length
       DC   A(T2)        String 2 text
       DC   A(ZERO)      End-of-string
       .
       .
L1     DC   FL1'12'
       .
       .
T1     DC   CL12'...'
       .
       .
L2     DC   FL1'37'
       .
       .
T2     DC   CL37'...'
       .
       .
ZERO   DC   FL1'0'
```

ROUTINES CALLED: RDM calls only the master input/output module (MIO), at its DIAGOUT entry. The DIAGOUT V-con and R-con are in excom. Standard linkage is used. The parameter list contains:

```
LIST   DC   A(Intercom) Same as in RDM calls
       DC   A(Text)     Message text
       DC   A(Length)   Message length, bytes
```

EXITS: Standard return linkage is executed to the calling program. The return code set is whatever code was returned by MIO.

OPERATION: RDM first assembles the diagnostic message as a line image from the indicated pieces of text. The DIAGOUT entry to MIO is then called to add the message to the source listing, and, if the compiler is running in conversational operation, to transmit the message to the terminal via PUTDIAG.

If a diagnostic message is greater than 80 characters, a diagnostic message is added to the source listing and only the first 80 characters of the message are sent to MIO.

CEKTF -- Constant Filers (CONFIL)

CONFIL receives numeric address and statement label constants, ensures that they have a symbol table entries, and provides symbol table pointers to the constants. See Chart AF.

CONFIL's CRL subroutine creates an internal statement number and files this number as a label. These labels may be used by compiler phases to mark points in the code.

RESTRICTIONS: Several references are made in text and tables to the filing of constants of one-byte length (referred to as *1 constants) and of length 16 (*16 constants). Currently, no compiler phases require that the Constant Files be able to file *1 constants, and no code is included for such filings, although space is left in various tables. For *16 constants, the only constant of such length currently is a C*16 constant. For such constants, only *8 alignment is required. In this case, the *16 alignment (not space creation) code exists, but is not entered. Similarly, C*8 constant filing uses *4 alignment code. The planning for these constants is based on the possibility that future modifications to the compiler would require the ability to file them.

ENTRIES: The CONFIL subroutines are reached via a standard call. The V-con and R-con values are available to the calling programs in intercom. Prior to calling a CONFIL routine (CRL excluded), the calling program places the constant to be filed in the intercom area TECONS. Upon return, CONFIL will have filled in the intercom item TEPNTR and will have set TEGNU (and, for CRL, TELINO).

The CONFIL entry points, entry symbol, and corresponding TECONS initialization are given in Table 5.

Table 5. Preparation of Constant Receiving Area by CONFIL

Entry Symbol	CONFIL Name	CONFIL V-Con (Intercom)	Description	TECONS Contents
CEKTFB	CONI2	TEVI2	Files I*2 constants	Constant to TECNS1
CEKTFC	CONI4	TEVI4	Files I*4 constants	Constant to TECNS1
CEKTFD	CONR4	TEVR4	Files R*4 constants	Constant to TECNS1
CEKTFE	CONR8	TEVR8	Files R*8 constants	High Order 4 to TECNS1 Low Order 4 to TECNS2
CEKTFF	CONC8	TEVC8	Files C*8 constants	Real 4 to TECNS1 Imag. 4 to TENCNS2
CEKTFG	CONC16	TEVC16	Files C*16 constants	High Order real 4 to TECNS1 Low Order real 4 to TECNS2 High Order imag. 4 to TECNS3 Low Order imag. 4 to TECNS4
CEKTFI	FLAD4	TEVFL4	Files storage class 4 constants other than R-cons	Constant to TECNS1
CEKTFJ	FLAD5	TEVFL5	Files storage class 5 constants	Constant to TECNS1
CEKTFK	FLADVR	TEVVR	Files V-con, R-con pairs	Constant to TECNS1
CEKTFL	FLL	TEVFLL	Files labels	Label to TECNS1
CEKTFM	CRL	TEVCRL	Creates & files labels	--

ROUTINES CALLED: CONFIL calls RDM if overflow occurred in the symbol table storage class table.

EXITS: CONFIL executes a standard return linkage to the calling program. A return code is set in register 15 as follows:

Code	Meaning
0	Normal
8	Symbol table or storage class 4 (the adcon page) overflow

CONFIL returns with register 15 containing zero or, if RDM was called, containing whatever code was returned by RDM.

OPERATION: CONFIL initially determines if a constant of the type being filed has previously been filed. This determination is made by inspecting the appropriate anchor for the chain in which the constant would be included. The constant types, their anchors, and the base of the tables containing the anchors are shown in Table 6.

If the appropriate anchor is empty (contains X'80--' meaning End-of-Chain), the constant is added to the symbol table, the storage class table is updated, and the anchor is made to point to the new entry. (This pointer, like all pointers in the symbol table, is a 2-byte offset from the symbol table base).

If the anchor is not empty, the chain to which it points is searched until either an identical constant is found or an end-of-chain indicator is found. If the constant has previously been filed, return is made with a pointer to the descriptive part of the previously filed constant. If the constant is not already in the chain, it is added, the storage class table is updated, and the previous end-of-chain entry is altered to point to the new entry.

Much of the code in CONFIL is shared by all constant filers. Not all constants are created identically, however. The major differences are:

1. The value part of the name part entries for 8- and 16-byte constants are longer (by 4 and 12 bytes, respectively) than for 2- and 4-byte constants.

2. The descriptive part for label constants is 12 bytes, rather than 8 bytes.

3. Adcons in storage class 5 \(list-entry adcons) are added to the end of the chain, without searching for an identical previous occurrence.

4. The code that searches the chains is divided into three sections for greater speed. The sections search chains for constants of length 2 and 4, 8, and 16 bytes, respectively.

5. One constant filer (FLADVR) files two identical constants -- a V-con and an R-con. The V-con is filed in the adcon storage class 4 chain; the R-con is filed in the R-con chain. The V- and R-cons will occupy adjacent locations in storage class 4, in the order V-con, R-con.

6. Create Label (CRL) creates a label, stores it in TECNS1, then files this label.

The possibility exists that constants in storage class 4 could be given locations in the storage class such that "holes" would exist. For example, addition to the storage class of three constants of 16 bytes, 4 bytes, and 16 bytes, respectively, in that order and with byte alignment, would create a 12-byte "hole". CONFIL fills such holes with items filed later, as described in Tables 7, 8, and 9.

CONFIL checks for symbol table overflow and storage class overflow, which can occur only in storage class 4. If either occurs, the TEOFLO item is set in intercom and return is made to the calling program with a return code of 8. A message is given by CONFIL in such an event.

Table 6. Constant Chain Anchors and Table Bases

CONFIL Name	Constant Type Filed	Anchors (in Excom)	Table Base (in Intercom)
CONI2	I*2	TCCHT (1)	TECHTB
CONI4	I*4	TCCHT (2)	
CONR4	R*4	TCCHT (3)	
CONR8	R*8	TCCHT (4)	
CONC8	C*8	TCCHT (5)	
CONC16	C*16	TCCHT (6)	
FLADVR	R-Cons(STCL = 4)*	TCCHT (7)	
FLAD4	Adcons (STCL = 4)	TCCHT (8)	
FLAD5	Adcons (STCL = 5)	TCCHT (9)	
FLL,CRL	Labels	TCLHT (1-16)	TELHTB

*STCL means Storage class

Table 7. CONFIL Storage Assignment No-Hole Branch Table TFNOHO

Length Constant Being Filed		*1	*2	*4	*8	*16
Corresponding Register P-3 Value		0	4	8	12	16
Current Alignment of Next Space in Storage Class 2	*1	TF610	TF620	TF630	TF640	TF650
	*2	TF610	TF610	TF660	TF665	TF670
	*4	TF610	TF610	TF610	TF675	TF680
	*8	TF610	TF610	TF610	TF610	TF685
	*16	TF610	TF610	TF610	TF610	TF610

Examples of Table Use:

1. An *4 constant is being filed and the alignment in Storage Class 2 is also *4. Branch to TF610. (No holes produced in Storage Class.)

2. An *4 is being filed, and the alignment is *8. Branch to TF610. (No holes produced).

3. An *8 is being filed and alignment is *2. Branch to TF665, at which point an *2 (and *4 if six bytes are required to create *8 alignment) hole will be made available as a result of *8 alignment being produced for the constant.

Table 8. CONFIL Storage Assignment Hole Availability Table

Length Constant Being Filed					*1[3]	*2[3]	*4	*8
Corresponding Register P-3 Value					0	4	8	12
Available Hole	*8	*4	*2	*1				
TDHOLE = 0					TF590	TF590	TF590	TF590
1				X	TF511	TF590	TF590	TF590
2			X		TF521	TF522	TF590	TF590
3			*X	*X	TF511	TF522	TF590	TF590
4		X			TF541	TF542	TF544	TF590
5		X		X	TF511	TF542	TF544	TF590
6		X	X		TF521	TF522	TF544	TF590
7		X	X	X	TF511	TF522	TF544	TF590
8	X				TF581	TF582	TF584	TF588
9	X			X	TF511	TF582	TF584	TF588
10	X		X		TF521	TF522	TF584	TF588
11	X		X	X	TF511	TF522	TF584	TF588
12	X	X			TF581	TF542	TF544	TF588
13	X	X		X	TF511	TF542	TF544	TF588
14	X	X	X		TF521	TF522	TF544	TF588
15	X	X	X	X	TF511	TF522	TF544	TF588

Examples of Table Use:

1. An *4 constant is being filed, no *4 constant hole is available (no X under *4 in the Available Hole columns), and no *8 hole is available. Branch to TF590, where space will be taken by increasing the size of the Storage Class.

2. An *2 constant is being filed, no *2 hole is available, but an *4 hole is available. Branch to TF542, at which point part of the *4 hole will be used, with the unused part of the hole assigned to the *2 hole.

3. Not implemented.

Table 9. CONFIL Storage Assignment Byte Alignment Branch Table TFBAL

Alignment of Next Available Byte					
Address Bits				Constant Length	Number Loaded Into N4
8	4	2	1		
				*16	80
			X	*1	0
		X		*2	20
		X	X	*1	0
	X			*4	40
	X		X	*1	0
	X	X		*2	20
	X	X	X	*1	0
X				*8	60
X			X	*1	0
X		X		*2	20
X		X	X	*1	0
X	X			*4	40
X	X		X	*1	0
X	X	X		*2	20
X	X	X	X	*1	0

CEKTH -- Master Input/Output (MIO)

All communication between interface programs supplying source line input to and producing edited line output for the compiler is accomplished my MIO. The compiler I/O operations are:

1. Calls on LPC GETLINE
2. Calls on LPC PUTDIAG
3. Opening of, additions to, and closing of the list data set

See Chart AG.

ENTRIES: The entry points to MIO are listed below. All are reached by standard calls.

Entry	Name	List in Parameter List (Address)
LDOPEN	CEKTHA	Intercom
LDCLOSE	CEKTHB	Intercom
LINEIN	CEKTHC	Intercom
DIAGOUT	CKTHD	Intercom, line address, 4-byte character count.
OLR	CEKTHE	Intercom, line address, 4-byte character count, and flag. The flag item will be zero or a PL4 format line number, with the following results. Zero. The output line will begin in column 1 of the list data set line. Lines presented are expected to be preceded by a carriage control character. PL4. The output line will begin in column 10 of a list data set line. The first nine characters will be XXXXXXXBB, where X = a numeric digit, and B = blank. A carriage control character of a blanks is associated with this line.
BFLUSH	CEKTHF	Intercom, Flag. Flag is 4, 8, or 12 for flushing old, new, or both buffers, respectively.

ROUTINES CALLED: MIO calls the LPC entries GETLINE and PUTDIAG and uses data management through VISAM I/O macro instructions.

The CALL and RETURN statements for MIO calling GETLINE and PUTDIAG are given below.

FORTRAN to GETLINE Call

Register 1 contains the address of the parameter list.

Register 13 contains the address of the FORTRAN save area to be used by GETLINE.

Register 14 contains the return address.

Register 15 contains the V-type Adcon for GETLINE (i.e., the entry point address).

```
[symbol] CALL GETLINE entry-symbol, (line
         number
                        (15)
              to GETLINE-addr, line number
              from GETLINE-addr, length of
              line-addr, source-addr,
              altered line number-addr)
```

GETLINE Entry

Entry point name for GETLINE.

Line Number to GETLINE

Specifies the address of a 1-word field containing a packed decimal number. (FORTRAN is requesting a source line which follows the line with this number.)

Line Number From GETLINE

Specifies the address of a 1-word field which will contain a packed decimal number when GETLINE returns to FORTRAN normally (i.e., return code = 0). This will be the

line number of the source line which GETLINE is giving to FORTRAN.

Length of Line

Specifies the address of a 1-word field which will contain a binary number when GETLINE returns to FORTRAN normally. This will be a count of the number of characters in the source line which GETLINE is giving to FORTRAN. This count will include the format character (see source line below).

Source Line

Specifies the address of a field which will contain the address of the source line when GETLINE returns to FORTRAN normally. This line will contain a maximum of 150 characters. The first character will be 0 or 1 (hexadecimal), depending upon whether the line is card or keyboard, respectively.

Altered Line Table

Specifies the address of a 1-word field which will contain (in packed decimal format) the line number of the lowest line modified when GETLINE returns to FORTRAN with a return code of 4.

GETLINE to FORTRAN Return

[Symbol] RETURN

Upon return from GETLINE, register 15 contains a code which may be interpreted as follows:

Code	Type of Return	Parameter Information Filled In
0	Normal (source line has been obtained).	Line number from GETLINE. Length of line. Source line
4	Lines have been altered.	Altered Line number
8	Batch -- EDOS (End-of-Data-Set. GETLINE was asked for a line after the last line in the data set).	None.
12	"abend-type"	Indeterminate.

FORTRAN to PUTDIAG Call

Register 1 contains the address of the parameter list.

Register 13 contains the address of the LP save area to be used by PUTDIAG.

Register 14 contains the return address.

Register 15 contains the V-type adcon for PUTDIAG (i.e., entry point address).

```
[symbol] CALL  PUTDIAG, entry - symbol
                          (15)
               [,(message-addr, length of
               message-addr, correction
               request indicator-addr)]
```

PUTDIAG Entry

Entry point for PUTDIAG.

Message

Specifies the address of an area which contains the message.

Length of Message

Specifies the address of a 1-word field which contains, in binary, the number of bytes in the message.

Correction Request Indicator

Specifies the address of a 1-byte field which indicates whether the message is to go to SYSOUT immediately (00000000) or is to be stacked by LPC and output as a correction request at the next entry to GETLINE (00000001).

PUTDIAG to FORTRAN Return

[Symbol] RETURN

Upon return from PUTDIAG, register 15 contains a code which may be interpreted as follows:

Code	Type of Return
0	normal
12	"abend-type"

EXITS: MIO exits to the calling program using a standard linkage. A return code is set in register 15 as follows:

Code	Description
0	Normal return.
4	GETLINE or PUTDIAG returned with an "abend-type" value in register 15. The program calling MIO will return to its caller with a return code of 4, until the phase controller is reached. The phase controller will then return to LPC with a return code of 4.

OPERATION

MIO has six entry points. These are described below.

1. List Data Set Open Entry -- LDOPEN

   The phase controller enters at LDOPEN to open the list data set. Opening will not occur again, unless FORTRAN is reached at its initial entry or a restart occurs. Restart will cause the list data set to be closed, and then reopened, thus discarding the contents of the previous list data set.

2. List Data Set Close Entry -- LDCLOSE

   The phase controller enters at LDCLOSE to close the list data set. Lines held in the list data set buffer (see ORL below) are output before closing.

3. Get a Line From LPC Entry -- LINEIN

   This entry is used by the exec subroutine CEKTC (GNSS) when a source statement is being formed for Phase 1 of the compiler. LINEIN will call the LPC subroutine GETLINE and pass the results to GNSS via excom.

   When processing card lines, GNSS requests the first line of each statement twice, once to determine that the previous statement is not to be continued, and once to obtain the first line of the new statement. LINEIN does not issue two calls on GETLINE under such circumstances. Rather, LINEIN saves the line after the first request, in anticipation of the second request.

   If GETLINE sets register 14 to note an abnormal end condition, LINEIN returns to GNSS with a return code that will force an abnormal end return by Phase 1 to the phase controller, followed by an abnormal end return by the phase controller to LPC.

4. Output a Diagnostic Message Entry -- DIAGOUT

   Any executive subroutine or any phase wishing to output a diagnostic message may do so by calling the receive diagnostic message subroutine, RDM. RDM forms the message and adds it to the terminal (unless in batch operation) and list data set (if any), using the DIAGOUT entry to MIO. In cases where executive modules have access to a complete line, they call DIAGOUT directly, for increased efficiency.

   During Phase 1 operation, diagnostic messages will frequently be output concurrently with addition of source lines to the list data set by the MIO entry OLR (see below).

   If the computer is running in conversational mode, OLR does not output source lines as soon as they are received, as a terminal correction may require deletion of lines. Instead, lines are stacked in one of the two MIO buffers (obtained by a PHC GETMAIN). Diagnostic messages concerning these lines are also stacked, in the same buffers, and added to the list data set only when the source line causing the diagnostic is added to the list data set.

5. Output Line Receiver -- OLR

   This entry is used by PHC (Phase Controller), GNSS, and Phase 5 of the compiler to add source lines to the list data set.

   OLR operation when called from anywhere except GNSS is quite simple, as lines to be added to the list data set will never be replaced. GNSS use, on the other hand, is more complex, since both the previous statement processed by Phase 1 and the current statement being prepared for Phase 1 may be deleted, due to conversational corrections. In such a case, the source lines for these statements must not be added to the list data set. This purpose could be accomplished by retaining the entire source program in virtual storage. The procedure adopted by OLR is to stack source lines (with diagnostic messages and comment lines received concurrently) in buffers until the source statement is irrevocably committed to inclusion in the Phase 1 tables and, thus, to further processing by following phases.

   The possibility exists that the capacity of any reasonably-sized buffer will be exceeded, due to an abnormally large number of comment lines contained within a source statement. In such a case, a message will be added to the list data set, to the effect that the statement will be repeated if corrected at the terminal.

6. Flush the Buffer -- BFLUSH

   This entry is used by GNSS to move a source statement and associated diagnostic messages from an output buffer to the list data set. This operation is performed only when it is determined that the statement cannot be replaced through conversational corrections.

CEKTI Analyze Console Source Line (ANALYZ)

ANALYZ, which is assembled into GNSS (CEKTC), analyzes a console-furnished source line to determine the location in the string of the statement number (if any) and the text. The statement number is moved to intercom; the location of the first text character and the number of text characters are returned to the calling program GNSS. See Chart AH.

ENTRIES: ANALYZ is reached only from GNSS, via a restricted linkage INVOKE. All information required by ANALYZ is in intercom. ANALYZ returns with TDLE in N3, and TDNUMC in V2.

ROUTINES CALLED: ANALYZ invokes subroutine INSCON (CEKTJ) for inspection of individual characters.

Information placed in registers for INSCON is:

Register	Contents
P2	LASTC, the address of the first character beyond the last text character.
V2	I, the address of the last character inspected by INSCON.

ANALYZ initializes V2, which is updated by INSCON. P2 is used, but not changed by INSCON.

EXITS: ANALYZ returns to GNSS via a RESUME, with no registers set. All information required by GNSS is in excom.

OPERATION: ANALYZ is invoked by GNSS with information in excom giving the line length (TDLONG) and the address of the area containing the line (TDLADD). There are too many possible legitimate combinations of text characters to describe all ANALYZ operations in writing, but the ANALYZ flowchart (Chart AY) gives all logic paths.

Refer to FORTRAN Programmer's Guide, "Appendix A: Entry and Correction of FORTRAN Source Statements," for information concerning the format of source statements.

CEKTJ -- Inspect a Console Character (INSCON)

INSCON is assembled into GNSS; its function is to inspect a character in a console source line to determine if it is tab, numeric, blank, or other. See Chart AI.

ENTRIES: INSCON is invoked by subroutine ANALYZ via restricted linkage. Information required by INSCON is all in registers prepared by ANALYZ, as follows:

Register	Contents
P2	LASTC, the address of the first character beyond the last text character.
V2	I, the address of the last character inspected prior to INSCON entry.

INSCON alters P1 and P3 for use by the calling program.

ROUTINES CALLED: None

EXITS: INSCON returns to the calling program with a RESUME, with a code in RC as follows:

Code	Description
0	Not used.
4	INSCON could not inspect a character, as the end of line was exceeded.
8	The next character was a tab.
12	The next character was numeric.
11	The next character was blank.
20	The next character was not tab, numeric, or blank.

OPERATION: INSCON tests the address next character in the console line to see if the line end has been reached. If so, the RC = 4 return is taken; if not, the character is converted, inspected, and return made with the RC code set appropriately.

CEKTK -- Move a Line to the List Data Set (LDMOVE)

LDMOVE is assembled as part of the master input/output module and is invoked by MIO via restricted linkage to move a line from a buffer to the list data set. LDMOVE counts lines moved, restores the page, and adds a page heading when required. See Chart AJ.

ENTRIES: LDMOVE is reached from the MIO subroutines FLUSH and BUILD, via an INVOKE. All information required by LDMOVE is in excom, intercom, the MIO PSECT, or registers N1 and N2:

N1 = text address
N2 = character count

ROUTINES CALLED: LDMOVE uses the VISAM PUT macro instruction to add lines to the list data set.

EXITS: LDMOVE sets no registers for invoking programs. Return is via a RESUME.

OPERATION: LDMOVE determines whether a new page is to be started, and, if so, moves the page heading from the internal files area to the list data set. The new page number is included in this heading.

LDMOVE adds the line to the list data set, updates line counters, and returns.

CEKTL -- Build a List Data Set Buffer (BUILD)

BUILD is assembled as part of the master input/output module and is invoked by MIO via restricted linkage to move a line to a list data set buffer or the list data set. This buffer is emptied using FLUSH (see CEKTM) when full, when the list data set is to be closed, or when a source statement is committed to further compilation. (See Chart AK.)

ENTRIES: BUILD is reached from the DIAGOUT and OLR entries to MIO, via an INVOKE. Programs, invoking BUILD, set registers as follows:

N1 = the address of the line to be processed
N2 = the number of characters in this line

ROUTINES CALLED: BUILD may enter LDMOVE or FLUSH via an INVOKE. No registers are set for, or expected to be set by, these subroutines.

EXITS: BUILD returns to its caller via a RESUME, with no registers set.

OPERATION: When BUILD is called in batch mode, it invokes LDMOVE at once, to move the line directly to the list data set.

In conversation, BUILD checks first to see if the list data set buffer currently being built is full; if it is, FLUSH is called. The line is then added to one of the MIO buffers.

CEKTM -- Flush a List Data Set Buffer (FLUSH)

FLUSH is assembled as part of the Master Input/Output module and is invoked to flush one or both of the list data set buffers by moving all lines resident in the buffers to the list data set. See Chart AL.

RESTRICTIONS: Register P6 must be set for FLUSH, as follows:

P6	Description
4	Flush the old buffer
8	Flush the new buffer
12	Flush both buffers

ENTRIES: FLUSH is reached from the MIO entry BFLUSH, LDCLOSE, and BUILD via an INVOKE. All items required by FLUSH are in excom, intercom, or the MIO PSECT.

ROUTINES CALLED: FLUSH invokes LDMOVE. No registers are set for, or expected from, this invocation.

EXITS: FLUSH returns via a RESUME, with no registers set for the calling program.

OPERATION: FLUSH determines if the buffer contains any lines to be removed; if it does, FLUSH repeatedly invokes LDMOVE until the buffer is empty. Otherwise, FLUSH returns at once.

CEKTQ -- Compiler File Dump (COMDUMP)

COMDUMP prepares hexadecimal dumps of compiler internal files, as part of the compiler diagnostic feature processing.

ENTRIES: COMDUMP contains one entry point, CEKTQA.

ROUTINES CALLED: COMDUMP calls LINDUMP at its CEKTSA entry, by means of the CEKTG macro.

EXITS: COMDUMP always makes a RETURN to the calling program, with no return code set.

OPERATION: COMDUMP is called with three parameters: the address of intercom and the low and high addresses of the area for which a hexadecimal dump is to be prepared. The COMDUMP output lines are issued via the CEKTG macro instruction. This macro instruction issues a call on the LINDUMP module, which in turn issues a VISAM PUT to pass the line to the compiler user.

An error message is given and no dump is produced if the parameters have the second address greater than the third.

CEKTS -- Compiler Line Dump (LINDUMP)

LINDUMP is called by the macro CEKTG, after CEKTG sets up parameters for the call. LINDUMP then forms one or more lines in accordance with parameters passed, and issues these via the VISAM PUT macro instruction.

ENTRIES: LINDUMP contains one entry point, CEKTSA.

ROUTINES CALLED: The PUT macro instruction, issued by LINDUMP, leads to an external call.

EXITS: COMDUMP always makes a RETURN, with no return code set.

OPERATION: Before describing LINDUMP, a description of the CEKTG macro instruction will be given.

Use of CEKTG

CEKTG can be used by macro instruction, in the forms:

1. CEKTG AREA,FORMAT,SIZE
(one area, one format)

where:

AREA - may be any symbol defined in the program or a term such as D(RN), where D is any displacement and RN any register.

FORMAT - may be:

0 or X for hexadecimal
1 or F for fullword integer
2 or H for halfword integer
3 or C for character
4 or Q for quarter-word integer
5 or B for binary

SIZE - may be any absolute or relocatable expression of up to eight characters. It is the byte size of AREA.

If FORMAT is C and SIZE is 133, only 132 characters are printed, and the first character is used to control printer skipping and spacing as follows:

1 = Skip to new page before printing the line
0 = Space one line before printing the line
+ = Space two lines before printing the line

...and any other character is ignored.

If AREA falls in the range 0 to 15, it is assumed to relate to an index register (general-purpose). SIZE then means the number of bytes, starting at the left-most (high-order) byte of that register. Wrap-around takes place, and the registers are printed as they were before the macro instruction was executed.

2. CEKTG A1,F1,A2,F2,....A6,F6
(up to six areas and formats)

where:

A1,A2,...A6 -- may be as specified for AREA above.

F1,F2,...F6 -- consist of a single letter, followed by an integer number in the range 1 to 999. The letter may be:

X for hexadecimal
F for fullword integer
H for halfword integer
Q for quarter-word integer
C for character
B for binary
N for name-indicator (see comment below)

Unless, and until, an N-type format is encountered, each area is printed separately on one or more lines, with the address of the area indicated, and the format letter shown. The area associated with the N-type format is printed in characters, starting at print position number 1. Other areas following the N-type format are printed alongside, up to a print line limit of 120 characters; additional lines are used if required. There will be spaces between individual items (bytes, halfwords, or fullwords) of multiple areas.

A1,A2,...A6 may refer to general-purpose registers, if they fall in the range 0-15. (See discussion on using the single-area CEKTG, above.)

There is no print control option with multiple areas.

The format parameters always specify length in bytes.

The standard CEKTG output line starts with REF, followed by the hexadecimal return address in the calling program, followed by ADR, followed by the decimal address of the area being printed, followed by the hexadecimal address of the area being printed, followed by a format letter (X, F, H, Q, or B), followed by data items. The data items are separated by spaces, except in the case of the Q format.

When a single area is printed in character format, or when multiple areas follow a name area (see above), the standard indication is dropped, and data starts at print position 1.

Except when using character format, there is always one space between the output of successive entries to CEKTG.

The CEKTG macro instruction saves and restores all registers around the call.

This is the initial PRF entry generated at the initialization of Phase 1

CEKTG -- Calling Sequence

1. The calling sequence for the single-area CEKTG is as follows:

```
        LA     0,N SET FORMAT CONTROL
        LA     1,PARAM  POINT TO LIST
        L      15,ADCEKT
        CALL   (15),MF=(E,(1))

PARAM   DC     A(AREA)
        DC     A(SIZE)SIZE IS IMMEDIATE
                      VALUE
ADCEKT  ADCON  IMPLICIT,EP=CEKTSA
```

...where the parameter N is a number in the range 0-5 (see notes on single-area CEKTG macro instruction above).

2. The calling sequence for the multiple-area CEKTG is as follows:

```
        LA     0,6  SET MULTIPLE AREA
        LA     1,PARAM  POINT TO LIST
        L      15,ADCEKT
        CALL   (15),MF=(E,(1))
        ...
PARAM   DC     A(A1)  FIRST AREA
        DC     CL4'F1'  FIRST FORMAT
        DC     A(A2)  SECOND AREA
        DC     CL4'F2'  SECOND FORMAT
        ...
        NOPR   0  END OF LIST
ADCEKT  ADCON  IMPLICIT,EP=CEKTSA
```

The parameters required for LINDUMP to prepare line(s) as described above are stored by CEKTG in intercom. CEKTG then calls LINDUMP. LINDUMP inspects these parameters, and builds one line of output. This output is issued via a VISAM PUT, a second line is prepared if requested, and so on.

INTRODUCTION

Phase 1 performs the initial scan of the source program, analyzes it for syntactical correctness, and encodes the information for subsequent processing. Figure 13 illustrates the operation of Phase 1.

On entrance from the compiler executive, Phase 1 initializes itself, calls GNSS to get the first source statement, and enters its main loop.

The main loop is traversed once for each source statement. It classifies the statement and calls an appropriate subroutine to process the statement. On return from an individual statement processor, GNSS is called for the next source statement. GNSS indicates, by the forget flag, whether the statement just processed should be compiled or ignored due to action by a conversational user. In the latter case or if the statement just processed contained serious errors, the results of processing that statement are expunged from all tables and output files, and the loop is reentered at the top to process the new statement just obtained. Otherwise, final housekeeping appropriate to the old statement is performed, and the loop is reentered at the top.

The processing for a statement includes producing appropriate output. Executable statements cause entries in the program representation file (PRF). Declarations may set fields in symbol table entries or produce output in the stoage specification list or preset data file. In addition, certain statements may affect Phase 1 internal tables and flags.

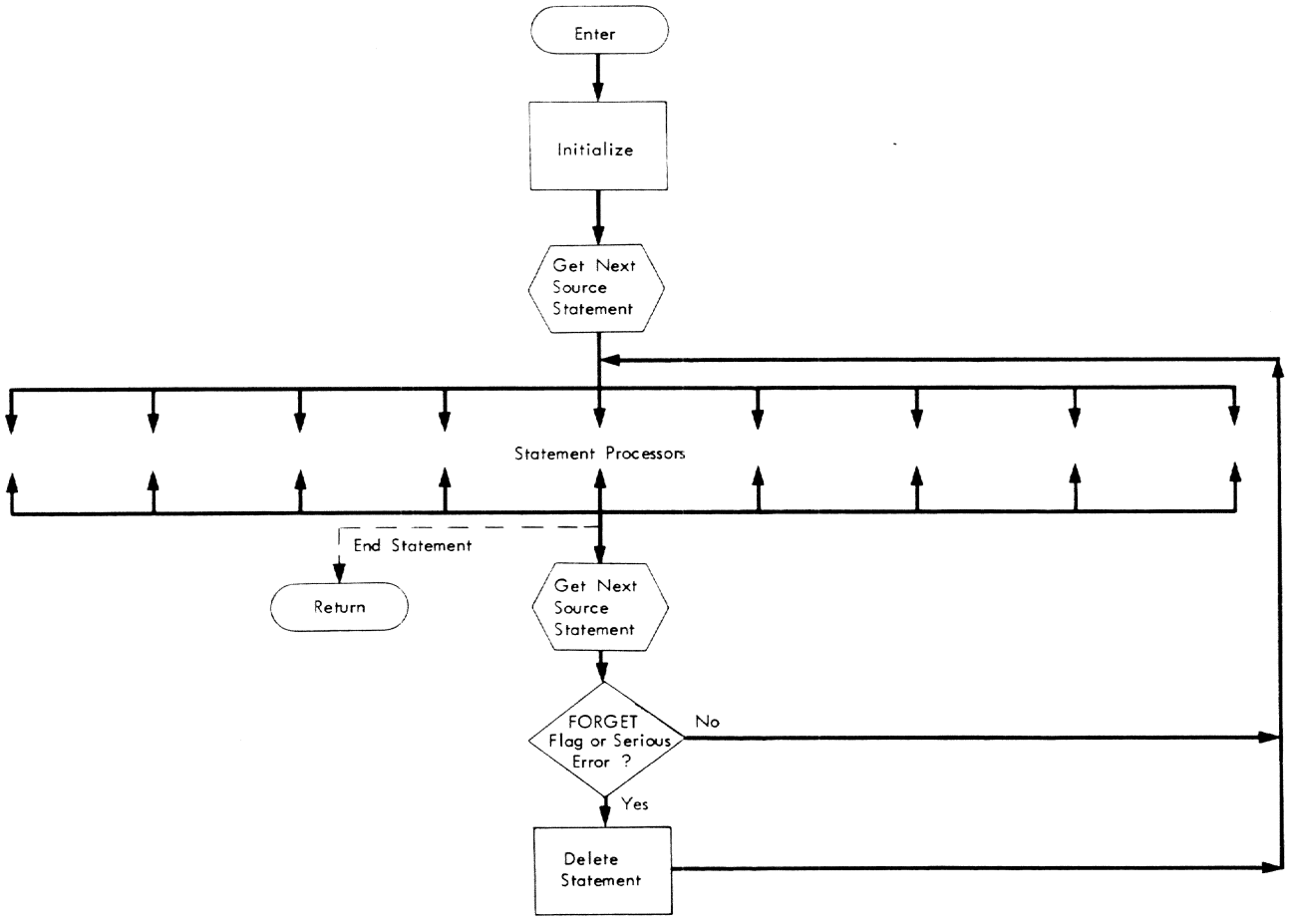


Figure 13. Phase 1 Interface

Access to the source text is through two subroutines: ESC and ACOMP. ESC supplies the next character on request. ACOMP supplies a pair of consecutive items: the first is a variable name, function name, constant, or statement number (label); the second is a delimiter. ACOMP calls subroutines to make symbol table entries, convert constants to binary, etc.

Statements containing arithmetic or logical expressions call the subroutine EXPR to process these expressions into Polish notation, which is output in the expression representation file (ERF). The subroutine SUBS processes subscripts, as a special category of expressions for EXPR and the statement processors.

When source program errors are detected by Phase 1, the subroutine ERR is called to prepare a diagnostic message and transmit it to the executive subroutines RDM. The message is determined by the parameters presented to ERR. A parameter may indicate a piece of prestored text to be included in the message or may direct the subroutine to obtain information from the compiler's tables (e.g., a name from a symbol table entry) and insert it in the message. Depending on the entrance used, ERR will also set the local maximum error code and may raise the delete flag.

After recognizing and processing the source program END statement, Phase 1 returns control to the executive.

Phase 1 has one PSECT that provides working storage for all Phase 1 modules. This PSECT is contained in module PHIM (CEKAI) and is organized as shown in Figure 14.

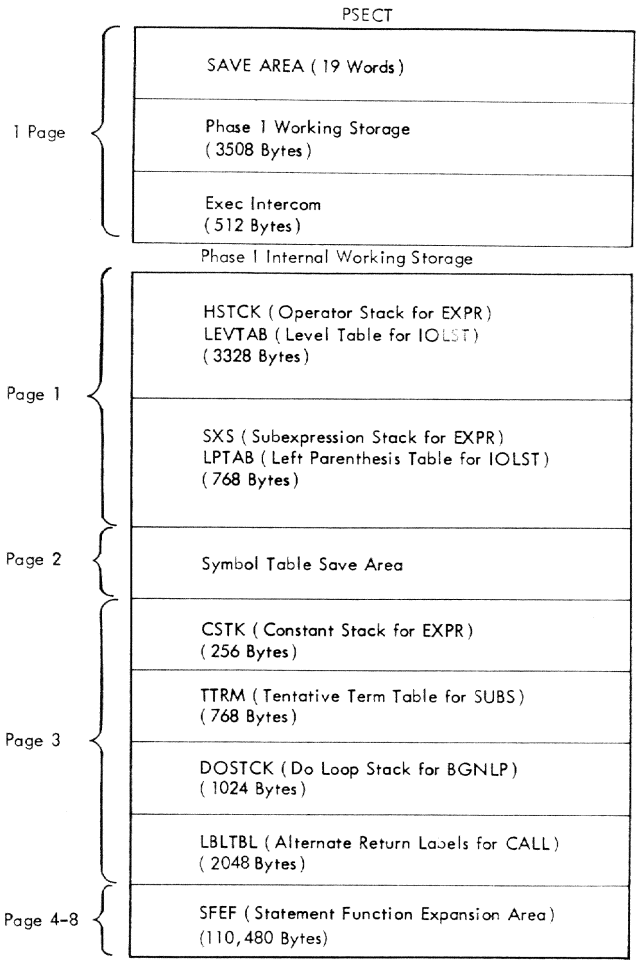


Figure 14. Phase 1 Storage

Phase 1 creates entries in the interphase files and tables listed below.

PROGRAM REPRESENTATION FILE (PRF)

The PRF consists of the executable elements of a source program. PRF entries are linked (chained) together in the sequence of their generation. Additional linking connects PRF entries by types.

Definition point analysis connects each definition point of each variable, connects the definition points of any formal arguments, and connects the definition points of all COMMON variables. Variables are defined when used as:

1. The expression to the left of an equal sign in an arithmetic or logical statement.
2. An induction variable of a DO statement.
3. A variable in an input list.
4. An argument of an external subprogram.

(All COMMON variables are defined when an external subprogram reference occurs in the source program.)

Statement number processing establishes the branching structure of the source program. Statement number definitions are entered in the PRF and are linked. All statement numbers referenced as branch points are linked.

DO statement processing establishes the looping structure of the source program. The beginning and terminating points of each loop are connected to each other and to other loop delimiting points. In addition to the loops specified by the source

program, a false loop is indicated before the first executable statement of the source program. This provides a position in the PRF for computation of expressions that are effectively constants in the program.

The program representation file, as generated by Phase 1, consists of the following types of entries. Additional entries are added by succeeding phases.

Begin Program Entry

This is the initial PRF entry generated at the initialization of Phase 1. Program type is set to indicate a main program. This setting is changed by the occurrence of a subprogram (SUBROUTINE or FUNCTION) statement. This entry is the terminal entry of the LINK chain.

Subprogram Entry

This multiple-purpose entry is a global (external) entry point. As such, it is linked into the label definition chain within the PRF. It has a pointer to the symbol table entry of a subroutine or function name and a pointer to a list of symbol table entries of the formal parameters of the subprogram. This list specifies the order of occurrence of the formal parameters. This is a false loop level entry. It is the primary entry point of a program.

Alternate Entry

This entry is generated for each occurrence of an ENTRY source statement and identifies a secondary entry point for a program. It is a global (external) entry point. As such, it is linked into the label definition chain within the PRF. It is a false loop level entry. It has pointers to the symbol table for the entry name and for entries of the formal parameters.

Label Definition Entry

The label definition entry is generated for each occurrence of a statement number in the source language and for each compiler-generated statement number. Label definition PRF entries mark possible entry points for local (internal) flow control. To facilitate the flow analysis by Phase 2, they are linked to the preceding entry point.

Equation Entry

This entry is generated from a FORTRAN assignment statement (arithmetic or logical). It contains a pointer to the expression representation file (ERF) entry representing the expression to the left, and another for the expression to the right, of the equal sign. An assignment statement is a variable definition point for the assigned-to identifier; it may be a common definition point if the defining expression contains a reference to an abnormal function. An "abnormal" function subprogram is one which does any of the following:

1. Refers to or changes the value of any COMMON variable.
2. Changes the value of any of its arguments.
3. Causes input or output.
4. Does not always return the same value when called with the same arguments.

   All external functions are treated as abnormal by the compiler.

GO TO Entry

A GO TO entry is generated for each occurrence of a GO TO source statement. Each is linked to the preceding label referencing PRF item, forming the referenced label chain used by Phase 2.

Assigned GO TO Entry

This entry is generated for each occurrence of an assigned GO TO source statement and contains a list of the statement numbers which may be assigned to the variable. Each statement number in the list is presumed to be referenced at this entry and, therefore, is linked to the preceding label referencing PRF item for analysis by Phase 2.

Computed GO TO Entry

Each occurrence of a computed GO TO source statement causes an entry that contains a list of the statement numbers to which control can be transferred. Each label in this list is assumed to be referenced at this entry and therefore, is linked to the preceding label referencing PRF item for analysis by Phase 2.

ASSIGN Entry

This entry is generated for each occurrence of an ASSIGN source statement. It is considered neither a reference to the label specified nor a redefinition of the assigned variable. Hence, the PRF entry is

not linked into the label reference chain or into the definition point chain. This entry is applicable to code generation only.

Arithmetic IF Entry

This entry is generated for each occurrence of an arithmetic IF source statement. Each label specified is assumed to be referenced at this entry. This item is linked to the preceding label-referencing PRF item. A label value of zero indicates fall-through to the next executable statement. If the expression contains a reference to an abnormal function, this item serves as a redefinition point for all COMMON variables and is linked into the common definition chain within the PRF. The test expression is in the ERF.

Logical IF Entry

This entry, generated for each occurrence of a logical IF source statement, combines the logical expression part with a conditional branch part to make the PRF entry very similar to the arithmetic IF. If the conditional statement is not a simple GO TO source statement, the expression is negated, a label is generated, and a transfer true to the generated label is indicated. This item is linked to the preceding label referencing PRF item. If the expression contains a reference to an abnormal function, this item serves as a redefinition point for all COMMON variables and is linked into the COMMON definition chain within the PRF.

CALL Entry

Statement numbers specified as actual arguments of a CALL source statement are entered into a list in a PRF entry. Each label in the list is assumed to be referenced at this PRF entry. Hence, CALL PRF entries are linked to the preceding label referencing PRF entry. The occurrence of a CALL source statement, which is a reference to an abnormal function, is a redefinition point for all COMMON variables and is linked into the COMMON definition chain within the PRF.

The occurrence of a CALL source statement also effects the generation of an argument definition point PRF entry for each actual argument of the call that is a simple or subscripted variable. Each of these PRF entries is linked to the previous definition point of the argument.

Argument Definition Entry

This entry is generated for each actual argument of an external reference (a call) that is a simple or subscripted variable. Each entry is linked into the definition chain of the particular variable.

RETURN Entry

This entry is generated for each occurrence of a RETURN source statement within a subprogram. STOP PRF entries are generated for RETURN statements occurring within a main program. These entries are either explicit or implicit references to global (external) labels. As such, they are linked to the preceding label referencing PRF entry.

Begin Loop Entry

For each DO statement, each implied DO statement, and around the total PRF excluding global (external) entry points, there are begin and end loop PRF entries. For each begin loop three successive PRF entries are made. Having three entries facilitates the optimization processing of Phase 3. Loop PRF entries are interlinked. Each begin loop links to the previous begin loop and end loop PRF entries. The begin loop entry is also linked to its own end loop PRF entry.

End Loop Entry

This entry is generated upon completion of the processing of a statement with a label that matches the last label in the DO pushdown list. An end loop entry is linked to the corresponding begin loop and to the previous loop PRF entry, begin or end.

CONTINUE Entry

An entry is included only to show presence of CONTINUE statements in the source program.

READ, READ Without Unit, and READ With NAMELIST Entries

An entry is generated for these source input statements. READ statements having either an EOF label or an ERR label specified are linked into the label reference chain of the PRF.

WRITE and WRITE With NAMELIST Entries

These are generated by WRITE source statements.

PRINT and PUNCH Entries

These are degenerate (particular) cases of WRITE source statements.

Input/Output List Representation in the PRF Entry

Each entry is a redefinition point when the list is associated with a READ statement. Begin and End Loop PRF entries from implied DOs are appropriately interspersed. A list sequence of PRF entries follows the I/O PRF entry with which they are to be associated.

End List Entry

This is a control entry in the PRF sequence to indicate the termination of an I/O list sequence to the code generation phase.

END FILE, REWIND, and BACKSPACE Entries

An entry is generated upon occurrence of each of these statements in the source program.

STOP Entry

This entry is generated upon occurrence of a STOP statement in the source program or for a RETURN statement appearing in a main program.

PAUSE Entry

This entry is generated upon occurrence of a PAUSE in the source program.

End Program Entry

PRF control entry to indicate the end of the PRF.

EXPRESSION FILE

The expression file consists of individual strings of entries which are operands and operators in the usual right-hand Polish notation. These strings represent all arithmetic and logical expressions occurring in the source program and any subscripts that are not constants. Expression file entries are generated by the following statements: equation, arithmetic and logical IF, READ, WRITE, PRINT, PUNCH, RETURN with variable index, ASSIGN, assigned and computed GO TO, and CALL. An entry may consist only of an operand, as is the case with the entries for ASSIGN, GO TO, RETURN, etc., statements.

Subscript Expressions

For subscript expressions, especially those containing loop variables, the occurrence of a loop variable causes its initial value to be incorporated into the expression. Also, the array item length is incorporated into the expression, so that the expression can be used directly in address computation. Wherever possible, terms are combined in order to increase efficiency. Finally, two additional plus operators are included before the special subscript operator to facilitate processing by Phase 3.

Special representations in the ERF are shown below.

Function and subroutine references:

```
F(x)        x F ;
F(x,y)      x y , F ;
F(x,y,z)    x y , z , F ;
 .
 .
 .
```

Max and Min function references:

```
AMAX1(X,Y, Z)    X Y MAX Z MAX
AMAX0(I, J, K)   I J MAX K MAX FLOAT ;
    .
    .
    .
```

Subscripts:

array - variable item with offset and flag

subscript expression - sum of products

subscript operator - :

STORAGE SPECIFICATION TABLES

The storage specification tables consist of two types of entries: a common entry and an equivalence entry. A common entry is made for each occurrence of a COMMON statement in the source program and represents each variable and its particular storage class (blank or named COMMON) in the statement. An equivalence entry is made for each occurrence of an EQUIVALENCE statement in the source program and represents the variables in each EQUIVALENCE group and their offsets, if any.

The way an equivalence entry is made depends on the dimension information preceding or following the equivalence statement.

In the event that dimension information for a particular variable (DIMENSION, COMMON or TYPE statement) or that a subscripted variable in the EQUIVALENCE statement contains only a single subscript, the offset in EE1 or EE6 is computed.

EE2-5 or EE7-10 are not used.

The type field in EE1 or EE6 indicates the type of variable.

When dimension information does not precede the EQUIVALENCE statement and a subscripted variable in the EQUIVALENCE statement contains more than one subscript, EE1 or EE6 contains the number of subscripts. In this case EE2 or EE7 are required, and EE3-5 or EE8-10 may be required.

The type field in EE1 or EE6 is set to 'FF', indicating that this variable contains the number of subscripts in EE1 or EE6 followed by EE2 or EE7 and possibly EE3-5 or EE8-10.

DIMENSION TABLE

The dimension table consists of entries in the preset data reference set. An entry is made for each array dimension specification occurring in the source program. These specifications may occur in DIMENSION, COMMON, or explicit type statements. If the array is not a formal argument, the entry represents the number of dimensions, total size, and the dimension products of the array. If the array is a formal argument, the entry represents the number of dimensions and the individual size specifications (value for a constant or symbol table pointer for a variable).

NAMELIST TABLE

The namelist table consists of entries in the preset data reference set. Each entry consists of a set of symbol table pointers to the variables in a given NAMELIST. An entry is made for each occurrence of a namelist name in a NAMELIST statement.

STORAGE CLASS TABLE

Phase 1 also adds certain information to the storage class table. Each COMMON block name occurring in a COMMON statement is entered into the storage class table and causes the count containing the number of COMMON block names to be updated. Also, for each occurrence of a FORMAT statement or a literal constant (except as initial values in a DATA or Type statement), the alphameric storage class counter is incremented by the number of bytes in the format or literal constant.

FORMAT PROCESSING

Format labels are entered into the symbol table and marked as defined. The current value of the alphameric storage class counter is entered as the storage location in the descriptive part of the symbol table entry.

The alphameric format information, including the initial open parenthesis and the terminal closing parenthesis, is output as an alphameric table entry in the preset-data-reference set. The location of this entry is entered into the descriptive part of the label symbol table entry. The alphameric storage class counter is incremented by the number of bytes of alphameric information.

The alphameric table entry consists of an identification element, an alphameric element, and either a termination element or a continuation element. All alphameric table entries are linked together. As each new entry is made after the initial entry, the terminal ID is changed to continuation, and the new entry location is entered as the continuation link.

ALPHAMERIC CONSTANTS

A label is generated for each occurrence of an alphameric constant as an actual argument of a subroutine call. This label is entered into the symbol table, in a manner analogous to a format label. The entrance constitutes both the definition and the reference of this label. The storage class is set to 3, and the current value of the class-3 location counter is entered into the label symbol table entry. The location counter is incremented by the size of the literal constant. An alphameric entry is made in the alphameric table (see "Format Processing").

Literal constants occurring as preset data are processed in the same manner as numeric constants occurring as preset data.

DATA PROCESSING

Each DATA statement and each data specification within a type statement produces a data entry in the preset data reference set transmitted to Phase 5. Each data entry consists of a variable element, one or more

value elements, and a continuation or terminal element. The variable elements have a pointer to the variable symbol table entry. The variable elements within a data entry are linked together. The continuation element links the data entries together. The address of the first data entry is in the intercom region.

CROSS REFERENCE INDEX LIST

If the user has selected the cross reference option, each occurrence of a statement number or variable identifier in a program causes an entry to be made in the cross reference index list. Each entry in this list consists of a symbol table pointer to the element name or label value, the line number of the occurrence, and an indicator. The indicator specifies that the occurrence is an assignment or a definition, rather than the usage of or reference to the element.

A variable identifier entry is marked as assigned when it occurs, as follows:

1. To the left of the equal sign in an assignment statement.
2. To the left of the equal sign in a DO statement.
3. In an ASSIGN statement.
4. As an element of an input list.
5. As an element of a NAMELIST referenced by a READ statement.

Statement number entries are marked as defined when the label PRF entry is made. All other occurrences of elements are usage or reference entries.

PHASE 1 ROUTINES, FUNCTIONAL DESCRIPTION

Phase 1 routines can be grouped according to the function they perform. A brief description of the function of each group and the routines belonging in each group follow.

Pass 1 Statement Processors

These modules control the analysis and encoding of each of the various FORTRAN source statements. The modules are EQUA, EXTE, GOTO, IF, TYPE, CONT, DIMN, COMM, EQUI, DO, ASSI, FCON, RWIO, FORM, PSR, NAML, BLDA, DATA, IMPL, BLNK, SUBE, CALL, and END.

Pass 2 Statement Processors

Due to the conversational nature of the compiler, certain operations pertaining to the processing of a statement are best delayed until it is sure the statement will not be deleted. These modules perform the final encoding and housekeeping operations for each of the various FORTRAN source statements. The modules are DCL2, EXEC2, BLDA2, IMPL2, SUBE2, CALL2, and STFN2.

Expression Processing and Translation

These routines perform the analysis and encoding of arithmetic and logical expressions wherever they may occur. Two of these routines are devoted exclusively to subscript processing. They are SUBS and TRMPRO. The other routines are EXPR, CNVRT, SFDEF, SFEXP, FNCLS, LIBN, ARITH, AARG, and CHKINT.

Source Extraction and Conversion

These routines perform the character-by-character source analysis of the basic language elements (variables, constants, and labels) and any conversions required. They also file these elements in the symbol table as required. The routines are ESC, ACOMP, FLRC, IVST, ICNV, FCNV, and FLIC.

Loop Processing Service Routines

The routines that perform the analysis and encoding of loops whenever they occur are BGNLP, ENDLP, CKLIM, and CLLIM.

I/O Statement Processor Service Routines

These routines perform analysis and encoding of parts of I/O statements for RWIO. The routines are IOLST, FLABL, RTRAN, and FNAME.

Initial Value Processing Service Routines

These routines analyze and encode the initial values occurring in the explicit type and DATA statements; they are IDATA and IVAL.

Miscellaneous Service Routines

There are a number of routines that perform specific functions as required by various statement processors and other routines. These routines and their functions are as follows:

ARDIM - analyze and encode the dimension specifications for an array when encountered in a dimension, common or type statement.

LBSTR - process the label string as encountered in the assigned and computed GO TO statement.

SID - classify each source statement and assign its ID number.

LABL - encode statement labels and determine if any loops are ended.

FALTH - determine if a statement number reference was to the next sequential statement and mark the reference for possible later optimization.

ERR - generate a diagnostic message and add it to the output data set.

ROUTINE DESCRIPTIONS

Phase 1 routines bear mnemonic titles as well as coded labels. The five-character coded labels begin with the letters CEK; the fourth and fifth letters identify a specific routine. Various entry points to a routine are identified by a sixth character appended to the coded label. Any mnemonic name beginning with the letters TEV refers to an Executive routine or entry point, rather than to a Phase 1 routine. The corresponding coded label is given in parentheses immediately following the mnemonic.

There are no hardware configuration requirements for any of the Phase 1 routines. All these routines are reentrant, nonresident, nonprivileged, and closed. Except for entry to the Constant Arithmetic Interrupt routine (CEKCS), which uses standard linkage, all entries must be by restricted linkage conventions. Each Phase 1 routine has only one exit; there are no special exits for error conditions.

Phase 1 is composed of 65 routines. The relationships of these routines are shown in the following nesting chart (Figure 15) and decision table (Table 10). The relationships are shown in terms of levels; a called routine is considered to be one level lower than the calling routine. The nesting chart is drawn to show only linkages to the fourth level. Phase 1 main loop is considered to be level 1.

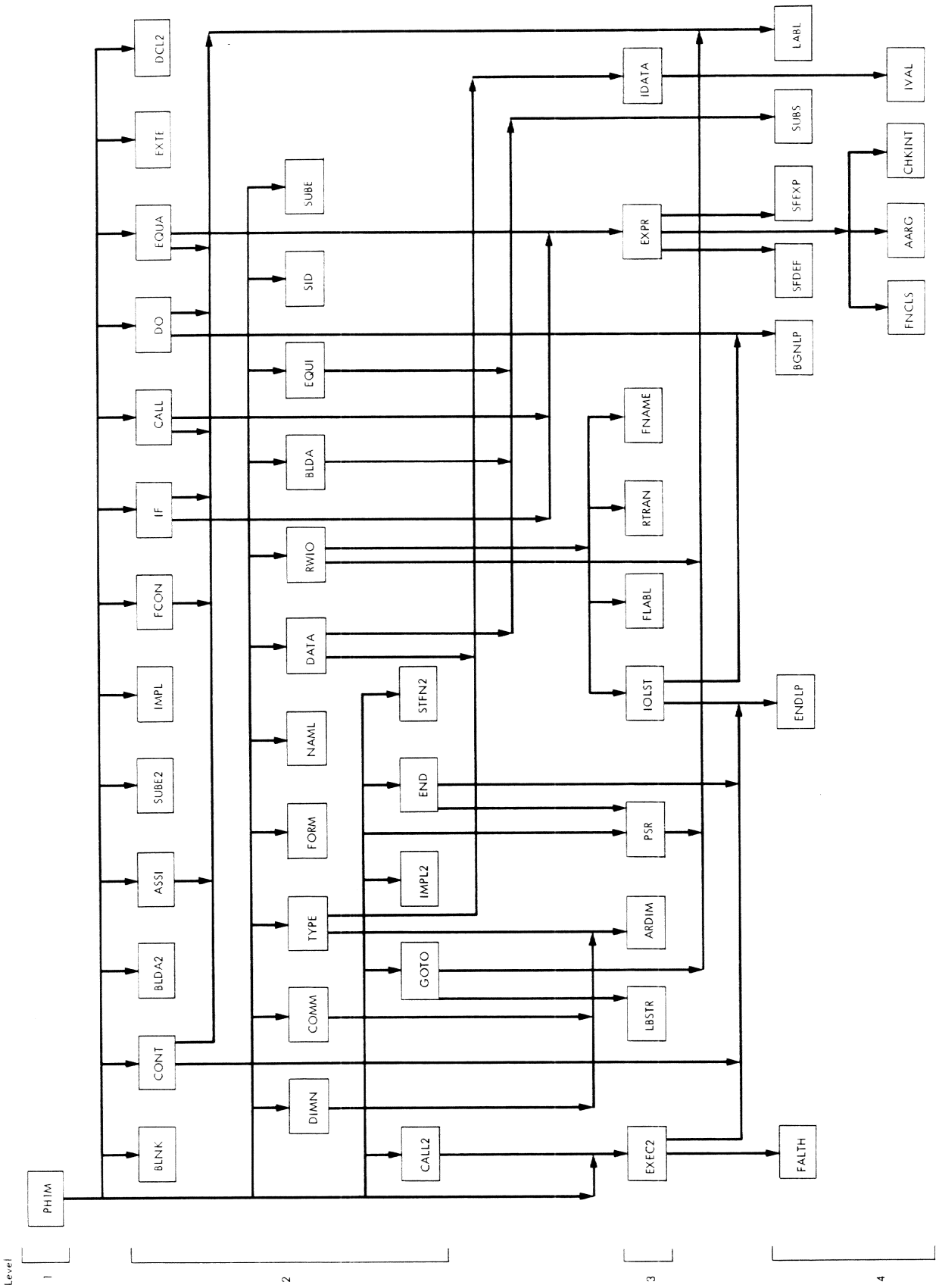


Figure 15. Phase 1 Nesting Chart

Table 10. Phase 1 Decision Table (Part 1 of 8)

Routine:----------Phase 1----------Level: 1----------

Routine	Usage	Called Routines	Calling Conditions
PH1M	Phase 1 Main Loop	SID	To identify the type of source statement.
		EQUA	To process logical and arithmetic assignment statements.
		EXTE	To process EXTERNAL statements.
		GOTO	To process GO TO statements.
		IF	To process Arithmetic and Logical IF statements.
		TYPE	To process type declaration statements.
		CONT	To process CONTINUE statements.
		DIMN	To process DIMENSION statements.
		COMM	To process COMMON statements.
		EQUI	To process EQUIVALENCE statements.
		DO	To process DO statements.
		ASSI	To process ASSIGN statements.
		FCON	To process BACKSPACE, END FILE, and REWIND statements.
		RWIO	To process READ, WRITE, PRINT, and PUNCH statements.
		FORM	To process FORMAT statements.
		PSR	To process PAUSE, STOP, and RETURN statements.
		NAML	To process NAMELIST statements.
		BLDA	To process BLOCK DATA statements.
		DATA	To process DATA statements.
		IMPL	To process IMPLICIT statements.
		BLNK	To process blank source statements.
		SUBE	To process ENTRY, FUNCTION, and SUBROUTINE statements.
		CALL	To process CALL statements.
		END	To process END statements.
		DCL2	To terminate processing of various declaration statements.
		EXEC2	To terminate processing of executable statements.
		BLDA2	To set program type for BLOCK DATA statements.
		IMPL2	To perform final housekeeping for IMPLICIT statements.
		SUBE2	To make PRF entries for ENTRY, FUNCTION, and SUBROUTINE statements.
		CALL2	To adjust the CALL PRF entry.
		STFN2	To terminate processing of Statement Functions.
		ESC	To obtain next source character.
		IVST	To make Symbol Table entry for alphameric names.
		ERR	To generate diagnostic messages.
		TEVGNS (CEKTC)	To get next source statement.

Table 10. Phase 1 Decision Table (Part 2 of 8)

Routine:----------Phase 1----------Level: 2----------

Routine	Usage	Called Routines	Calling Conditions
SID	Source statement identification	ESC	To obtain next source character.
		ERR	To generate diagnostic messages.
EQUA	Equation statement processor	LABL	To process statement label.
		EXPR	To translate source language expressions into Polish Notation.
		ERR	To generate diagnostic messages.
EXTE	EXTERNAL statement processor	ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic messages.
GOTO	GO TO statement processor	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		LABL	To process statement labels.
		LBSTR	To process a string of labels.
		ERR	To generate diagnostic messages.
IF	IF statement processor	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		EXPR	To translate source language expressions into Polish Notation.
		LABL	To process statement labels.
		ERR	To generate diagnostic messages.
		TEVCRL (CEKTFM)	Exec routine that creates a label for a code file.
TYPE	Explicit type statement processor	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		ARDIM	To process dimension specifications for an array.
		IDATA	To process initial value data for type and DATA statements.
		ERR	To generate diagnostic messages.
CONT	CONTINUE statement processor	ESC	To obtain next source character.
		LABL	To process statement labels.
		ERR	To generate diagnostic messages.
DIMM	DIMENSION statement processor	ACOMP	To assemble source characters into basic components.
		ARDIM	To process dimension specifications for an array.
		ERR	To generate diagnostic messages.
COMM	COMMON statement processor	ACOMP	To assemble source characters into basic components.
		ARDIM	To process dimension specifications for an array.
		ERR	To generate diagnostic messages.
EQUI	EQUIVALENCE statement processor	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		SUBS	To translate subscript expressions into Polish Notation.
		ERR	To generate diagnostic messages.

Table 10. Phase 1 Decision Table (Part 3 of 8)

Routine:----------Phase 1----------Level: 2--(Cont'd)--

Routine	Usage	Called Routines	Calling Conditions
DO	DO statement processor	ACOMP	To assemble source characters into basic components.
		LABL	To process statement labels.
		BGNLP	To process Begin Loop information.
		ERR	To generate diagnostic messages.
ASSI	ASSIGN statement processor	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		LABL	To process statement labels.
		ERR	To generate diagnostic messages.
		TEVFLL (CEKTFL)	Exec routine that makes Symbol Table entry for created label.
FCON	File control statements processor (BACKSPACE, END, FILE, REWIND)	ACOMP	To assemble source characters into basic components.
		LABL	To process statement labels.
		ERR	To generate diagnostic messages.
		IVST	To make Symbol Table entries for alphameric names.
RWIO	I/O statements processor (READ, WRITE, PRINT, PUNCH)	ACOMP	To assemble source characters into basic components.
		LABL	To process statement labels.
		IOLST	To process list elements for READ, WRITE, PRINT, and PUNCH statements.
		ERR	To generate diagnostic statements.
		FLABL	To process FORMAT statements.
		RTRAN	To process ERR and END labels.
		FNAME	To process variable FORMAT designators or NAMELIST names.
		IVST	To make Symbol Table entries for alphameric names.
		TEVI4 (CEKTFC)	Exec routine that files an Integer *4 constant.
FORM	FORMAT statement processor	ESC	To obtain next source character.
		ERR	To generate diagnostic messages.
		TEVFLL (CEKTFL)	Exec routine that makes Symbol Table entries for created labels.
NAML	NAMELIST statement	ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic messages.
BLDA	BLOCK DATA statement processor	ESC	To obtain next source character.
		ERR	To generate diagnostic messages.
DATA	DATA statement processor	ACOMP	To assemble source characters into basic components.
		SUBS	To translate subscript expressions into Polish Notation.
		IDATA	To process initial value data for type and DATA statements.
		ERR	To generate diagnostic messages.
IMPL	IMPLICIT statement processor	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic messages.

Table 10. Phase 1 Decision Table (Part 4 of 8)

Routine:----------Phase 1----------Level: 2--(Cont'd)--

Routine	Usage	Called Routines	Calling Conditions
BLNK	Blank statement processor	ERR	To generate diagnostic messages.
SUBE	Subprogram entry statements processor (ENTRY, FUNCTION, SUBROUTINE)	ACOMP ERR TEVCRL (CEKTFM)	To assemble source characters into basic components. To generate diagnostic messages. Execute routine that creates a label for the code file.
CALL	CALL statement processor	LABL EXPR	To process statement labels. To translate source language expressions into Polish Notation.
END	END statement processor	ENDLP PSR ERR	To encode the End Loop entries. To process PAUSE, STOP, and RETURN statements. To generate diagnostic messages.
DCL2	Declaration statements final processing	none	
BLDA2	BLOCK DATA statement final processing	none	
IMPL2	IMPLICIT statement final processing	none	
SUBE2	Subprogram entry statements final processing	none	
CALL2	CALL statement final processing	EXEC2	To terminate processing of executable statements.
STFN2	Statement function statement final processing	none	

Table 10. Phase 1 Decision Table (Part 5 of 8)

Routine:----------Phase 1----------Level: 3----------

Routine	Usage	Called Routines	Calling Conditions
PSR	PAUSE, STOP, RETURN statement processor	ACOMP	To assemble source characters into basic components.
		LABL	To process statement labels.
		ERR	To generate diagnostic messages.
		TEVCRL (CEKTFM)	Exec routine that creates a label for the code file.
EXEC2	Executable statements final processing	ENDLP	To encode the End Loop entries.
		FALTH	To check for references to current label.
EXPR	Process expression	ACOMP	To assemble source character into basic components.
		SUBS	To translate subscript expressions into Polish Notation.
		CNVRT	To convert constants to new type.
		FNCLS	To determine proper class of a function.
		LIBN	To select appropriate Library Function name.
		SFDEF	To make entries in the Statement Function Expression File.
		SFEXP	To make entries in the Expression File.
		AARG	To make Argument Definition entries in the PRF.
		ERR	To generate diagnostic messages.
		CHKINT	To treat floating point overflow and divide checks.
ARDIM	Process array dimension specification	ESC	To obtain next source character.
		ACOMP	To assemble source character into basic components.
		ERR	To generate diagnostic message.
IDATA	Process initial data specifications	ERR	To generate diagnostic message.
		IVAL	To process constants as initial values in type or DATA statements.
IOLST	Process I/O statement list	ESC	To obtain next source character.
		ACOMP	To assemble source character into basic components.
		SUBS	To translate subscript expressions into Polish Notation.
		BGNLP	To process Begin Loop information.
		ENDLP	To encode End Loop entries.
		ERR	To generate diagnostic messages.
		IVST	To make Symbol Table entries for alphameric names.
FLABL	Process FORMAT statement number in I/O statement	ERR	To generate diagnostic message.
		TEVFLL (CEKTFL)	Exec routine that makes Symbol Table entry for created label.
RTRAN	Process END and ERR statement numbers in READ statements	ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic message.
FNAME	Process FORMAT and NAMELIST name in I/O statements	ERR	To generate diagnostic message.
LBSTR	Process label string in Assigned and Computed GO TO statements	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic message.

Table 10. Phase 1 Decision Table (Part 6 of 8)

Routine:----------Phase 1----------Level: 4----------

Routine	Usage	Called Routines	Calling Conditions
SUBS	Process subscripts	ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic message.
		TEMPRO	To process a tentative subscript term prepared by SUBS.
		TEVI4 (CEKTFC)	Exec routine that files an Integer*4 constant.
LABL	Process statement number	ERR	To generate diagnostic message.
		TEVCRL (CEKTFM)	Exec routine that creates a label for the code file.
BGNLP	Process and generate Begin Loop elements	ACOMP	To assemble source characters into basic components.
		CKLIM	To check loop parameters for validity.
		ERR	To generate diagnostic message.
		TEVCRL (CEKTFM)	Exec routine that creates a label for the code file.
ENDLP	Generate End Loop	CLLIM	To remove loop parameter information from Symbol Table.
FALTH	Determine fall-through on GO TO and IF statements.	ERR	To generate diagnostic message.
SFDEF	Initialize for statement function definition	ESC	To obtain next source character.
		ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic message.
SFEXP	Expand Statement Function reference	ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic messages.
FNCLS	Classify function name	none	
IVAL	Process initial values in DATA or type statements	ESC	To obtain next source characters.
		ACOMP	To assemble source characters into basic components
		CNVRT	To convert constants to new type.
		ERR	To generate diagnostic message.
AARG	Process function argument	none	
CHKINT	Check for arithmetic interrupt during expression processing	none	

Table 10. Phase 1 Decision Table (Part 7 of 8)

Routine:----------Phase 1----------Level: 5----------

Routine	Usage	Called Routines	Calling Conditions
ACOMP	Assemble component (operand-operator pair)	ESC	To obtain next source character.
		FLRC	To file real and complex constants in Symbol Table.
		IVST	To make Symbol Table entries for alphameric names.
		ICNV	To convert a decimal integer to a binary integer.
		FLIC	To file integer constants in the Symbol Table.
		ERR	To generate diagnostic message.
		TEVCRL (CEKTFM)	Exec routine that creates a label for the code file.
CNVRT	Checks types and converts constants	LIBN	To select appropriate Library Function name.
		ARITH	To perform all constant arithmetic.
		ERR	To generate diagnostic message.
TEMPRO	Process subscript term	ERR	To generate diagnostic message.
CKLIM	Check loop parameters for correctness and validity	ACOMP	To assemble source characters into basic components.
		ERR	To generate diagnostic message.
CLLIM	Clear flags on loop parameters at End Loop	none	

Routine:----------Phase 1----------Level: 6----------

ESC	Extract source character	none	
LIBN	Select Library Function name	IVST	To make Symbol Table entries for alphameric names.
		ERR	To generate diagnostic message.
FLRC	File real constant in Symbol Table	FCNV	To convert a decimal constant to floating binary.
		ERR	To generate diagnostic message.
		TEVR4 (CEKTFD)	Exec routine to file a Real*4 constant.
		TEVR8 (CEKTFE)	Exec routine to file a Real*8 constant.
		TEVC8 (CEKTFF)	Exec routine to file a Complex*8 constant.
		TEVC16 (CEKTFG)	Exec routine to file a Complex*16 constant.
FLIC	File integer constant in Symbol Table.	ICNV	To convert a decimal integer to a binary integer.
		ERR	To generate diagnostic message.
		TEVI4 (CEKTFC)	Exec routine to file an Integer*4 constant.
ARITH	Perform constant arithmetic during expression scan	ERR	To generate diagnostic message.
		CHCBGA	
		CHCBKC	
		CHCBIA	FORTRAN Math Library exponentiation routines.
		CHCBKA	
		CHCBMC	

Table 10. Phase 1 Decision Table (Part 8 of 8)

Routine:----------Phase 1----------Level: 7----------

Routine	Usage	Called Routines	Calling Conditions
IVST	File variable name in Symbol Table	ERR	To generate diagnostic message.
FCNV	Convert floating-point number from decimal to binary.	ICNV ERR	To convert a decimal integer to a binary integer. To generate diagnostic message.

Routine:----------Phase 1----------Level: 8----------

ICNV	Convert integer from decimal to binary.	none	
ERR	Generate diagnostic message	TEVRDM (CEKTE)	Exec routine that issues a diagnostic message.

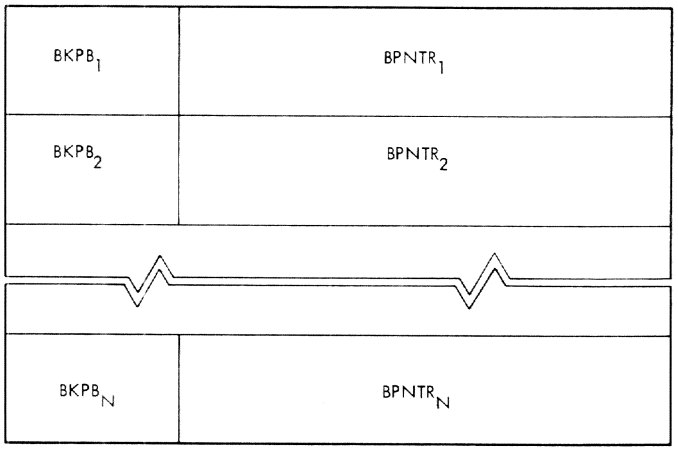
CEKAD -- Phase 1 Main Loop (PH1M)

PH1M controls the identification, analysis, and encoding of source data in Phase 1. See Chart AM.

ENTRIES: PH1M has one entry point CEKAD1. Exec intercom base is expected in parameter register P2.

EXIT: No output parameters.

OPERATION: PH1M performs all initialization for Phase 1. This includes generation of a begin program PRF item, followed by the begin loop PRF items for the false loop. Following initialization, a source statement is read, identified, analyzed, and encoded by calling appropriate subroutines. At this point, the next source statement is read, and the forget and delete flags are tested. If either the forget flag (set by GNSS) or the delete flag (set by any of the statement processing subroutines) is raised, the previously encoded statement is deleted. The statement deletion is accomplished by resetting appropriate items in intercom from their respective backup values. These backups are set for each statement prior to statement processing. The symbol table is restored for variable items, through use of a symbol table save area (Figure 16). Backups for all variable symbol table entries except the NAME, DPP, LINK, TYPE, and LINKF items are entered into the save area, and if deletion is required, these backups are used to restore the symbol table entries. After deletion, the next statement is processed.



BKPB - Backup Byte for Symbol Table change
BPNTR - Byte Pointer into Symbol Table for BKPB

Figure 16. Symbol Table Save Area

If the statement is not deleted, the appropriate subroutine is called to complete the processing for that statement. All tables which may have been updated are then checked for overflow. If no overflow occurred and the statement just processed was not an END statement, control is transferred to the beginning of the loop, to process the next statement.

CEKAK -- Assignment Statement Processor (EQUA)

EQUA analyzes and encodes logical and arithmetic assignment statements. See Chart AN.

ENTRIES: EQUA has one entry point (CEKAK1) and no input parameters.

EXIT: No output parameters.

OPERATION: EQUA generates an equation PRF entry and than calls the Expression Scan subroutine. If the expression is a statement function definition, the PRF entry is deleted, and the statement ID number is changed from assignment to statement function.

CEKAM -- EXTERNAL Statement Processor (EXTE)

EXTE analyzes and encodes the EXTERNAL statement. See Chart AO.

ENTRIES: EXTE has one entry point (CEKAMA) and no input parameters.

EXIT: No output parameters.

OPERATION: EXTE checks to see that the statement is not in a BLOCK DATA program and is not the conditional statement of a logical IF. If not, the statement is scanned, and the variables listed are marked as "external function" in the symbol table. If the statement is in a BLOCK DATA program or is the conditional statement of a logical IF, an error message is produced and the scan is terminated.

CEKAQ -- GO TO Statement Processor (GOTO)

GOTO analyzes and encodes all forms of the GO TO statement. See Chart AP.

ENTRIES: GOTO has one entry point (CEKAQA), with no input parameters

EXIT: No output parameters.

OPERATION: After calling the Label Processing routine, GOTO determines whether the statement is an unconditional GO TO, assigned GO TO, or computed GO TO. In each case, the appropriate PRF entry is made. If an unconditional GO TO is the conditional statement of a logical IF, the sign of the ERF entry for the logical IF is changed, and the GO TO label value is inserted as the true transfer label in the logical IF PRF entry. For the assigned and computed GO TO, internal subroutine LBSTR is called to process the label list into the PRF entry.

CEKAR -- IF Statement Processor (IF)

IF analyzes and encodes the arithmetic and logical IF statements. See Chart AQ.

ENTRIES: IF has one entry point (CEKARA), with no input parameters.

EXIT: No output parameters.

OPERATION: After calling the label processing routine, IF generates an arithmetic IF PRF entry. It then calls upon the Expression Processing routine to analyze and encode the conditional expression. If the expression type is logical, the logical-IF indicator is set and the PRF entry ID is changed to logical IF. A non-source label is created and entered as the "true" transfer label in the PRF entry. If the expression type is arithmetic, the three transfer labels are entered in the PRF entry.

CEKAS -- Type Statements Processor (TYPE)

TYPE analyzes and encodes the type statements, including INTEGER, REAL, COMPLEX, LOGICAL, and DOUBLE PRECISION. See Chart AR.

ENTRIES: TYPE has five entry points, each of which requires no input parameters. The five entry points are INTE (CEKASI) for the INTEGER statement, REAL (CEKASR) for the REAL statement, COMP (CEKASC) for the COMPLEX statement, LOGL (CEKASL) for the LOGICAL statement, and DOBP (CEKASD) for the DOUBLE PRECISION statement.

EXIT: No output parameters.

OPERATION: A type switch is set to show which type statement is used. Where the statement is not DOUBLE PRECISION, TYPE scans it for a length indication; if there is a length indication, the type switch is adjusted to show the length. TYPE then continues the scan, picking up variables and making entries in the symbol table to specify the variable type. If the variable is dimensioned, TYPE also makes entries in the dimension table to specify array length. If a dimension specification is encountered (indicated by a variable followed by a left parenthesis), the Dimension Scan routine is called to process the dimension. If an initial value specification is encountered (indicated by a slash), the Data Scan routine is called to process the initial values.

CEKAT -- CONTINUE Statement Processor (CONT)

CONT analyzes and encodes the CONTINUE statement.

ENTRIES: CONT has one entry point (CEKAT1) and no input parameters.

EXIT: No output parameters.

OPERATION: CONT calls the label processing subroutine to convert the label, if any, and to see if any loops are ended. The logical IF indicator is tested to see if this is the conditional statement of a log-

ical IF statement, and a warning message is issued if it is. A CONTINUE PRF entry is made, and the statement scanned to see that the remainder of the statement is blank.

CEKAU -- DIMENSION Statement Processor (DIMN)

DIMN analyzes and encodes the DIMENSION statement. See Chart AS.

ENTRIES: DIMN has one entry (CEKAUA), with no input parameters.

EXIT: No output parameters.

OPERATION: DIMN first checks to see that the DIMENSION statement is not a conditional statement of a Logical IF. It then proceeds to scan the statement calling the dimension specification processing routine, to process the dimension values as they are encountered for each variable. Appropriate diagnostics are generated if any source errors or incongruities are encountered.

CEKAV -- COMMON Statement Processor (COMM)

COMM analyzes and encodes the COMMON statement. See Chart AT

ENTRIES: COMM has one entry point (CEKAV1) with no input parameters.

EXIT: No output parameters.

OPERATION: COMM first checks to see that the COMMON statement is not a conditional statement of a logical IF. If this is the case, it then opens the common list entry and begins the scan of the statement. Components are acquired with the assemble components routine, and variables are entered into the common list. The symbol table entry common flag is raised, and, if the variable is followed by a left parenthesis, the array dimension specification processor routine is called to process the dimension values.

Variables enclosed in slashes initiate a search of the storage class table for named COMMON blocks, and if any are found, the storage class is appropriately set. Otherwise, the name is entered as a named COMMON block, and a new storage class is established. If there are two slashes without an intervening variable, the storage class will be set to 9 for blank COMMON.

Appropriate diagnostics are generated if any source errors or incongruities are encountered.

CEKAY -- EQUIVALENCE Statement Processor (EQUI)

EQUI performs the analysis and encoding for the EQUIVALENCE statement. See Chart AU.

ENTRIES: EQUI has one entry point (CEKAYA) and no input parameters.

EXIT: No output parameters.

OPERATION: EQUI determines that the statement is not the conditional statement of a logical IF. If this is the case, the heading information for the equivalence table is entered into the storage specification table. Source elements are acquired with ACOMP and analyzed for syntactical correctness. Variables are entered into the equivalence table as they are encountered, and subroutine SUBS is called to determined any offsets indicated by a left parenthesis following a variable. If an offset cannot be completed because the dimension information (TYPE, COMMON, or DIMENSION statement) has not yet been specified for an equivalence variable, the actual subscripts are stored in the Storage Specification List.

Appropriate diagnostics are generated if any source errors or incongruities are encountered.

CEKAZ -- DO Statement Processor (DO)

DO analyzes and encodes the DO statement. See Chart AV.

ENTRIES: DO has one entry point (CEKAZ1) and no input parameters.

EXIT: No output parameters.

OPERATION: DO determines that the statement is not the conditional statement of a logical IF. If it is not, A COMP is called to acquire the label for the end loop. If the label value is satisfactory, BGNLP is called to process the loop variable, range, and increment. Appropriate diagnostics are generated if any source errors or incongruities are encountered.

CEKBC -- ASSIGN Statement Processor (ASSI)

ASSI analyzes and encodes the ASSIGN statement. See Chart AW.

ENTRIES: ASSI has one entry point (CEKBCA) and no input parameters.

EXIT: No output parameters.

OPERATION: ASSI generates a PRF entry for the ASSIGN statement and then scans the source characters. ACOMP is called to acquire the assigned label and the vari-

able. The intervening characters "TO" are checked individually after calls on ESC.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKBD -- File Control Statement Processor (FCON)

FCON analyzes and encodes the BACKSPACE, END FILE, and REWIND statements. See Chart AX.

ENTRIES: FCON has three entry points: BKSP (CEKBD1), ENDF (CEKBD2), and REWI (CEKBD3) for the BACKSPACE, END FILE, and REWIND statements, respectively. FCON has no input parameters.

EXIT: No output parameters.

OPERATION: FCON has three entry points and sets a switch to one of three values, depending upon which entry was taken. A PRF entry is generated and the switch setting entered in that entry, to indicate whether the source statement was BACKSPACE, END FILE, or REWIND. ACOMP is called to acquire the unit number, which is entered into the PRF entry. The I/O initialization library routine entry name (CHCIA1) is filed in the symbol table and marked as class external.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKBE -- Input/Output Statement Processor (RWIO)

RWIO analyzes and encodes the READ, WRITE, PRINT, and PUNCH statements. See Chart AY.

ENTRIES: RWIO has four entry points: READ (CEKBE1), WRIT (CEKBE2), PRNT (CEKBE3), and PUNC (CEKBE4), for the READ, WRITE, PRINT, and PUNCH statements, respectively. RWIO has no input parameters.

EXIT: No output parameters.

OPERATION: RWIO has four entry points. Each entry point generates a PRF entry corresponding to the type of source statement. For the READ statement, RWIO first determines whether or not it is a READ without unit statement. For all statements, the Assemble Components routine is called to acquire statement components as required. If no FORMAT reference is given, the FORMAT pointer in the PRF entry is set to X'8000'.

If a NAMELIST reference is given in place of a FORMAT reference, the PRF ENTRY ID is changed accordingly. For the READ statement, END and ERR condition transfer options are checked and entered into the PRF if present. If they are not given, the statement number items in the PRF are set to zero. Subroutine IOLST is called to process the list elements if required. The I/O Initialization Library routine's entry name (CHCIA1) is filed in the symbol table and marked as class external.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKBF -- FORMAT Statement Processor (FORM)

FORM analyzes and encodes the FORMAT statement. See Chart AZ.

ENTRIES: FORM has two entry points: CEKBF1, for Phase 1 FORMAT statement processing, and SYSPFMT, for FORTRAN I/O-time FORMAT statement processing. CEKBF1 has no input parameters; SYSPFMT has the following input parameters:

P2 -- FIO Translate Table
P3 -- Address of FORMAT statement
P4 -- FORMAT table output area

EXITS: Only the normal exit is made, with no output parameters.

OPERATION: FORM begins by determining that the statement is not the conditional statement of a logical IF statement and not inside a BLOCK DATA program. If this is the case, the statement label is then converted to its binary value and filed in the symbol table (see Table 11).

The FORMAT table is initialized in the Preset Data area, and encoding of the FORMAT statement begins.

Table 11. Encoding of FORMAT Symbols

Character	ID Code
0-9	1
A,I,L,Z	2
D,E,F	3
G	4
H	5
P	6
T	7
X	8
+,-	9
/	10
(	11
)	12
,	13
.	14
'	15
EOS	16
Other	17

The encoding consists of filling out a FORMAT table (see Table 12), through which the compiler communicates format information to FORTRAN I/O routines. An entry is placed in the table whenever a valid FORMAT statement code is found. In addition, syntax is checked, and diagnostics are issued for errors. FORMAT statement processing continues after diagnostics.

FORM terminates the scan when it finds a level-zero right parenthesis.

CEKBG -- PAUSE, STOP, RETURN Statement Processor (PSR)

PSR analyzes and encodes the PAUSE, STOP, and RETURN statements. See Chart BA.

ENTRIES: PSR has four entry points: PAUS (CEKBG1), STOP (CEKBG2), and RETU (CEKBG3) for the PAUSE, STOP, and RETURN statements, respectively, and ESTOP (CEKBG4) for the call by the END statement processor (END). None of the entry points has input parameters.

Table 12. Translation of Format Codes

FORMAT CODE	FORMAT TABLE ENTRY						
	SIZE (BYTES)	BYTE 0	BYTE 1	BYTE 2	BYTE 3	BYTE 4	BYTE 5
H	STRING LENGTH+2	X'0'	LENGTH	CHARACTER STRING AS MANY BYTES AS NEEDED (MAX 255)			
/	1	X'1'					
X	2	X'2'	REPEAT COUNT				
T	2	X'3'	W				
P	2	X'4'	SCALE FACTOR				
)	2	X'5'	NEST LEVEL				
(	3	X'6'	NEST LEVEL	REPEAT COUNT			
A	3	X'7'	REPEAT COUNT	W-1			
Z	3	X'8'	REPEAT COUNT	W-1			
L	3	X'9'	REPEAT COUNT	W-1			
I	3	X'A'	REPEAT COUNT	W-1			
G	4	X'B'	REPEAT COUNT	W-1	D		
F	4	X'C'	REPEAT COUNT	W-1	D		
D	4	X'D'	REPEAT COUNT	W-1	D		
E	4	X'E'	REPEAT COUNT	W-1	D		
SPECIAL H	6	X'F'	LENGTH	ADDRESS OF CHARACTER STRING			

EXIT: No output parameters.

OPERATION: PSR has four entry points, one each for the PAUSE, STOP, and RETURN statements, and one for a call from the END statement processor (to generate a stop when there is flow into an END statement). A PRF entry is generated for the PAUSE, STOP, and RETURN statements, respectively. An appropriate literal constant is filed for the pause and stop entries and for a return entry in a main program. A call from the END statement processor causes a stop PRF entry to be generated. The pause and stop library routine entry names are filed in the symbol table and marked as class external.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKBH -- NAMELIST Statement Processor (NAML)

NAML analyzes and encodes the NAMELIST statement. See Chart BB.

ENTRIES: NAML has one entry point (CEKBH1) and no input parameters.

EXIT: No output parameters.

OPERATION: NAML first checks to see that the statement is not the conditional statement of a logical IF or in a BLOCK DATA program. NAML then gets the Namelist name, which must be enclosed in slashes. After the Namelist name is checked for correct class; a Namelist table entry is made and the symbol table pointer for each variable in the list is entered into the table. Appropriate diagnostics are printed for any source errors or incongruities encountered.

CEKBI -- BLOCK DATA Statement Processor (BLDA)

BLDA analyzes and encodes the BLOCK DATA statement. See Chart BC.

ENTRIES: BLDA has one entry point (CEKBI1) and no input parameters.

EXIT: No output parameters.

OPERATION: BLDA first determines that the statement is not the conditional statement of a logical IF. If this is the case, BLDA checks the program type code to determine whether it is unknown. If it is, a normal exit is taken; otherwise, BLDA prints a diagnostic and exits.

If the statement is the conditional statement of a logical IF, BLDA prints a diagnostic and exits.

CEKBM -- DATA Statement Processor (DATA)

DATA analyzes and encodes the DATA statement. See Chart BD.

ENTRIES: DATA has one entry point (CEKBM1) and no input parameters.

EXIT: No output parameters.

OPERATION: DATA first checks to see that the statement is not the conditional statement of a logical IF statement. The variables in the statement are then extracted and entered into a parameter list, until a slash is encountered. Subroutine IDATA is called at entry DDATA to process the initial value specifications for the list of variables. The process is repeated until an end of statement or a source error is encountered. Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKBN -- IMPLICIT Statement Processor (IMPL)

IMPL analyzes and encodes the IMPLICIT statement. See Chart BE.

ENTRIES: IMPL has one entry point (CEKBN1) and no input parameters.

EXIT: No output parameters.

OPERATION: IMPL first determines that the statement is not the conditional statement of a logical IF statement. The implicit type table is then copied into a temporary hold area, where it can be modified without destroying the current status of the table. The type specification is extracted from the source statement and identified, and the corresponding type code is established. The letters being typed are then extracted and used as a index to modify the implicit type table in the temporary hold area.

CEKBR -- Blank Statement Processor (BLNK)

BLNK processes a blank source statement.

ENTRIES: BLNK has one entry point (CEKBR1) and no input parameters.

EXIT: No output parameters.

OPERATION: BLNK first checks the logical IF indicator. If it is nonzero, a diagnostic is printed to the effect that no conditional statement is given for a logical IF statement. If the logical IF indicator is zero, the label field is checked to see if it was blank. If so, a normal exit is taken; otherwise, a diagnostic message is printed.

CEKBS -- Subprogram Entry Statements Processor (SUBE)

SUBE analyzes and encodes the ENTRY, FUNCTION, and SUBROUTINE statements. See Chart BF.

ENTRIES: SUBE has two entry points: ENTR (CEKBS1) and FUNC and SUBR (CEKBS2) for the ENTRY FUNCTION, and SUBROUTINE statements, respectively. None of the entry points has input parameters.

EXIT: No output parameters.

OPERATION: SUBE has an entry point for each of the three statements it processes. For the ENTRY statement, the program type is checked to ensure that it is a subprogram. If the no flow flag is down (indicating that the previous executable statement transfers control only to the current statement), a label is created and filed, and the symbol table pointer is entered into the PRF. This is done so that a branch around the ENTRY statement can be generated. The DO level is also checked for zero, to ensure that the ENTRY statement does not occur within a DO loop.

For FUNCTION and SUBROUTINE statements the program type is checked to ensure that it is unknown, thus indicating that no statement except an IMPLICIT statement has preceded it.

The PRF entries for these statements are built in a temporary area, due to their variable length. The entries assembled by this routine are then copied into the PRF as permanent entries during Pass 2, in subroutine SUBE2.

The entry name is acquired and classified. For the FUNCTION statement the type option is processed and coded if given. The dummy arguments are then scanned and entered into the PRF. The symbol table entries for each argument are flagged, and the symbol table pointers are entered into the storage class table.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKBU -- CALL Statement Processor (CALL)

CALL analyzes and encodes the CALL statement.

ENTRIES: CALL has one entry point (CEKBU1) and no output parameters.

EXIT: No output parameters.

OPERATION: CALL first calls LABL to process the statement label, ifone is present. CALL then generates a PRF entry for the CALL statement. Finally, the expression scan routine (EXPR) is called to analyze and encode the subroutine name and the arguments.

CEKAL -- END Statement Processor (END)

END performs the required processing for an END statement. See Chart BG.

ENTRIES: END has one entry point (CEKAL1) and no input parameters.

EXIT: No output parameters.

OPERATION: If the statement is the conditional statement of a logical IF statement, a diagnostic is produced and control is returned to the caller. If the program type is BLOCK DATA, the data flag is checked and control is returned to the caller. For all other conditions the executable flag and the DO loop level are checked. If the DO loop level is nonzero, enough end loop PRF entries are generated to reduce it to zero. Then the end loop for the false loop is generated. If the ISD option is on, the false loop is set to "unsafe." The no flow flag is checked to see if execution flow has been terminated. If it has not, a stop PRF item is generated. Finally, an end program PRF item is generated and control is returned to the caller.

CEKAW -- Declaration Statements, Pass 2 (DCL2)

DCL2 performs the housekeeping operations and terminates the processing for the following declaration statements: COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, NAMELIST, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, REAL, FORMAT, and DATA.

ENTRIES: DCL2 has two entry points, COMM2 (CEKAW1) and DCL2 (CEKAW2), neither of which has an input parameter.

EXIT: No output parameters.

OPERATION: DCL2 sets the program type to "main" if it was unknown. In any case, the implicit flag is set to 1 before returning to the caller. A special entry for the COMMON statement also updates the total number of named COMMON blocks in the storage class table before joining the path for other declaration statements.

CEKAX -- Executable Statements, Pass 2 (EXEC2)

EXEC2 performs the housekeeping operations and terminates the processing for the executable statements. See Chart BH.

ENTRIES: EXEC2 has two entry points, FL2 (CEKAX2) and NF2 (CEKAX1), neither of which requires any input parameters. FL2 is the entry point for the following statements: assignment, ASSIGN, BACKSPACE, CONTINUE, END FILE, PAUSE, PRINT, PUNCH, REWIND, WRITE, DO, READ, and CALL. NF2 is the entry point for the following statements: STOP, RETURN, GO TO, and arithmetic IF.

EXIT: No output parameters.

OPERATION: EXEC2 has two entry points:

1. For statements that do not transfer control to statements other than the ones immediately following them.

2. For statements that do transfer control to statements other than the ones immediately following.

If the logical IF indicator is not on, the entry for the second class (above) raises the no-flow flag, indicating that the next executable statement must have a label or there is a logical flaw in the source program. The remaining operations are common to both entries.

If the executable statement flag is down, it is raised and the program type is checked. If the program type is unknown, it is set to "main" before EXEC2 returns to the caller. If the executable statement flag was up, the fall-through processing routine is called to optimize the code in case fall-through occurs from any statement which causes branching. The logical IF indicator is then tested. If it is on, the created label for the conditional GO TO statement is entered into the PRF. The end loop processing routine is called to test for and process any end loops.

CEKBJ -- BLOCK DATA Statement, Pass 2 (BLDA2)

BLDA2 sets the program type for the BLOCK DATA statement.

ENTRIES: BLDA2 has one entry (CEKBJ1) and no input parameters.

EXIT: No output parameters.

OPERATION: BLDA2 sets the program type code to BLOCK DATA and exits.

CEKBP -- IMPLICIT Statements, Pass 2 (IMPL2)

IMPL2 performs the final housekeeping for the IMPLICIT statement after it is accepted.

ENTRIES: IMPL2 has one entry point (CEKBP1) and no input parameters.

EXIT: No output parameters.

OPERATION: IMPL2 copies the implicit type table back from a temporary hold area where it was updated by IMPL and sets the implicit flag to 2.

CEKBT -- Subprogram Entry Statements, Pass 2 (SUBE2)

SUBE2 sets the program type code and makes the permanent PRF entries for the ENTRY, FUNCTION, and SUBROUTINE statements. See Chart BI.

ENTRIES: SUBE2 has three entry points: ENTR2 (CEKBT1), FUNC2 (CEKBT2), and SUBR2 (CEKBT3); none of which has input parameters.

EXIT: No output parameters.

OPERATION: SUBE2 has a unique entry point for ENTRY, FUNCTION, and SUBROUTINE statements. For the ENTRY statement the number of entry points total is incremented. The FUNCTION and SUBROUTINE statements set the program type code to the appropriate value. The remaining operations are performed for all three of the possible statement entries.

An end loop for the false loop is generated, after which the PRF entry is copied from its temporary area into the PRF. A new begin loop for a false loop is then generated, and the number of alternate returns total is updated.

CEKBV -- CALL Statement, Pass 2 (CALL2)

CALL2 adjusts the CALL PRF entry to insert the statement numbers for the alternate returns. See Chart BJ.

ENTRIES: CALL2 has one entry point (CEKBV1) and no input parameters.

EXIT: No output parameters.

OPERATION: If the count of alternate returns in intercom (TENAR) is zero, a normal return is taken. If the count is nonzero, the PRF entries for the argument definition points and the CALL are moved up by the appropriate number of words. The statement numbers are then inserted in the CALL PRF entry. During the pass through the argument definition point PRF entries, the FDP fields in the symbol table are updated if required. The statement numbers are also entered into the cross reference list.

CEKBZ -- Statement Function Definition, Pass 2 (STFN2)

STFN2 performs the housekeeping operations and terminates the processing for the Statement Function.

ENTRIES: STFN2 has one entry point (CEKBZ1) with no input parameters.

EXIT: No output parameters.

OPERATION: STFN2 restores symbol table class and flag fields of variables which were used as statement function arguments. It then checks the program type and, if it is unknown, sets it to "main." Then STFN2 returns to the caller.

CEKAG -- Subscript Processor (SUBS)

SUBS scans subscripted variables and translates the subscript expressions into the internal language (Polish notation) form. See Chart BK.

ENTRIES: SUBS has one entry point (CEKAG1) with no input parameters.

EXIT: No output parameters.

OPERATION: A subscripted variable has the form:

$$A\ (S_1, S_2, \ldots, S_n)$$

where:

$$S = C_1*V_1 \pm C_2*V_2 \pm \ldots \pm C_n*V_n$$

SUBS expands the subscripts into a single expression of the form:

$$S*L - L + S_2*L*d_1 - L*d_1 + \ldots + S_n*L*d_1*\ldots*d_{n-1} - L*d_1*\ldots*d_{n-1}$$

where:

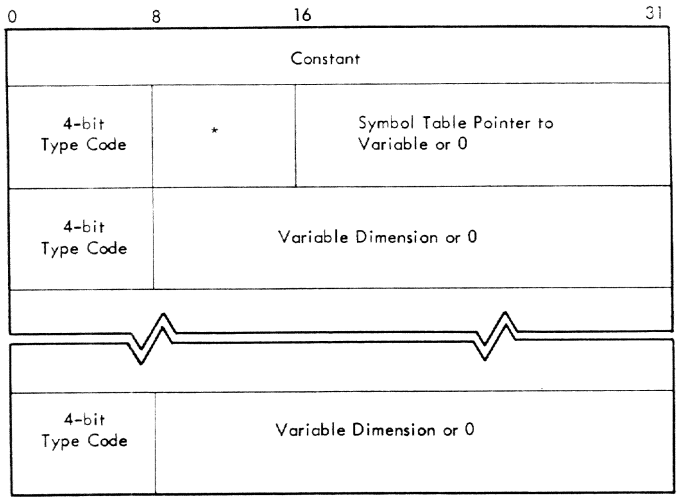
L = length in bytes of an array element
d = ith dimension of N-dimensional array

Constant terms and like variable terms are combined, and the resultant expression is translated into Polish notation for output to the expression representation file (ERF).

SUBS operates in two stages. The first stage scans the source, term by term, and makes up tentative output terms which are stored in an area called TTRM. A subroutine called TRMPRO checks TTRM and adds the contents to a list (TERMS) if it cannot combine the new term with one already in TERMS.

The second stage (PUTOUT) translates the terms of TERMS into Polish notation and puts them in ERF.

Each tentative output term of TTRM has the following format:



* If Induction Variable = ULEV; otherwise = 0

Each entry of a term is one word. The number of entries per term is

NUMDM+1

where:

NUMDM = number of dimensions.

A typical term of subscript S is

$$C*V*L*d_1*\ldots*d_{K-1}$$

The "constant" entry is the product of all the constant factors of the term. The variable entry is the symbol table pointer to V , or to zero if V is missing from term. The variable dimension entries are symbol table pointers to any of the d (i = 1, ---, K-1) that are variable.

Terms of TERMS have the same format as TTRM, except that all nonzero entries are moved to the top of a term.

During the course of processing a subscript, branches are made to NEWTRM, SCAN, and LOOP within the main loop of SUBS.

NEWTRM updates the dimension product (DMPR) with an entry from dimension table, if the entry is a constant: otherwise; NEWTRM enters the symbol table pointer of the variable dimension in TTRM. NEWTRM then branches to SCAN.

SCAN puts DMPR in TTRM and calls subroutine TRMPRO, which adds constant terms to offset of array name entry in ERF or processes variable terms as explained earlier. SCAN then calls ACOMP (assemble component routine) for the next operand-operator pair.

LOOP tests all operators that separate terms in a subscript expression. If a right parenthesis is found, LOOP branches to PUTOUT. If a comma is found, LOOP branches to NEWTRM. If a plus or minus sign is found, TNEG is set accordingly. LOOP then calls ACOMP.

SUBS begins processing by entering the array name in ERF. Various flags and counters are initialized, and a branch is made to SCAN. ACOMP is called, and a subscript term put in TERMS. If the term contains a loop variable, a new term is generated, containing the lower limit of the loop variable in place of the loop variable. A branch is made to LOOP to check the operator. This process is repeated until the loop finds a right parenthesis which signals end of subscript, and a branch is made to PUTOUT.

If the statement ID is DATA or EQUIVALENCE, PUTOUT determines that there are no entries in TERMS. Otherwise, PUTOUT puts the entries from TERMS into the ERF. If a term contains a loop variable, the variable is entered into ERF ahead of the constant. If the statement ID is EQUIVALENCE and dimension information from a TYPE, COMMON or DIMENSION statement had not yet been specified for an equivalence variable, SUBS processes each subscript on an equivalence variable, and saves the subscript and sign, if any, for subsequent processing by EQUI.

CEKAI -- Expression Processor (EXPR)

EXPR translates the source language expression into the internal language (Polish notation) expression. See Chart BL.

ENTRIES: EXPR has one entry point (CEKAI1) with no input parameters.

EXIT: No output parameters.

OPERATION: Subroutine EXPR is the arithmetic and logical expression scanner, and produces in the Expression File (EF) the internal-language equivalent of a FORTRAN IV expression in the source program. EXPR scans expressions on the left and right side of equation statements, the conditional expression in IF statements, and the subroutine name and argument list in CALL statements.

EXPR sees the source language through ACOMP (assemble components) which provides EXPR with an operand-operator pair (component) each time it is called.

The internal-language expression is contained in EF as a string of operators (delimiters) and operands in right-hand Polish notation. An oversimplified statement, then, of EXPR's task is "to transfer operators from their position between their operands to a position following their operands." This implies that, in scanning over the source expression and putting the internal form in EF, an operator must be held back during the scanning and putting out of its second operand (which may be a large expression itself). Two main tables, HSTCK and SXS, are used by the subroutine largely for just this purpose. Each incoming operator is placed in the HSTCK until required for output. SXS contains information about the operands that have been put out.

To oversimplify again, the syntax of algebraic expression requires that operators and operands should alternate, as in "X*Y+Z". This is reflected in EXPR in that the subroutine is always in one of two states, controlled by the condition of a cell HS. The H state means, roughly, that the last item scanned went into the HSTCK (i.e., was an operator) so an operand can be expected next. The S state means that the last item scanned caused an entry in SXS (i.e., was an operand) so an operator can be expected next. Clearly, the scan should begin in the H state and end in the S state.

The situation is, in reality, much complicated by the presence in expressions of unary operators (such as the logical negative .NOT.), function calls, subscripted variables, and parentheses. These complications are best described by examining EXPR's methods for handling them. These methods are variations on, or elaborations of, the basic idea.

The main loop of EXPR begins with a call on ACOMP for the next component of the source-language expression. The characteristics of the next component determine the processing it receives, after which the subroutine returns to the top of the main loop to obtain the next component. Some objects are illegal if received when the subroutine is in the H state; some are illegal in the S state; and, a few are legal in either state, but have their meaning determined by the state during which they arrive. Each object processed sets the state for the next. From this viewpoint, there are four classes of objects: H to S, those that are legal in the H state and leave the subroutine in the S state; H to

H, those that are legal in the H state and leave the subroutine in the H state; S to S, those that are legal in the S state and leave the subroutine in the S state; and S to H, those that are legal in the S state and leave the subroutine in the H state.

The two basic classes most easily handled by EXPR are H to S and S to H. The other two classes handle the more complicated situations involving functions (except functions as arguments), and special operators (i.e., parentheses, unary operators, equal sign, and end of statement).

H to S Class

Constant: When a constant item is received from ACOMP, a constant entry is made in EF and SXS with the appropriate type.

Simple Variable: A simple variable is processed like a constant.

Array Variable: When an array item is received, the next operator is checked for a left parenthesis. If one is not found, the array is treated like a simple variable. If the parenthesis is found, EXPR calls the subroutine SUBS, which processes the subscript and enters the array variable into EF.

Function as Argument: When a function item is received and is not the first item of a CALL statement, it may be a function used as an argument to another subprogram. The item is accepted as such if the next operator is either a comma or a right parenthesis, and the top item in HSTCK is a comma or semicolon (see function call in "H to H Class" below). If the function is subject to automatic typing, it is checked, and the function name changed, if necessary, before outputting to EF and SXS.

S to H Class

This class contains the comma and all the binary operators: arithmetic, relational, and logical. When an item for one of these operators is received by EXPR, it is compared with the top item of HSTCK. If this new item represents an operator of lesser precedence than the top item, the HSTCK operator is output to EF and is appropriately processed. It is then removed from the HSTCK, and the new item is compared with the new HSTCK top item. This process continues until an item of less precedence is brought to the top of the HSTCK (the bottom of the HSTCK will always look like such an item), at which point the new item is added to become the top item of the HSTCK, unless it is a comma or equal sign, in which case it receives special treatment. Comparisons include a check for illegal pairs.

Table 13 explains operator precedence table, called PRECTAB. The operators that appear at the top of each column are the new items that can legally come from ACOMP. The operators that appear at the beginning of each row are the items that can appear at the top of HSTCK. Indexes to the action taken when a new item is compared with a HSTCK item are given as elements of the table. The various actions taken are discussed after the table.

S to S Class

Right parenthesis and end of statement (EOS) are the only items which are received from ACOMP is the S state and leave EXPR in the same state. See Table 13 for further discussion.

H to H Class

Left parenthesis, .NOT., and unary + or - are the only legal operators that can be received from ACOMP in the H state. See Table 13 for further discussion.

Explanation of PRECTAB

DD1:
Illegal operator pair.

DD2:
New operator has greater precedence than HSTCK item. New operator is put in HSTCK.

DD3:
New) meets (. Left parenthesis is deleted from HSTCK.

DD4:
New EOS meets =. This indicates that the right side of the equation statement has been processed. EXPR calls subroutine CNVRT, which uses the last two entries in SXS and EF to check for legal type mix and enters a conversion function in EF so that the expression type on the right side will conform to the variable type on the left side. If expression is a constant, it is converted to variable type.

DD5:
New = meets BOT. This indicates that the variable on the left side of the equation statement has been processed. The equation PRF entry is updated and the variable is linked into VDP chain. The = operator is put in HSTCK.

DD6:
New) meets BOT. This indicates that an IF statement has been processed.

Table 13. Operator Precedence

HSTCK ITEM	NEW OPERATOR +	-	/	*	**	)	,	=	(	EOS	.LT.	.LE.	.EQ.	.NE.	.GE.	.GT.	.NOT.	.AND.	.OR.	UN +	UN -
+	DD13	DD13	DD2	DD2	DD2	DD13	DD13	DD1	DD2	DD13	DD13	DD13	DD13	DD13	DD13	DD13	DD1	DD13	DD13	DD1	DD1
-	14	14	2	2	2	14	14	1	2	14	14	14	14	14	14	14	1	14	14	1	1
/	13	13	13	13	2	13	13	1	2	13	13	13	13	13	13	13	1	13	13	1	1
*	13	13	13	13	2	13	13	1	2	13	13	13	13	13	13	13	1	13	13	1	1
**	13	13	13	13	2	13	13	1	2	13	13	13	13	13	13	13	1	13	13	1	1
,	2	2	2	2	2	11	10	1	2	1	2	2	2	2	2	2	2	2	2	2	2
(	2	2	2	2	2	1	1	1	2	4	2	2	2	2	2	2	2	2	2	2	2
=	2	2	2	2	2	3	1	1	2	1	2	2	2	2	2	2	2	2	2	2	2
BOT	2	2	2	2	2	6	1	5	2	7	2	2	2	2	2	2	2	2	2	2	2
.LT.	2	2	2	2	2	16	16	1	2	16	1	1	1	1	1	1	1	16	16	2	2
.LE.	2	2	2	2	2	18	18	1	2	18	1	1	1	1	1	1	1	18	18	2	2
.EQ.	2	2	2	2	2	16	16	1	2	16	1	1	1	1	1	1	1	16	16	2	2
.NE.	2	2	2	2	2	17	17	1	2	17	1	1	1	1	1	1	1	17	17	2	2
.GE.	2	2	2	2	2	19	19	1	2	19	1	1	1	1	1	1	1	19	19	2	2
.GT.	2	2	2	2	2	16	16	1	2	16	1	1	1	1	1	1	1	16	16	2	2
.NOT.	2	2	2	2	2	21	21	1	2	21	2	2	2	2	2	2	2	21	21	2	2
.AND.	2	2	2	2	2	20	20	1	2	20	2	2	2	2	2	2	2	20	20	2	2
.OR.	2	2	2	2	2	20	20	1	2	20	2	2	2	2	2	2	2	2	20	2	2
UN +	22	22	22	22	2	22	22	1	2	22	2	2	2	2	2	2	1	22	22	1	1
UN -	23	23	23	23	2	23	23	1	2	23	2	2	2	2	2	2	1	23	23	1	1
:	2	2	2	2	2	12	8	1	2	1	2	2	2	2	2	2	2	2	2	2	2
::	2	2	2	2	2	12	9	1	2	1	2	2	2	2	2	2	2	2	2	2	2
SF	2	2	2	2	2	24	24	1	2	1	2	2	2	2	2	2	2	2	2	2	2
MAX	2	2	2	2	2	25	26	1	2	1	1	1	1	1	1	1	1	1	2	2	2

DD7:

New EOS meets BOT. This indicates that a CALL statement has been processed.

DD8 AND DD9:

New , meets ; or ;;. This indicates that the first of several arguments has been processed. The . is put in HSTCK (,, for intrinsic functions). Subroutine AARG is called to determine if argument should be linked into VDP chain. ARG type is put in SXS.

DD10:

New , meets , or ,,. This indicates that the N'th argument of a function call has been processed. Subroutine AARG is called (see DD8) to check argument type, increment the argument count, and output, or ,, to EF.

DD11 and DD12:

New) meets . or ;. This indicates that the last argument has been processed (only one argument in function-call if HSTCK item is ;. Subroutine AARG is called (see DD8 and DD10). The correct number and type of arguments are checked. Functions with class LIBA call subroutine LIBN, which selects a function name based on the argument type. Functions with class OPEN A use table FNUM to select a function number. The function is output to the EF followed by a ; or ;; EF entry.

DD13 through DD23:

New operator has less or equal precedence. This means that the HSTCK item is to be output to EF. The HSTCK item may be one of the following:

> Unary + or - or .NOT. operator.
> Binary arithmetic, relational, or logical operator.

The unary +, -, and .NOT. are the simplest to output. The top item in SXS is their operand and is checked for legal type; then, the last EF entry has its sign changed for unary - and .NOT.

For a binary arithmetic, relational, or a logical operator, the top two entries in SXS represent its operands. In addition to putting an operator item in EF, the processing requires replacing the two operands in SXS with a single entry for the result of the operations.

The types of the operands are checked for legal combinations. The top SXS item is deleted, and the next SXS item is given a subexpression class with the maximum type of the two operands. A relational operator is assigned Logical*4 type.

A binary arithmetic operator is also checked for constant operands. If both SXS operand entries are class constant, the arithmetic called for by this operator will be done on the last two EF entries. The subroutine CNVRT does constant arithmetic and type checking for binary-arithmetic operators.

If the HSTCK item is a -, .NE.,.LE. or .GE., it is changed to a +,.EQ.,. GT. or .LT., respectively, and the sign of the last EF entry is changed. Then the above processing is done.

DD24:

New , or) meets SF. This indicates that an argument of a statement function has been processed. EXPR calls subroutine SFEXP (at entry point SFEXPC) which continues processing the statement function expression.

DD25:

New) meets MAX. This indicates that all arguments have been processed in a MAX/MIN function. The argument type is checked and the last MAX operator is put in EF. The top three bytes of the multiple byte entry for MAX/MIN function are deleted from HSTCK. The top item of HSTCK is now (, unary -, or a conversion function which was entered in HSTCK by the function-call processing.

DD26:

New , meets MAX. This indicates that an argument of MAX/MIN function has been processed. The argument type is compared with the type in HSTCK. If this is first argument processed, and automatic typing is called for, the argument type is put in HSTCK. The comma flag is set, and the sign of last EF entry is changed if the MIN flag is set. The MAX operator is put in EF, except after the first argument.

Function/Subroutine Call

A function call is recognized if one of the following conditions exists:

1. The item is an external, intrinsic, or library function with the next item a left parenthesis. Six bytes are added to HSTCK. Bytes 1 and 2 contain Symbol Table pointer to function entry. Byte 3 contains type of arguments observed. Byte 4 contains number of arguments observed. Byte 5 contains

flag for argument definition PRFs (set to 1 if abnormal function). Byte 6 contains semicolon operator.

For external functions, the ABN flag is set in the PRF entries for IF and equation statements. This flag is used by Phase 2 to find common definition points. EXPR then calls ACOMP for the next component.

2. The item is the first item of a CALL statement. The subroutine flag is raised in the symbol table, and the function is handled the same as case 1 if the next item is a left parenthesis. If the next item is EOS, the function is entered in EF as a no-parameter function.

3. The item's class is "unknown" or "unknown function" and the next item is a left parenthesis. EXPR calls the subroutine FNCLS, which determines the class of the function (OPEN, OPENA, external, LIB, LIBA, or MAX). FNCLS sets the function flag and appropriate class in the symbol table and returns to EXPR. If the function class is external, library or intrinsic the function is processed like case 1. If the function is a member of the MAX-MIN family, it is processed as described in case 4.

4. The item is a member of the MAX-MIN family. Members of the MAX-MIN family require special treatment. They are interpreted not as functions, but in terms of a new operator, MAX, which is like + in that it takes two operands and has its type determined by the type of its operands. MIN is expressed by changing the signs of MAX and its operands. A conversion function is entered in HSTCK per case 1, if needed; otherwise, a left parenthesis is put in HSTCK. Either of these HSTCK items will correctly terminate the MAX function processing after the last argument has been processed. A unary - is then put in HSTCK if function is MIN. This will negate the last MAX operator in EF. The next HSTCK byte contains two flags: a MIN flag (set if MIN function), and a comma flag (set after first argument processed). The next byte contains argument type required by function (FS if automatic typing). The top byte contains the operator MAX. EXPR then calls ACOMP for the next component.

5. The item is a statement function. EXPR calls the subroutine SFEXP (at entry point SFEXPI), which initializes and controls the statement function processing. EXPR's machinery is used to process the statement function arguments.

CEKAN -- Conversion Subroutine (CNVRT)

CNVRT converts constants to new type, if specified, and checks legal type mixes for arithmetic and logical expressions, and across the equal sign in assignment statements. See Chart BM.

ENTRIES: CNVRT has two entry points: CNVRT (CEKAN1), which is called by EXPR to perform all functions mentioned above, and CNVRTD (CEKAN2) which is called by IVAL and is concerned only with converting constants to the type of the variables into which they will be stored.

Input Parameters:

P2 -- Variable Symbol Table Pointer (CNVRTD entry only)
P5 -- HSTCK address (CNVRT entry only)
P6 -- SXS address (CNVRT entry only)

EXIT: P5 contains the HSTCK address, and P6 contains the SXS address.

OPERATION: The types of the top two operands in SXS, SXS(J) and SXS(J+1), are compared by using the table CNVTAB, and appropriate action is taken. The action taken depends on whether the top HSTCK item is = or +, /, * or **.

If the HSTCK item is =, then SXS(J) is the operand on the left side in an assignment statement or a variable of a DATA statement, and SXS(J+1) is converted to the type of SXS(J), if they are different. If SXS(J+1) is a constant, CNVRT converts the constant, and files the new constant in the symbol table and EF. If SXS(J+1) is not a constant, the appropriate conversion function is entered in EF. Symbol table and EF entries are not made for DATA statement.

If the top HSTCK item is an arithmetic operator (except**), the two operands are checked to see if they are constant. If just one operand is constant, it is converted to the maximum type of the two, if different. If both operands are constant, one is converted to the maximum type, if different, and subroutine ARITH is called. It combines the constants according to the HSTCK operator. The new constant is filed in the symbol table and EF. A special case occurs if the operand types are R*8 and C*8. The maximum type in this case is C*16, and all constants are converted to this type.

There are three cases to consider if the HSTCK item is:

1. Both exponent and base are constant.

2. The base is a real or integer variable and the exponent is an integer constant in the range 0 through 16 for integer base, or -16 through +16 for real base.

3. Neither of the above cases.

For case 1, the subroutine ARITH is called and constant arithmetic is performed.

For case 2, a series of one or more special open functions are entered in the EF from a table called EXPF (Table 14). This, in effect, causes the power to be expanded as a series of products of the base multiplied by itself. Another special open function (RECIP) is also entered in EF, to take the reciprocal of the power if the exponent is negative and the base is real.

For case 3, the subroutine LIBN is called (at entry point LIBNX) which selects the appropriate exponential library function. Upon return from LIBN, the function is entered in the EF.

CEKBK -- Statement Function Definition (SFDEF)

SFDEF enables EXPR to translate a statement function expression into Polish notation and to store the translated expression in the statement function expression file (SFEF). See Chart BN.

ENTRIES: SFDEF has one entry point (CEKBK1) with P5 = HSTCK(I) address and P6 = SXS(J) address as input parameters.

EXIT: P5 contains the HSTCK address, and P6 contains the SXS address.

OPERATION: SFDEF scans the argument list and temporarily changes the class and flag fields of all symbol table entries whose names are the same as the dummy arguments. The class is changed to "statement function argument" and the flag field to contain an offset to be used in locating the argument in ARGSTCK (see SFEXP routine). These fields are restored after EXPR finishes scanning the expression. In scanning the argument list, SFDEF checks for legal arguments and maximum number of arguments.

Table 14. EXPF Entries (Real Base)

Exponent	Entries								
2	SQ	; ;	Term.						
3	Cube	; ;	Term.						
4	SQ	; ;	SQ	; ;	Term.				
5	FIFTH	; ;	Term.						
6	SQ	; ;	Cube	; ;	Term.				
7	SEVEN	; ;	Term.						
8	SQ	; ;	SQ	; ;	SQ	; ;	Term.		
9	Cube	; ;	Cube	; ;	Term.				
10	SQ	; ;	FIFTH	; ;	Term.				
11	11	* *	Term.						
12	SQ	; ;	SQ	; ;	Cube	; ;	Term.		
13	13	* *	Term.						
14	SQ	; ;	SEVEN	; ;	Term.				
15	Cube	; ;	FIFTH	; ;	Term.				
16	SQ	; ;	SQ	; ;	SQ	; ;	SQ	; ;	Term.

The base type is attached to semicolon entries.

After completion of argument scanning, SFDEF alters the EF pointer and base, so that EXPR will enter the expression into SFEF. SFDEF then returns to EXPR, which processes the statement function expression. Upon completion of expression processing, the EF pointer and base are restored.

CEKBL -- Statement Function Expansion (SFEXP)

SFEXP inserts a statement function expression into EF when the function is referenced in arithmetic or logical expression, and uses EXPR to process the actual arguments. See Chart BO.

ENTRIES: SFEXP has two entry points: SFEXPI (CEKBL1) and SFEXPC (CEKBL2). The input parameters are P5 = HSTCK(I) address and P6 = SXS(J) address.

EXIT: P5 contains the HSTCK address, and P6 contains the SXS address.

OPERATION: There are two entry points to SPEXP: SFEXPI, which is the entry to the initializing portion, and SFEXPC, which is the entry to the expansion portion.

The initializing part scans the arguments and stores a pointer to the first character of each argument in the source statement. It also stores a pointer (SFEP) to the function expression in SFEF, and a pointer to show EXPR where to resume scanning the source after the statement function has been expanded. These pointers are stored in a portion of the SFEF called ARGSTCK. An "SF" item is entered in HSTCK, which enables EXPR to process the function arguments one at a time. After the initializing is complete, SFEXP begins expanding the function by entering SFEF entries into EF, using SFEP as a pointer. When a statement function argument entry is found, the offset of this of this entry is used to obtain the correct argument pointer from ARGSTCK. This pointer is stored in SOURCE, and SFEXP returns to EXPR, which processes the argument. When a ", " or ")" meets the "SF" item in HSTCK, the argument has been processed and EXPR calls SFEXP (via SFEXPC entry). SFEXP checks the actual argument type with the dummy argument type. If the type is correct, SFEXP resumes transferring SFEF entries to EF until another argument entry is found. This cycle is repeated until an end of expression entry is found in SFEF. This terminates expansion, and SFEXP returns to EXPR with the SOURCE pointer set to scan the remainder of the statement following the statement function reference.

CEKBX -- Function Classifier (FNCLS)

FNCLS determines the proper class of a function whose class was originally "unknown" or "unknown function." See Chart BP.

ENTRIES: FNCLS has one entry point (CEKBX1) and no input parameters.

EXIT: No output parameters.

OPERATION: A function with "unknown" or "unknown function" class is assigned one of the following classes: LIBA, LIB, OPEN, OPENA, MAX, or external.

If the function name is found in the LIBA name list (library function with automatic typing) and its type is not frozen, it is given LIBA class. If its type is frozen, then it is classed external.

If the function name is found in the LIB name list, and its type is not frozen or its type is the same as the library function, then it is classed LIB. If the type is different, it is classed external.

If the function name is not in the LIBA or LIB name lists and its class is unknown function (i.e., delcared in an EXTERNAL statement), it is classed external.

If the function is in the intrinsic function name list (includes OPEN, OPENA, and MAX class functions), and its type is not frozen or its type is the same as the intrinsic function, then the symbol table name part of the function is linked to the intrinsic function descriptive part. If the function type is different, it is classed external.

If the function name is not found in any of the three lists, LIBA, LIB, or intrinsic, it is classed external.

CEKBY -- Library Function Selector (LIBN)

LIBN selects the appropriate library function name, based on the argument type. See Chart BQ.

ENTRIES: There are three entry points: LIBN (CEKBY1), LIBNA (CEKBY2), and LIBNX (CEKBY3). P = SXS(J) address is the input parameter.

EXIT: P6 contains the SXS address.

OPERATION: LIBN has three entry points, LIBN, LIBNA, and LIBNX. LIBN and LIBNA are the entry points for functions with automatic typing. LIBNA is the entry for automatic functions being used as arguments. LIBNX is the entry point for exponential library function selection.

Using the argument type and the function index, the proper function name is selected (see Table 15). The function name is inserted in the symbol table, and the descriptive part entries filled if class is unknown.

Table 15. Library Function Names

Automatic Function Name	Argument Type			
	R*4	R*8	C*8	C*16
EXP	EXP	DEXP	CEXP	CDEXP
LOG	ALOG	DLOG	CLOG	CDLOG
LOGIO	ALOGIO	DLOGIO	CLOGIO	CDLOGIO
ATAN	ATAN	DATAN	0	0
SIN	SIN	DSIN	CSIN	CDSIN
COS	COS	DCOS	CCOS	CDCOS
SQRT	SQRT	DSQRT	CSQRT	CDSQRT
TANH	TANH	DTANH	0	0
Implicit Exponential Functions				
Base Type	Exponent Type			
	I*2	I*4	R*4	R*8
I*2	FJXPJ	FJXPI	FJXPR	FJXPD
I*4	FIXPJ	FIXPI	FIXPR	FIXPD
R*4	FRXPJ	FRXPI	FRXPR	FRXPD
R*8	FDXPJ	FDXPI	FDXPR	FDXPD
C*8	FCXPJ	FCXPI	0	0
C*16	FCDXJ	FCDXI	0	0

CEKCB -- Constant Arithmetic Subroutine (ARITH)

ARITH performs all constant arithmetic. See Chart BR.

ENTRIES: ARITH has one entry point (CEKCB1) with input parameters as follows:

P2, P3 -- Integer Constants
F0, F2 -- Real Constants
F0, F2 -- Complex Constants of type C*8
F0 thru F6 -- Complex Constants of type C*16

EXIT: No output parameters.

OPERATION: If the operator is **, the appropriate FORTRAN library function is called, based on the type of the base and exponent. If the operator is +, *, or /, ARITH does the arithmetic necessary based on operator and operand type.

ARITH may be called upon to perform arithmetic which will cause overflow or divide check exceptions to occur. In order to diagnose these situations properly, system macro SIR is called to enable module CEKCS to trap these interruptions and set appropriate flags. Prior to exit, system macro instruction DIR is called to disable these interruptions.

CEKCG -- Term Processor (TRMPRO)

TRMPRO processes a tenative subscript term prepared by SUBS and either combines it with a previous term or adds it to the TERMS list. See Chart BS.

ENTRIES: TRMPRO has one entry point (CEKCG1), with the address of TTRM and TERMS in P5 as input parameter.

EXIT: There is a single output parameter: the address of TERMS in P5.

OPERATION: If the tentative term has no variable factors, its constant factor is combined with OFFSET. If the tentative term has the same variable factors as a previous term already in TERMS, the terms are combined by adding their constant factors. Otherwise, the tentative term is added to TERMS as a new term.

TRMPRO checks for too large a subscript expression.

CEKCR -- Actual Argument Service Routine (AARG)

AARG performs certain functions in connection with actual arguments of function and subroutine calls. See Chart BT.

ENTRIES: AARG has one entry point (CEKCR1) and two input parameters: the address of last HSTCK entry in P5, and the last SXS entry in P6.

EXIT: P5 contains the address of HSTCK, and P6 contains the address of SXS.

OPERATION: AARG puts an argument definition entry in the PRF for a variable as argument of an abnormal subprogram. If this is not the first argument (comma flag up), the type is checked and a comma (,) or double comma (,,) operator is put out to the EF.

CEKCS -- Constant Arithmetic Interrupt (CHKINT)

CHKINT provides for treatment of interruptions from ARITH and sets flags for issuance of a proper diagnostics. See Chart BU.

ENTRIES: This routine is called by the standard linkage convention. There are three entry points:

CEKCS1	-- Set flag for divide check interruptions
CEKCS2	-- Set flag for exponent overflow interruptions
CEKCS3 (CHKINT)	-- Return flags to caller

Entry CEKCS3 returns the interruption flags to the fields specified in the parameter list, which is one word long and contains the address at which to store the two flag words. No other input or output parameters are used.

EXIT: No output parameters.

OPERATION: The CHXINT routine is called by ARITH to enable and disable CEKS, for fielding of exponent overflow and divide check interruptions during constant arithmetic. Any interruptions due to divide checks or exponent overflow cause the system interruption processor to enter CHKINT at entries CEKCS1 or CEKCS2, where a flag will be set, indicating that an interruption has occurred. On an exponent overflow, the contents of the R1 register in the ISA save area is set to infinity before exiting.

This routine is called at entry CEKCS3 by EXPR after a complete expression has been processed, to see if any of the above interruptions occurred.

CEKAB -- Extract Source Character (ESC)

ESC is used to obtain the next source character. See Chart BV.

ENTRIES: There are two entry points: ESC (CEKAB1) returns the next nonblank source character; ESCB (CEKAB2) returns the next source character, including blanks. One input parameter, the address of the next available character in the source string, is passed by value in parameter register P3. The high-order 24 bits of parameter register P1 are expected to be zeros. This routine uses only registers P1, P2, and P3. The contents of any other registers except the linkage registers are not destroyed.

EXIT: Output parameters are:

1. Original source character, in register P1.
2. Internal code source character, in register P2.
3. Updated source pointer, in register P3.

OPERATION: Input source data stored in the compiler intercom region is transmitted to the requesting routine, one character at a time. As each source character is extracted from the source input data, a translation is made from either EBCDIC or BCD character codes. This translated character set is a dense set value and is used for identification purposes only. The original character set is used for variable names in the symbol table and preset data in the object program. Values of the dense set are as follows:

Character	Dense Code
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15
G	16
H	17
I	18
J	19
K	20
L	21
M	22
N	23
O	24
P	25
Q	26
R	27
S	28
T	29
U	30
V	31
W	32
X	33
Y	34

Z	35
$	36
Blank	37
+	38
-	39
/	40
*	41
(Not Used)	42
)	43
,	44
=	45
(	46
EOS	47
.	48
'	49
(Not Used)	50
&	51
Others	52

CEKAE -- Assemble Components (ACOMP)

ACOMP assembles source characters into basic components for syntactical analysis. See Chart BW.

ENTRIES: ACOMP has one entry point (CEKAE1) with no input parameter.

EXIT: No output parameters.

OPERATION: Each request for next components returns an operand and the operand delimiter. The possible operand types are variable, constant, label, and null. The delimiter may be any of the arithmetic, logical, or relational operators, the right or left parenthesis, the comma, the end-of-statement, or the label terminator delimiter.

The assembled elements are placed in the component storage area of intercom (see Figure 17). Source characters, both original and converted forms, are acquired by request to the extract source routine. The converted internal code is used as an index into the assemble components character table (see Table 16). By branching upon the value derived from the components character table, using the decision table status as the base, the appropriate action may be effected (see Table 17).

As variables are assembled, a symbol table hash index from the variable hash table is derived for use by INVST in filing the name in the symbol table.

As constants are assembled, their type is determined and appropriate filing routines invoked.

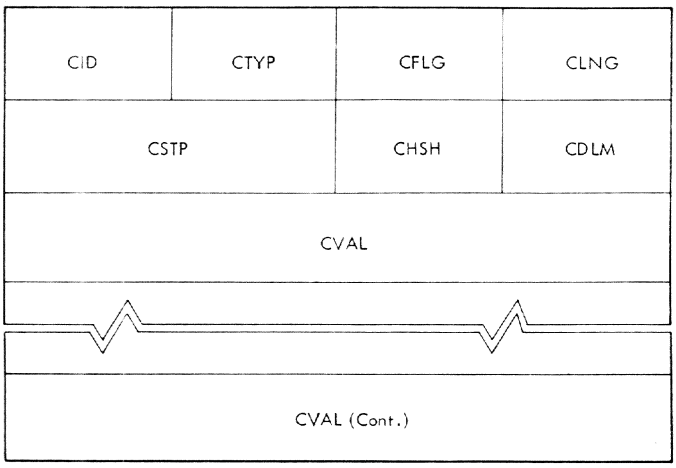


Figure 17. Component Storage Area

Legend for Figure 17

Field	Description
CID	Component ID: Null = 0 Variable = 1 Constant = 2 Label = 3
CTYP	Operand Type Code in Hexadecimal Unknown = 00 Integer *2 = 12 Integer *4 = 32 Real *4 = 33 Real *8 = 73 Complex *8 = 74 Complex *16 = F4 Logical *1 = 01 Logical *4 = 31 Literal = 02
CFLG	Flags
CLNG	Length of CVAL in bytes (maximum = 256)
CSTP	Symbol Table Pointer for Operand
CHSH	Variable Hash Total
CDLM	Component Delimiter Code + = 0 - = 1 / = 2 * = 3 ** = 4) = 5 , = 6 = = 7 (= 8 EOS = 9 .LT. = 10 .LE. = 11 .EQ. = 12 .NE. = 13 .GE. = 14

.GT. = 15
.NOT. = 16
.AND. = 17
.OR. = 18
a = 19 (Label Delimiter)

CVAL Component Operand -- Value or Name

Table 16. Assemble Components Character Table

Internal Code	Component Code
0	1
1-9	2
A-$	0
+	4
-	5
*	6
.	3
'	7
&	8
/ = ( ) , EOS	9
Others	10

CEKCH -- File Real Constant (FLRC)

FLRC files real and complex constants in the symbol table. See Chart BX.

ENTRIES: FLRC has one entry point (CEKCH1) and no input parameters.

EXIT: No output parameters.

OPERATION: FLRC calls FCNV to convert the constant to floating binary. If the constant is not part of a complex constant and the "don't file" flag is down, the constant is filed in the symbol table.

If the constant is the real part of a complex constant, the value is saved and the routine returns to the caller. If the constant is the imaginary part of a complex constant, the real and imaginary parts are combined as a single constant. If the "don't file" flag is down, the real and imaginary parts are also filed in the symbol table.

Appropriate diagnostics are generated if the types of the two parts of a complex constant do not agree. The parts are made to agree with the larger type.

CEKCI -- Insert Variable in Symbol Table (IVST)

IVST finds or makes a symbol table entry for an alphameric name. See Chart BY.

ENTRIES: IVST has one entry point (CEKCI1), with no input parameters.

EXIT: No output parameters.

OPERATION: IVST uses the name hash value to select a chain anchor in the variable hash table. If the chain is not empty, the chain entries are searched for one with the present name. If the chain is empty or an entry is not found, a new entry is made for this name and added to the chain. The symbol table descriptive part pointer for the found or made entry is set in CSTP.

CEKCN -- Decimal to Binary Integer Conversion (ICNV)

ICNV converts a decimal integer to a binary integer. See Chart BZ.

ENTRIES: ICNV has one entry point (CEKCN1) and no input parameters.

EXIT: No output parameters.

OPERATION: ICNV performs the conversion by extracting the digits from left to right, multiplying the intermediate value by 10 for each digit, and adding that digit to the intermediate value. A maximum of 16 digits is allowed, and the result is a doubleword binary integer. The first word of the result is placed in the second word of CVAL, and the second word of the result is placed in the first word of CVAL.

CEKCP -- Decimal to Floating Binary Conversion (FCNV)

FCNV converts a decimal constant to floating binary. See Chart CA.

ENTRIES: FCNV has one entry point (CEKCP1) and no input parameters.

EXIT: No output parameters.

OPERATION: FCNV calls ICNV to convert the decimal digits to a binary integer. This integer is then converted to floating point and normalized. The number is then scaled, to account for the exponential and fractional portions. Appropriate diagnostics are generated if the exponent and magnitude ranges are exceeded.

CEKCQ -- File Integer Constant (FLIC)

FLIC files integer constants in the symbol table.

ENTRIES: FLIC has one entry point (CEKCQ1).

EXIT: No output parameters.

OPERATION: FLIC calls ICNV to convert the constant to integer binary. If the "don't file" flag is down, the integer is then

Table 17. Assemble Components Decision Table

STATUS	Code	A-$	0	1-9	.	+	-	*	'	&	EOS /=(,)	Non FORTRAN
	S \ C	0	1	2	3	4	5	6	7	8	9	10
	0	A	D	F	I	Z	Z	ZZ	AZ	AC	Z	E
	1	B	B	B	U	C	C	CC	E	E	C	E
	2	G	D	F	V	H	H	CC	E	E	H	E
	3	G	FF	FF	V	H	H	CC	E	E	H	E
	4	J	M	M	E	E	E	E	E	E	E	E
	5	K	E	E	L	E	E	E	E	E	E	E
	6	N	MM	MM	S	T	T	CC	E	E	T	E
	7	E	O	O	E	P	Q	E	E	E	E	E
	8	E	O	O	E	E	E	E	E	E	E	E
	9	E	R	R	S	T	T	CC	E	E	T	E
	10	E	E	E	S	T	T	CC	E	E	T	E
	11	X	E	E	E	E	E	E	E	E	E	E
	12	W	M	M	S	T	T	CC	E	E	T	E
	13	E	E	E	AB	Z	Z	ZZ	E	E	Z	E
	14	Y	O	O	E	P	Q	E	E	E	E	E
	15	AY	AY	AY	AY	AY	AY	AY	AX	AY	AY	AY
	16	AT	AT	AT	AT	AT	AT	AU	AT	AT	AT	E
	17	E	E	E	E	AV	AV	AV	E	E	AV	E
	18	E	E	E	AB	Z	Z	ZZ	E	E	Z	E

filed in the symbol table. A diagnostic is generated if the constant exceeds $2^{31}-1$.

CEKBA -- Begin Loop Processor (BGNLP)

BGNLP analyzes and encodes the begin loop information for the DO statement and for implied loops within an I/O list. See Chart CB.

ENTRIES: BGNLP has one entry point (CEKBA1) and no input parameters.

EXIT: No output parameters.

OPERATION: BGNLP begins by making a call on ACOMP to acquire the induction variable for the loop. If the characteristics of the induction variable are satisfactory, flags are set to indicate the special use of this variable for the duration of the loop. Calls are then made on subroutine CKLIM to acquire, analyze, and appropriately code the lower loop limit, upper loop limit, and the loop increment, if present. If the loop increment is not given, the pointer for integer 1 is supplied. A non-source label is created and filed in the Symbol Table for the loop top, and the begin loop PRF entries are generated.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKBB -- End Loop Processor (ENDLP)

ENDLP encodes the end loop entries for explicit loops specified by a DO statement and for implied loops within an I/O list. See Chart CC.

ENTRIES: ENDLP has one entry point (CEKBB1) and no input parameters.

EXIT: No output parameters.

OPERATION: The end loop PRF entry is generated, and successive calls for the four loop parameters are made on subroutine CLLIM. If the loop parameter is a variable, CLLIM determines whether this is the lowest loop in which it is active. If it is, CLLIM clears the symbol table flags and indicators which distinguish the variable for the duration of a loop.

CEKCJ -- Check Limits (CKLIM)

CKLIM checks DO loop parameters for validity. See Chart CD.

ENTRIES: CKLIM has one entry point (CEKCJ1) and one input parameter: symbol table pointer to loop induction variable in P1.

EXIT: No output parameters.

OPERATION: For each lower limit, upper limit, and increment of a DO statement or I/O loop, CKLIM verifies that the limit is either a variable or constant integer. For a variable the symbol table entry is marked to indicate the level of end loop at which the unredefinable property of a loop limit should be terminated.

CEKCK -- Clear Limits (CLLIM)

CLLIM removes information from the symbol table entries for loop parameters at the loop end.

ENTRIES: CLLIM has one entry point (CEKCK1) and one input parameter: symbol table pointer to loop parameter in P2.

EXIT: No output parameters.

OPERATION: For a variable loop parameter, if the loop being ended is the outermost loop in which this variable is a parameter, CLLIM clears the ULEV field and lowers the "must not be defined" flag in the variable symbol table entry.

CEKBW -- I/O List Processor (IOLST)

IOLST analyzes and encodes the list elements for READ, WRITE, PRINT, and PUNCH statements. See Chart CE.

ENTRIES: IOLST has one entry (CEKBW1) and no input parameters.

EXIT: No output parameters.

OPERATION: IOLST makes two scans over the list elements. The first scan detects and codes the presence of any implied loops. The second pass classifies the variables (if required), generates the EF and PRF entries, and generates the begin and end loop entries, as required. The I/O transmission and end transmission library routine entry names are filed in the symbol table and marked as class external.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKCD -- Format Label Processor for I/O Statements (FLABL)

FLABL processes a FORMAT statement number, as used in an I/O statement. See Chart CF.

ENTRIES: FLABL has one entry point (CEKCD1) and no input parameters.

EXIT: No output parameters.

OPERATION: FLABL checks the label, files it in the symbol table, and fills in the LABF field in the PRF entry being built.

CEKCE -- Read Transfer Processor for I/O Statements (RTRAN)

RTRAN processes ERR and END labels, as used in I/O statements. See Chart CG.

ENTRIES: RTRAN has one entry point (CEKCE1), with no input parameters.

EXIT: No output parameters.

OPERATION: RTRAN performs the necessary checking and sets the PRF entry fields for the error (ERR) and end of file (END) condition transfer labels.

CEKCF -- FORMAT or NAMELIST Name Processor (FNAME)

FNAME processes variable FORMAT designators or NAMELIST names, as used in I/O statements. See Chart CH.

ENTRIES: FNAME has one entry point (CEKCF1), with no input parameters.

EXIT: No output parameters.

OPERATION: For a namelist name FNAME sets the ID and LABN fields in the PRF entry being built. For a variable FORMAT, FNAME sets the LABF field.

CEKAH -- Initial Value Data Specification Processor (IDATA)

IDATA analyzes and encodes initial value data for the type (integer, real, complex, and logical) and DATA statement processors. See Chart CI.

ENTRIES: IDATA has two entry points, TDATA (CEKAH1) and DDATA (CEKAH2), for calls by the type and DATA statements processors, respectively. The input parameter for TDATA is a symbol table pointer for the variable in parameter register P2. The input parameters for DDATA are a parameter list address in parameter register P2 and the number of items in the parameter list as fullword, right-justified binary value in parameter register P1. The parameter list is made up of 2-word items. The first word is a symbol table pointer, and the second word is a fullword offset value.

EXIT: No output parameters.

OPERATION: IDATA has two entry points, one for calls by the type statements and one for calls by the DATA statement. After initialization, each entry point calls on internal subroutine IVAL, which processes the actual value specifications.

Appropriate diagnostics are printed if any source errors or incongruities are encountered.

CEKCL -- Initial Value Processor (IVAL)

IVAL processes constants used as initial values in Type or DATA statements. See Chart CJ.

ENTRIES: IVAL has two entry points, IVAL (CEKCL1) and IVAL1 (CEKCL2). Input parameters are the symbol table pointer of variable in P2 and the current preset data top in P6.

EXIT: Output parameters are

P5 = 0 if constant not entered into data table
= 1 if constant successfully entered into data table
P6 = Updated Preset Data Top

OPERATION: IVAL first checks the variable to which the initial value is being assigned and opens the preset data entry. It then joins with entry point IVAL1 to process the initial data. If a repetition factor is present, it is converted and placed in the preset data entry. The initial value constant is then processed, converted, and added to the preset data entry.

CEKAF -- Array Dimension Specification Processor (ARDIM)

ARDIM analyzes and encodes the dimension specifications for an array, when encountered with a DIMENSION, COMMON, or type statement. See Chart CK.

ENTRIES: ARDIM has one entry point (CEKAFA) and one input parameter. The symbol table pointer of the array name is required in parameter register P2.

EXIT: ARDIM returns with parameter register P2 unchanged. No other parameters are returned.

OPERATION: If the array name class is "unknown" or "simple variable," it is changed to "array variable," and the dimension values are scanned. If the class is already array variable, the source characters, through the next right parenthesis, are spaced over before returning to the caller.

The dimension values may be either integer constants or integer variables. If they are constants, the appropriate dimension table entry is made, depending upon whether the array name is a subprogram argument. If they are variables, the dimension values and the array name must both be subprogram arguments. If so, the symbol table flags are appropriately set to reflect the use of this variable as a variable dimension, and a dimension table entry for a variable dimension is made.

CEKCC -- Label String Processor (LBSTR)

LBSTR processes a string of labels, as encountered in assigned and computed GO TO instructions. See Chart CL.

ENTRIES: LBSTR has one entry point (CEKCCA) and one input parameter: P2 contains the PRF address of the line number field of the PRF entry being formed.

EXIT: Output parameters are

P2 = contains the source character following the right parenthesis of label string.

P3 = contains the PRF address of the last label added.

P4 = contains the count of the number of labels in the string.

OPERATION: For each label ACOMP is called for the label value. The value is checked and added to the PRF entry being built. When a right parenthesis is found, LBSTR returns.

CEKAC -- Statement of Identification (SID)

SID is used to identify the type of source statement. See Chart CM.

ENTRIES: SID has one entry point (CEKAC1), and no input parameters are required.

EXIT: One output parameter, the statement ID number, is returned in parameter register P2.

OPERATION: An initial recognition is made to identify the statement as either an assignment or a nonassignment statement (see Table 18). A precedence table (Table 19) is used, from which new status values are extracted and used for branching. Nonassignment statements are analyzed further, until a unique identification is made. This is done in two steps. First, the first two characters of the name are matched against List 1 (see Table 20). If this does not yield a unique identification, the first four characters of the name are matched against List 2 (see Table 20). The resulting ID numbers are shown in Table 21.

Table 18. Assignment/Nonassignment Character Table

Internal Code	Identification Code
A-G	2
H	1
I	2
J-K	1
L	2
M	1
N	2
O	1
P	2
Q	1
R-S	2
T-V	1
W	2
X-Z,$	1
0-9	3
(	4
)	5
,	6
=	7
EOS*	8
'	10
All Others	9

*EOS - End of Statement

CEKAJ -- Statement Label Processor (LABL)

LABL processes the statement label and determines if any loops are ended. See Chart CN.

ENTRIES: LABL has one entry point (CEKAJA), with no input parameters required.

EXIT: No output parameters.

OPERATION: LABL checks to ensure that the statement to be processed is not a conditional statement of a Logical IF and is not inside a BLOCK DATA program. If this is the case, the label is converted to its binary value. If the statement is in a BLOCK DATA program or is a conditional statement of a Logical IF, an error message is produced, and the scan is terminated.

Table 19. Assignment/Nonassignment Precedence Table

STATUS	ID Code	1	2	3	4	5	6	7	8	9	10
	1	9	2	10	10	10	10	10	11	10	10
	2	2	2	2	3	8	8	7	8	8	8
	3	4	4	5	8	6	3	8	8	3	8
	4	4	4	4	8	6	3	8	8	3	8
	5	8	8	5	8	6	3	8	8	3	8
	6	8	8	8	8	8	8	9	8	8	8
	7	7	7	7	9	9	8	9	9	9	9

8 - Nonassignment Exit
9 - Assignment Exit
10 - Error Exit
11 - Blank Statement

Table 20. Nonassignment Type Statement Identification

List 1	Statement Name	# Characters in Name
IF	IF	2
GO	GO TO	4
DO	*	
CO	*	
WR	WRITE	5
RE	*	
FO	FORMAT	6
CA	CALL	4
DI	DIMENSION	9
AS	ASSIGN	6
NA	NAMELIST	8
EQ	EQUIVALENCE	11
LO	LOGICAL	7
IN	INTEGER	7
IM	IMPLICIT	8
EX	EXTERNAL	8
EN	*	
BA	BACKSPACE	9
SU	SUBROUTINE	10
FU	FUNCTION	8
PR	PRINT	5
PU	PUNCH	5
BL	BLOCK DATA	9
DA	DATA	4
ST	STOP	4
PA	PAUSE	5

List 2	Statement Name	# Characters in Name
READ	READ	4
END Eos	END	3
COMM	COMMON	6
RETU	RETURN	6
REWI	REWIND	6
CONT	CONTINUE	8
ENDF	END FILE	7
REAL	REAL	4
COMP	COMPLEX	7
ENTR	ENTRY	5
DOUB	DOUBLE PRECISION	15

* Not unique.

If the label field is nonzero, a label definition PRF entry is generated, and the DO loop pushdown list is scanned to see if any loops are terminated. If the no-flow flag is up, the no-flow indicator in the PRF entry is set.

If the label field is blank and if the no-flow flag is up, a diagnostic is printed, indicating that the statement is not accessible. If the ISP option is on, TEVCRL is called to create a label, and processing continues with forming the PRF entry.

CEKBQ -- Fallthrough Determination (FALTH)

FALTH is called by EXEC2 to determine if a label reference in the statement preceding the current one refers to the current statement. See Chart CO.

ENTRIES: FALTH has one entry point (CEKBQA) and no input parameters.

EXIT: No output parameters.

OPERATION: FALTH checks to see if the current statement was labeled. If it was not, a normal exit is taken. If it was labeled, the PRF links are followed until the label definition entry is reached. The statement number in the label definition PRF entry is saved, and the link followed to the previous PRF entry. If that PRF entry is a label definition or an argument definition entry, the link is followed to the next entry, and so on. If the PRF entry is any of the GO TO entries, or a CALL, arithmetic IF, READ, or READ with namelist entry, the label references in the statement are matched with the statement number saved from the label definition. If a match is found, the label reference number is set to negative.

The occurrence of a negative statement number in succeeding phases results in object code optimization. If the PRF entry is other than those mentioned above, a normal exit is taken.

Table 21. Statement ID Numbers

Executable	ID No.	Nonexecutable	ID No.
BLANK	0	BLOCK DATA	18
ASSIGNMENT	1	COMMON	19
ASSIGN	2	DATA	20
BACKSPACE	3	DIMENSION	21
CONTINUE	4	END	22
END FILE	5	ENTRY	23
PAUSE	6	EQUIVALENCE	24
PRINT	7	EXTERNAL	25
PUNCH	8	FORMAT	26
REWIND	9	FUNCTION	27
WRITE	10	IMPLICIT	28
READ	11	NAMELIST	29
CALL	12	SUBROUTINE	30
STOP	13	COMPLEX	31
RETURN	14	DOUBLE PRECISION	32
GO TO	15	INTEGER	33
IF	16	LOGICAL	34
DO	17	REAL	35
		STATEMENT FUNCTION	36*

*This ID is never set by SID, but is set by FYPR.

CEKCA -- Diagnostic Message Generator (ERR)

ERR generates diagnostic messages for the statement processors whenever any source errors are encountered. See Chart CP.

ENTRIES: ERR has four entry points. ERR1 (CEKCAA) is used for warning messages, ERR2 (CEKCAB) is used for serious error messages. ERRD (CEKCAC) is used for serious error messages associated with statement deletion, and ERR3 (CEKCAD) is used for fatal error messages associated with abortive end of compilation. The input parameter for all entry points is the message number in register P2.

EXIT: No output parameters.

OPERATION: This routine prepares a parameter list for the compiler executive subroutine RDM (CEKTE), and calls RDM to put out a diagnostic message. The parameters for RDM are determined by the message number presented to this routine. Each message number indicates a list of four halfword indicators. Each nonzero indicator either specifies a piece of prepared text, whose length and location are to be added to the RDM parameter list, or specifies a code branch to perform a special operation to obtain material for the RDM parameter list. A message number for which indicators have not been provided causes a special RDM parameter list to be prepared, giving the message number.

The local maximum error code is updated by this routine, according to the entry used. The delete flag is raised when the ERRD entry is used.

INTRODUCTION

Phase 2 performs several major functions. Storage assignments are made for all source program variables, taking into account the effects of COMMON, EQUIVALENCE, and DIMENSION statements. The source program flow and DO loop structure are analyzed to verify that all referenced labels are defined, to determine that all flow across loop boundaries is legal and to mark loops for materialization of the loop variable (keeping it in its memory cell), or for marking the loop unsafe (minimum optimization) when flow conditions demand it.

ROUTINE DESCRIPTIONS

Phase 2 routines bear mnemonic titles as well as coded labels. The 5-character coded labels begin with the letters CEKJ; the fifth letter identifies a specific routine. Various entry points to a routine are identified by a sixth character, a digit, added to the coded label; for example, the coded label for the diagnostic message generator variable routine is CEKJH, and there are entry points CEKJH1, CEKJH2, and CEKJH3. When reference is made to a compiler executive routine or entry point, the mnemonic title is used, followed immediately by the corresponding coded label enclosed with parentheses.

There are no hardware configuration requirements for any of the Phase 2 routines. All these routines are reenterable, nonresident, nonprivileged, and closed. Except for PHASE2 (CEKJA), which uses standard, type I linkage, all Phase 2 routines use restricted linkage.

The relationships of routines in this phase are shown in the following nesting chart (Figure 18) and decision table (Table 22). The relationships are shown in terms of levels; a called routine is considered to be one level lower than the calling routine. Phase 2 controller PHASE2 is considered to be level 1.

CEKJA -- (PHASE2)

PHASE2 controls the overall processing of Phase 2. See Figure 19.

ENTRIES: This routine is entered using standard, type I linkage. It calls the other routines used in Phase 2 by restricted linkages. It has one entry point, CEKJA1, and one parameter, the executive intercom region.

EXITS: PHASE2 has one normal exit to Exec.

Abnormal exits are converged to the Exec with return codes (RC) 8 and 4. Return code 8 specifies an irrecoverable condition and is referred to mnemonically as the ABORT return code. Machine or compiler errors (MCERR) are indicated by return code 4.

OPERATION: On entrance from Exec, PHASE2 initializes itself and invokes the two main routines: VSCAN and FSCAN. (See Figure 19). VSCAN makes storage assignments using the storage specification list for information about COMMON and EQUIVALENCE statements. FSCAN scans the PRF to perform the flow and loop analysis.

CEKJC -- Storage Assignments for Variables (VSCAN)

VSCAN makes storage assignments for all variables in a source program, and consists of three parts: VSCAN1, VSCAN2, and VSCAN3. See Chart CQ.

ENTRIES: VSCAN has one entry point (CEKJC1) and is invoked by PHASE2. There are no input parameters.

EXITS: VSCAN returns to the Phase 2 executive with the normal, ABORT, or MCERR return codes.

OPERATION: VSCAN assigns storage space for all variables. Assignments are made in certain storage classes. The status of each storage class is kept up to date in its storage class table entry (see Appendix A). Non-COMMON variables are assigned in storage class 6, blank COMMON in storage class 9, and named COMMON in as many of classes 10-127 as are needed. Each symbol table variable entry has its STCL field filled with the appropriate storage class and its SLOC field filled with an assignment relative to the base of that storage class. The storage class table entry for each class includes the number of bytes already assigned in that class; the entry also indicates the next available space.

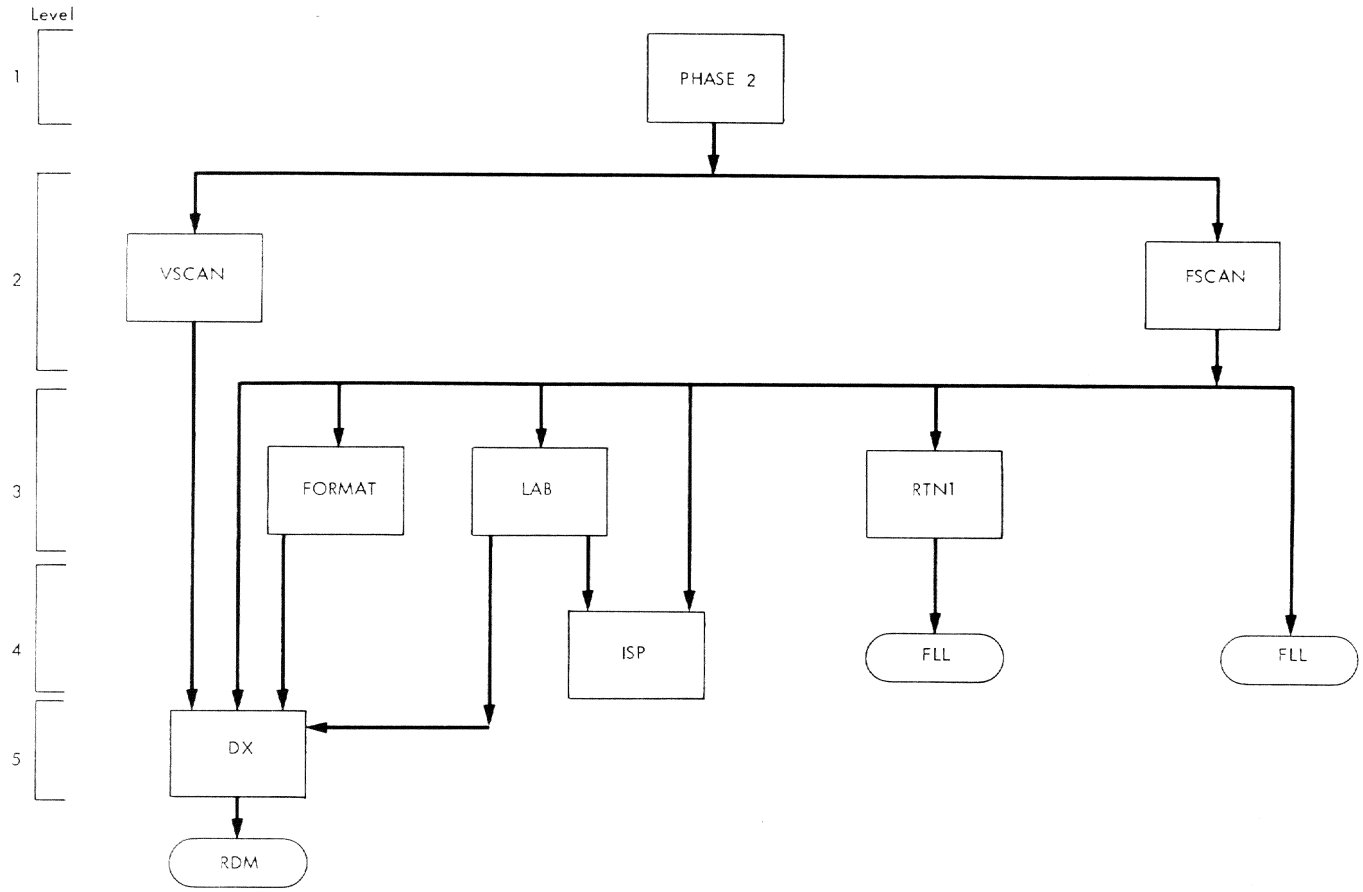


Figure 18. Phase 2 Nesting Chart

Table 22. Phase 2 Decision Table

Routine:----------Phase 2----------Level: 1 ----------

Routine	Usage	Called Routines	Calling Conditions
PHASE2	Controls the operation of Phase 2.	VSCAN	Entered unconditionally to make the memory assignments.
		FSCAN	Entered unconditionally to scan the PRF to perform the flow and loop analysis.

Routine:----------Phase 2----------Level: 2 ----------

VSCAN	Makes the memory assignments for all variables.	DX	If an error condition is found, entered to print the error message.
FSCAN	Does the flow and loop analysis including label processing, illegal transfers, unsafe loops, and COMMON and formal argument definition points.	RTN1	Entered for each label reference to place an entry in the Symbol Table.
		LAB	Entered for each label reference to check the legality of transfers into and out of loops.
		ISP	Entered for each label reference to check for a proper Symbol Table entry.
		FORMAT	Entered for each I/O statement reference to a FORMAT number to check for a proper Symbol Table entry.
		TEVFLL (CEKTFG)	An Exec routine entered for each label definition to file an entry in the symbol table.
		DX	Entered when an error condition is found to print the error message.

Routine:----------Phase 2----------Level: 3 ----------

RTN1	Places label references in the Symbol Table.	TEVFLL (CEKTFL)	An Exec routine entered to make the label entry in the Symbol Table.
LAB	Checks the flow as related to DO loops.	ISP	Entered for each label reference to determine if a legitimate Symbol Table entry exits.
		DX	Entered if an error condition is found to print out the error message.
FORMAT	Checks to see that referenced FORMAT statements are properly defined in the Symbol Table.	DX	Entered if an error condition is found to print out the error message.

Routine:----------Phase 2----------Level: 4 ----------

ISP	Checks to see that referenced statement labels are properly defined in the symbol table.	DX	Entered if an error condition is found to print out the error message.

Routine ----------Phase 2----------Level: 5 ----------

DX	To generate the error message	RDM (CEKTE)	An Exec routine entered for each error to print the line.

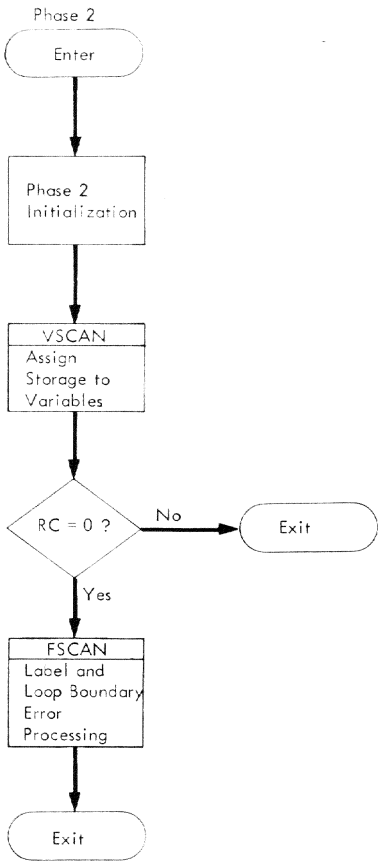
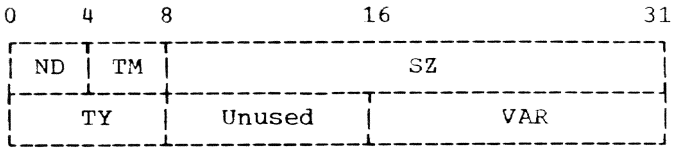


Figure 19. Phase 2 General Flow

All symbol table entries except constants and labels are relinked from the hash-table-based chains used in Phase 1. The variables of each storage class now form a chain, linked in order of assignment. External reference, namelist, and entry entries form three additional chains.

Description (VSCAN1): All chains based on the variable hash table are scanned, and each entry is examined. Those marked as external reference, namelist, or entry are linked into their appropriate chains. Those marked variable but flagged as COMMON or EQUIVALENCE are ignored, since they will be processed later. Those marked variable but flagged as formal argument are also ignored. All other nonvariable entries are ignored.

For each non-COMMON, non-EQUIVALENCE, non-formal-argument variable encountered, an entry in a sort table is made (see Figure 20) containing the number of dimensions, the type size mask, the amount of storage required, the type indicator, and the symbol table pointer. When all symbol table entries have been scanned, the sort table is sorted to increasing value of these fields. The result is that all



ND Number of dimensions (0-7)
TM Type/size mask:

0	Logical*1	3	Real*4
1	Integer*2	7	Real*8
3	Logical*4	7	Complex*8
3	Integer*4	F	Complex*16

SZ Total storage requirement
TY Type: 1 Logical
2 Integer
3 Real
4 Complex
VAR Symbol Table name part pointer

Figure 20. Sort Table Entry

simple (undimensioned) variables come first, then all 1-dimensional arrays, etc. Within a dimensionality, variables of the same type fall together, those requiring less storage preceding those needing more.

The variables are then assigned in the sorted order, to maximize the possibility for sharing address constant cover and subscripts between variables. The assignments are made in storage class 6, with each variable being assigned to the next available byte on a boundary suitable for the type. At the same time, the variable symbol table entries are linked into the variable chain.

Description (VSCAN2): VSCAN2 scans the storage specification list, processing the COMMON variables and providing preliminary processing for variables appearing in EQUIVALENCE statements. The information from COMMON statements filed in the storage specification list is scanned. Each variable is given an assignment (STCL and SLOC) in the storage class for its COMMON block and linked into the symbol table chain for that block. The size of the block is increased for each variable by the space required for the variable.

If the available assignment for a COMMON variable is not at the proper boundary (halfword, fullword, doubleword) for the type, a warning message is produced. (In the object program storage layout all storage classes will start on a doubleword boundary.)

As the storage specification list is being scanned and COMMON variables are processed, the information from EQUIVALENCE statements also receives preliminary processing. The material appearing in a set of parentheses in an EQUIVALENCE statement is called a group; group numbers are

assigned to groups sequentially, in order of occurrence. As an EQUIVALENCE entry is encountered, VSCAN2 must determine whether a variable is of tne type 'FF'. If so, VSCAN2 computes the offset by searching for dimension information in the Dimension Table (whose specifications may occur in DIMENSION, COMMON or explicit TYPE statements). If insufficient or no dimension information is found, an E level diagnostic is issued and an offset is computed by defaulting to the first subscript. Processing continues to the next variable or group, if any, until all groups in the EQUIVALENCE statement are exhausted.

If a variable is of the type 'FF' and dimension information permits an offset to be computed, VSCAN2 overlays the last subscript entry (EE2, EE3, EE4 or EE5) with the newly formed EE1. With this technique VSCAN2 can consistently step through tne EQUIVALENCE entries by an increment of one (1). For each occurrence of a variable in an EQUIVALENCE statement, an entry is made in the variable list (see Appendix A for variable list format). The entry consists of the symbol table pointer, the group number, and the offset in bytes, and represents the equation

(base of group) = (base of variable) + offset

Base-of-group and base-of-variable are unknowns. Base-of-variable represents the eventual storage assignment to be made for the variable. Base-of-group represents the assignment which would be made to a variable appearing in the group with no subscript.

Description (VSCAN3): After the storage specification list scan is completed, the variable list is sorted by increasing order, with the symbol table pointer as the major key and the group, there will be consecutive entries for that variable in the sorted list. These consecutive entries indicate connections between different groups. See Figure 21.

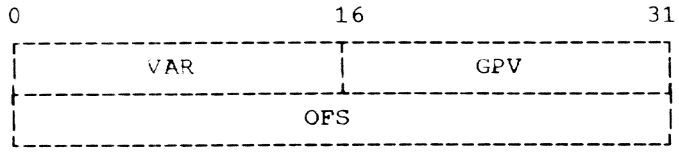
The sorted list of variables is scanned. In the case of consecutive pairs involving the same variable, each such pair represents a pair of equations:

(base of $group_1$) = (base of variable) + $offset_1$,

(base of $group_2$) = (base of variable) + $offset_2$,

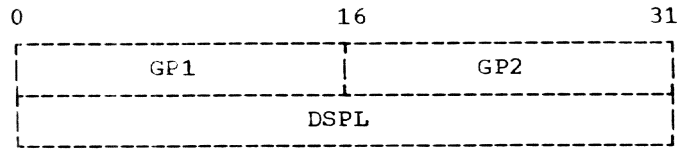
where the number of $group_2$ ≥ number of $group_1$.

Variable List Entry



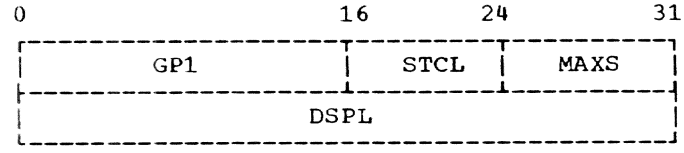
VAR Symbol Table name part pointer
GPV EQUIVALENCE group number
OFS Offset in bytes

Group Connection List Entry



GP1, GP2 Group numbers
DSPL Displacement

Group Table Entry



GP1 Group number
STCL Storage class
MAXS Maximum byte boundary over group
DSPL Displacement

Figure 21. Variable List, Group Connection List, and Group Table Entries

Eliminating the base-of-variable gives the equation

(base of $group_2$) = (base of $group_1$) + ($offset_2$-$offset_1$) ,

which is represented by an entry in a new list, the group connection list, consisting of group $number_1$, group $number_2$, and a displacement computed as $offset_2$ minus $offset_1$.

After completion of this scan, the group table is initialized. It contains one entry per group, and will eventually indicate, for each group, the lowest numbered group with which it is connected and its displacement from the base of tht group. Each entry consists of a group number and displacement, representing the equation

(base of $group_j$) = (base of group) + displacement ,

and is initialized to

(base of $group_j$) = (base of $group_j$) + 0 .

Each group connection entry is processed against the group table. The group connection entry gives the equation

(base of $group_k$) = (base of $group_i$) + $displacement_1$

and the group table entry for group gives the equation

(base of $group_k$) = (base of $group_j$) + $displacement_2$.

Comparing the numbers of $group_i$ and $group_j$, there are three possible cases.

Case 1. i<j The group connection entry relates $group_k$ to a lower-numbered group than that with which it is already connected. $Group_j$ and $displacement_2$ are saved, and the group table entry for $group_k$ is changed to indicate

(base of $group_k$) = (base of $group_i$) + $displacement_1$.

If k = j (as initially), no further processing is needed. However, if k > j, eliminating base-of-$group_k$ from our two equations gives a new group connection entry representing

(base of $group_j$) = (base of $group_i$) + ($displacement_1$ - $displacement_2$) .

This entry is formed and processed against the group table entry for $group_j$.

Case 2. i = j. If $displacement_1$ = $displacement_2$, this entry is consistent but redundant, and needs no processing.

Case 3. i > j. The group table entry relates $group_k$ to a lower-numbered group than the group connection entry. Eliminating base-of-$group_k$ from the two equations gives a new group connection entry representing

(base of $group_i$) = (base of $group_j$) + ($displacement_2$ - $displacement_1$) .

This entry is formed and processed against the group table entry for $group_i$.

After the group connection list has been processed, a final pass is made over the group table. Each entry represents an equation

(base of $group_r$) = (base of $group_s$) + $displacement_1$

If the earlier entry for $group_s$ indicates

(base of $group_s$) = (base of $group_t$) + $displacement_2$,

then substitution yields

(base of $group_r$) = (base of $group_t$) + ($displacement_1$ + $displacement_2$) .

This substitution is carried out for each entry to which it applies.

The group table is now in final form and ready for use. The variable list is scanned again. Each entry represents

(base of variable) = (base of $group_m$) - offset .

The group table entry for $group_m$ represents

(base of $group_m$) = (base of $group_n$) + displacement .

Substitution yields

(base of variable) = (base of $group_n$) - (offset - displacement)

and the variable entry is changed to represent this equation. (If two consecutive entries for the same variable occur, both must transform to the same new entry.) This entry now relates the variable to the base of the lowest-numbered group in the connected set of groups in which the variable occurs.

During this scan the STCL field in the symbol table entry for each variable is checked. This field is zero for a non-COMMON variable, but indicates the COMMON block for a variable which has already been assigned in the COMMON processing. If any variable in a group is in a COMMON block, the group table entry receives the appropriate storage class; otherwise, this entry is set to storage class 6. Also during this scan, the size of the largest variable in a group (1, 2, 4, 8, or 16 bytes) is associated with the group.

The variable list is sorted by decreasing order, with the group number as the major key and the offset as the minor key. (The offset may be negative, so the sort must use algebraic comparisons.) This sort brings together the variables within the same group and arranges them in order of storage assignment.

Now the list is scanned, and assignments are made for each variable. For each non-COMMON group the current size of storage class 6 is adjusted to the proper byte

boundary for the largest variable in the group. The first variable in the group (the one with the largest offset) is given this assignment, and each successive variable is assigned to this location plus the difference of maximum-offset minus the -variable's-offset. Also for each variable, its size (total size if an array) is added to its assignment, and the maximum of these over each group is used to update the size of storage class 6 when all the group variables have been processed. Each variable is linked to the end of the non-COMMON variable chain.

For a group including a COMMON variable, that variable is located and its existing assignment is taken as a group base.

Each variable is given, as an assignment, the assignment of the base COMMON variable plus the difference of COMMON-variable-offset minus variable-offset. A check is made for negative assignments and assignments to improper byte boundaries. Each variable assigned is linked into the chain for the COMMON block, in order of increasing assignment. If the size of the COMMON block is increased by these assignments, the storage class table entry is updated.

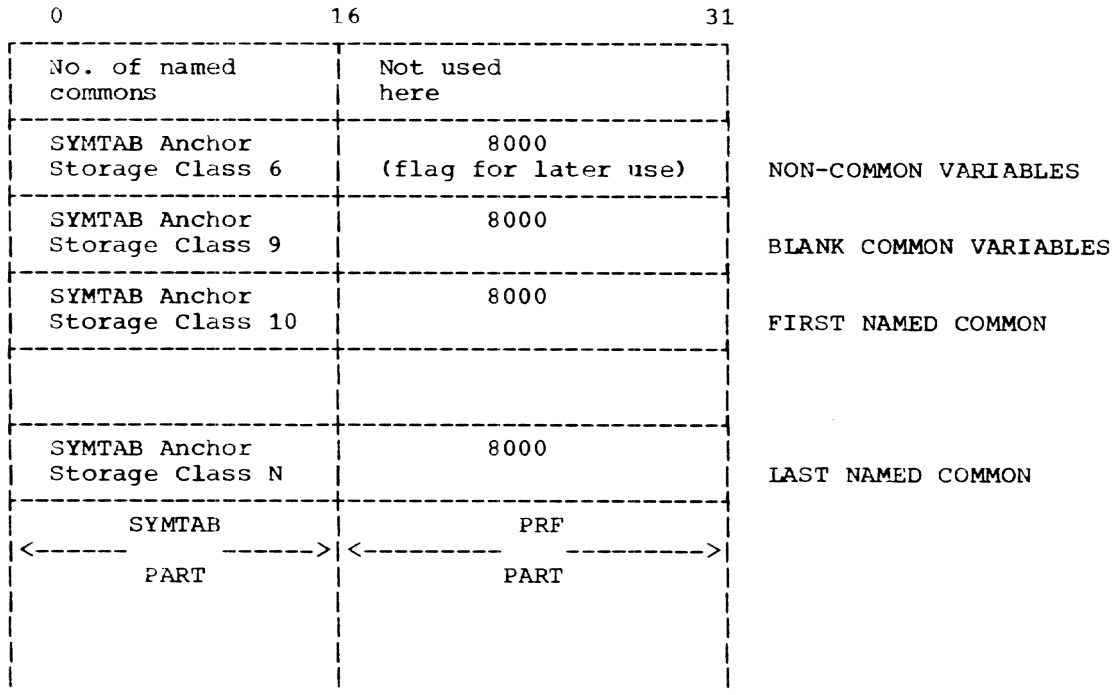
In addition, as the assignments are made for common variables, checks are made to ensure that those appearing in DATA statements are permissible.

VSCAN detects and issues diagnostic messages for source program errors related to storage assignments for variables. These include inconsistencies in EQUIVALENCE relations and assignments forced by COMMON or EQUIVALENCE statements that place variables on byte boundaries which are not proper for the variable type.

VSCAN may issue a diagnostic message and branch to PHASE2 with the ABORT code if the internal tables used for sorting exceed the maximum available space. VSCAN may branch to the Phase 2 executive with the MCERR code if certain conditions are detected which must be due to machine or compiler error.

If the ISD option is OFF, another symbol table scan is made to find interfering variables. In each storage class, the variables are scanned in order of storage assignment by following the existing chains built by VSCAN. For each variable, the storage assignment plus the size is compared with the storage assignments of succeeding variables. When overlap is detected, the "Equivalence Flag" in the symbol table description part is raised and the variables are linked, using the FDP anchor field in the symbol table entries, in a chain of interfering variables for that storage class. At the end of VSCAN, these chains are anchored in a new table, "Intble", which has the format below.

INTBLE FORMAT



CEKJB -- Process Label References and Definitions (FSCAN)

FSCAN is in two parts:

FSCAN1 - is concerned with labels which must be entered into the symbol table and marked if referenced in the source program.

FSCAN2 - is concerned with the following: undefined label references and illegal flow conditions across DO loop boundaries, unsafe loops and the need for materialization, and definition points for COMMON and formal arguments.

See Chart CR.

ENTRIES: FSCAN has one entry point (CEKJB1) and is invoked by the Phase 2 executive.

EXITS: FSCAN returns to the Phase 2 executive with the normal, ABORT, or MCERR return codes. There are no parameters.

OPERATION: FSCAN1 constructs symbol table entries from label references and label definitions, together with information pertaining to each label. One scan is made over the PRF, by simultaneously progressing along three separate chains. These chains are as follows:

1. CLNK chain -- Links all transfer of control statements in the PRF. For each different place to which control can be transferred, there is a label reference.

2. LLNK chain -- Link label definition entries in the PRF.

3. PDLNK chain -- All DO statements have a begin loop (BL3) and a special entry end loop (ENDL) in the PRF, just prior to the next executable statement outside the loop. These are the loop boundary items, and all such items in the PRF are linked into a loop boundary chain called the PDLNK chain.

After any chain entry has been processed, the three links (one for each chain) are compared. The chain having its next entry closest to the present scan position in the PRF is selected for processing next.

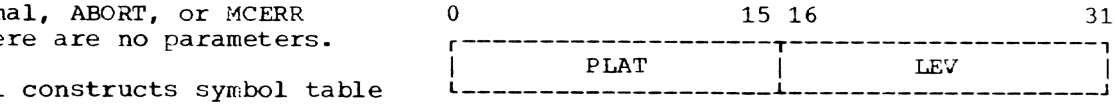
When a GLNK chain entry is selected, each label reference (there are NOEL of them) is placed in the symbol table and denoted as a referenced label. The number of references is given by the NOEL field. The LLNO field in the PRF is changed to contain the pointer to the label entry in the symbol table. A negative label value indicates a reference to the next statement and is not marked "referenced."

When an LLNK chain entry is selected, the label is placed in the symbol table, together with corresponding level and plateau values, and the LLNO field in the PRF is changed to contain the label address in the symbol table. Multiply defined labels are detected in this scan.

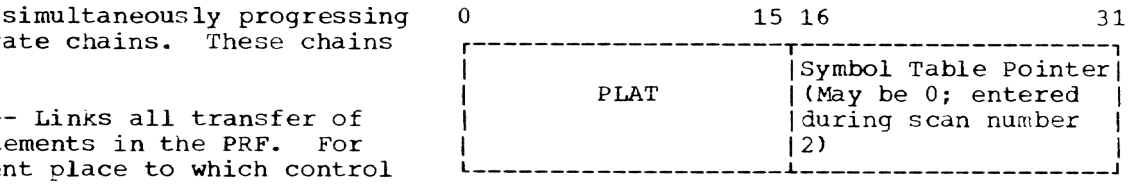
When a PDLNK chain is selected, the level and plateau values are incremented and saved, and the level value is placed into the LEV field of the PRF.

Two tables, formed during FSCAN1, are used during FSCAN2 for detection of illegal flow conditions:

The Barrier Table



The Innermost Loop Table



The plateau value is entered into the innermost loop table whenever a BL3 entry is preceded by an ENDL entry. The level and plateau values are entered into the barrier table whenever a BL3 entry is followed by an ENDL entry.

A plateau is any area between loop boundary entries. The PRF link to a loop boundary entry (End Loop or BL3) is used as a name for the plateau which starts with that entry.

FSCAN2 is concerned primarily with three things:

1. Discovering any label references which are not defined.

2. Processing DO loop items for flow conditions.

3. Forming the CDP and ADP chains.

These three processes are carried out simultaneously while FSCAN2 is scanning the PRF along the ILNK chain (which links together successive PRF items).

Label Processing

If the current item in the PRF scan is a label reference item, each symbol table pointer associated with the label reference is used to locate the define bit in the symbol table for that particular label reference. If the define bit is not on, this particular label is not defined, and an appropriate error diagnostic is given. This particular PRF item is then deleted from the ILNK chain.

Label definition items are similarly checked, to see if they have been referenced (i.e., the reference bit is set in the symbol table). If not, this item is deleted from the LLNK chain. After this scan the LLNK chain will link together successive label definitions for referenced labels only. Label definition items which have been referenced are checked for the no-flow condition. If the no-flow bit is set in the symbol table, a diagnostic is issued indicating that the statement cannot be reached.

Flow Processing

If the current item in the PRF scan may cause an illegal flow condition to occur, the item will be investigated for all such conditions, and a diagnostic given if any is found.

The flow processing is broken into two areas for investigation. These conditions are described below. "Level zero" denotes a plateau not inside any DO loop.

1. Jumps from or to Level Zero

   Jumps from Level Zero. If the jump is not to an innermost loop or to level zero, a diagnostic is given. If the jump is to an innermost loop, the plateau and symbol table pointer of the definition are entered into the E loop list.

   Jumps to Level Zero. If a jump is made to level zero from an innermost loop, the plateau value of the label reference is entered into the X Loop List. In this case, all intervening levels from the label reference to the level preceding the label definition are marked as materialized in the "Materialization List."

2. Jumps Other than those from or to Level Zero

   Jumps into Loops (Jumps to Higher Levels). A jump from a level other than level zero to a label definition whose level is greater than that of the label reference is an illegal jump into a loop, and an appropriate diagnostic message is given.

   Jumps Out of Loops (Jumps to the Same or Lower Levels). If a jump is made (from a reference level other than level zero) to a label definition whose level is less than or equal to that of the label reference, the barrier table must be inspected for any plateau values with level lower than that of the definition intervening between the plateau value of the label reference and that of the label definition. If there are no such plateau values between these limits, the jump is legal. If there is such an intervening plateau value between these limits, a diagnostic message is given, indicating an illegal jump into a loop. All loops from the reference level to the definition level are marked "materialize."

DO Loop Processing

1. Unsafe Loops

   Two lists are formed during FSCAN2:

   E Loop List -- consists of plateau values of label definitions which occur at an innermost loop, and are referenced from level zero.

   X Loop List -- consists of plateau values of label references which occur at an innermost loop.

   Every entry in the E loop list should also be in the X loop list; therefore, each innermost loop entered from level zero also has a jump out of this innermost loop to level zero. If this condition is not met, an appropriate diagnostic is given. A third scan is made over the PDLNK chain. If any entries exist in the E loop list, those end loop entries which lie between the plateau values of the label reference and the label definition are marked as unsafe.

2. RETURN Loops

   Loops containing RETURN statements are marked "materialize" if the loop variable is in COMMON or is a formal argument called by name.

3. Definition Point Chains

   The CDP chain connects PRF entries which must be considered as definition points for all COMMON variables or formal arguments. An entry is linked in this chain if the statement involves a call on an abnormal func-

tion or subroutine (one which may redefine COMMON), or if a formal argument is explicitly assigned a new value.

The ADP chain connects PRF entries which must be considered as definition points for all formal arguments. An entry is linked in this chain if a COMMON variable is explicitly assigned a new value (and the entry is not already in the CDP chain).

The processing of interfering variables takes place during the second PRF scan. Chains of the interfering variables within each storage class are formed within the PRF and anchored in the second halfword (the PRF part) of the corresponding Intble entries. When a variable is being defined, that is, wherever ID = 5 (equation), ID = D (argument definition point), ID = 10 (Begin Loop 2), or ID = 21 (Input list Element), the equivalence flag in the symbol table descriptive part for the variable is checked. If the flag is raised, the storage class of the variable is used to locate the correct Intble entry. The variable is added to its chain by setting the VDP field to the PRF part of the Intble entry, and the chain's anchor is updated by setting the PRF part of the Intble entry to the ILNK field.

The interfering variable chains are reversed during the reversing of the CDP and PDLNK chains. Another table of the same size as Intble, LNKSAVE, is used to hold the saved links. During the chain reversal, the chain whose current link is in the highest location is chosen for reversal at each step. The set of interfering variable chains is searched to find the highest link, and the result is compared with the current CDP and PDLNK links to find the highest current chain link.

After the chain reversals, Intble is scanned for storage classes containing interfering variables. These symbol table chains of variables are followed and FDP anchor fields are set to the beginning of the PRF VDP chain for that storage class.

A halfword cell "LXT" is used to hold a symbol table pointer and a flag "ACGTFL" is used to indicate that the current PRF item is either an assigned or a computed GO TO statement.

ACGTFL is lowered before starting FSCAN2 and raised at each computed (ID 8) or assigned (ID 7) GO TO item before calling LAB (CEKJE) and lowered when returning from LAB.

LXT is set to 8001 at each end-loop item (ID12) unless the global flag is raised (i.e., loop is flagged "Innermost no calls"). In this case, LXT equals 8000. At each begin-loop-2 item (ID 10), the current value of LXT is put into the EXITLB field and the LXT is set to 8001.

CEKJD -- Label Reference Processor (RTN1)

RTN1 places label references in the symbol table. See Chart CS.

ENTRIES: RTN1 has one entry point, CEKJD1. Input parameters are

P1 -- Number of label references in PRF item
P2 -- Index to first label number in PRF item

EXITS: RTN1 returns to the invoking routine with the normal or ABORT return code.

OPERATION: RTN1 checks the sign of the label number. If the sign is positive, it is replaced by the pointer to the corresponding symbol table entry, and the symbol table entry is marked as "referenced." If the sign is negative, indicating a reference to the next statement, it is replaced by the symbol table pointer, but is not marked as "referenced."

Negative values appearing in arithmetic IF statements are simply replaced with X'8000' to indicate fall-through.

CEKJE -- Label Reference Processor (LAB)

LAB checks flow as related to DO loops. See Chart CT.

ENTRIES: LAB has one entry point, CEKJE1. Input parameters are

P1 -- Number of label references in PRF item
P2 -- PRF index to first symbol table pointer
P5 -- Pointer to PRF item

EXITS: Only the normal exit is made, with no output parameters.

OPERATION: LAB checks the legality of jumps from and into DO loops, as described under "Flow Processing" in FSCAN.

The materialization list is marked as required, the necessary X loop and E loop list additions are made, and appropriate diagnostics are given when illegal flow conditions are detected.

LXT and ACGTFL are processed at the two points where a branch out of a loop is detected to a level of zero or greater than zero. If LXT equals 8000 or if LXT equals

the Symbol Table pointer for the current label, and if ACGTFL is down, then set LXT to equal the Symbol Table pointer for the current label, and omit marking the MAT stack for the currently innermost level. Otherwise set LXT to 8001.

CEKJF -- Statement Label Reference Inspection (ISP)

ISP determines whether referenced statement labels are properly defined.

ENTRIES: ISP has one entry point, CEKJF1. Input parameter is

P2 -- PRF index to symbol table pointer

OPERATION: The symbol table item of the referenced label is checked to see if it is marked "defined." If it is not, a diagnostic is issued and the undefined flag is raised. ISP is not entered to check the validity of FORMAT label references. A diagnostic is issued and the undefined flag is raised when erroneous references to FORMAT lables are encountered.

CEKJG -- Format Reference Inspection (FORMAT)

FORMAT determines whether referenced FORMAT statements are properly defined.

ENTRIES: FORMAT has one entry point, CEKJG1. Input parameters are

P2 -- PRF index to symbol table pointer
P5 -- Pointer to PRF item

OPERATION: The associated symbol table entry of the referenced label is checked to see if it is marked "defined." If it is not, a diagnostic is issued and the undefined flag is raised. A diagnostic is also issued and the undefined flag is raised if the class of the label item is not FORMAT.

CEKJH -- Diagnostic Message Generator (DX)

DX generates diagnostic messages whenever error conditions are encountered. See Chart CU.

ENTRIES: DX has three entry points: CEKJH1, CEKJH2, CEKJH3. The input parameters, for all three entry points, are

P1 -- The Phase 2 diagnostic code
P2,P3 -- Pointers to the symbol table or PRF item from which information is to be extracted.

EXITS: Only the normal exit is made, with no output parameters.

OPERATION: DX generates a diagnostic message by operating on a parameter list, from which another parameter list is generated for RDM. An input parameter may be one of two types. The first type is a parameter which merely points to a piece of prepared text. In this case, the address of a word containing the text length in characters and the address of the text are entered into a parameter list for RDM.

The second type is a parameter which specifies that a certain predefined operation is to be performed. In this case, an indexed branch on the parameter is made to the operation to be performed. Each of the operations extracts specified information from some table or file, such as the symbol table, performs any conversions required, and makes appropriate entries in the parameter list for RDM.

A parameter word containing zeros indicates the end of the input parameter list, and RDM is called to output the diagnostic message.

INTRODUCTION

The major function of Phase 3 is global optimization, which is the process of minimizing the number of object code instructions to be generated by Phase 4. There are four categories of global optimization.

1. Removable Expressions. A "removable expression" is one whose individual operands do not have "definition points" inside the loop. A definition point is a statement in which the variable has, or may have, a new value stored in it (e.g., appears on the left-hand side of an equal sign). In removing an expression, Phase 3 does not remove the left-hand side of an assignment statement nor a label. The "store" operation remains inside the loop.

   In the following example the expression (B+C) is removable from the indicated loops, but the expression (A+D) is not, since the variable A has a definition point inside the loop (statement 10).

```
            .
            .
        DO 30 I=1,N
10      A=B+C
20      E=A+D
30      CONTINUE
            .
            .
```

2. Common Expressions. Two occurrences of an expression are considered to be common if the value of the expression cannot change between the occurrences, i.e., there are no definition points for any of the variables involved and there is no intervening referenced label.

   In the following example, the occurrences of A+B in statements 10 and 30 are considered common with each other, but not with the occurrence in statement 40 because there is an intervening referenced label. That is, statement label 40 (which is referenced from statement 60) intervenes between the occurrence of A+B in statement 30 and its occurrence in statement 40. Labels 20 and 30 intervene between the occurrence of A+B in statement 10 and its next occurrence; however, since neither label 20 or 30 is referenced, the occurrences are considered to be common. The expression (C+(A+B)) cannot be marked common in statements 10 and 30 because the value of C changes (i.e., has a definition joint) in between, at statement 20.

```
            .
            .
10      D=C+(A+B)
20      C=D+F
30      E=C+(A+B)
40      IF(A+B)50,10,70
50      A=F+E
60      GO TO 40
            .
            .
```

3. Subscript Expressions. Subscript expressions determine which individual element of a dimensioned array is referenced. The expression may contain four types of constituents:

   a. An address constant (adcon)

   b. Induction variable parts

   c. Removable parts

   d. Nonremovable parts

   Each subscript has exactly one associated adcon. It is determined from the base address of the array variable itself and the collection of constant terms (done by Phase 1).

   The induction variable is the variable referenced in the DO statement of the loop. In the statement

   DO 10 I=1,N

   I is the induction variable (also referred to as the loop variable).

   For the removable and nonremovable parts the same criteria are applied, as described in "Removable Expressions" above.

   In the following example, the terms of the subscripts involving I and J are induction variable parts (statements 30 and 40). Removable terms are found in statement 30. The terms involving N are removable from both loops, and the terms involving I are removable from the inner loop (statement 2 loop).

The subscript terms involving M in statement 20 are nonremovable because of the M definition point in statement 10.

```
          .
          .
 1      DO 50 I=1,K
 2      DO 40 J=1,L
10      M=J+3
20      Z(M)=A+M
30      X(I)=Z(N)+Y(N)
40      Y(J)=Y(J)+M
50      CONTINUE
          .
          .
```

4. DO loop control. It is a Phase 3 responsibility to determine the method that is going to be used by Phase 4 to generate the loop control instructions. The following are the types of loop control and the criteria for each.

   a. BXLE on recursive.

      This loop is controlled by a BXLE instruction of the form:

      ```
      BXLE R1,14,LOOPTOP
      ```

      where R1 contains the recursive expression that has been initialized to zero at the loop top, register 14 contains the increment to be added to the recursive, and register 15 contains the test value.

      The requirements for this type of loop are:

      - There must be no reason to materialize the induction variable; e.g., the ISD option must be off, and the induction variable must not appear in the loop outside of a subscript.
      - Loop must be save, innermost, with no external calls. This is necessary since Phase 4 is going to globally assign registers 14 and 15.
      - There must not be branches out of the loop to more than one label. If there is a branch to only one label, the induction variable is materialized on the exit path.
      - There must appear in the loop at least one subscript expression containing the induction variable as the least-removable term. The coefficient of the recursive must be a positive constant.

      The following is an example of a BXLE on recursive loop:

      ```
            REAL*4 A (10)
            REAL*8 B (10)
             DO 10 I = 1,10
       10 B(I) = B(I)+A(I)
      ```

      In the example, both recursives are candidates for the BXLE, but the recursive on B, having more uses in the loop is selected.

   b. BCTR loop.

      This loop is controlled by a BCTR instruction of the form:

      ```
      BCTR 15,14
      ```

      where register 15 has been initialized at the loop top with the count of times through the loop, and register 14 contains the address of the loop top.

      The BCTR loop requirements are:

      - Induction variable does not need materialization.
      - Loop must be save, innermost, with no external calls.

      The BCTR instruction is never selected if the loop also contains the recursive requirements to qualify for a BXLE on recursive loop.

      An example of a BCTR loop is:

      ```
            DO 10 I=J,K,L
       10 B(I) = B(I)+A(I)
      ```

      Since the loop step (and hence the recursive increment) is not a constant, the loop does not qualify for BXLE on recursive.

   c. Materialize and BXLE on induction variable.

      This loop is controlled by a BXLE instruction of the form:

      ```
      BXLE 1,14,looptop
      ```

      where register 1 contains the induction variable, register 14 contains the loop step, and register 15 contains the test value that has been created by Phase 4. The instruction at LOOPTOP is always a store out of

register 1 into the induction variable. The prime requirement of this loop is that the induction variable must be materialized; when a loop fails the requirements for the other loop control methods, it is always materialized, since there is no other way to count the loop.

Phase 4 recognizes two versions of this loop. One, when the loop is innermost, safe, and has no external calls. In this case registers 14 and 15 can be globally assigned in the loop. Otherwise temporary storage must be used to save and restore the registers. Two examples of the BXLE-on-induction-variable loop are:

```
      DO 10 I = 1,10
      IF (I .EQ. 1) GO TO 10
      A(I) = 0.
   10 CONTINUE
   DO 20 I = 1,10
   IF (I .EQ. 1) GO TO 20
   A(I) = SQRT (A(I))
   20 CONTINUE
```

In both cases, I must be materialized since it is referenced outside of a subscript. In the first example, registers 14 and 15 can be globally assigned. In the second, they cannot be because of the call to the SQRT function.

d. Compare and Test Recursive.

This loop is controlled by a compare and branch-not-equal (BNE) of the form:

```
CLR R1,R2
BNE LOOPTOP
```

where R1 contains a recursive value; R2 contains the test value initialized outside the loop and (in this example) is globally assigned. If R2 did not have enough weight to be globally assigned, then the compare would be to temporary storage.

The only requirements for this loop is that there must be no need for materialization, and there must be at least one recursive expression (a subscript term containing the induction variable). An example is:

```
      DO 10 J = 1,10
      DO 10 I = 1,10
   10 A(I,J) = 0.0
```

The outer loop, where j is the induction variable, will be controlled by a compare and branch.

5. Global Register Assignment. In order to facilitate minimizing generated instructions, Phase 3 considers certain items for permanent assignment to registers across a loop. The selections are made by maintaining a popularity count for each item. The count is weighted for each type of candidate, considering the value of having it globally assigned versus the value of not having it globally assigned.

   a. Adcons. Weight = 5.

   b. Removable integer expressions. Weight = 5.

   c. Recursive expressions. Weight = 10.

   d. Constant steps on recursive expressions. Weight = 3.

   e. Expressions for testing the end of a compare-and-branch loop. Weight = 5.

Some items that are used for generating loop control instructions do not follow the normal selection methods. For example, in a loop where registers 14 and 15 will be globally assigned by Phase 4, the items in those registers will mot be considered for assignment by Phase 3. In a BXLE on-recursive loop, Phase 3 always gives global assignment to the recursive expression, but never to the constant step on the recursive or to the test value.

Phase 3 also considers one floating point quantity to be pseudo-globally assigned into FP register 6.

This assignment can take two forms.

a. Where the variable or subscripted variable on the left of the equal sign can be kept in FP register 6 through the loop, and stored when the loop is completed.

   The requirements for this assignment are:

   - Loop must be innermost, with no external calls.
   - ISD option must be off.
   - Loop must contain only one assignment statement, plus any number of blank or CONTINUE statements.

- Loop must contain no complex nonremovable operation or 2-argument intrinsic function.

- Assignment variable must be real and must not be flagged "interfering" in the symbol table.

- If the assignment variable is subscripted, the subscript must be removable to at least BL1 of the inner loop.

- If the assignment variable is an array element, references on the right side of the equal sign must be to the same element, or to an element which is known to never be the same. The loop is flagged only if all references are to elements known to be either the same element or never the same element. For example:

  ```
  A(I) = A(I) + A(I+1)
  ```

  On the right of the equal sign A(I) is flagged as being in FP register 6, but A(I+1) is not flagged. The loop is flagged for global assignment, since A (I+1) can never reference the same element as A(I).

  In the following examples, the loop is not flagged:

  ```
  A(3) = A(I)
  A(I) = A(J)
  ```

  In this case, since it is not known whether I can equal 3 or whether I can equal J, the loop is disqualified.

  The reason for this restriction is that the array element on the left will not be updated in storage if it is globally assigned. Therefore, it is necessary to know at compile time which references to array elements on the right should obtain values from storage and which should obtain values from the globally assigned registers. If this determination cannot be made, a global assignment is not made.

  The first 3 requirements are determined by Phase 1, the last 4 by Phase 3. If all requirements are met, the begin-loop entries are flagged for Phase 4, along with each EF item on the right-hand side that matches the assignment variable.

b. When a loop does not meet the requirements of (a), one optimization which might still be performed is to select a floating point quantity that can be loaded into FP register 6 outside the loop.

   The requirements for this optimization are that the loop must be innermost, safe, with no external calls.

   In its backward scan over the loop, Phase 3 selects the candidate that is last processed. This candidate is deleted if a referenced label is reached, or if tne current candidate appears as an assignment variable. In all cases, the candidate must be a simple, real variable or a simple, real constant.

In addition to the major function of Phase 3, many other functions are performed. A more complete description of Phase 3 is listed below.

1. The program file (PRF) is scanned backwards. The expression file (ERF) is scanned, when required, for the PRF item; and, a triad table is created for internal use with one entry for each unique expression. An operand pushdown table (OPT1) is created to assist in scanning the ERF. The PRF and ERF are modified to form the PF and EF, which are treated separately by Phase 3. These files are relinked in the forward direction and interleaved into a new program file (PF) that is the input for Phase 4.

2. All variable and constant entries in the ERF are changed, with the OFFSET field replaced by a reference to an address constant and an immediate value of the displacement. The adcon is represented by a new type of entry introduced into the symbol table.

3. All subscripts are rearranged. The adcon for the variable is placed in the expression, which is rearranged to remove the largest subexpression from loops, to handle loop variables by recursion, and to make use of double indexing.

4. Common expressions are recognized and named. The point at which they are last used is marked.

5. Expressions that can be computed outside of loops for use inside ("removed") are recognized and named. They are inserted in the EF at the loop top and removed from the EF inside, leaving a short "residue" entry there.

6. Expressions that can be computed by recursive additions around a loop are identified, and one (for each loop) for use as a test of the loop is determined. The initial value, step values, and test value expressions are formed and treated as other expressions (see items 4 and 5 above).

7. Quantities that are to be placed in registers and kept there over loops are determined and specified. These may be integer arithmetic operations, subscript expressions, or address constants. They are determined on the basis of total time saved and number of registers that can be used for such purposes.

8. Each statement label entry in the symbol table is changed to contain a reference to an address constant (see item 2 above). The adcon entry is given an estimated value based on the estimated location, which is then cleared.

9. The formal arguments have variable adcons that must be computed at the preamble of a subprogram. These are listed in the formal argument adcon table (FAAT) for Phase 4. (The format of FAAT is explained in the module description "CEKKS Phase 3 storage (PSECT).")

A general description of the procedures used by Phase 3 to carry out its functions is given in the following paragraphs.

Phase 3 makes a backwards scan over the PRF, rewriting it. By means of the links in the PRF, each value a variable assumes can be analyzed to determine the loops from which that variable can be removed and a point at which it can be first used in computation. This point is preliminary and can be moved if the PRF scan reaches a point where the value may change. Each reference to the ERF string from PRF entries causes a local forward scan over the ERF. An operand pushdown table is built and used during this scan. All operators are entered into the triad table for commonality determinations. Certain operators are put in a compute and removal item table (CRT). Subscript expressions in the ERF are scanned twice. First, determinations of computation points, removal levels, and use of loop variables are made. Then the ERF is sorted into a new order, new operators are introduced, and the address constant is introduced into the expression. Finally, this new ERF is scanned again, and entries are made in the triad table and the compute and removal item table, as for other expressions.

For DO loop processing, Phase 3 maintains a set of loop tables for use in determining global register assignments, identifying loop variables, and determining removal levels.

MEMORY REFERENCE PROCESSING

During the processing of Phase 3, all references to variables and constants (including address constants) are replaced by references to an address constant and a displacement. Adcons are supplied for each storage class, with one separate adcon for each 4080 bytes. Adcons are supplied only when needed and are entered into the symbol table to be shared with all other parts of the program. Negative adcons (e.g., storage class base -4080) are allowed.

In computing the values for adcon and displacement, Phase 3 uses the storage class, the assigned location within that class, and, in some cases, the offset from that location supplied by Phase 1. At all times the FORTRAN object program makes use of one register that can cover the special page, part of which is assigned to adcons.

The adcon reference and displacement are placed in the Polish string for the variable. In addition, the reference to the original variable entry in the symbol table is kept for purposes of editing in Phase 5. When a subscript expression modifies a variable and the adcon is referenced in that expression, the adcon reference from the variable in the Polish string remains zero.

Whenever constant subscripts result in a reference to other than the first byte of an array but no subscript expression occurs, a special subscripting operator is entered in the triad table. This operator is called the "@" or "Addressing" operator. This operator is strictly internal to Phase 3; it is used only in the triad, not in the EF, to distinguish between references to other than the first byte of an array.

COMMON EXPRESSIONS

A common expression is one that is used more than once, but needs to be computed only once. Phase 3 determines the existence of these in most cases; it gives each a distinct identifying number (name) and

marks one occurrence of this expression in the last Polish string in which it occurs as "last use." Three separate determinations must be made:

1. It must be determined which expressions are identical in form.

2. It must be determined that occurrences are necessary or "valid" (i.e., two occurrences as part of the same larger common expression require only one use of the smaller expression).

3. It must be determined that the expression cannot change in value between occurrences.

Identity is determined by the triad table. Every expression is changed to a triad (operator plus references to its two operands) that is identical for identical expressions. For each expression the triad table is searched to determine whether the expression is already there; if it is not, it is inserted. If the expression is already in the table, it may be a common expression, depending upon the results of the other two determinations.

The determination of the occurrence of two valid uses is made by another algorithm: whenever an expression occurs after the first time and either is part of a larger expression that is occurring for the first time or stands alone in a statement, this is the second valid occurrence. When this situation exists, it becomes necessary to mark previous occurrences (sometimes, in a previous larger expression, there may be more than one that were not all valid). This is done by keeping a pointer in the Triad Table entry to the "last occurrence." Whenever an unnamed expression is entered or located in the triad table, a reference to the triad is made in the ERF string entry. Then, when an ERF string entry is copied into the EF without receiving a name, the pointer in the triad table is set to point to this EF string. At the time an expression is named, the EF string pointed to by the triad table entry is scanned, and the occurrences of this expression in that string are recognized by the reference to the triad. The field used for that triad reference is now used for the name. Whenever a name is entered in an EF entry, the expression is changed to indicate a named expression; and, if this is "last occurrence," the "last use" flag is turned on. If a second valid occurrence occurred in the same string as the first, the "last occurrence" field is X'8000'. In this case, the "last use" bit is turned on in the current EF, and other occurrences are named later when the ERF string is copied into the EF.

In order to determine that the value of an expression does not change, Phase 3 must consider the component of each expression. Every variable entry in the symbol table has pointers (FDP and BDP) to forward-and backward-linked chains in the PRF of its definition points. There are also linked chains for definitions of COMMON variables and definitions of arguments. COMMON variables are defined at their own definition points. Arguments are defined at every definition point for any argument and for COMMON definition points. Every variable is also marked when it is a loop parameter over any loop in the current nest, and it is, therefore, of fixed value over that loop. The loop tables indicate the range of every loop in the current nest (in terms of PRF entries) and indicate which loops are "unsafe" (have entry points from an outer loop). By scanning this information, Phase 3 can determine which loop, if any, is the outermost loop from which expressions containing only this variable can be removed. This level is entered in the symbol table entry for the variable as the RLEV field (removal level). If the nearest definition point (forward) lies outside a loop from which the variable cannot be removed, the forward compute point (FCP) is set just inside that loop; otherwise, it is set at the nearest definition point forward.

This determination does not take into account all possibilities that may limit the range of commonality. The other limitations are the occurrence of referenced labels, the occurrence of loop endings of unsafe loops, and the occurrence of the FCP (determined above) inside a new loop. These are determined as the PRF is scanned further and are used to terminate the range of the commonality if it has been set up or to prevent its being set up.

The mechanism for keeping track of the range and terminating commonality is the compute and remove item table. Here every named expression has an entry that is keyed to the FCP (or the removal point) in the PRF. These are scanned in parallel with the PRF. For compute point entries the range of commonality is terminated when the point is reached. Whenever a loop end or a referenced label is reached that might limit the range of commonality of some entries, this table is consulted and the commonality terminated. Commonality is terminated by removing the name from the triad table entry and appropriately marking that entry.

When a common expression is named, it must be determined that referenced labels

or terminating loop ends have not intervened between the "last occurrence" and this one. This can be checked by the loop tables and the backward chain of referenced labels. If such intervention has occurred, the expression is not named, and the current EF string becomes the last occurrence.

REMOVING EXPRESSIONS FROM LOOPS

Computing time can be saved when an expression that occurs within a loop can be computed outside that loop for use inside. Expressions that contain the loop variable and are common over the loop comprise a special case. Since a new occurrence is introduced at the point of removal, expressions that are removed are always treated as common, even if they occur only once within the loop. However, for purposes of producing better object code, it is not always desirable to remove an expression from Level 0 (the false loop) if its only occurrence is at Level 0. Therefore, such an expression is treated as not removable at its first occurrence. If there are other occurrences, the first is marked common with them and the expression is treated as a normal removable expression.

Some of the mechanism for handling removal is discussed above in the explanation of the determination of the removal level (RLEV) of expressions. If an expression is part of a larger expression that can be removed, it is ignored within the loop and treated at the loop top when the larger expression is inserted there for computation outside the loop. If an expression can be removed but the expression that contains it cannot, or if it stands alone, it is removed. At the first occurrence of the expression within the loop, it is removed after it is given a name, after commonality with any previous occurrence (outside the loop) is established, and after last usage is marked. Removal consists of placing a pointer in the compute and removal table at BL1, marking the triad entry properly, and replacing the Polish Expression in the EF by a residue entry with the expression name.

A special type of removal expression occurs when an induction variable is involved in an expression that is not in a subscript. Here the variable has a compute point inside the loop (BL3) and is removable from the inner level. When such an expression is encountered, either as a common expression or a removal item from an inner loop, a special test inside the commonality routine allows the expression to be common, despite the possible intervention of labels. These items are removed to BL2, after their commonality is established and their last use marked. It is possible for such an expression to occur only as a residue and to occur elsewhere as a named expression that is still present.

When the removal point is reached at BL1, BL2, or BL3, the Polish expression is reconstructed from the triad table and inserted in the EF, except for subexpressions that may be further removed. At this time, the commonality of all expressions and the removability of all subexpressions is treated in the same manner as expressions that occur elsewhere in the code.

OPTIMIZING SUBSCRIPT COMPUTATION

Terms involved in the subscript expressions are divided into four categories:

1. An adcon
2. Induction variable (or recursive) terms
3. Removable terms
4. Nonremovable terms

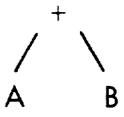
The adcon for the subscript expression is always determined by Phase 3 from the SLOC field of the array variable and the offset that computed Phase 1.

The terms of the expression are grouped according to the four types and then sorted by removal level. The adcon is always the most removable entry. The induction variable or recursive terms are considered special within a removal level group. A recursive expression is one that increases by a fixed amount each time through a loop. These expressions are treated specially, in that they are considered removable only to begin loop-2 of a level rather than to begin loop-1 as for a normal removable expression. Therefore, if two terms of a subscript expression have the same removal level but one is a recursive term and the other is not, the recursive term is considered less removable for the purposes of sorting the terms.

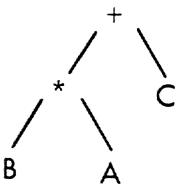
Special operators are introduced into the expression to separate the groups of sorted terms. A recursive operator (!) is used to mark the induction variable terms internally for Phase 3. Another special operator, the base/index split operator (?), is used to separate the nonremovable terms from all the removable terms. If there are no nonremovable terms, but there is a term of the form 'I a *' where I is the induction variable of an innermost loop with no external calls, and 'a' is a con-

stant or a multiply expression (only possible if the array has adjustable dimensions), then 'I a *' is converted to 'a o!' and the major operator is converted to a '?'. The ? operator can appear only once in any subscript expression, and it is passed on to Phase 4 to indicate that a base-index method can be used to reference this array. All other terms are connected by plus signs.

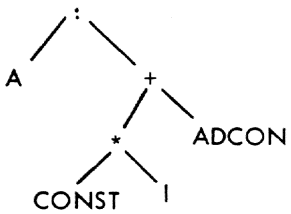
The best method for explaining this function of Phase 3 is by the use of expression diagrams. The diagram, called a "tree," can show the exact relationship of all terms of an expression. By convention, the left-hand operand of an operator represents the left-most term of the original FORTRAN expression. Therefore, the expression (A+B) is written.



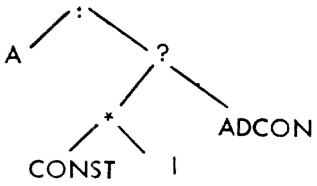
and the expression (B*A)+C is written



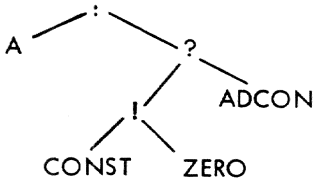
With this basis, the subscript expression processing can be examined further. In the following discussion, all constants are referred to as CONST and all address constants as ADCON, since their actual values are not relevant. The operator that links the subscript expression to the array variable is the colon (:). A reference to the variable A(I) that comes into Phase 3 is essentially the same under any conditions. However, the Phase 3 output to Phase 4 depends on the conditions of the variable I. The four possible output expressions are:



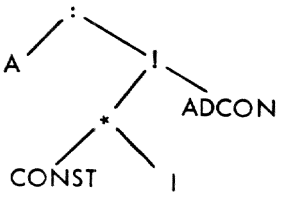
In this case, I is removable but is not an induction variable.



In this case, I is nonremovable.

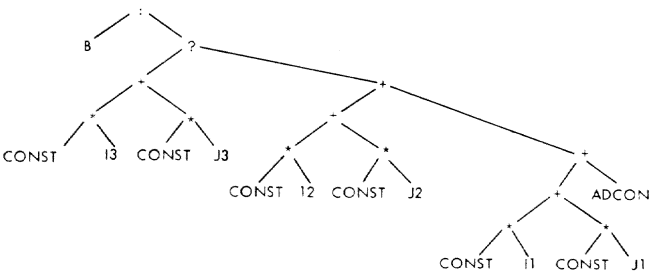


In this case, I is the induction variable of an inner loop with no calls.



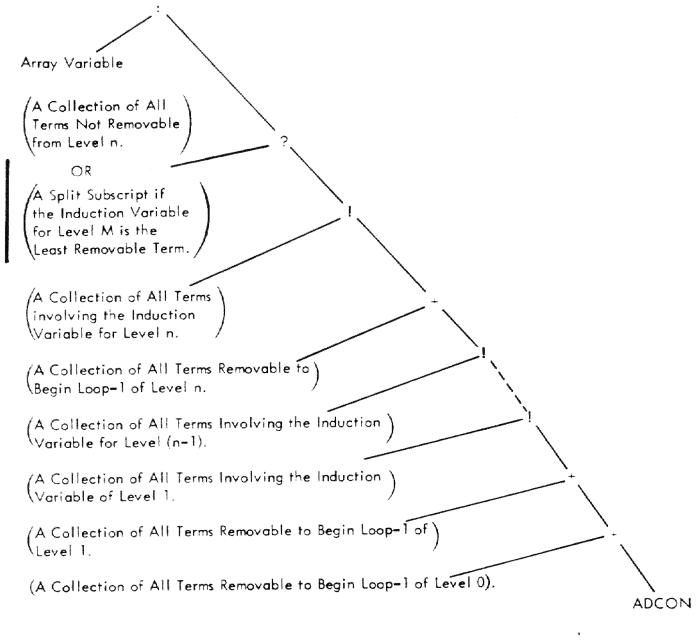
In this case, I is the induction variable, but the loop does not qualify to have a split subscript as in (c).

To show the gathering of terms according to removal level, the tree for the variable B(I1,J2,I3,J1,I2,J3) is shown:



I1 and J1 -- removable furthest to level 1; i.e., they are the most removable (the lower the removal level, the more removable the item is),
I2 and J2 -- removable to level 2,
I3 and J3 -- not removable.
None are induction variables.

Therefore, the completely general subscript expression for an array variable appearing at level n can be described as follows:



If there are no terms of the type indicated, the associated operator does not appear and, all terms below it are moved up. The only possible occurrence of the ? operator is immediately below the colon. Therefore, if there are any nonremovable terms, the right-hand operand of the colon is a question mark, since the adcon is always considered removable. Therefore, the operands of a question mark are always categorized in the same manner: the left-hand operand is the index expression and the right-hand operand is the base expression (always removable), thus setting up the base/index method of addressing for Phase 4.

The form shown above for the generalized subscript expression is used only internally to Phase 3. In the final form all removable and recursive expressions are replaced in the EF by a residue entry that indicates to Phase 4 the name of the removed expression. The removed expressions are then attached to the appropriate BL1 or BL2.

Two passes are made across each subscript expression in order to accomplish the optimization.

During the ERF scan the occurrence of a subscripted variable is indicated by a flag for that variable set by Phase 1. This flag is checked immediately after the adcon has been formed, and at that time the adcon is saved for later use. During the first scan, triads in the triad table are left unchanged with regard to occurrence flags, and new entries are marked as having no occurrences since the last compute point. During this pass no changes are made in the compute and remove item table. When a + is reached, the operand pushdown table contains entries representing a string of operands to be added to form the subscript expression (in addition to any earlier entries). The remaining + and : operators are ignored on this first scan, which is terminated at this time.

The rearrangement subroutine (CEKKM) determines the number of operands (including the one it inserts; i.e., the adcon) and sorts entries in the operand list. It then moves the ERF strings representing these operands to a temporary area. The program that moves them back in resorted order operates at two levels: one for the individual operand expression and another for each block of expressions that have the same sort key. At the end of each such block, an operator (+, !, or ?) is placed in a pushdown list and later used to connect this block to the others. Sufficient + operators are inserted in the string at the end of each block to connect the operands in the block. Tests are made to determine the location of the adcon. The main ERF scan is reset to start again at the beginning of the subscript expression. Phase 1 has inserted two extra + operators following the : to allow for the insertion of the adcon and the proper operator (+, ! or ?).

LOOP VARIABLE EXPRESSIONS

The induction variable is always given a forward compute point at begin loop-2 of its respective loop. When an induction variable is recognized inside a subscript, it causes the recursive operator as explained above. When an induction variable is recognized outside of a subscript, the materialize flag is set for the loop and special processing is applied to common expressions involving the induction variable. The special @ operator is also used whenever an induction variable occurs outside a subscript. This is necessary to distinguish references inside and outside its loop and to prevent erroneous marking of common expressions.

When the begin loop-3 entry is reached at the loop top, all expressions removed to begin loop-2 are examined. All those that do not have an exclamation point as the major operator are really BL3 items. A flag indicates whether any labels have occurred in the loop. If none have

occurred, the nonsubscript loop variable expressions are common within the loop and require no special treatment at the loop top, except to terminate the range of their commonality. If a label has occurred, these expressions must be computed at the loop top just inside the loop. They are reconstructed in the EF and attached to BL3 for this purpose.

The recursive expressions are attached to BL2, with modification to perform the recursion optimally. Each recursive expression is considered in two parts. The right-hand operand of the exclamation point can be an expression or an adcon; it represents the initial value of the expression. The left-hand operand represents the amount by which the recursive expression is stepped at the bottom of the loop. Phase 3 locates the induction variable in the step expression and replaces it with the step parameter of the loop.

When the materialize flag is on for a loop, Phase 4 tests for the end of the loop with the loop parameters. When the materialize flag is not on for a loop, Phase 3 creates a test expression from one of the recursive expressions for use in testing the end of the loop.

GLOBAL REGISTER ASSIGNMENT

The general registers are used for integer arithmetic expressions, subscript expressions, adcons for variables and constants, address constants for control transfers, and other purposes. Phase 3 considers certain uses of them for permanent assignment over one or more loops and if the most time can be saved this way, makes such permanent assignment. (The actual register to be used is determined by Phase 4.) A limit (currently 8) exists for the number that can be assigned to ensure that sufficient registers remain to allow Phase 4 to generate efficient code.

When the loop lists are initialized at any end loop PRF entry, a null chain is set up of candidates for global assignment over that loop. Whenever any expression occurs as a candidate, the chain is searched and that expression is found or inserted. A popularity count associated with this usage is added to the cumulative popularity count for that expression over the current loop. At the loop top (BL1) these entries are sorted by popularity, and the process of determining successful candidates is started.

The loop lists have a linked chain connecting all the loops one level higher (inner) and parallel to each other. The process of determining successful candidates at this level, readjusting the lists for these internal loops, and placing the results for Phase 4 requires that there be three passes over the chain of internal loops.

A first pass over the chain of parallel inner loops actually determines a successful candidate. For each inner loop, a test is made to determine whether the expression has already been assigned globally or if the total count of global registers is less than maximum. If neither condition prevails, the candidate cannot be assigned globally, and it is removed from the outer loop list. A count is kept for the outer loop (there may be no inner loops); and, if that count reaches the maximum, the remainder of the list is removed.

Immediately after the first pass, a second pass over the chain of inner loops is made for each successful candidate at an outer loop. This pass updates each internal list by increasing the count for that inner loop, if the candidate was not present in the inner loop.

The final pass over the chain of inner loops is made after the outer loop candidate list has been exhausted. This pass takes the assigned global expressions remaining for each inner loop (those that are not global in the outer loop) and links them into the chain of the outer loop. This makes a candidate available through an unsafe loop to level 0.

Phase 3 also selects, for Phase 4, one floating point constant or (in its backward scan) one floating point variable which is referenced in an inner, safe loop with no external calls. Phase 4 will load this quantity into a register outside the loop.

ROUTINE DESCRIPTIONS

Phase 3 routines bear only coded labels. These 5-character labels begin with the letters CEK; the fourth and fifth letters identify a specific module. Various entry points to a module are identified by a sixth character added to the coded label; for example, the coded label for the Phase 3 master control routine is CEKKR, and there are entry points CEKKRA and CEKKRE. Any mnemonic name beginning with the letters TEV refers to a compiler executive routine or entry name, rather than to a Phase 3 routine. The corresponding coded label is given in parentheses immediately following the mnemonic.

There are no hardware configuration requirements for any of the Phase 3 rou-

tines. All these routines are reentrant, nonresident, nonprivileged, and closed. All except the Phase 3 PSECT (CEKKS) and the Phase 3 Master Control Routine (CEKKR) use restricted linkage, are entered by the INVOKE macro instruction, and return to the calling routine by the RESUME macro instruction.

The relationships of routines constituting this phase are shown in the following nesting chart (Figure 22) and decision table (Table 23). The relationships are shown in terms of levels; a called routine is considered to be one level lower than the calling routine. Phase 3 Master Control routine is considered to be level 1.

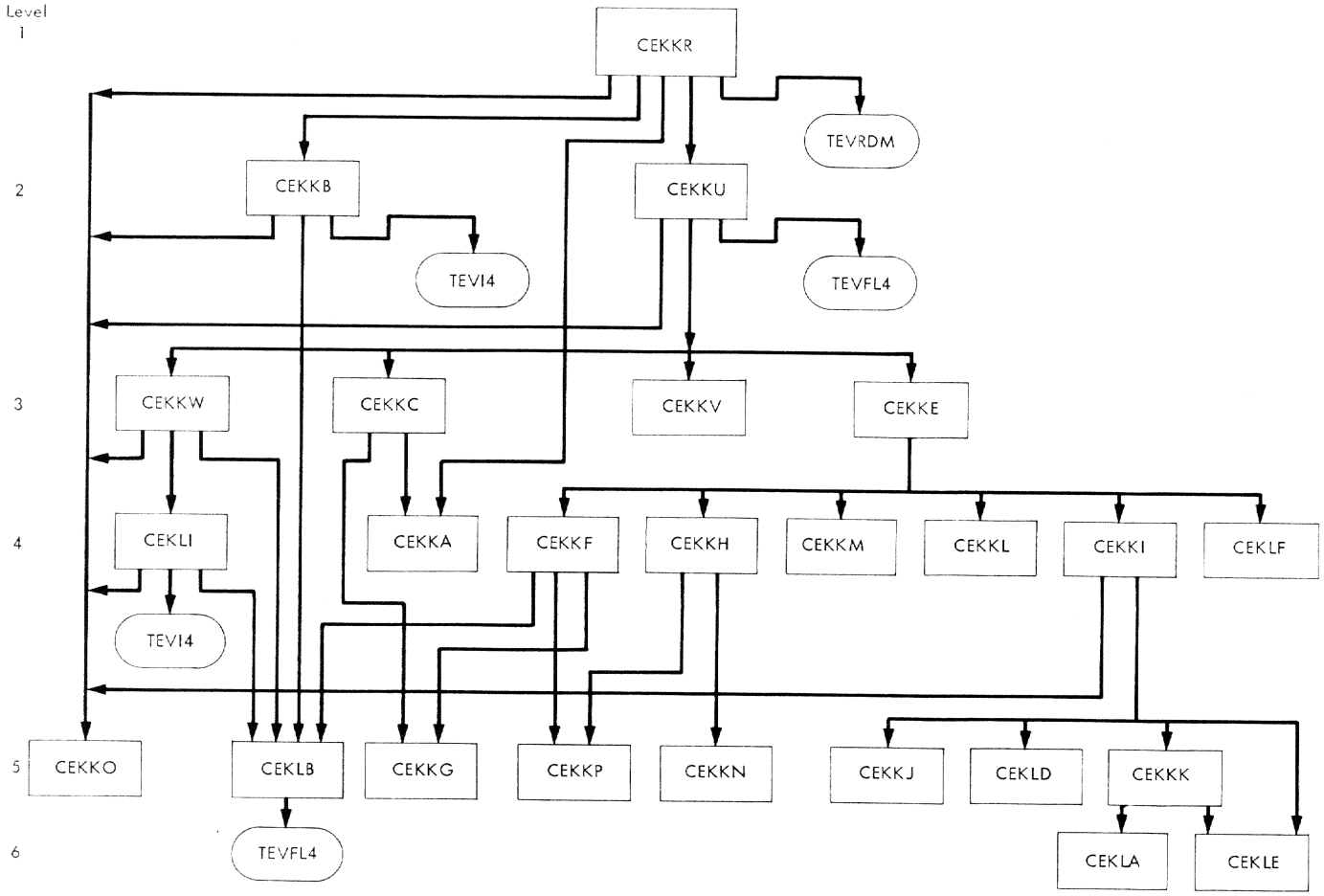


Figure 22. Phase 3 Nesting Chart

Table 23. Phase 3 Decision Table (Part 1 of 4)

Routine:----------Phase 3----------Level: 1 ----------

Routine	Usage	Called Routines	Calling Conditions
CEKKR	Directs the sequence of processing prior to editing of each of the PRF/ERF entries into PF/EF Output.	CEKKU	To edit each PRF entry.
		CEKKB	To generate removed, recursive, and induction variable expression at loop tops.
		CEKKA	To search the Compute and Remove Table for Triad entries pointing to the current PRF entry.
		TEVRDM	To print a diagnostic message when an error condition is encountered in any part of Phase 3.
		CEKKO	To tally popularity for global register assignment of recursive expressions.

Table 23. Phase 3 Decision Table (Part 2 of 4)

Routine:----------Phase 3----------Level: 2

Routine	Usage	Called Routines	Calling Conditions
CEKKU	To edit each PRF entry into an equivalent PF entry performing the necessary functions to accomplish this. In addition uses subroutines to edit the associated ERF entries into the EF format	CEKKE	When an ERF expression is to be processed.
		CEKKV	To process the Begin Loop-1 PRF entries.
		CEKKW	To process the Begin Loop-2 PRF entries.
		CEKKC	To process the End Loop PRF entries.
		CEKKO	To tally global register popularity for the code covering Adcon.
		TEVFL4 (CEKTPI)	To file a Symbol Table entry for an Adcon to cover branches in the object code.
CEKKB	To generate removed expressions at Begin Loop-1; recursive expression at Begin Loop-2; and induction variable expression at Begin	CEKLB	To file an Adcon entry in the Symbol Table covering a variable and compute the displacement.
		CEKKO	To tally the popularity for global register assignment for Adcons and integer expressions.
		TEVI4 (CEKTFC)	To file a Symbol Table entry for a new generated constant for recursive expressions.

Routine:----------Phase 3----------Level: 3

CEKKE	Controls the processing of an expression in the by the use of subroutines, and the editing into the EF.	CEKKF	Entered when a primitive is encountered in the ERF to generate the OPT1 entry and modify the ERF entry.
		CEKKH	To create a Triad entry when an operator ERF entry is encountered.
		CEKKI	Entered for each operand of an expression.
		CEKKM	Entered to process a subscript expression when the first plus is encountered.
		CEKKL	Entered when an expression's operands have been processed to generate an OPT1 entry for the expression.
		CEKLF	Entered when the complete expression has been processed to move it from the ERF to the EF.
CEKKW	To process the Begin Loop-2 PRF entry. If the loop is to be materialized, loop parameters are put into the EF; otherwise, a test expression is generated by subroutine.	CEKLI	Entered when a test expression is to be generated for the loop.
		CEKLB	Entered to file a covering Adcon and determine the displacement for each loop parameter when the loop is materialized.
		CEKKO	To tally popularity for global register assignment for the induction variable if the loop is materialized.
CEKKV	To process the Begin Loop-1 PRF entries. Determines which candidates are to be globally assigned.	None.	
CEKKC	To process the End Loop PRF entries	CEKKA	To find those Triads whose Forward Compute Point falls within the loop so they can be deleted.
		CEKKG	Entered for each loop parameter to determine the removal level and the forward compute point.

Table 23. Phase 3 Decision Table (Part 3 of 4)

Routine:----------Phase 3----------Level: 4 ----------

Routine	Usage	Called Routines	Calling Conditions
CEKKI	To process each operand of an expression. If it is primitive, the Adcon is considered for global assignment. If it is an expression, it is considered for removability or commonality.	CEKKJ	To determine if two expressions can be considered as common.
		CEKKK	Entered to name each common and removed expression.
		CEKLE	Entered for removable expressions to file as entry in Compute and Remove Table.
		CEKLD	Entered for removable expressions to replace the expression with a residue entry in the ERF.
		CEKKO	To tally popularity for global register assignment for Adcons and removable integer expressions.
CEKKA	To search the Compute and Remove Table for Triads which fall within the requested PRF limits.	None.	
CEKKL	To form an entry in the OPT1 representing an expression as an operand.	None.	
CEKLF	To copy the edited ERF entries for an expression from the ERF to the EF, inserting the newly created entries where indicated.	None.	
CEKKF	To update the Symbol Table entry of a variable (by subroutine), to change the ERF entry to reference an Adcon and displacement, and to form an entry in the OPT1 for the operand. Also, a dummy expression is generated for special conditions.	CEKKG	Entered for each variable ERF item to find its removal level and forward compute point.
		CEKLB	Entered for each variable ERF item to file a covering Adcon in the Symbol Table and compute the displacement.
			Entered when a dummy ("at") Triad is generated for an induction variable or a variable with an associated offset, to file the entry in the Triad Table.
CEKKH	To generate a Triad from an ERF operator and two operands in the OPT1.	CEKKN	Entered for all expressions except those inside subscripts to put the operands into canonical form.
		CEKKP	Entered for all expressions to file the Triad entry or locate its previous existence.
CEKKM	To revise a subscript expression to include the Adcon and optimize the loop variable and removed expressions.	None.	
CEKLI	To generate and insert into the EF a test expression to be used at bottom of a loop to test for the end conditions of the loop.	CEKLB	Entered for each loop parameter which not a constant to file a covering adcon in Symbol Table and compute displacement.
		CEKKO	To tally popularity for global register assignment for covering adcons and integer expressions.
		TEVI4 (CEKTFC)	To file a Symbol Table entry for a constant generated from the combination of other constant forms in the test expression.

Table 23. Phase 3 Decision Table (Part 4 of 4)

Routine:----------Phase 3----------Level: 5

Routine	Usage	Called Routines	Calling Conditions
CEKKJ	To determine if two occurences of an expression are common.	None.	
CEKLD	To replace a removed expression in the ERF with a residue entry.	None.	
CEKKO	To find a GIRL entry and add in the new popularity count or, if none already exist, to create a new GIRL entry.	None.	
CEKKN	To put the operands of an expression into canonical form to facilitate finding common expressions.	None.	
CEKKK	To assign a name to a common or removed expression, and by subroutine 1) put the name into the previous occurrence of a common expression, and 2) file a CRT entry for the expression.	CEKLA CEKLE	Entered when a previous occurrence of the expression is to be marked common in EF. Entered for every named expression to file an entry in the Compute and Removal Table at the forward compute point of the expression.
CEKKP	To enter new expressions into the Triad Table, locate common Triads, and delete obsolete Triads.	None.	
CEKKG	To determine the removal level and forward compute point for a variable and store the information in its Symbol Table entry.	None.	

Routine:----------Phase 3----------Level: 6

CEKLB	To file a covering Adcon entry in the Symbol Table, compute a variables's displacement, and file entries in the Formal Argument Adcon Table	TEVFL4 (CEKTFI)	Entered for each variable to file covering Adcon in the Symbol Table.
CEKLA	To replace a removed expression's ERF representation with a residue entry.	None.	
CEKLE	To file an entry in the Compute and Removal Table at the indicated PRF location.	None.	

CEKKS -- Phase 3 Storage (PSECT)

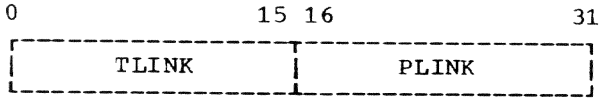
This routine supplies storage for Phase 3.

A brief description of most table and item formats follows. Most other variables will occupy one word. The Phase 3 Storage map is shown in Figure 23.

Phase 3 Loop Tables

1. LEV - 1 full word, contains the level of the current loop.

2. Current Nest Table (CNT) (Fixed, 55 entries long):



TLINK Loop in current nest, this level (PLP)
PLINK Chain of parallel loops, next level (PLP)

CNT is 224 bytes; the 55 allowable entries require 220 bytes, and the false loop over the whole program accounts for the remaining four bytes.

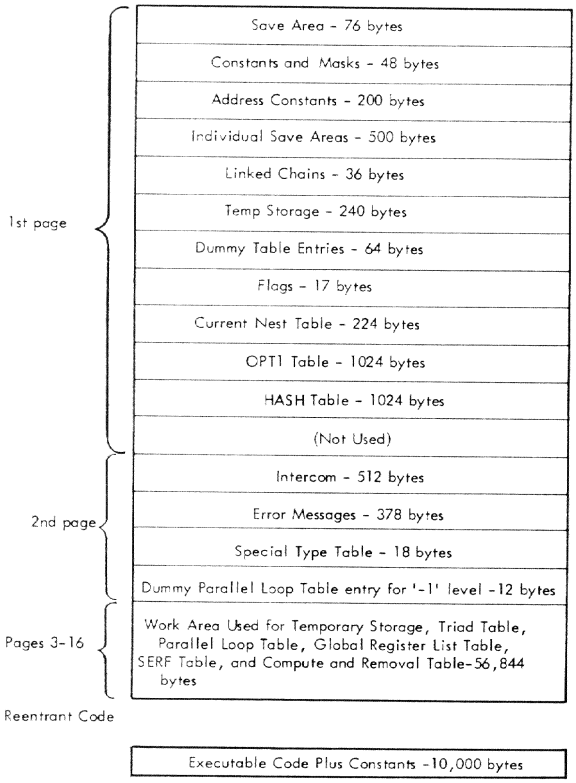
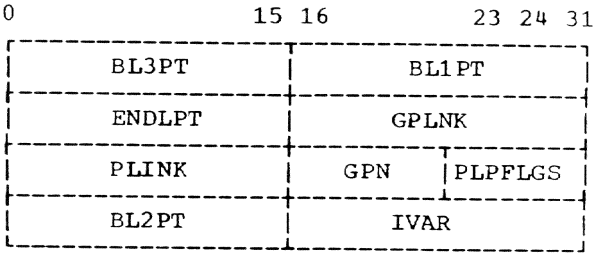


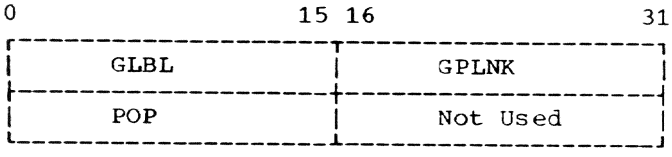
Figure 23. Phase 3 Storage Map

3. Parallel Loop Table (PLP)



BL1PT,BL2PT, BL3PT — Begin-loop entries (PRF)
ENDLPT — End-loop entry (PRF)
GPLNK — Chain of global register expressions
PLINK — Link in parallel loop chain
GPN — Number of global register expressions
PLPFLGS — 80 = Labels occurred in the loop
40 = Unsafe loop
20 = Materialize loop variable flag
10 = Parameter
08 = Global flag
04 = BXLEREC flag
02 = ONEASN flag
IVAR — Symbol Table entry of the induction variable

Global Register List (GIRL) (Linked, Permanent)



GLBL The name of a global expression for this loop, or a Symbol Table pointer for an Adcon. The name will have 7000_1 added to it to distinguish it from a pointer

GPLNK Link in global register chain

POP Popularity count for candidates

Link Pointers

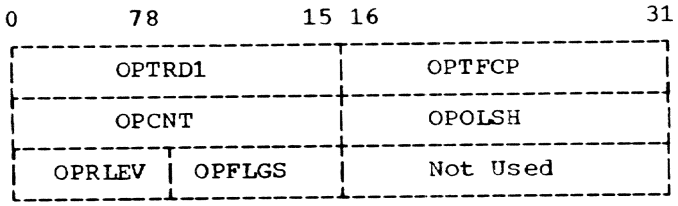
During the PRF scan in Phase 3, chains of PRF items occur in pairs, one going forward (unprocessed) and one going backwards (has been relinked by Phase 3 in opposite direction). Phase 3 keeps a pair of pointers to head each chain. Compute point entries are processed before the PRF entry to which they are attached.

Chain	PRF Field	Forward Head	Backward Head
Variable Definition	VDP	FDP (SYM)[1]	BDP (SYM)[1]
Common Definition	CDP	CFDP	CBDP
PRF Entries	ILNK	FLINK	BLINK
Label Definitions	LLNK	LFDP	LBDP

[1]These fields exist for each variable and are in the symbol table entry for that variable.

Operand Pushdown

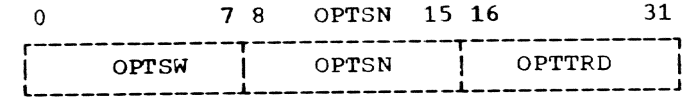
Type 1 Entry - For Generating Triad Table



OPTRD1 Reference to Triad Table or Symbol Table
OPTFCP Forward Compute Point
OPTCNT Length of expression in Polish
OPOLSH ERF pointer to Polish expression (right end)
OPRLEV Removal Level
OPFLGS There are seven flags:

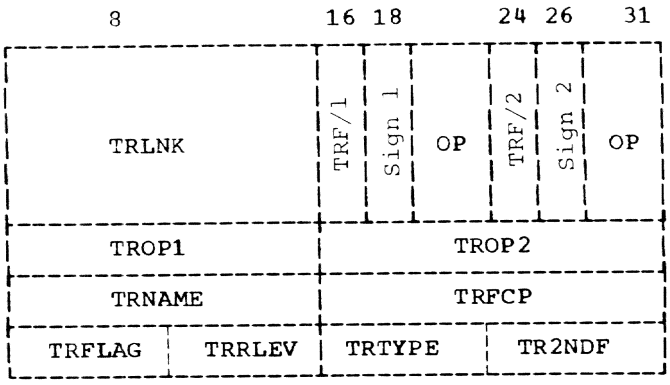
Name	Bit	Setting	Meaning
OPFI	8	0	OPTRD1 is a Symbol Table Pointer = Primitive
		1	OPTRD1 is a Triad Table Pointer
OPSIGN	9	1	Sign Flag
OPLVF	10	1	Loop Variable Flag
OPPLF	11	1	ERF Insert Flag
ATF	13	1	At Operator Flag
RSEF	14	1	Removable Subexpression Flag
IVARF	15	1	Induction Variable Flag

Type 2 Entry - For Generating Expression File Entries for Removed Expressions



OPTSW Switch, used to determine the stage of processing an item
OPTSN Sign, used to store sign of the operand
OPTTRD A Triad or Symbol Table pointer (determined by OPTSW setting)

Triad Table Entry



TRLNK Links to next entry in a chain from a hash table.
TRFI1, TRFI2 File Indicator 1, File Indicator 2
SIGN1, SIGN2 Sign for OP1 and OP2, respectively
OP Operator--Same as code in EF Form 2 format
TROP1 First operand (Triad or Symbol) [for : and @, displacement]
TROP2 Second operand (Triad or Symbol) For : and @, variable (symbol). Also see TRTYPE, below.

NAMEF = 0, link in chain of
OFLAG = 1 Triads for last occurrence here.
NAMEF = 0, link to new Polish
QFLAG = 0 expression for last occurrence.
NAMEF = 1 name of this expression.

TRFCP Forward compute point (PRF)
TRFLAG See TRTYPE, below.
TRRLEV Removal level
TRTYPE The ID field from the ERF entry of the operator is saved to determine the type of expression.

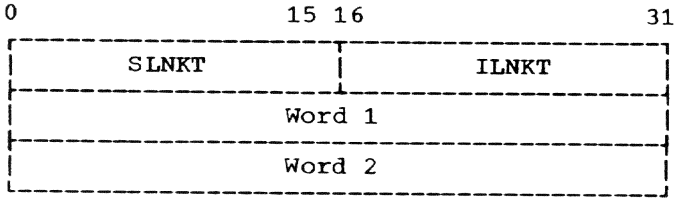
(See Code and Type fields in the EFID in EF form 1 format)

The search key for entering the triad table consists of the seven fields: OP, OP1, OP2, FI1, FI2, SIGN1, and SIGN2.

	Flag Name	Bit Position	0 Meaning	1 Meaning
TRFLAG	ZEROF	X'80'	No Reference since Compute point	Other
	FIRSTF	X'40'	Other	First reference since compute point
	NAMEF	X'20'	(See NAME above)	
	QFLAG	X'10'	Other	This EF location to be saved in Triad
	REMOVF	X'08'	Other	Has been removed
	LUF	X'04'	Other	EF 'Last-use' bit has been set
	NCOMPF	X'02'	Other	Expression not removable on its own
	COMAF	X'01'	Other	Operator is a comma, double comma, or question mark
TR2NDF	FRCFLG	X'80'	Other	Exp. must be generated at BL3.
	CPFLAG	X'40'	Other	Level Zero removable expression.
	CRTF	X'20'	Other	
	TRLVF	X'10'	Other	
	SPLTTRD	X'08'	Other	Special BXLE or Recursive Triad
	COMAF	X'01'	Other	

Polish Insertion Entries

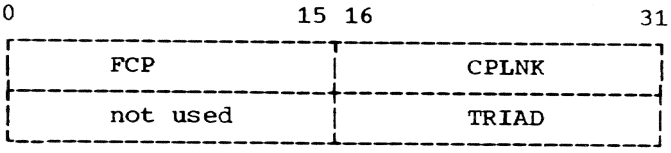
When an integer is an operand of a non-integer expression, Phase 3 inserts a float operator by means of two entries in the expression file, a primitive "FLOAT" connected by the operator "FUNCTION." Prior to the time that the ERF is copied to the PF, insertions are indicated by entries in a linked file SERF. An entry to this file is as follows:



SLINKT Link to next entry in this chain
ILNKT Link to ERF entry this insert precedes

Word 1 and WORD 2 A normal ERF entry

Compute and Removal Item Table

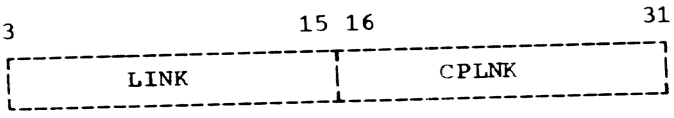


FCP Forward compute point (or removal point)
CPLINK Link to other entries
TRIAD Expression in Triad Table

For insertions a hash table is entered, using the low order n-bits of FCP and linking to a chain. Entries in the chain are sorted on FCP (highest first). Later insertions precede earlier insertions. The table is scanned by a pointer which is synchronized with the PRF scan for removal of entries at the proper time. Removal entries are distinguished by having FCP at Begin Loop-1 PRF entries.

Hash Table for Compute and Removal Table (CRT) and Triad Entries (HCRT)

Those entries serve as dummy first entries for the linked chains of CRT entries and for the linked chains of triad entries. This table has 256 entries.

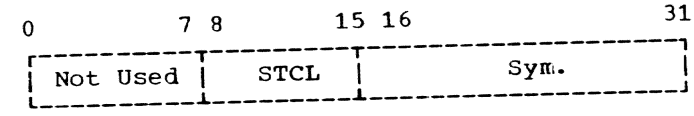


LINK — For Triad items the fields OP1 and OP2 are added together, multiplied by 4, and the low-order 12 bits of that result are used as the index into this table.

CPLINK — For CRT items the low-order 12 bits in a PRF address form the index into this table.

Formal Argument Adcon Table

Some adcons assigned to storage class 4 are actually not constants but are variables which must be computed by the preamble at any entrance of the subroutine. These are the adcons referring to storage classes 128 through 253 reserved for the parameters (dummy arguments), one per argument. In order to specify these adcons to Phase 4 for preamble generations, a list is prepared by Phase 3. The entries to this list are



STCL — Storage Class

Sym — Symbol Table pointer for the Adcon

CEKKR -- Phase 3 Master Control Routine

This is the entry point from the Exec. The intercom area is initialized. The PSECT is moved into the GETMAIN area, and the adcons that point to areas within the PSECT are relocated. The work area is initialized, necessary parameters stored, and proper registers filled. The proper subroutines are entered for processing each PRF item. These items are scanned in reverse order. When the end of the PRF is reached, CEKKU sets a flag, and CEKKR returns to the Exec. All errors found by other routines of Phase 3 are handled by a special entry in CEKKR. See Chart CV.

ENTRIES:

CEKKRA — This is the point where the Exec enters Phase 3 by a standard linkage (CALL macro instruction).

CEKKRE — This is the error exit for all routines within Phase 3. The entry is made by restricted linkage (INVOKE macro instruction). The only parameter is in register P2. This is a pointer to the error message parameter list.

EXITS: The routine exits to the Exec by a standard linkage (RETURN macro instruction). The value in register L3 indicates whether it is a normal return (value = 3) or an abort return (value = 8).

The routine detects no error conditions of its own, but does handle the errors of all the other routines of Phase 3.

OPERATION: Upon entry, the standard procedure is used to save the registers and locate the PSECT belonging to Phase 3. The intercom area is moved from the exec's PSECT to Phase 3's PSECT. Register N1 is loaded with the intercom location. After the PSECT has been moved into the GETMAIN area, the location of the first page of the GETMAIN area is loaded into register L1. The location of the second page of the GETMAIN area, which is the new intercom location, is loaded into register N1. Register N2 is loaded with the first available location in the working storage area. The location of the first entry in the PRF is calculated and put into register P5. The location of the first available word in the PF is calculated and put into P6. The limit of the PF is calculated and saved in PFLIM.

The CEKKA subroutine is invoked, for each PRF item, until no qualifying entry in the compute and removal table is found. For each CRT entry returned by CEKKA, CEKKB is entered if the current PRF entry is a Begin Loop 1, 2, or 3 item. If it is not, the CRT entry is deleted from the table.

Each begin loop 3 item goes through the CEKKA circuit twice, first pointing to itself then pointing to the begin loop 2 PRF entry. On the first pass, a qualifying CRT entry returned from CEKKA is deleted if the forward compute point is less than the current PRF location, or if the FRCFLG is not on in the triad entry and the "label"

flag is zero in the PLP entry. Otherwise, CEKKB is entered to generate the expression. On the second pass, the CRT entry is deleted if the operator of the traid is not an ! (indicating a recursive expression). For the recursive expression triads, a flag is set indicating that this loop need not be materialized. Then, the popularity of the expression is increased by CEKKO. When CEKKA returns a zero in register P2, the flag for the first begin loop 3 pass (HOLDB3) is checked. If it is on, the second pass is set up and the flag turned off. During the second BL3 pass, all recursives are examined to determine if there is a 'BXLE on recursive' candidate. The following checks are made:

1. The recursive must be a special split subscript. That is, the least removable part of the subscript must be the induction variable.

2. The step on the recursive must be a positive constant. Note that the loop step is not included in the subscript expression at this time. CEKKB inserts it at BL2 processing.

3. The loop step must be a constant.

4. The loop must not be marked for materialization.

Of all the candidates passing the requirements, the one with the highest popularity count is retained.

At the end of the second pass, if a BXLE recursive was selected, the CRT entries are relinked so that the selected recursive is the one last seen by Phase 4, and the one for which the test expression will be created by CEKLI.

CEKKO is then called to increase the popularity of the recursive to the maximum, to ensure its global assignment by CEKKV. When both passes at begin loop 3 have been completed, the materialized byte is checked to see if a recursive expression was found. If none was found and the global flag is off, or if the ISD option is on in intercom, the materialize flag is set in the PLP item.

For each completion of the CEKKA circuit, the ID of the PRF entry is checked for a "label" or an "alternate entry" item. For labels, the "referenced" flag in the symbol table is tested. If it is on, CEKKA is entered and all the CRT entries deleted. For alternate entries, all CRT entries are deleted by using CEKKA and pointing to X'7FFF' or to the absolute beginning of the PRF. No triads can be carried past an alternate entry. Upon completion, or if the PRF entry was not one of those two, the CEKKU routine is invoked. Upon return from CEKKU, the end flag (ENDIT) is checked. If it is off, the routine returns to enter CEKKA again for the next PRF item. If the end flag is on, Phase 3 ends. The registers are restored, and return is made to the executive by standard linkage (RETURN macro instruction).

At the entry point for errors (CEKKRE), the executive subroutine CEKTE is called to output the error message. Upon return, register L3 is set to 8, to indicate an abort condition. The routine exits through the same procedure as for a normal exit.

CEKKU -- PRF Processing Routine

This routine manipulates the PRF entry into its proper PF format, performs any necessary linking, and writes the entry into the PF. See Chart CW.

ENTRIES: The entry point is CEKKUA. Register P5 contains the location of the current PRF entry, register P6 contains the location of the next available word in the PF, register N1 covers the Intercom area, and register L1 covers the work area.

EXITS: Register P5 contains the location of the next PRF entry, register P6 contains the location of the next PF word, register N1 covers the intercom area, register N2 covers the unused working storage, and register L1 covers the work area.

Two error conditions are detected:

1. An illegal ID in the PRF.

2. The PF table overflowed.

For all cases register P2 is set with the location of an error message parameter list in the PSECT, and the standard phase error processor (CEKKRE) is invoked.

OPERATION: The registers are saved by the STORE macro. The ID code of the PRF is converted to an index by multiplying it by four. An internal branch table is used to direct the routine to the proper processing section. If the ID is zero, or greater than the maximum, the error exit is taken.

Each item is processed by rearranging (if necessary) the fields of the PRF to conform to the PF format. In many cases entire fields are deleted; in other cases fields are modified (refer to the format diagrams of the PF items in Appendix A).

1. Begin Program (ID = X'1'). The end flag (ENDIT) is set. If the program being compiled is a main program, a dummy ENTRY statement is inserted into the PF. Return is to CEKKR.

2. Enter (ID = X'2') and Alternate Entry (ID = X'3'). For a subprogram entry (ID = 2) the end flag (ENDIT) is set to mark the end of the PRF. Each argument symbol table index (ASTX) is used to obtain the corresponding storage class (STCL) from the symbol table. These are stored in the PRF. The label relink (a Phase 3 internal subroutine described below) subroutine is entered. Upon return, the length of the PF entry is computed and the Variable Move routine (a Phase 3 internal subroutine, described below) is entered.

3. Label Definition (ID = X'4'). The label relink subroutine is entered. Upon return, the label flag bit in the PLP entry of the current loop level is turned on if the "Reference" flag is on in the symbol table. The fields are packed and the Two-Word Move routine (a Phase 3 internal subroutine, described below) is entered.

4. Equation (ID = X'5'). EXSN (expression scan subroutine, a Phase 3 internal subroutine, described below) is entered for OPD1. Upon return, the symbol table location stored by CEKKE is used to enter the Variable Relink subroutine. EXSN is then entered for OPD2. Upon return, the Common Relink subroutine is entered. Upon return, the fields are rearranged and the Four-Word Move routine entered. (Variable Relink, Common Relink, and Four-Word Move are all Phase 3 internal subroutines, described below.)

5. GO TO (ID = X'6'). The symbol table index of the label (LLNO) is used to enter the Adcon Assignment subroutine. Upon return, the fields are rearranged and the Three-Word Move routine is entered. (Adcon Assignment and Three-Word Move are both Phase 3 internal subroutines and are described below.)

6. Assigned GO TO (ID = X'7'). The number of label elements (NOEL) is used to find the line-number word (LINO), which is moved to the third word of the entry. The OPD field is used to enter EXSN. The AVAR field is moved up two bytes, and the Three-Word Move routine is entered.

7. Computed GO TO (ID = X'8'). The OPD field is used to enter EXSN. The fields are rearranged, with the LLNOs being packed into two bytes. The new length is computed, and the Variable Move routine entered.

8. ASSIGN (ID = X'9'). The OPD is used as a parameter for entry to the EXSN routine. On return, the Three-Word Move routine is entered.

9. Arithmetic IF (ID = X'A'). EXSN is entered with the test value (TVAL). Upon return the Common Relink subroutine is entered. Upon return, the Adcon Assignment subroutine is entered for the three branch points. The fields are then rearranged and the length set at 20 bytes. The variable move is then entered.

10. Logical IF (ID = X'B'). EXSN is entered with the test value. Upon return, the Common Relink subroutine is entered. The last two words are moved up one word. The Adcon Assignment subroutine is entered. Then the Four-Word Move routine is entered.

11. CALL (ID = X'C'). The fields are rearranged, with the LLNOs being packed into two bytes. The new length is computed, and the Variable Move routine entered.

12. Argument Definition Point (ID = X'D'). The VAR field is converted to a symbol table location, and the Variable Relink subroutine is entered. Upon return, the Two-Word Move routine is entered.

13. RETURN (ID = X'E'). The RIND field is checked. If it is nonzero, the RVAR field is used as a parameter to enter the CEKKE routine. Upon return, or if RIND was zero, the RVAR field is moved forward two bytes, and the Three-Word Move routine entered.

14. Begin Loop 1 (ID = X'F'). CEKKV is entered. Upon return, the length is set at 28 bytes and the Variable Move routine entered.

15. Begin Loop 2 (ID = X'10'). CEKKW is entered. Upon return, the Four-Word Move routine is entered. The Four-Word Move routine adds the IVAR and EXITLB pointers to make the BL2 a five-word entry.

16. Begin Loop 3 (ID = X'11'). The RMVAL word is obtained from the PSECT and put into the PRF entry. A hexadecimal 8000 is put into RMVAL. The subroutine to relink common is then entered. On return, the CDP and GLAB fields are rearranged to the PF format. If the level is not zero, the Adcon Assignment subroutine is entered. The flags are then checked. If the 'global' flag is on, indicating an inner loop with no external calls, and the loop is safe, the symbol table pointer for the current floating point candidate

is put in the PF. The candidate pointer is then replaced with X'8000'. The Three-Word Move routine is entered.

17. End Loop (ID X'12'). CEKKC is entered. The symbol table pointer for the current floating point candidate is replaced with X'8000'. Upon return, the One-Word Move routine is entered. (One-Word Move is a Phase 3 internal subroutine and is described below.)

18. CONTINUE (ID = X'13'). No processing is necessary, so the Two-Word Move routine is entered directly.

19. READ (ID = X'14') and WRITE (ID = X'17'). The flag field is checked to see if the LABF field is an EF pointer. If it is, the EXSN routine is entered. Upon return or if it was not an EF pointer, the UNIT field is used to enter EXSN. Upon return, the ID is checked. If it is a WRITE entry, the Three-Word Move routine is entered. For a READ, the ERR and EOF fields are packed into one word and the Four-Word Move routine entered.

20. READ (ID = X'16'), PRINT (ID = X'19'), and PUNCH (ID = X'1A'). The flag field is checked. If it indicates that the LABF is an EF pointer, the EXSN routine is entered. Upon return or if LABF is a symbol table pointer, the Three-Word Move routine is entered.

21. READ with NAMELIST (ID = X'15') and WRITE with NAMELIST (ID = X'18'). The UNIT field is used as a parameter to enter the EXSN routine. If the entry is a WRITE with NAMELIST, the Three-Word Move routine is entered. For the READ with NAMELIST entry the fields are rearranged, with ERR and EOF being packed into two bytes. Then the Four-Word Move routine is entered.

22. Output List Element (ID = X'1B'). EXSN is entered with OPD. Upon return, the Two-Word Move routine is entered.

23. End List (ID = X'1C'). The One-Word Move routine is entered directly.

24. File Control (ID = X'1D'). The UNIT field is used as the parameter on an entry to EXSN.

25. STOP (ID = X'1E') and PAUSE 'ID = X'1F'). The Three-Word Move routine is entered directly.

26. End Program (ID = X'20'). The One-Word Move routine is entered directly.

27. Input List Element (ID = X'21'). EXSN is entered. Upon return, the symbol table location generated by CEKKE is used to enter the Variable Relink subroutine. Upon return, the Common Relink subroutine is entered. When finished, the Three-Word Move routine is entered.

28. Adcon Assignment Subroutine. The symbol table pointer, assumed to be in register P2, is converted to an address. The storage class of the entry is checked. If it is equal to 255, the routine returns to the caller. If not equal to 255, it is set to 255. Register P2 is set to zero, and CEKLB is entered to file the constant. The pointer to the symbol table entry for the adcon is moved from the PSECT (TEPNTR) to the symbol table entry being processed. The parameters are set, and CEKKO is entered to tally the popularity of the adcon for global register assignment. Return is to the calling routine.

29. Label Relink Subroutine. The pointer of the current PRF entry (FPT) is compared to the forward label link (LFDP). If they are not equal, the routine returns. If they are equal, the label field of the PRF (LINK) is converted to the PF chain, and the current LLNK saved as LFDP.

30. Common Relink Subroutine. The pointer of the PRF entry is compared to the forward common link (CFDP). If they are equal, the PRF field is chained and the PRF pointer is saved. The routine then returns to the caller. If they are not equal, the PRF pointer is compared to the forward formal argument link (AFDP). If these are not equal, the routine returns. If these are equal, the PRF field is linked, and the PRF link is saved.

31. Variable Relink Subroutine. The PRF pointer is compared to the forward definition field in the indicated symbol table entry. If they are not equal, the routine exits. If they are equal, the forward and backward definition fields of the symbol table are relinked, and the forward link saved.

32. One-Word, Two-Word, Three-Word, and Four-Word Move Routines. The PF location is moved to register P3 and increased by the indicated number of bytes (4, 8, 12, or 16). Then the Special Move subroutine is entered.

Upon return, the proper number of words is transferred from the PRF table to the PF table. The PF location is then updated by the proper number of bytes. The Exit Routine is then entered.

33. Variable Move Routine. The indicated length (i.e., number of bytes to be moved) is put on the next full-word boundary. The PF location is put into register P3 and the length added to it. The Special Move subroutine is then entered. Upon return PRF entry is moved in blocks of 256 bytes or fewer. The PF location is then increased by the length, and the Exit Routine entered.

34. Special Move Subroutine. The extent of the new PF entry (register P3) is compared to the PL limit (LIMSAV) for PF table overflow. If the table limit is exceeded, the error exit is taken. If not, the PF link field (ILINK) is updated, and return is made to the calling routine.

35. EXSN - Expression Scan Subroutine. The ERF location is obtained from the indicated location, and CEKKE is entered. Upon return, the EF location is stored in the indicated field. The PRF location is restored, and return is to the entering routine.

36. Exit Routine. The forward and backward links for the PRF and PF are updated. The current PRF location is calculated. The routine then exits to CEKKR by a restricted linkage (RESUME macro instruction).

CEKKC -- End Loop PRF Entry Routine

This routine processes the end loop entries in the PRF table. It sets up the loop tables for the loop, terminates commonality of expressions as required, marks loop variables appropriately, and determines their compute points if necessary. See Chart CX.

ENTRIES: The entry point is CEKKCA. Register P5 contains the location of the current PRF entry, register N1 covers the intercom area, register N2 contains the location of the first available word in the working storage area, and register L1 covers the work area.

EXITS: Registers N3 through L2 are restored. Registers P5, P6, and N1 return unchanged. Register N2 reflects any use of working storage by pointing to the new first-available word.

This routine uses the working storage area of the PSECT. An error condition exists if this area is overflowed. The Phase 3 error exit (CEKKRE) is taken.

OPERATION: The level (LEV - see "Phase 3 Loop Tables") is increased by 1, and a new PLP entry is generated and linked into the CNT Table. If the current loop is safe, all entries in the CRT (found by CEKKA) below the BL3 of the loop are deleted. If the loop is unsafe, level zero is checked. If level zero is safe, all CRT entries to BL3 of level zero are deleted.

The "active induction variable" flag is turned on in the symbol table entry of the loop induction variable. The current level is stored as ULEV, and the forward compute point is set at BL2 of the loop. For the three loop parameters, CEKKG is entered to set the ULEV and FCP. Upon return, these entries are checked to ensure that the ULEV is at least the current loop and FCP is at BL1 of the loop. Also, if any of the loop parameters is an induction variable for a previous loop, that loop is set for materialization.

CEKKV -- Begin Loop 1 PRF Processor

This routine processes the begin loop 1 entries in the PRF. The global register candidates are computed and moved into the PRF. Since this is the end of the processing for a loop, the level (LEV - see "Phase 3 Loop Tables") is reduced by 1. See Chart CY.

ENTRIES: The entry point is CEKKVA. Register P5 contains the location of the current PRF entry, register N1 covers the intercom area, and register L1 covers the work area.

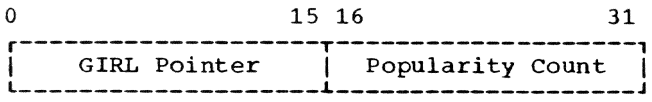
EXITS: Registers P6, N1, N2, and L1 are the same as when entered. Register P5 points to the BL1 work area in the PSECT (BL1WORK).

This routine uses the working storage area of the PSECT. If this area is overflowed, an error condition exists. The Phase 3 error exit (CEKKRE) is taken.

OPERATION: Upon entry, the 2-word PRF BL1 item is moved from the PRF to the BL1 work area in the PSECT, and register P5 is loaded with that location. (This is done because the BL1 entry to be created for the PF will be 28 bytes long. If the PRF is near the end of the allotted area, working in the PRF could cause errors by accessing past the end on the legitimate PRF area.) The line number is moved from the second word to the seventh. The loop level (LEV) is used to find the PLP location through the CNT table. The PLP flags are moved to

the PRF entry. The global register candidate counter (GPN) is saved and the field cleared in the PLP. If the original GPN is zero, there are no global register assignments, and the routine skips to the end.

For the sort, a temporary table (the first entry of which is set to zero) is set up in working storage with the following format for each entry:



THE GPLINK chain is followed from the PLP. The GIRL entries are then picked up one at a time to be stored in the table at the point where the popularity count of the next item in the table is less than the popularity count of the current GIRL entry. The table is pushed down at that point, and the new entry is inserted. The entry for the first or lowest negative popularity is stored immediately preceding the end of the table. When the end-of-chain is encountered, the sort is completed. The GPLINK chain is now relinked so that the entries are encountered in order of decreasing popularity.

Now each candidate is checked to see if global assignment can be accomplished. In order to be globally assigned, each inner loop must have the same candidate already globally assigned, or have fewer than the maximum number of registers assigned (8). Starting with the most popular candidate, each inner loop is checked to see if there can be global assignment. If there cannot be, the candidate is deleted from the GPLINK chain. If there can, the GPN of the current loop is increased by 1. Also, each loop in which it is not already globally assigned has its GPN increased by 1.

When either the candidate list is exhausted or the GPN of the current loop reaches the maximum, the search is completed. An X'8000' is stored in the GPLINK of the last entry as the end-of-chain. The GLBL fields in the BL1 entry are preset to X'8000'. Then the GPLINK chain is followed to the end storing the GLBL field into the BL1 entry.

For each of the parallel inner loops, the GPLINK chain is followed. Each entry is compared to each GPLINK entry in the current loop. If an entry is found in an inner loop which is not in the current chain, it is linked into the current chain, and the GPN is increased by 1. This allows a GLBL to go from level "n" to level "n-2" with no occurrences in level "n-1".

Since this is the end of the loop processing, the CNT table entry is set to X'8000' for both the TLINK and PLINK fields. The pointer to the removed expression generated by CEKKB is moved from RMVAL in the PSECT to the BL1 entry and X'8000' is stored in RMVAL. The loop level (LEV) is reduced by 1. The exit is then taken.

CEKKW -- Begin Loop 2 PRF Processor

This routine processes the begin loop 2 entries in the PRF. The EF entries for the loop parameters generated for materialized loops. If a test value exists, the test expression is generated by entering CEKLI. See Chart CZ.

ENTRIES: The entry point is CEKKWA. Register P5 contains the location of the current PRF entry, register N1 covers the intercom area, and register L1 covers the work area.

EXITS: Register P5, P6, N1, and N2 are the same as when entered. Registers N3 through L2 are saved and restored.

Two errors are detected:

1. The PF table overflowed.
2. An illegal type code found in the symbol table.

For all cases the Phase 3 error exit (CEKKRE) is taken.

OPERATION: The registers are saved by the STORE macro instruction. The link to the Polish chain (RMVAL) is moved from the PSECT to the PRF, and the value X'8000' is stored in RMVAL. The symbol table pointer of the induction variable (IVAR) is converted to a symbol table location. The pointer to the current PRF entry is compared to the forward definition point (FDP) of the symbol table. If they are equal, the FDP and BDP fields are relinked.

The PLP flags are checked to determine which path the processing then follows:

1. If the materialize flag is on, the symbol table pointer for IVAR is converted to a location. The PRF pointer is stored in the symbol table entry as the forward compute point. The undefined level (UNLEV) field is set to 55, and DUNL (a Phase 3 internal subroutine) is entered.

   The expression file and DUNL subroutines are entered for the initial value (BEG), the final value (END), and the increment value (INC). The link to each expression is stored in the proper place in the PRF.

2. If the GLOBAL flag is on, but the BXLE on recursive flag is off, the EXITLB field is checked. If EXITLB is not equal to X'8000', the materialize flag

is set; in either case, processing continues as in 1.

3. If the BXLE on recursive flag is on, the constant step on the recursive is deleted as a global assignment candidate. This is accomplished by calling CEKKO with a zero weight, then returning to CEKKO with a weight equal to (-TOTPOP). If EXITLB is not X'8000', a PF entry is generated for the induction variable and the initial value via FEFP. The adcon for the exit label then has its popularity reduced by 5.

   CEKLI is then called to generate the test expression for the recursive.

4. If none of the above conditions prevail, CEKLI is called to generate a test expression.

The routine then exits.

File EF and Point (FEFP) Subroutine

This subroutine (internal to CEKKW) is used to file an adcon in the symbol table for the loop variables. The routine is entered with a symbol table pointer in register P1, which is converted to a location in register P4. The location (SLOC) is divided by 4096 (to put it on a page boundary). The remainder is stored as the displacement of the EF entry. The result plus the storage class (STCL) are stored for the executive subroutine.

Then the subroutine CEKTFI is entered to file the adcon. The cell TEPNTR contains the pointer to the symbol table. This is stored in the EF entry. The type and ID are also stored in the EF entry. The pointer to the EF is put into register P2. Register P6 (the PF location) is updated by two words. Return is then made to the calling routine.

Delete the Undefined Level (DUNL) Subroutine

This internal subroutine resets the undefined level (ULEV). It is entered with the symbol table location in register P4. The variable class is found, and its ULEV checked against the loop level. If ULEV is equal to or greater than the loop level, ULEV, is set to 55. If ULEV is smaller, the forward compute point is checked against FLINK. If FCP is less than or equal to FLINK, the ULEV is set to 55, and the routine exits. If FCP is greater than FLINK, the routine exits.

CEKKE -- Expression Scan Routine

This is the main control routine for a series of subroutines which scan an entry in the expression file (ERF), put it into canonical form, generate or locate triad table entries for all expressions and subexpressions, update symbol table entries as necessary, determine commonality and removability, rearrange and expand subscript expressions, generate adcon entries as needed, and rewrite the ERF as part of the program file (PF). See Chart DA.

ENTRIES: The entry point is CEKKEA. Register P2 points to the last ERF entry of the expression to be processed. Register P5 contains the location of the current PRF entry. Register P6 contains the location in the PF.

EXITS: Register P2 points to the last ERF entry made in the PF. Register N1 is returned unchanged. Register P6 points to the new location in the PF. Register N2 points to the new first available word of working storage.

OPERATION: Upon entry the ERF pointer, passed in Register P2, is converted into an ERF location. If the pointer is an X'8000', the exit is taken. It is assumed that the location is the "right-end" or major operator of an expression, or a primitive (a constant or variable). A counter is set to 1, and the expression is scanned backward (from high core to low) to find the "left-end" (or beginning). The ID of each entry in the ERF is examined. If it is an operator, the counter is increased by 1. If it is a primitive (anything other than an operator), the counter is reduced by 1, the ERF location is reduced by eight bytes, and the next entry is checked. When the counter reaches zero, the beginning of the expression has been found, and the ERF location is saved in LOWERF. When a subscript expression is being scanned, special conditions exist. Since CEKKM must insert two entries into the ERF, two extra plus operators have been inserted by Phase 1 immediately preceding the colon operator. Therefore, when a colon is encountered during a scan, the ERF location is reduced by 16 bytes in order to skip the pluses, which are not part of the expression.

Once the left-end has been determined, the control words and flags are set to initial conditions. Register P5 is loaded with the -1 entry of the OPT1, so that CEKKF will start at the first entry point. Register P2 contains the ERF location of the left-most or first entry of the expression.

The general processing scheme is to check the ID of the ERF item pointed to by

register P2. If the ERF ID indicates a primitive, CEKKF is invoked. CEKKF forms an entry in the OPT1 Table for the primitive. Upon return the current ERF location is compared to the end or right-end location. If they are not the same, the ERF location (P2) is increased by eight bytes to point to the next item of the expression, and its ID is checked.

If the ERF ID indicates an operator entry, the subscript switch (SWCHSB) is checked. If it is set to 1, the subscript expression is on the first pass. The operator is checked for a plus. If it is a plus, the first pass is completed and CEKKM is invoked. Upon return from CEKKM, the conditions have been reset for another complete pass over the subscript expression, so the process of checking each ERF ID is restarted. The switch SWCHSB is set to zero, so that the second pass appears as normal processing.

If SWCHSB is zero or, if the operator is not a plus, CEKKH is entered to form and file a triad generated from the operator and the last two OPT1 entries. Upon return from CEKKH, CEKKL is entered under one of two conditions:

1. The triad is removable from the current loop, and the triad flag indicates that it has already been removed, except for expressions removed to BL3 (i.e., FRCFLAG is on).

2. SWCHSB is set to 1, indicating the first pass over the subscript expression.

CEKKL forms a new OPT1 entry for the previously created triad, so that it will be an operand of the next operator. Upon return from CEKKL, the insert flag (INSW) is checked. If this flag is nonzero, there was a "Float" function inserted by CEKKN. In this case, CEKKL has restored conditions, so CEKKH is now entered to form another triad, and processing is continued from there. If INSW is zero, the routine loops back to check for the end of the expression.

After the return from CEKKH, if neither of the two above conditions exists, CEKKI, is entered, with register P4 pointing to the first (or lower) of the last two OPT1 entries. CEKKI operates on the operands of each triad. For primitive operands adcons are counted for global register assignment where appropriate. For triad operands the removability and commonality are determined and the necessary action taken (naming, creating residues, etc.). Upon return from CEKKI, the flag SWCHFL is checked. If it is nonzero, a "Float" insert was made by CEKKN, in which case CEKKL is entered. Upon return from CEKKL, CEKKH is reentered for a new triad as before.

If SWCHFL is zero, register P4 is set to point to the last OPT1 entry (the second of the two operands), and CEKKI, is reentered. Upon return from CEKKI, CEKKL is entered. Upon return from CEKKL, the INSW is tested with the same branches as described previously.

When the right-end of the expression is reached, register P3 is set to point to a dummy triad entry (which represents the "equals" operator). CEKKI is then entered to process the last entry in the OPT1 table. Upon return from CEKKI, CEKLF is entered to copy the ERF string into the PF, inserting any float functions as necessary. Upon return from CEKLF, the JOINTE chain is followed. This chain points to triad entries which need the location in the PF to be saved in the name field of the triad. The chain starts with JOINTE and is continued in the name field of each triad in the chain. When an end-of-chain (X'8000') is found, the exit is taken.

CEKLF -- Copy and Edit an Expression

The function of CEKLF is to copy an expression from ERF into the PF. During this process null entries will be deleted, special expression file entries will be inserted, and any expressions which have been named as common will be changed to so indicate, if necessary. See Chart DB.

ENTRIES: This routine is entered at CEKLFA with the following input parameters:

P2 = Location of the right-end of the expression in the ERF
P3 = Location of the left-end of the expression in the ERF
P6 = Location of next available entry in the PF

EXITS: Register P4 contains the location of the last entry made in rhe EF portion of the PF. Register P6 contains the updated next available work in the PF. If the PF is filled during the copying, the Phase 3 error exit is taken.

OPERATION: Initially, the SETFLAG is checked. If it is turned on, indicating that the left end of an assignment statement is being copied, the 'global load' flag is checked in the PLP. If it is on, the GFLSW is set to 3, indicating that this loop may have a removed floating-point candidate. The SERF cell contains the location of the first insert entry. If SERF is zero, there are no insert entries. In this case, the insert location is set to X'FFFFFFFF' to prevent any insertions. If there are insert entries, the location of

the first entry is loaded from SERF. The insert location is loaded for comparison with the ERF location. If the ERF location is equal to the insert location, the SERF entry is moved into the PF. The ID field is then checked for an "operator." If it is an operator, the triad location is loaded and the name flag checked. If it is on, the name is moved from the triad to the EF entry and the ID is changed to "CSX." In any case, the PF location is updated to the next location. The location of the next SERF entry is loaded, and the routine returns to load and check its insert locations.

When the insert location is not equal to the ERF location, the ID of the ERF entry is checked for a null ID. If it is null, the entry is not put into the PF, but the routine skips to update the ERF location.

If the ID is not null, it is checked for an operator. If it is an operator, the following checks are made.

1. If it is either a complex operator or an intrinsic-function-argument operator, the GFLSW is turned off.

2. If it is a colon operator, and the COLONF flag is on, it is checked to see if this is a removed floating load candidate. The COLONF flag is set when an array operand is processed and meets the 'removed floating point quantity' requirements. The SETFLAG is checked to determine whether the expression is the left or right end of the assignment statement.

   If this is the left end of the assignment statement, the removal level of the subscript operand is checked. If it is removable from the current loop, the GFLSW is set to 2, indicating that a subscripted variable appeared to the left of the equal sign. The subscript-triad pointer is saved for comparison when the right end of the equal sign is processed.

   If the SETFLAG is off, then the right end of the equal sign is being processed. The GFLSW is checked. If it equals 1, it is set to zero. This means the statemnet A(3) = A(I) will not be considered for global assignment in floating-point register 6.

   The subscript triad is then compared to the one saved from the left end. If they are not equal, the GFLSW is set to zero. This protects the statement A(J) = A(I) from being considered. The adcon-displacements of the two array items are then checked. If they are equal, the colon operator is flagged for Phase 4. If they are not equal, the colon operator is not flagged and the GFLSW is left unchanged. This allows the statement A(I) = A(I+3) to be recognized as an allowed condition in that the elements being referenced will never be the same.

   In any case, the triad name is moved to the ERF entry, and the ERF is copied to the PF.

If the ERF entry is an operand, it is checked for a variable. If it is a variable and the SETFLAG is on, it is examined for being a removed floating-point-quantity candidate. The requirements are that it must be REAL*4 or REAL*8, and must not have the INTERFERING flag set in its symbol table entry.

The ERF flags are then checked to see if the quantity is a subscripted variable. If not, GFLSW is set to 1, indicating a simple variable to the left of the equal sign. If it is a subscripted varialbe, COLONF is set to one, indicating that the next colon operator processed corresponds to this array operand.

If the SETFLAG is off, GFLSW is checked. If it is set to 2, and the symbol table pointer matches the saved one, the ERF flag is checked to see if it is a subscripted variable. If not, GFLSW is set to zero. This disallows A(I) = A(3).

If GFLSW is set to 1, the symbol table pointer matches the saved one, and the ERF flag indicates a subscripted variable, GFLSW is set to zero. If it is not subscripted, and the adcon displacement equals the saved one, the ERF operand item is flagged for Phase 4.

In all cases, the ERF is copied to the PF. The next available word in the PF (register P6) is updated. The ERF location is increased by eight bytes and compared to the right-end location. If they are not equal, the routine loops back to check for an insertion. If they are equal, the expression has been copied. The new PF location is converted to a pointer and stored in TEPFT. If the SETFLAG is off and GFLSW is set to 1 or 2, CEKKO is called to reduce the popularity of the subscript or adcon by the number of times the item was flagged. CEKKO is called a second time to increase the popularity of the item on the next outer loop by a weight of 10. The pointer to the EF for the left end of the expression is then stored in GBLREAL for CEKKU.

If the SETFLAG is on, the pointer to the expression in the PF is saved, and exit is taken.

CEKKF -- Pushdown Primitive Operand Routine

The functions of the CEKKF are to update the symbol table entry of an operand (if a variable), to change the ERF entry to reference an address constant and displacement, to create a dummy expression (@ operator) if required, and to form an entry in the operand push-down list (OPT) from the information created in these processes. See Chart DC.

ENTRIES: This routine is entered at CEKKFA by restricted linkage (INVOKE macro instruction). Register P2 contains the location of the current entry in the ERF. Register P5 contains the location of the current OPT1 entry.

EXITS: This routine returns to CEKKE by the restricted linkage (RESUME macro instruction). Registers P2, P5, and P6 are returned unchanged.

OPERATION: The OPT1 location is stepped to the next entry and the area is cleared. The current ERF pointer (register P2) is stored in the Polish (OPOLSH) field. The forward compute point (OPTFCP) is set to X'7FFF', which forces it to the beginning of the object program. If the ID of the ERF item indicates a loop variable, the loop variable flag (OPLVF) is turned on. The count field (OPCNT) is set to 1 (since this routine is only entered for primitives). The symbol table pointer for the primitive is moved from the ERF to the OPT1 (OPTRD1). The left-end switch (DSWT) is tested. If it is zero, the symbol table location is stored in DEFSYM for CEKKU and the switch set to nonzero. If the flag (SET FLAG) is nonzero, the operand being processed is an assignment value. If its symbol table pointer is equal to the current floating point candidate for Phase 4, the candidate is deleted. The sign flag (OPSIGN) in the OPT1 entry is set to agree with the sign in the ERF.

Next, CEKKG is entered to determine the forward compute point (FCP) and undefined level (ULEV) of the primitive. However, CEKKG is not entered for loop parameters, functions, and constants. Loop parameters receive special processing (see below). Functions go directly to the exit, leaving the OPRLEV set to zero, and the OPTFCP set to X'7FFF'. The fields are the same for constants, but they do get the remainder of the processing in order that an adcon may be filed for them.

For ERF entries that are flagged 'split subscript', the FCP and ULEV are transferred from the symbol table entry for the induction variable. The symbol table pointer was temporarily put in the EF adcon halfword by CEKKM for this purpose.

Upon return from CEKKG (or the loop parameter processing), the FCP and ULEV are moved from the symbol table to OPTFCP and OPRLEV, respectively, in the OPT1 entry. A "storage location" is formed by adding the SLOC field from the symbol table to the offset word in the ERF. This, along with the storage class (STCL) from the symbol table, is used as a parameter to enter CEKLB, which files a covering adcon in the symbol table.

When control is returned from CEKLB, the subscripted variable flag is checked in the ERF. If it is on, a special processing section is entered (see below). If the flag is not on, the pointer to the adcon entry in the symbol table is stored in the ERF (in EFADCON). Special processing is then given to variables with an offset of nonzero and to loop variables, to insert an @ operator (see below). For all others, the type is checked. If it is real, and neither a subscripted nor a class array item, its symbol table pointer is saved as the current floating point candidate in GBLREAL. The exit is then taken.

"AT" Operator Insertion

The @ operator is used to distinguish between different bytes of an array for constant-subscript items. (In other words, it is to distinguish between A (3) and A(5).) It also makes occurrences of the loop variable inside the loop different from references outside the loop. This is accomplished by forming a dummy triad, with the operator an @. The displacement is stored as the first operand (TROP1), so that only references to the same byte of an array will be common. The prototype triad is formed, and CEKKP is entered to file it in the TRIAD Table. Upon return, the triad pointer is stored in the OPT1 table and the indicator (FI) set for a triad. The @ flag (ATF) is set in the OPT1 to mark this as a dummy triad. The exit is then taken.

Subscripted Variable Processing

The remaining entries in the ERF, up to the first + preceding the :, are inside the subscript. These entries require two passes for complete processing. For the first pass special paths are taken in CEKKE, CEKKH, CEKKF, CEKKI, and CEKKL. The switch (SWCHSB) is set to 1, to mark the first path. In addition, the symbol table pointer to the adcon covering the array, the SLOC of the adcon, and the symbol table pointer of the array variable are saved for CEKKM. A branch back to check for @ operator is made.

Loop Parameter Processing

If SWCHSB is set to 1 the processing is inside a subscript. The loop variable flag in the ERF ID is cleared. This makes a loop variable look like a loop parameter. On the second pass the loop variable will get the special loop parameter processing, but not the special @ processing, and the loop will not be materialized (see below). The main section is entered just after the return from CEKKG.

If SWCHSB is set to zero, processing is outside a subscript. If the loop variable flag is on in the ERF, the loop level is extracted from the ERF ID field, and used to set materialize flag in the PLP table of the proper loop

For all loop parameters the ERF ID is changed to variable or constant, with the proper type code from the symbol table. The main section is reentered just after the return from CEKKG. (CEKKG should not be entered for loop parameters, since the end-loop routine, CEKKC, has put in the proper FCP and ULEV, and CEKKG is not set to recognize the special case.)

CEKKG -- Variable Compute Point and Removal Level Routine

The purpose of this routine is to determine the forward compute point and removal level for a variable. See Chart DD.

ENTRIES: The entry point is CEKKGA. Register P1 contains the symbol table pointer for the variable to be processed.

EXITS: Register P4 contains the symbol table location of the variable processed. Registers P1, P2, P4, and P5 are returned unchanged.

OPERATION: If the symbol table entry for the variable has the "not computable" flag on, the variable is an adjustable dimension; therefore, the varible cannot be redefined across level zero. If level zero is safe, the undefined level flag (ULEV) is set to zero; otherwise, ULEV is set to 1.

If the "not computed flag" is not on, the FCP in the symbol table entry of the variable is compared with the current PRF pointer (FPT). If the FCP is smaller than the FPT, the FCP is still valid. The ULEV field is obtained from the symbol table entry and compared with the current loop level (LEV). If ULEV is larger than the current level, it is still valid and the routine returns to the invoking routine. If ULEV is not larger than the current looplevel, the new undefined level is determined by checking the FCP against the PRF pointer of the BL3 entry of each loop from the current ULEV to the current loop level. If the FCP is greater (above in the PRF) than the BL3 entry of a loop, that level is made the undefined level. If no level is found for which the FCP is outside, the current level plus 1 is made the undefined level. This means that any expression involving this variable cannot be removed from the current loop.

In the case where the FCP is not smaller than the FPT, a new FCP and ULEV must be determined. First, a tentative forward compute point, TFCP (in the PRF), and a tentative backward compute point, TBCP (in the PF), are determined. Two types of variables are present:

1. Normal Variables. The forward definition point (FDP) and backward definition point (BDP) from the symbol table entry are set as the tentative compute points.

2. Variables in Common. For the TFCP, the lower of the FDP and the forward definition point of the common chain (TECPAN) is used. For the TBCP, the higher of the BDP and the backward definition point of the common chain (CBDP) is used.

Once the TFCP and TBCP are fixed, the removal level is determined. This is the first safe loop between ULEV and the current loop for which TFCP is higher than the BL3 PRF entry for the loop, and for which TBCP is lower than the end loop PF entry. In other words, there is no definition point inside the loop. If the TBCP is inside the loop but TFCP is outside, the TFCP is reset to the BL3 point before the next higher loop is tested.

When the removal level is determined, it is stored as ULEV in the symbol table, and TFCP is stored as the FCP.

CEKKL -- Operand List Expression Formation Routine

CEKKL produces an entry in the OPT1 table, representing an expression, formed from an operator and the last two operands in the operand list (which are thereby deleted). See Chart DE.

ENTRIES: This routine is entered at CEKKLA with the following input parameters:

P3 = Address of expression in triad table
P5 = Index into OPT table
P2 = Current ERF pointer

EXITS: Register P2 points to the updated current ERF entry. Register P5 contains the updated OPT1 entry location. Registers P3 and P6 are unchanged.

OPERATION: A new OPT1, created to represent the previous triad, is formed from information in the triad entry and the last two OPT1 entries. I replaces the lower of these two entries.

The location of the triad is converted to a pointer and stored in the new OPT1 entry. The FCP and RLEV are moved from the triad to the OPT1.

The OPFI is set for a triad entry. (OPFI refers to the triad table entry F1 or F2 with which the phase is currently concerned.) The sign flag is moved from the SIGNOP cell in the PSECT (stored by CEKKH). If the loop variable flag (OPLVF) is on for either of the two OPT1 entries, it is set for the new OPT1 entry.

The cell SWCHFL indicates whether CEKKN made an insert of a float function during the processing of the previous expression. If SWCHFL is zero, no insertion was made. If SWCHFL is nonzero, it contains the ERF location of the entry to which the float is to be applied.

CEKKL tests SWCHFL. If it is zero, the ERF location in register P2 is converted to a pointer and stored in the OPT1 entry OPOLSH. The count fields (OPCNT) from the two OPT1 entries are added together, the total increased by 1, and the result stored in the new count field. This gives the number of EF entries in the expression to this point (used by CECKM and CEKLD). The exit is then taken.

If SWCHFL is nonzero, the contents are put into register P2 as the ERF location, and SWCHFL is cleared to zero. The byte INSW is set to nonzero, to indicate to CEKKN that a float function was inserted on the previous entry. The two OPCNT fields are added together, but not increased by 1 before being stored in the new OPCNT. This is because the float operator is not in the ERF, but in the SERF.

CEKKH -- Triad File Manipulation Routine

The purpose of CEKKH is to locate or insert in the triad table an entry formed from an operator entry in the expression file and two operands in the operand push-down table. See Chart DF.

ENTRIES: The entry point is CEKKHA. Register P2 contains the location of the ERF entry of the operator of the expression. Register P5 contains locations of the current OPT1 entry.

EXITS: Register P3 contains the triad location. Registers P2, P5, and P6 are unchanged.

OPERATION: Immediately upon entry, CEKKH calls CEKKN to put the operands into canonical form. Upon return from CEKKN, a prototype triad entry is generated in working storage from the last two entries in the OPT1 table and the operator in the ERF. If the operator is a semicolon, a function is involved. The lower address entry in the OPT1 is assumed to be the function. The symbol table entry for this item is checked for a library function. If it is not a library function, the expression is not removable. This nonremovability is forced by setting the RLEV in the Triad to 55. Also, the expression can be common only with a similar expression in the same statement. This commonality is forced by setting the forward compute point (FCP) to the current PRF location plus 1. Before the next PRF item is processed, the triad entry will be deleted by CEKKR.

For all operators other than semicolons, '?', and for library functions, the removal level (RLEV) in the triad is set as the higher of the two OPT1 entries of the operands. The triad prototype is now completed, so CEKKP is entered to find a common triad or to file the prototype as a new entry.

If the operator is '?', and both of its operands are removable , the RLEV in the triad is set to LEV+1 so that the triad appears nonremovable. This is necessary since both operands of the '?' are constants for the split-subscript expression, yet the expression for the base/index split is by nature nonremovable.

After control is returned from CEKKP, the name flag of the triad indicated by CEKKP is checked. If the flag is on, the triad is already correct. If SWCHSB is off (set to 1), the expression is being processed for subscripts for the first pass; therefore, the exit is taken. If SWCHSB is not 1, the name in the triad and the triad pointer are put into the EF entry.

CEKKP -- Search and Insert Triads

CEKKP enters new expressions in triad table, locates old ones, and removes obsolete ones. See Chart DG.

ENTRIES: The entry point is CEKKPA. Register P2 contains the ERF location of the operator of the expression. Since a prototype triad has been built in the working storage area, register N2 will contain its location.

EXITS: Register P3 contains the location of the triad. Register P2 is unchanged. Register N2 is the same or is updated to point to available working storage.

OPERATION: Upon entry CEKKP locates at the point indicated by register N2 the prototype triad, which was put together by CEKKH.

The triad entries are linked together through a hash table (TRIH). The hash table index is formed by adding together the OP1 and OP2 fields of the triad (these are either symbol table pointers or other triad pointers) and reducing the sum to modulo 1020 (X'3FC'). The resulting hash table entry is the anchor for a chain of triad entries. This chain is followed until an end-of-chain or a matching triad entry is found.

As the chain is followed, the forward compute point (FCP) of each triad is first compared with the current PRF location (FPT). If the FPT is higher than the FCP, the compute point has been passed in the PRF and the triad is now obsolete. Thus, this triad should not be considered as a candidate for commonality with the prototype. The triad is relinked out of the chain, so it is permanently unavailable as a common expression.

If the triad is still active (FPT not greater than FCP), the key fields are compared to the prototype's fields for common expressions. For two triads to be common, the following fields must match exactly: OPD1, OPD2, FI1, FI2, SIGN1, SIGN2, OP, and RLEV. If a match is found, the location is put into register P3 and the exit taken.

CEKKN -- Canonical Form Routine

CEKKN puts an expression into a canonical form so that expressions differing only in nonessential variations will be recognized as common. See Chart DH.

ENTRIES: The entry point is CEKKNA. Register P2 contains the location of the ERF entry for the operator of the expression. Register P5 contains the location ofthe OPT1 entry for the second (or right) operand of the expression.

EXITS: Registers P2, P5, and P6 are returned unchanged unless there was a float function inserted, in which case P2 points to the EF insert in the work area and P5 points to the generated OPT1 entry for the insert.

OPERATION: If the operator is not plus, multiply, divide, greater than, less than, AND, OR, or equal, the routine exits. For plus, multiply, and divide, the type is checked. If the operator is a Real*4 or Real*8, the types of each operand are checked. If one type is Integer*2 or Integer*4, a Float function is to be inserted in the EF at that point. The current ERF location is stored in SWCHFL, and P2 set to point at the preset EF entry for the function in working storage. The necessary OPT1 entries are inserted and the routine exits. On the next entry INSW indicates that the last entry had a float insert, thus preventing another from being inserted, and normal processing continues.

The processing is an attempt to make as many triads common as possible. This is done by always putting the smaller OPD field as OPD1 and by moving any minus signs up to the operators (since the operator sign does not affect commonality, but the operand sign does).

CEKKI -- Expression Removal and Commonality Determination Routine

The function of CEKKI is to determine commonality or removability of an expression, to make entries in the compute and remove table (CRT), to mark last occurrences, and to tally popularity counts. See Chart DI.

ENTRIES: The entry point is CEKKIA. This routine uses the two common registers of Phase 3: N1, covering intercom and the rest of the second page of the phase PSECT; and N2, covering the unused area start point of the phase PSECT. Input parameters are

P2 = ERF location
P3 = Address of triad entry is dominant operator
P4 = OPT entry address this operand

EXITS: This routine has one normal exit. Except for catastrophic returns to the Phase 3 abort point, there are no other exits and no output parameters.

OPERATION: This routine processes one operand of a triad on each entry. Upon entry, CEKKI loads the ERF location from the OPT entry. If the operand is a primitive or an @ operator, the conditions are tested for counting the popularity of the covering adcon for global register assignment.

The adcon is not counted for any one of the following conditions:

1. The "adcon" field of the EF is X'8000'.
2. The I/O flag, set by CEKKU, is on.

3. The variable is an array variable.

4. The processing is in the first pass of a subscript.

5. The operand is part of a normal expression.

6. The EF entry is a function.

7. The triad entry is a removed expression.

8. The triad entry is a common expression.

If the triad is removable to level zero and is occurring at level zero, the adcon is counted when the expression occurs for the first time. If it is not occurring for the first time, the count is set to -5 to reduce the previous count, since the expression will now be removed. CEKKO is used to tally the popularity.

If the indicated operand is an operator (other than @ or ,), it is checked for commonality or removability. An expression is considered removable if its removal level is not greater than the current loop level. If the removable expression has not expression has not been named, CEKKK is entered to give it a name. If the last-use flag has not been set, it is now set. If the forward compute point for the expression is above the begin-loop-1 entry of the current loop, CEKLE is entered to file a compute-and-remove table entry at begin loop 1 to insure expression generation at the loop top. The expression is replaced in the EF by a residue in CEKLD. Finally, if the expression is integer and not part of a larger removable expression, it is tallied by CEKKO as a candidate for global register assignment.

If the expression is not removable, it undergoes the following processing. If it is named, no further processing occurs. If this is the first occurrence, it is linked into the JOINTE chain. This chain causes the PF location to be stored in the last occurrence field of the triad entry by CEKKE. If this is not the first occurrence, the next larger expression is checked for commonality. If the larger expression is common, the current expression is not processed further except to insure that it is in the JOINTE chain. If the next larger expression is not common or if this is its first occurrence, the current expression is checked by CEKKJ for commonality. If the current expression is common, it is named by CEKKK and the last use bits are set.

Expressions involving the induction variable are processed for removability in a special manner. If the removal level is higher than the current level, the expression is not removable. If the removal level is lower than the current level, the expression is removable. If the removal level is equal to the current level, another test is used to determine removability. The forward compute point for the expression is compared to the begin loop 1 PRF location of the current level. If the FCP is higher than or equal to BL1, the expression is treated as removable. If it is less than BL1, it is treated as a special nonremovable.

CEKKJ -- Check Commonality

CEKKJ determines whether any entries in the PRF between first and second occurrences of an expression rule out their being common. See Chart DJ.

ENTRIES: The entry point is CEKKJA. Register P2 contains the ERF location of the operator of the expression. Register P3 contains the location of the triad entry for this expression.

EXITS: There are two exit points from CEKKJ. The "common" exit (KJ900) returns with a return code of nonzero. The "not common" exit (KJ950) returns with a return code of zero. Registers P2, P3, P4, P5, and P6 are returned unchanged.

OPERATION: This routine is entered if there have been two occurrences of the same expression, the current one is not removable, and the expression is unnamed. The name field of the triad entry contains the PF pointer to the previous occurrence (which was inserted by the JOINTE chain at the end of the processing by CEKKE). If the name field contains an X'8000', the previous occurrence is in the statement currently being processed. In this case the expressions must be common, so the "common" exit is taken.

If the name field is not X'8000', the pointer to the last label definition point in the PF (LBDP) is compared with the PF pointer to the last expression. If the label pointer is higher, then a label has intervened and the expressions cannot be common. The "not common" exit is taken.

If the label pointer was not higher than the PF pointer, the expressions still may not be common if the end loop of an unsafe loop intervenes between the two occurrences. The end loop location (found in the PLP Tables) of each loop in the current nest is checked down to level zero. If the end loop is between the two occurrences, the "unsafe" flag of the PLP is checked. If it is on, the "not common" exit is then taken. If the flag is not on, the next

lower level is checked until an end loop entry is found which is lower than the previous occurrence (the level zero end loop location will always be lower). Then the "common" exit is taken.

CEKKK -- Establish Common Expression Routine

This routine's purpose is to assign a name to an expression, to put that name in the previous occurrence that is common, and to enter the name in the CRT at the forward compute point. See Chart DK.

ENTRIES: The entry point is CEKKKA. Register P2 contains the ERF location of the operator of the expression. Register P3 contains the triad entry location.

EXITS: Registers P2, P3, P4, P5, and P6 are unchanged.

OPERATION: Upon entry, CEKKK loads the last assigned name from TENCSX and adds 1 to it to create a new name. The new name is checked for a value greater than 4095. If it is greater, the error exit is taken. If the name has a value of 4095 or less, it is stored in TENCSX as the new last-used name.

If the QFLAG of the triad is zero or if the name field of the triad is not X'8000', the name contains a PF pointer to the previous use of the expression. The pointer is loaded in a register, and the expression name is stored in the name field. Then CEKLA is entered to mark the previous occurrence as 'CSX' and "last-use." If the QFLAG is on or if the name field is X'8000', CEKLA is not entered, because the last occurrence is in the current statement.

Upon return from CEKLA or if it was skipped, the name flag is turned on in the triad.

The forward compute point is loaded into a register from the triad to file a compute and removal table (CRT) entry at that point. CEKLE is entered to file the CRT entry. Upon return from CEKLE, the exit is taken.

CEKLA -- Label Common Expressions

When it is determined that an expression is a common expression, this routine locates in the PF the previous occurrence, labels it "CSX," and marks it as last usage. See Chart DL.

ENTRIES: The entry point is CEKLAA. Register P1 contains a PF pointer for the major operator of an expression containing a common subexpression. Register P3 contains the location of the subexpressions TRIAD entry.

EXITS: Register P1, P2, P3, P4, P5, and P6 are returned unchanged. There are no output parameters.

OPERATION: The location of the expression in the EF containing the previous occurrence is computed. A counter is started with a value of 1. The ID of each EF entry in the expression is checked, starting from the right end (major operator) and going to the left end. If the entry is a primitive, the counter is reduced by 1, the EF location is reduced by 8, and the next entry is checked. If the entry is a 'CSX' (common expression), the count is increased by 1, and the next EF entry checked.

When an operator is encountered in the EF, the triad pointer saved in the EF (EFTRD) is compared with the pointer to the current triad. If they do not match, the count is increased by 1 and the next EF entry checked.

If the triad pointers match, the name is moved from the triad to the EF. The operation code in the ID field of the EF entry is changed to a CSX (the type is retained). The last use flag is set in the triad and in the EF.

There can be more than one occurrence of the triad within an expression, if it is a subexpression of two larger expressions which are also common. Therefore, each expression must be scanned to the end. The end is reached when the counter is reduced to zero. Then the exit is taken.

CEKLE -- File CRT Entries

CEKLE locates a previously filed entry, if present, or files a new entry in the compute and remove table. See Chart DM.

ENTRIES: The entry point is CEKLEA. Register P1 contains a PRF pointer, indicating where the compute and remove point is. Register P3 contains the location of the triad entry.

EXITS: Registers P1, P2, P3, P4, P5, and P6 are returned unchanged.

OPERATION: The PRF pointer to the filing location is converted to a hash table index, by taking the PRF location module 1020. The chain from that hash entry is followed until one of the following conditions exists:

1. An end-of-chain is encountered, which causes the entry to be inserted as the last entry in the chain.

2. The same triad entry is found, which causes an exit.

3. A PRF pointer is found that is less than or equal to the indicated pointer. This causes the entry to be inserted into the chain at this point. The PRF pointers should be in descending order as the chain is followed out from the hash table.

CEKLD -- Expunge a Removable Subexpression

This routine replaces a subexpression with a series of null entries and one residue entry, indicating the expression by name. See Chart DN.

ENTRIES: The entry point is CEKLDA. Register P2 contains the ERF location of the operator of the expression. Register P3 contains the triad location. Register P4 contains the OPT1 entry location for the expression.

EXITS: Registers P2, P3, P4, P5, and P6 are returned unchanged.

OPERATION: The count field from the OPT1 entry indicates the number of EF entries in the expression. This number is reduced by 1, multiplied by 8 (for eight bytes per EF entry), and subtracted from the "right-end" location to give the "left-end" location. These two locations are used as limits to check the SERF chain for inserts. When the insertion point into the ERF falls between the two limits, that SERF entry is deleted by relinking the SERF chain. When the end-of-chain is encountered, the ends of the SERF chain are restored.

Each ERF entry in the expression is then checked. If the ID indicates a residue, the last use flag on the ERF entry is checked. If it is on, the triad location is loaded and the last use flag in the triad is cleared.

The ID is then set to null (zero), which causes CEKLF to skip the entry when copying the ERF into the PF.

When the right end is reached, the ID is set to a residue and the exit taken.

CEKKM -- Subscript Expression Revision Routine

This routine's purpose is to revise a subscript expression to include the address constant as a term and to be optimal as regards computation with loop variables and removable expressions. See Chart DO.

ENTRIES: The entry point is CEKKMA. Register P2 contains the ERF locations of the first + operator encountered in the subscript expression. Register P5 contains the current OPT1 entry location.

EXITS: Register P2 contains the ERF location of the first operand of the sorted subscript expression. Register P5 contains the OPT1 location for the first entry of the sorted subscript expression.

OPERATION: This routine is entered when the preliminary scan has reached the first of a string of + signs prior to the :. It counts these to determine the range of the expression to be revised, and the number of operand entries in the OPT Table which are involved. Then it sorts these entries in OPT on removal level, loop variable indicator, and forward compute point. A new OPT entry is preset for the new operand (address constant). The entire Polish string, including the preformatted address constant entry, is copied into working storage. From there it is copied back in the new order, in blocks of operands.

Each block consists of all operands with the same removal level and loop variable indicator. For each block a string of + signs with the maximum type code, is placed in a pushdown list and copied into the revised string to connect the operands within that block. For each block, an operator is set aside in a pushdown list: the operator being ! if the loop variable flag is raised, or + otherwise. The pushdown list of block operators is moved into the revised string when the removal level drops below the current level, and the pushdown list is reinitialized with the ? operator. When a '!' operator is selected, if a '?' has not been put on the block operator list, then the induction variable is the least removable term. If the loop is innermost with no external calls (i.e., the GLOBAL flag is on) a split subscript is created. This is done by taking the expression in the ERF which has the form:

IVAR OPND2 *

where OPND2 can be a constant or an expression, and changing it to the form

OPND2 ZERO !

where ZERO is an ERF operand for the constant 0. A '?' operator is then added to the block operator list so that it will split the removable parts of the expression (e.g., the adcon) from the recursive. When all blocks have been moved, the pushdown list of block operators is moved into the revised string.

CEKKA -- Acquire Entry from Compute and Removal Table

This routine locates one entry in the compute and removal table which falls within a given range of PRF locations, if at least one such entry exists. See Chart DP.

ENTRIES: The entry point is CEKKAA. Register P2 contains the location of a byte, four bytes in front of a CRT pointer. Register P3 contains the hash table index of the original entry point (also in LOCHCR). Register P4 contains the limiting PRF location. Register N1 convers the intercom area, and register L1 covers the work area.

EXITS: If register P2 is zero, no valid entry was found. Otherwise, register P2 contains a pointer to the CRT table. LOCCRT contains the previously found CRT entry (used for deleting). Registers P5 and P6 are returned unchanged.

OPERATION: Upon entry, CEKKA saves the register P2 in the cell LOCCRT (the procedure is also done when the routine loops around). This is the location of either an HCRT entry or a CRT entry. (See "Hash Table for Compute and Removal Table and Triad Entries".) The location is used to relink the chain and to delete an entry from the CRT.

The pointer to the next CRT entry is checked for an end-of-chain. If it is not an end-of-chain, the location of the indicated entry is loaded into register P2. The forward compute point in the CRT entry is compared to the limiting PRF location (register P4). If the FCP is not higher than the limiting PRF location, a legitimate CRT entry exists, and the exit is taken.

If the FCP is higher than the limiting PRF location or if the CRT pointer was an end-of-chain, register P2 is set to zero, so that any exits will indicate "no valid entry found." The limiting PRF location is converted to a hash index and compared to register P3 in order to determine if the search should be continued. If they are equal, the limiting PRF location is checked against the current PRF location, to determine if the entire table should be checked.

If the limiting PRF location is within 1024 bytes of the current PRF location, the search is completed and the exit is taken. If not, the entire table must be searched. In this case, or if the limiting hash index does not equal register P3, register P3 is checked to see if the top of the HCRT has been reached (255). If it has, register P3 is set to zero, the bottom of the HCRT. If not, 1 is added to the index to examine the next hash entry. The index is then compared to the cell LOCHCR, which contains the original index. If they are equal, the entire table has been searched and the exit is taken. If they are not equal, the new HCRT location is loaded into register P2, and the routine returns to the top to save this value in LOCCRT.

CEKKB -- Polish Expression Generation Routine

The function of CEKKB is to convert an expression from triad table format to expression file (ERF) format to be placed in the program file. It is used to reform removed and loop control expressions. See Chart DQ.

ENTRIES: The entry point is CEKKBA. Register P2 contains the location of the triad pointer for the start of the expression. Register P5 contains the location of the current PRF entry. Register P6 contains the location of the next available word in the PF (used to store the EF entries). Register N1 covers the intercom area, register N2 points to the first available word in working storage, and L1 covers the work area.

EXITS: Register P6 points to the new first work in the PF. Register N2 points to the new first word in working storage. Registers P5 and N1 are the same as when entered. Registers N3, L2, and P2 through P4 are saved and restored. A pointer to the last EF entry filed is in RMVAL.

Three error conditions are detected in this routine:

1. Overflow of the program file.
2. Overflow of the working storage area.
3. An illegal type code in the symbol table.

For all conditions, the standard Phase 3 error exit (CEKKRE) is taken.

OPERATION: This routine uses an operator pushdown list (Type 2) of its own to build a left-hand Polish string from the triad table entries. It follows the left (OP1) branch of each expression until a primitive is reached, then it takes the first right branch (OP2) above the primitive and repeats. The output expression is formed in the program file and is followed by a field (RMVAL) which links it in a chain of expressions connected to a BL1, BL2, or BL3 entry.

The dummy operator @, which is introduced for a constant subscript or an induction variable outside a subscript, is

deleted in this routine, and the two operands (variable and adcon) are combined to produce one variable entry in the Polish. If the induction variable is recognized (in a subscript, not under the @), it is replaced with the increment expression, and, if that is a constant, the product of two constants is replaced with a new constant.

When the right operand of a triad flagged as 'split subscript' is recognized, special processing takes place. As in normal subscript expressions, the loop step is introduced into the expression; constant arithmetic being done where required. If the split subscript is also the 'BXLE on recursive' candidate, the symbol table pointer to the constant recursive step is saved so that its GIRL entry can be deleted by CEKKW.

For a special begin loop 2 entry (BL2GT is on), a dummy triad entry is built in working storage and pointed to by a new OPT2 entry. The program is repeated to file a test expression in the EF.

CEKKO -- Save Popularity Counts for Register Assignment

The function of this routine is to create new entries as needed in the list of expressions to be considered for global register assignments and to keep a popularity of usage count. See Chart DR.

ENTRIES: This routine is entered at CEKKOA with the following register assignments:

P1 = Weight
P2 = Address of Triad entry, Symbol Table entry of Adcon, or expression name
P3 = Indicator (0 = Symbol entry, 1 = Triad entry, 2 = expression name)
P4 = Level in which the popularity is to be counted.

EXITS: Registers P5 and P6 are returned unchanged.

The new GIRL entries are stored in working storage. If working storage is overflowed, the Phase 3 error exit is taken.

OPERATION: Upon entry CEKKO checks the indicator in P3. If it is set to zero, register P2 contains a symbol table pointer which is to be saved. If the indicator is set to 1, P2 contains a triad pointer. In this case the triad location is computed, and the name field is loaded into P2. If the indicator is set to 2, the name is already in P2. In both cases, a X'7000' is added to the name, so that Phase 4 can distinguish names from symbol table pointers.

The location of the PLP table entry for the level given in register P4 is determined. The GPLINK chain points to GIRL table entries for the loop; this chain is followed until an end-of-chain or until the GLBL of a GIRL entry matches register P2.

If a match is found, the weight (register P1) is added to the popularity count in the GIRL entry, and the exit is taken.

If the end-of-chain is found, a new GIRL entry is linked into the GPLINK (it will be the new end-of-chain). Register P2 is stored as the GLBL, and the weight is stored as the popularity count. The exit is then taken.

CEKLB -- File Constant and Covering Adcon

The function of this routine is to file a constant, compute and file its covering adcon, and compute the displacement. See Chart DS.

ENTRIES: The entry point is CEKLBA. Register P1 contains the 4-byte address constant (adcon) to be filed in the symbol table format of SLOC and storage class (STCL). Register P2 contains either the ERF location of where the adcon pointer is to be stored or zero if the adcon pointer should not be stored. Register P4 contains the symbol table location of the original entry for which the adcon is being filed.

EXITS: Registers P2, P4, P5, and P6 are returned unchanged.

When the return code found in register L3 in nonzero, and error occurred in the executive subroutine. The Phase 3 error exit is taken.

OPERATION: Upon entry CEKLB saves the storage class byte (STCL), which is assumed to be the right-most byte of register P1. The parameter is put into an even-numbered working register and shifted into the following odd-numbered register, with the sign being extended so that the SLOC value is right-justified in the odd-numbered register. This value is now divided by the value 4080, which represents the number of bytes covered by an adcon. The result is a page number in the even-numbered register and a displacement in the odd-numbered register. If the sign of the resulting displacement is negative, the page number is reduced by 1 and the displacement is increased by 4080. If subscripts are being processed (SWCHSB≠0) or if the EF location (register P2) is zero, the displacement is not saved; otherwise, it is stored in the indicated EF entry.

In either case, the page number is multiplied by 4080, to compute the page bound-

ary location. The result is the new SLOC, which is shifted, and the saved STCL is inserted in the right-most byte. The resulting word is stored in TECNS1 for the executive subroutine which is now entered. The executive subroutine checks the symbol table, to see if an entry has already been filed for the parameter. If one has not, a new entry is created in the symbol table.

Upon return from the executive subroutine, the return code (register L3) is tested. If it is nonzero, an error occurred in the executive subroutine and the error exit is taken. If the return code is zero, the symbol table entry for the parameter (input in register P4) is checked to see if the item is a formal argument. If it is not, the exit is taken. If it is a formal argument, the cell TEGNU in Intercom is checked. If TEGNU is nonzero, a new adcon was filed for the last parameter. In this case, an entry is made into the formal argument adcon table (FAAT). The location of the next available entry in FAAT is found in TEFAAT in intercom. The adcon's symbol table pointer in TEPNTR in intercom is moved to FAAT. The STCL is moved from the argument's symbol table entry to the FAAT entry. The FAAT location is increased by four bytes, to point to the next entry, and restored in TEFAAT. The exit is then taken.

CEKLI -- Loop Test-Expression Generator

This routine generates the test expression, used for determining the end of a DO loop, by modifying the last recursive expression. See Chart DT.

ENTRIES: The entry point is CEKLIA. Register P6 contains the EF point for the next entry, register N1 covers the intercom area, register N2 contains the next available word in the work area, and register L1 covers the PSECT .

EXITS: Register P6 points to the new next entry in the EF; registers N1, N2, and L1 are unchanged.

OPERATION: The location of the recursive expression to be used for generating the test expression is found in the word EFSAV. It was saved as CEKKB regenerated the recursive expression at the BL2 entry. The form of the recursive expression is assumed to be:

(OP1) (OP2) !

where

OP1 can be a constant or an expression
OP2 can be an adcon, residue, expression, or constant

The location given in EFSAV points to the ! operator. By using a backward scan, the start and end of OP2 are found and saved. OP1 is then copied into the EF, and its last use flag is cleared in the original expression.

If the loop is not marked as BXLE on recursive, a new term is inserted into the expression to generate an expression of the following form:

(OP1) (T) * (OP2) + [the + replaces the !]

where

T is an expression of the form

$$\frac{U - L + S}{S}$$

U = Upper limit
L = Lower limit
S = Step size

This routine will perform constant arithmetic wherever possible to reduce the T expression. The following cases are considered.

1. L = Constant, U = Constant, S = Constant

   A new constant,

   $$t = \frac{U-L+S}{S},$$

   is calculated. Two subcases then exist:

   a. If OP1 is a constant, a new constant $t_2 = t * OP\ 1$ is formed, filed in the symbol table, and entered in the EF to replace OP1 forming

      (t_2) (OP2) +

   b. If OP1 is not a constant, t is filed in the symbol table and inserted to form the expression

      (OP1) (t) * (OP2) +

2. L = Constant, U = Constant, S = Variable

   A constant, t = U - L, is calculated, filed in the symbol table, and inserted to form

   (OP1) (tS + S /) * (OP2) +

3. L = Variable, U = Constant, S = Constant

A constant, t = U + S, is calculated and filed in the symbol table. Two subcases are considered:

a. If the step size is 1, the divide is omitted to form the expression

(OP1) (t $\overline{L}$ +) * (OP2) +

b. If the step size is not 1, the expression is

(OP1) (t $\overline{L}$ + S /) * (OP2) +

4. L = Constant, U = Variable, S = Constant

A constant, t = S - $\overline{L}$, is calculated and filed in the symbol table.

a. If step is equal to 1, the expression formed is

(OP1) (U t + 1MAX) * (OP2) +

b. If step is not equal to 1, the expression is

(OP1) (U t + S / 1MAX) * (OP2) +

5. L = Variable, U = Variable, S = Constant

a. If step is equal to 1, the expression formed is

(OP1) (U $\overline{L}$ S ++ 1MAX) * (OP2) +

b. If step is not equal to 1, the expression generated is the same as given in 6 (below).

6. L = Variable, U = Constant, S = Variable
L = Constant, U = Variable, S = Variable
L = Variable, U = Variable, S = Variable

The full expression is inserted to form the expression

(OP1) (U $\overline{L}$ S ++ S / 1MAX) * (OP2) +

If the loop is marked as a BXLE on recursive, the recursive expression has the form:

constant1 constant2 !

where constant1 is the value that is put into register 14 by Phase 4, and constant2 is zero, the initial value of the recursive.

The test expression generated has the form:

constant3 U $\overline{L}$ + *

where constant3 represents a new constant generated by dividing constant1 from above by the loop step.

When both the upper and lower values are constants, the test expression is merely one constant EF item.

In either case, an EF item for constant3 is created, and its pointer is set in the INC field of the BL2. This is used by Phase 4 when the loop is "materialized on exit."

An internal subroutine, SETUP, is entered to generate an EF entry for each of the three loop parameters. A flag is set for each parameter, to indicate whether it is a constant or a variable. These flags are then tested to determine which case exists, and the proper subsection is entered to generate the test expression.

INTRODUCTION

The objective of Phase 4 is to produce from the program file (PF), which is its primary input, a representation of the object program in a form very close to machine code, the code file, which is its primary output. Other output consists of entries made in the symbol table; parameter list entries and V/R adcon-pair entries arising from external references; location counter values associated with statement labels; numeric constants filed as complements of constants filed as complements of constants referenced in the PF.

The major functions of Phase 4 are primarily oriented about source program statements and expressions as they are represented in the PF. A documentation module, or component, is associated with each of these major functions.

PHASE 4 PROCESSING

Processing is directed by the phase controller (PHAS4), which simply performs a single scan of the PF. During this scan PHAS4 passes control to the particular PF entry processor appropriate to the identification of each PF entry encountered. Its sole functions are to perform this scan, to select the processors, and to terminate Phase 4 processing when the end program item is encountered in the PF. The following list indicates the relationship between the source language statements and the PF entry processing routines:

Routine Name	Source Language Features Processed
ENT	Main program or subprogram main or alternate entry -- entry prologue generation.
LABEL	Source- or compiler-created statement label.
EQUAT	Arithmetic statement.
AIF	Arithmetic IF statement.
LIF	Logical IF statement.
GOTO	Unconditional GO TO statement.
ASSIGN	ASSIGN statement.
AGO	Assigned GO TO statement.
CGO	Computed GO TO statement.
CALL	CALL statement.
RTRN	RETURN statement.
BL1, BL2, BL3, ENDLP	DO statement
RD	All I/O statements. Generate transmission initialization call.
OLIST	Any I/O statement that includes a list. Generates list element transmission call(s).
NDLST	Any I/O statement that includes a list. Generates termination call.
STOP	STOP and PAUSE statements.

A major component of Phase 4 is the arithmetic generator, AGEN. Its function is to generate code to evaluate the expressions that are represented in the PF. AGEN is called to process the operands of any PF statements which may reference either general arithmetic expressions or subscript expressions.

AGEN is primarily a control routine which directs the activities of expression-operator generating routines. These lower level routines are tailored to process specific arithmetic operators, or even specific operator/type combinations. These routines, as well as the higher level PF-entry processing routines, make use of a collection of service routines, which are categorized and whose functions along with those of the higher level routines are summarized below.

Expression Generator Control Routines

AGEN	Expression generator control.
TRBLD	Expression tree formation.
WGHT	Order of evaluation determination.
CSX	Common expression usage count determination.

Expression Operator Generating Routines

RPLUS	Real addition and subtraction.
RMUL	Real multiplication.
RDIV	Real division.
IPLUS	Integer addition and subtraction.
IMPLY	Integer multiplication.

IDVDE	Integer division.
LADDR	Special addition by means of LA instruction.
CPLUS	Complex addition or subtraction.
CMUL	Complex multiplication.
CDIV	Complex division.
RLTNL	Relational operations.
ANDOR	Logical operations.
MAX	Maximum and minimum operations.
FUNC	External function reference operations.
COMMA	External function argument processing.
DCOM	Open function processor selector.
OPEN1 OPEN2 OPEN3 OPEN4 OPEN5 OPEN6	Inline (open) function processors.

Memory Reference Covering Routines
MEMAC
COVER
SADDR
LBL
FETCH

Operand-Reference Optimizing Routines
SELOP
SLPOS
SLONE
SELGM
SELGD

Operand Locating Routines
KEY
KEY1
FNDAR
FNDFR

Operand-Usage Processing Routines
OPND
RSLT

Register Selection Routines
SELSR
SELDR
SELFR

Register Assignment Routines
ASAR
ASARS
ASFR
ASFRS

Temporary storage Allocating Routines
FNDWS
RLSWS

Miscellaneous Routines
INSOT
FLUSH
EDIT

Expression Generation

The first stage of expression generation converts the expression form, in which each binary operator is preceded by first its left and then its right operand, to a tree form (see Figure 24). In expression form the relationship between operator and operand is implicit in the ordering of the expression. In tree form the relationship is made explicit by linking each operator to its operands with explicit address pointers. The tree is also backlinked so that each nonprimitive operand (operator) is linked to the operator upon which it depends. The tree is now equivalent to a push-down table with space at each level (tree node) to record information about the generation status at that node.

During the process of conversion to tree form, conversion function operations are introduced, where necessary, to obtain type compatibility between the operands of certain of the operators. This is done to reduce the number of individual cases presented to the expression-operator generators.

When the expression tree has been created, the order in which the component operations of the expression are to be generated is determined. The language rules require that expressions be associated from left to right. This association is explicit in the expression form input to Phase 4, and converting to tree form does not change this association. However, it does allow easy change in order of computation at any level in the tree. For example, consider the expression

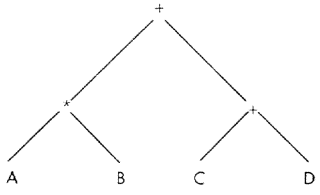
```
(A*B) + (C+D)
```

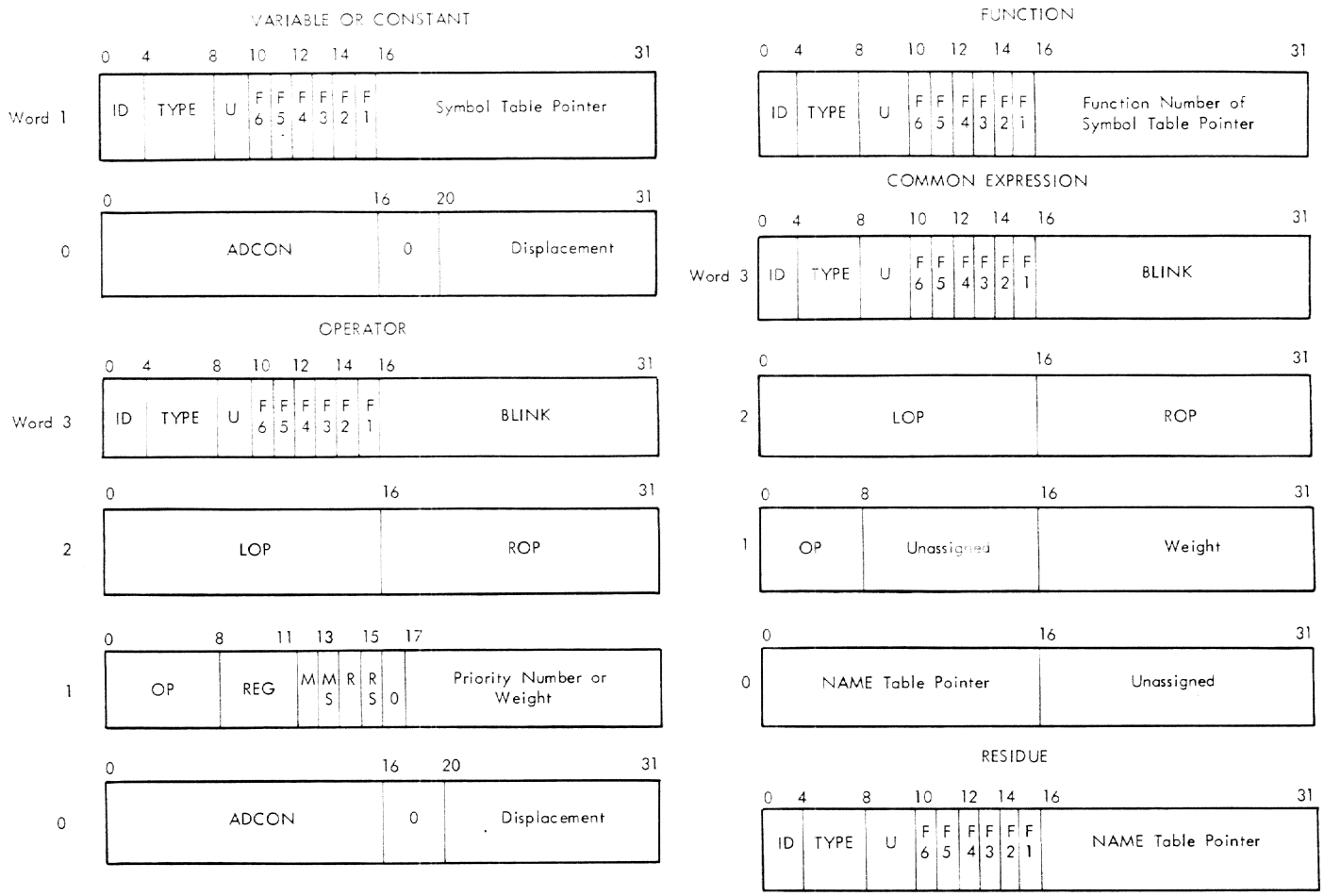
which is represented by Phase 1 as

```
AB*CD++
```

If generation is to proceed directly from the latter, the easiest and most natural way to proceed would be first to compute A*B, then C+D, and then to add the two partial results. However, the language does not require this ordering.

The expression in tree form is written





Legend

ID identifies the Tree Table entry:

1 = Operator
2 = Common Expression
3 = Adcon
5 = Variable
6 = Constant
7 = Function
8 = Residue

TYPE identifies the entry type:

1 = Logical *1
2 = Logical *4
3 = Integer *2
4 = Integer *4
5 = Real *4
6 = Real *8
7 = Complex *8
8 = Complex *16

U unassigned

F6 I/O flag

F5 use secondary temp.

F4 examined

F3 weighted

F2 computed

F1 sign

BLINK link to first byte of operator of next node up (back-link)

LOP link to first byte of left operand

ROP link to first byte of right operand

OP operator code

REG register (if R = 1)

M in memory

MS memory sign

R in register

RS register sign

Figure 24. Expression Tree

Given this form, with direct address links as indicated, it is no easier to compute first along one branch of the highest level operator than it is along the other. Thus, this representation allows a choice of order of computation based upon criteria which are designed:

1. To minimize the numbers of active partial results and thereby to use fewer registers.

2. To favor computation of denominators before numerators in order to avoid unnecessary loads and stores.

3. To compute first along paths containing function references so as to minimize the possibility of having to store partial results which are in registers volatile with respect to the function calls.

The order of computation is determined by the routine WGHT which assigns at each tree node a priority number (or weight) and records it in the tree.

Starting with the major operator of the tree, AGEN examines the left and right operands. If neither is primitive (a variable, a constant, or an already computed operator), the link is followed from the operator to its higher weighted operand. If the weights are equal, then arbitrarily the left link is followed. This new, lower level, operator is then examined in the same manner. If at any time only one operand is primitive, the other link is followed. The search is concluded when an operator with two primitive operands is found. At this point generation for the operator proceeds.

The operator code and expression type are used to select an expression-operator generator, and the appropriate module is invoked.

The expression-operator generator is tailored to the operation to be performed and to the types of its operands, with consideration given to the location of operands (in registers or storage); the requirement for even/odd register pairs; the availability of the register containing an operand; the selection of which operand register is to contain the result; etc. Lower level routines are invoked for various functions: to select registers; to determine when operands are no longer needed and to free the registers in which they reside; to protect an operand by moving it to another register before the contents of the former register are altered; to assign temporary storage and store operands in temporary storage for later use; to obtain cover for and assign B2, X2, and D2 instruction fields for storage references to operands; to record the location of the operation result for later reference in generation; etc.

When the expression-operator generator has completed its task, it returns control to the arithmetic expression generator, which marks the tree node "computed" (primitive).

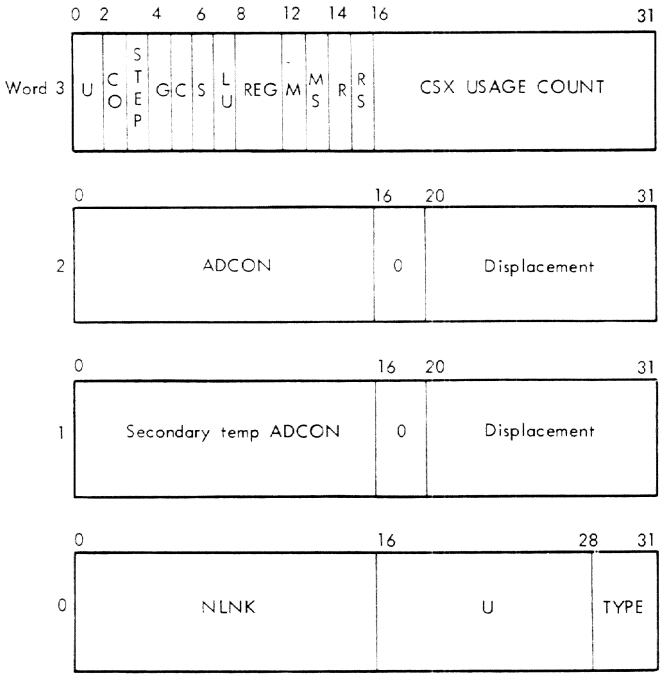
The subscript connector (:), open function argument connector (,,), recursive operator(!), and base/index connector(?) require no generation, and the tree node at which one of these occurs in simply marked "computed". The back-link is inspected next. If it is null (zero), generation is complete and return is made to the module that called the arithmetic expression generator. If the back-link is not null, it is followed to the next higher level operator, from which point generation proceeds as above.

Expression Storage

Whenever a noncommon operation is complete, whenever a noncommon operand is loaded into a register without the intention of immediately operating upon it, or whenever a quantity is stored in temporary storage, a record of the transaction is made. If the operand is being loaded into a register, the corresponding register number is recorded at the appropriate node of the expression tree. If the operand is being stored, the temporary storage assignment is recorded in the tree. Note, however, that such information concerning variables and constants is not recorded in the tree. Thus, the expression tree contains the current location of any computed noncommon partial result.

Common-Expression Storage

Whenever a transaction such as the above involves a common expression, the transaction record is made in the Name Table (Figure 25). Each common expression has a name (number) which is its identification. Associated with each distinct common expression is a Name Table entry which is used to record the location of the common expression in the same way that the expression tree is used to record the location of noncommon expressions.



Legend

U	unassigned	LU	last use
CO	Operator is a colon	REG	register (if R = 1)
STEP	loop increment	M	in memory
G	globally assigned	MS	memory sign
C	computed	R	in a register
S	secondary temp assigned	RS	register sign

NLNK link to loop Table entry of last use of recursive increment, if applicable; otherwise, zero.

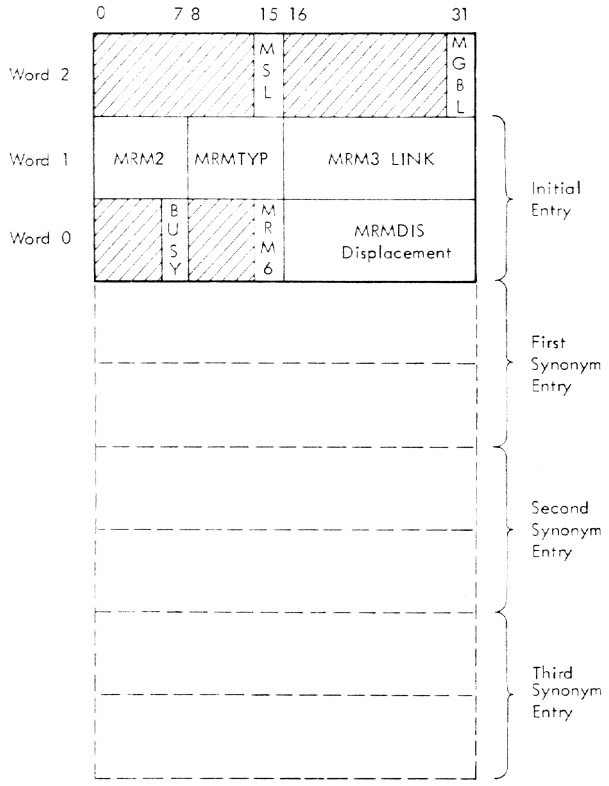
TYPE EF Type Code

Figure 25. Name Table

Register Storage

Two tables are used to keep a running record of the contents of the arithmetic registers. The MRM table (Figure 26) contains one entry for each of general registers 1 through 15 (since general register 0 is used simply as a transient register and its contents are never retained, this register is not represented in the MRM table). The MRMFR table (Figure 27) contains one entry for each of the four floating-point registers. MRM is the symbol applied generically to the register tables, and often, when no confusion can result, MRM is used interchangeably for MRM and MRMFR.

The two tables have generally similar structures. The first word of each entry contains two status indicators, the first of which indicates whether the register is available for selection. The second indicator records whether or not the general register is globally assigned over the scope of a DO loop, or whether or not a floating-point register (0 or 4) is linked as a complex quantity pair to the next higher register. This first word applies to the register as a whole.

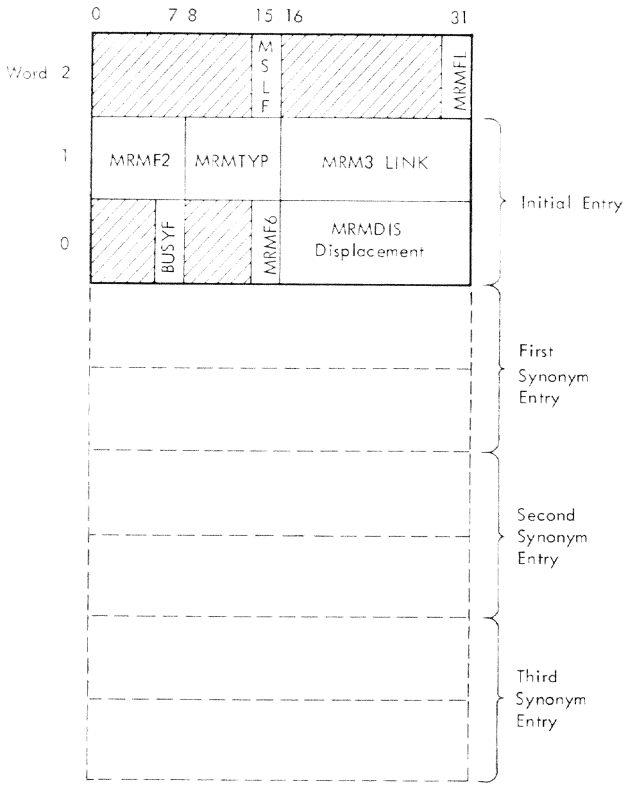


Legend

MSL	0 = Selectable 1 = Nonselectable
MGBL	0 = Nonglobal 1 = Global
MRM2	ID
MRMTYP	EF Type Code
BUSY	0 = Inactive 1 = Active
MRM6	0 = True Sign 1 = Negated

Word 2 occurs once for each general register. Words 0 and 1 occur four times for each general register.

Figure 26. MRM Table



Legend

MSLF	0 = Selectable 1 = Nonselectable
MRMFL	0 = Not Linked 1 = Linked
MRMF2	ID
MRMTYP	EF Type Code
BUSYF	0 = Inactive 1 = Active
MRMF6	0 = True Sign 1 = Negated

Word 2 occurs once for each floating register. Words 0 and 1 occur four times for each floating register.

Figure 27. MRMFR Table

Next, for each entry, is a set of four pairs of words; the first pair is called the initial entry and the rest, synonym entries. The initial entry may or may not be active; but, whenever there is at least one active synonym, the initial entry is active. Whenever a quantity is loaded into and assigned to a register, or computed in and assigned to a register, a record is made in the initial entry of the appropriate MRM table entry to indicate the presence of the assigned quantity in the corresponding register. Synonym entries are sometimes made for quantities which appear on the left-hand side of arithmetic statements. Consider, for example, the sequence of statement:

```
1  A = B
2  C = A
3  D = B
```

The quantity B is loaded into some register, and an assignment for B is recorded in the initial entry of the corresponding MRM table entry. The store into A causes the insertion of A into the first synonym entry. In statement 2, A is found to be in a register, so no load is generated, simply a store into C which is then recorded in the second synonym entry. In statement 3, B is found to be in a register so a store into D is generated, and D is recorded in the third synonym entry.

When an attempt is made to record a fourth synonym, the first (oldest) synonym entry is erased, the remaining two are moved up one slot, and the new synonym is recorded in the third synonym position. The initial entry is never changed by this procedure.

Whenever a quantity which has a current MRM table entry changes value, the corresponding position of the appropriate entry is vacated. If an initial entry thus becomes empty, the first active synonym is installed as the initial entry; or if there are no active synonyms, the MRM entry becomes empty.

General Register Selection

The general registers are used to contain virtual storage addresses and to perform address arithmetic and integer, logical, and relational computations. General registers 1 through 11 and 14 and 15 are treated as equivalent for purposes of register selection, with the single exception that if there is no other basis upon which to make a selection, the lowest numbered available register is selected arbitrarily. General register 13 is used to cover the first page of the object program PSECT (sometimes called the adcon page), and general register 12 is used to cover local temporary storage. Both of these registers are made unavailable for any other use simply by raising their MSL and MGBL flags in the corresponding MRM Table entries. General register 0 may be used in the normal way as a member of the 0/1 register pair, but register 0 itself may never be selected or assigned. It is used only in extremely local context.

In certain situations use of registers 1, 14, and 15 is restricted. For example, when at the top of a DO loop an address constant must be loaded into a register and assigned globally to that register over the scope of the loop, none of these three registers is selected because of its specially required use in the subroutine linkages. Otherwise, the register selection routines make their selection on the basis of the register contents.

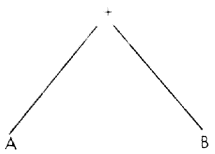
The general criteria for selection of registers involve the relative cost of having to reload the quantity which is in the registers at the time of selection. Clearly, if a register is empty, it is a prime choice for selection. In turn a register which contains a constant whose absolute value is less than 4096 and which therefore may be reconstructed with a relatively fast LA (load address) or SR (subtract register) instruction is likewise a good selection. On the other hand, registers which contain partial expression results (operators) are the poorest choices since they must be stored in temporary storage and later fetched from there. A register which contains an unstored common expression is a somewhat better choice, since while the partial result is known to be last used in the current expression, the common expression has more than one use and may therefore have to be stored eventually, so that not storing it now may only be postponing the inevitable.

The general register selection routines apply these criteria by determining the contents of each register by an examination of its corresponding MRM table entry. A weight is assigned to each register according to those criteria, and the register with the highest weight is selected.

Storage Reference Processing

Cover is obtained for storage references (simple and subscripted variables, constants, and temporaries) and most branches by one of two subroutines: MEMAC and COVER. MEMAC is a more general routine which is used to obtain cover for a reference to any given expression tree quantity. COVER is used in situations where it is known that all that is needed for cover is an adcon, and the symbol table pointer for the adcon is at hand.

Consider first the expression A+B in which neither A nor B is in a register and which is represented in expression tree form as:



The plus generator first selects a floating-point register, say 4, in which to perform the addition. It then requests from MEMAC cover for A. The latter routine recognizes that the operand is a variable. It obtains the adcon pointer from the variable item and searches the MRM table for a register containing that adcon. If one is found, the values X2 = 0, B2, and D2 are returned, where B2 is the register and D2 is the displacement indicated in the variable item. If the adcon is not in a register, MEMAC selects one and loads the adcon into it. It then returns X2, B2, and D2 as above.

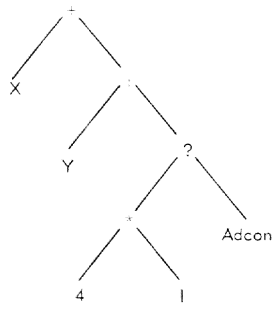
The plus generator will now generate the instruction

```
LE  4, D2(0,B2)
```

MEMAC is then entered to obtain cover for B. In similar fashion it returns new (although possibly the same) values of B2 and D2, and the plus generator produces

```
AE  4,D2(0,B2)
```

Consider next the expression X + Y(I) where X and Y are of type REAL*4, Y is an array, and I is neither a loop index variable nor removable from a DO loop. The expression is represented in the expression tree as:



where the effective displacement, D2, is given in the variable item Y, and the rest of the storage assignment and subscript offset has been subsumed in the item indicated by "adcon". It should also be pointed out that if generation is now taking place for the plus operator, generation of all lower level operations has taken

place -- in particular, the formation of the product 4*I.

MEMAC is first asked for cover for X which -- as before -- results in generation of, perhaps,

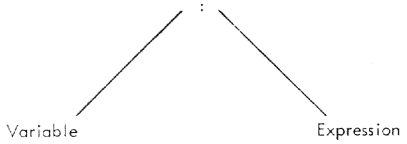
```
LE  2,D2(0,B2)
```

where D2 plus the contents of B2 is the address of X.

Then MEMAC is asked for cover for the right operand of the plus, the subscript connector (: operator). MEMAC looks beneath the colon and finds a base/index connector (? operator). Each of the operands of the ? operator is now individually obtained in general registers, either by locating the quantity in a register, or by selecting a register and loading the quantity as above. D2 is obtained from the variable item Y, and the registers containing the left and right operands of the ? operator are returned as X2 and B2, respectively. The plus generator now completes the addition by generating

```
AE  2, D2(X2,B2)
```

Subscripted variables containing only expressions that are removed from and computed outside a DO loop have the form



where the right-hand operand of the colon operator is the removed expression. In this case, the expression is not introduced by a ? operator, and MEMAC simply ensures that the expression is in a register, as above, and returns D2 from the variable item, X2 = 0, and B2 as the register containing the removed expression.

DO Loop Processing

The generation associated with DO loop control, removed expressions, and recursive expressions is governed by the begin loop 1(BL1), begin loop 2 (BL2), begin loop 3 (BL3), and end loop PF items. The processes performed and the code generated at BL1 and BL2 are "out of the loop"; that is, they are considered as loop initialization and are performed only once prior to entry to a loop, not each time through the loop. BL3 and end loop mark the scope of the loop, and any code generated or processing done therein is considered to be within the loop.

The BL1 entry contains a list of as many as eight quantities (adcons or common expressions) that are to be globally assigned to general registers across the upcoming loop. This list is scanned and transcribed to the current level of the loop push-down table, or loop table (see Figure 28). During this process, for each entry which is a common expression, the G (globally assigned) flag of the corresponding name table entry is set.

After the global assignment list has been transcribed to the loop table, a second list is scanned. This is a linked list of removed, nonrecursive common expressions in the PF. Each of these expressions is presented to the arithmetic expression generator for processing. This generation completes the processing at BL1.

The processing at BL2 generates the computations associated with recursive expressions, assures that all quantities which are to be globally assigned in the loop are now in registers and globally assigned, and clears register storage in all registers which contain quantities not to be globally assigned. If any such quantity is not also in storage (not a variable, constant, subscripted variable, or a previously stored common expression), temporary storage is obtained, and the quantity is stored.

Recursive expressions are those whose highest level operator is the recursive operator (!). A recursive expression is a constituent of, or perhaps all of, an effective relative address. The recursive operator has been introduced in a subscript expression in place of any + operator whose left-hand operand is a function of a loop induction variable. At the same time, the occurrence of the induction variable has been replaced by the induction variable increment size specified in the corresponding DO statement. The recursive expression is to be initialized outside the loop and incremented after each pass through the loop. The recursive operator has the property that its right-hand operand is the initial value of the corresponding recursive expression, and its left-hand operand is the increment to be applied to the expression at the end of the loop.

A loop table entry is made for each recursive processed at BL2. The entry contains the name of the recursive (the recursive is always a common expression) and an expression-tree-like item representing the increment to the recursive entry. This information is retrieved from the loop table at the loop end in order to increment the appropriate recursive expression.

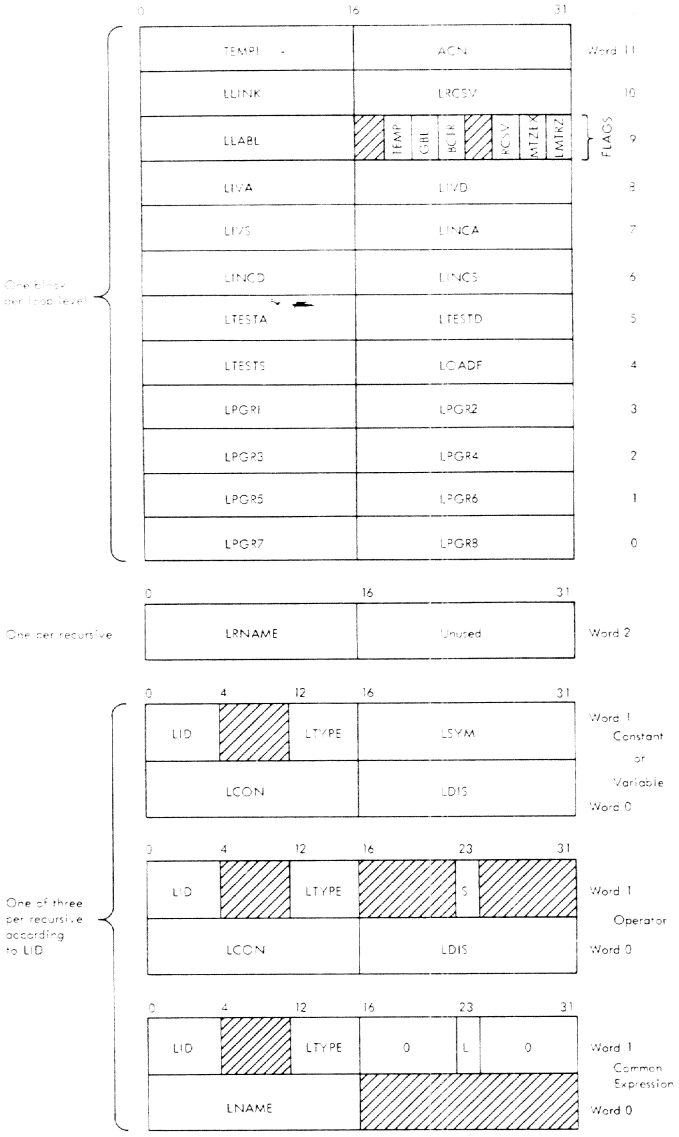


Figure 28. Loop Table

Legend for Figure 28

TEMP1 Byte number of lowest temporary assigned at next outer DO Loop.

LLINK Link to Loop Table entry for next outer DO Loop.

LRCSV Number of recursive expression to be incremented at this end DO.

LLABL Symbol Table pointer to compiler-created label marking beginning of DO scope. Not applicable to level zero loop.

FLAGS

Mnemonic	Mask value	Meaning
LMTRZ	X'01'	Loop variable materialized; there is no test expression. Otherwise, loop variable not materialized; last recursive expression is followed by a test expression.
MTZEX	X'02'	Loop variable is to be materialized (calculated) on exit from loop.
RCSV	X'04'	BXLE on recursive. Loop controlled at loop bottom by BXLE instruction.
	X'08'	Not used.
BTR	X'10'	Loop controlled at loop bottom by BCTR instruction.
GBL	X'20	Registers 14 and 15 may be globally assigned over loop. Loop bottom instruction will be BXLE 1,14,d (B1).
TEMP	X'40'	Registers 14 and 15 must be put in temporary storage before loop is entered, and must be restored at loop bottom before the BXLE 1,14,d (B1) is generated.
	X'80'	Not used.

LIVA Loop variable covering-Adcon pointer.

LIVD Loop variable D2 field.

LIVS Loop variable Symbol Table pointer.

LINCA Variable or constant increment covering-Adcon pointer.

LINCD Variable or constant increment D2 field.

LINCS Variable or constant increment Symbol Table pointer.

LTESTA Variable or constant upper limit covering-Adcon pointer.

LTESTD Variable or constant upper limit D2 field.

LTESTS Variable or constant upper limit Symbol Table pointer.

LPGR1 Up to 8 globally assigned quantities.

LPGR8
LPGRi = 8000 marks end of list.

LPGRi<7000 indicates Adcon Symbol Table pointer.

7000<LPGRi<8000 indicates common expression whose name is LPGRi-7000.

LRNAME Common expression name of recursive expression.

LID ID of increment or test expression:

1 = Operator
2 = Common Expression
5 = Variable
6 = Constant

LTYPE Type of increment or test expression:

3 = Integer*2
4 = Integer*4

LSYM Symbol Table pointer of variable or constant increment.

LCON Symbol Table pointer of Adcon covering constant or variable increment or temporary assigned to increment or test expression.

LDIS D2 field for variable, constant, or temporary reference.

S Tree sign of increment or test expression operator at creation of expression.

LNAME Name of increment or test expression if a common expression.

L L = 1 if last use of increment or test expression was encountered prior to Loop end; otherwise, L = 0.

When the recursive processing is complete, the BL2 PF entry is examined to determine whether materialization of the loop induction variable is required. Such materialization implies that the value of the induction variable will be maintained in the storage location assigned to it. If materialization is indicated, instructions are generated to load the initial value of the induction variable into a general register and store it into its assigned location. Symbol Table pointers to the induction variable, increment quality, and upper limit quantity are now recorded in the appropriate loop table fields.

If materialization is not indicated, the arithmetic expression generator is called to generate the "test expression."

It is assumed that whichever recursive expression has been processed last will also be processed last at the end of the loop (incremented there), and that this recursive will be tested against the test expression to determine whether the loop has been traversed the requisite number of times. The information identifying the test expression is added to the loop table following the entry for the last recursive expression, in just the same manner.

Next, any quantity to be globally assigned over this loop and not already so assigned is processed. If it is already in one of general registers 2 through 12, the corresponding MRM table entry is marked "globally assigned". If it is in some other general register, or not in a register at all, then one of the registers 2 through 12 is selected and made available, and the quantity is loaded into the selected register. Finally, all the floating-point registers and any general registers that are not now globally assigned are stored and cleared if their contents are not already in storage, or are cleared, otherwise. This completes the processing at BL2.

At BL3, four tasks are performed. First, the compiler-created statement label, marking the first instruction inside the loop, is placed in the code file and is identified by an entry in the loop table. Second, the arithmetic expression generator processes any expressions (dependent upon the induction variable) which may have been removed to BL3. Third, any temporary storage locations assigned at this loop top or at the next higher level are protected from reuse within the loop. Finally, the loop table is "pushed down" one level, ready to record information about any inner loop that may be encountered.

When an end-loop item is encountered in the PF, the loop table is "popped up" one level so that the information regarding the loop now ending is once again in evidence. Instructions are generated to increment each recursive expression listed in the loop table entry by the amount specified therin. Next, the end-of-loop test is generated. If materialization was required for this loop, instructions are generated to increment and test the induction variable and to branch conditionally (BNL) to the loop top. If materialization was not required, the last recursive incremented is now tested against the test expression, and a conditional branch (BNE) to the loop top is generated.

Next, the global assignment list is scanned, and for each entry the global assignment flag of its corresponding MRM table entry is cleared. Finally, all tem-

poraries assigned to the next outer DO level and released within the loop now ending are made available for reassignment.

ROUTINE DESCRIPTIONS

Phase 4 routines bear mnemonic titles as well as coded labels. The 5-character coded labels begin with the letters CEK; the fourth and fifth letters identify a specific routine. Most routines have only one entry point; for those that have multiple entries, both the coded labels and the mnemonics are given for the alternate entries. Any mnemonic name beginning with the letters TEV refers to an Exec routine or entry point, rather than to a Phase 4 routine. The corresponding coded label is given in parentheses immediately following the mnemonic.

There are no hardware configuration requirements for any Phase 4 routines. All these routines are reentrant, nonresident, nonprivileged, and closed. All except Phase 4 Master Control (CEKNX) use the restricted linkage conventions. Return codes and output parameters, if any, are noted in the routine descriptions that follow Table 24.

The relationships of routines constituting this phase are shown in the following nesting chart (Figure 29) and decision table (Table 24). The relationships are shown in terms of levels; a called routine is considered to be one level lower than the calling routine. Phase 4 Master Control is considered to be level 1.

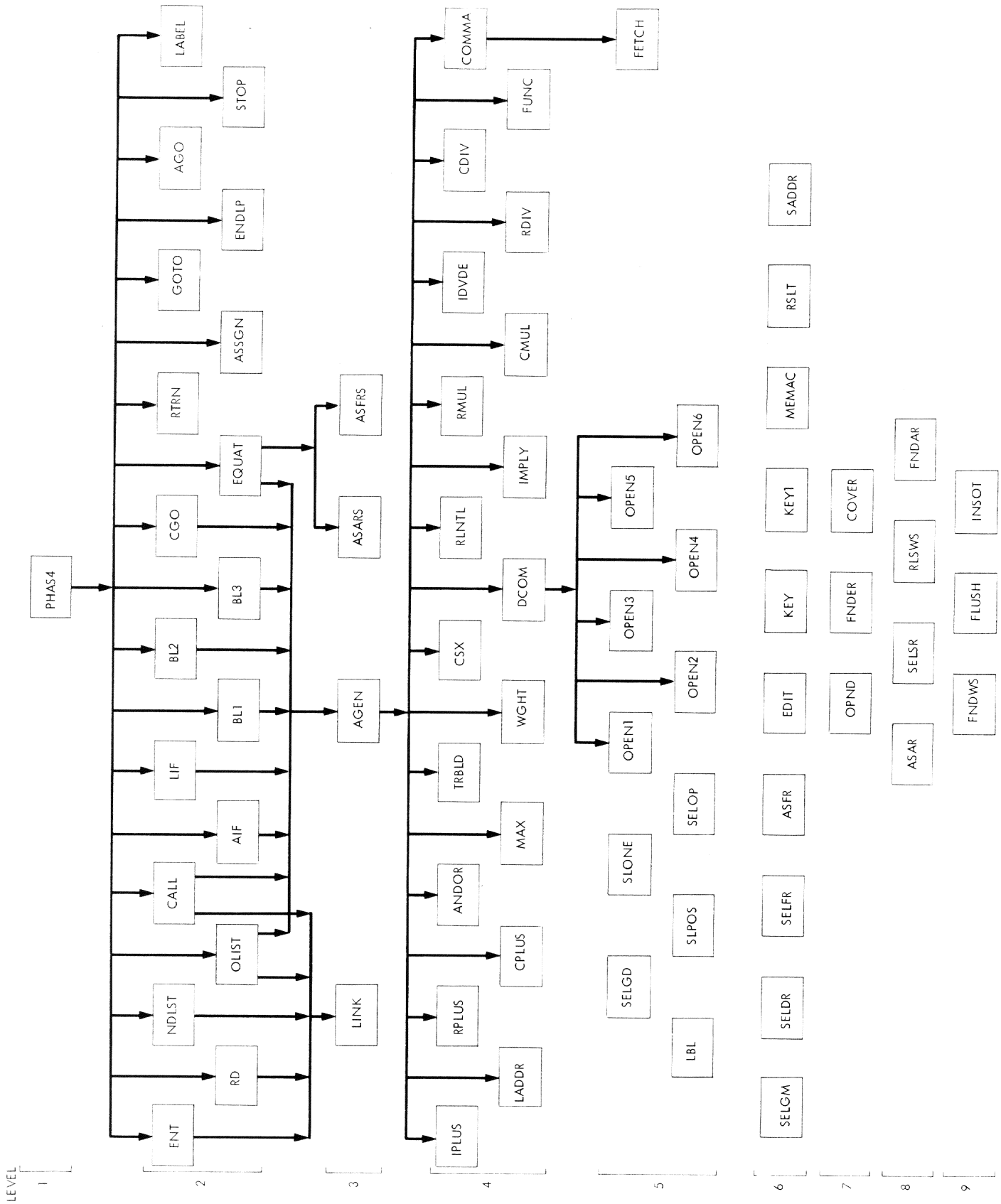


Figure 29. Phase 4 Nesting Chart

Table 24. Phase 4 Decision Table (Part 1 of 12)

Routine:----------Phase 4----------Level: 1----------

Routine	Usage	Routines Called	Calling Conditions
PHAS4	Phase 4 master controller	ENT	To generate main program, subprogram, or entry prologue.
		LABEL	To process Statement Label definition.
		EQUAT	To generate for arithmetic statement.
		GOTO	To generate unconditional GO TO.
		AGO	To generate assigned GO TO.
		CALL	To generate CALL.
		RTRN	To generate epilogue.
		BL1	To generate removed expressions.
		BL2	To generate recursives and make global assignments.
		BL3	To generate DO loop top.
		ENDLP	To generate for end of DO loop.
		OLIST	To generate for I/O list.
		NDLIST	To generate for end of I/O list.
		RD	To generate for I/O initialization.
		INSOT	To make code file entry.
		CGO	To generate computed GO TO.
		ASSGN	To generate ASSIGN.
		AIF	To generate arithmetic IF.
		LIF	To generate logical IF.
		STOP	To generate STOP and PAUSE statements.
		TEVRDM	To issue diagnostic message.

Routine:----------Phase 4----------Level: 2----------

ENT	Generate main program, subprogram, or entry prologue.	INSOT	To make code file entry.
		LBL	To generate for branch to label.
		LINK	To generate load of V/R Adcon pair.
		SELSR	To select single general register.
		ASAR	To assign a general register.
		FNDAR	To search general register table.
		TEVI4	To file an INTEGER*4 constant.
		TEVFL4	To file an address constant.
LABEL	Process Statement Label definition.	INSOT	To make code file entry.
		FLUSH	To reset and/or transfer register table entry.
EQUAT	Generate for arithmetic statement.	AGEN	To generate expression.
		SELSR	To select single general register.
		SELFR	To select single floating register.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		ASAR	To assign a general register.
		ASFR	To assign a floating register.
		ASARS	To assign a general register synonym.
		ASFRS	To assign a floating register synonym.
		OPND	To process operand.
		INSOT	To make code file entry.
		KEY1	To determine status of single operand.
		TEVI2	To file an INTEGER*2 constant.
		TEVI4	To file an INTEGER*4 constant.
		TEVR4	To file a REAL*4 constant.
		TEVR8	To file a REAL*8 constant.
		TEVC8	To file a COMPLEX*8 constant.
		TEVC16	To file a COMPLEX*16 constant.

Table 24. Phase 4 Decision Table (Part 2 of 12)

Routine:----------Phase 4----------Level: 2-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
GOTO	Generate unconditional GO TO.	INSOT	To make code file entry.
		LBL	To generate for branch to label.
AGO	Generate assigned GO TO.	SELSR	To select single general register.
		COVER	To load specified Adcon into any general register.
		INSOT	To make code file entry.
CGO	Generate computed GO TO.	INSOT	To make code file entry.
		FNDAR	To search general register table.
		SELSR	To select single general register.
		MEMAC	To get cover for storage reference.
		AGEN	To generate expression.
		SADDR	To get local branch cover.
		TEVFL4	To file an address constant.
		TEVFL5	To file a parameter list entry.
		TEVCRL	To create a label for the code file.
		TEVFLL	To make Symbol Table entry for created label.
ASSIGN	Generate ASSIGN.	INSOT	To make code file entry.
		MEMAC	To get cover of storage reference.
		TEVFL4	To file an address constant.
AIF	Generate arithmetic IF.	AGEN	To generate expression.
		FNDAR	To search general register table.
		FNDFR	To search floating register table.
		MEMAC	To get cover for storage reference.
		SELSR	To select single general register.
		SELFR	To select single floating register.
		ASFR	To assign a floating register.
		ASAR	To assign a general register.
		INSOT	To make code file entry.
		OPND	To process operand.
		LBL	To generate for branch to label.
LIF	Generate for logical IF.	AGEN	To generate expression.
		OPND	To process operand.
		FNDAR	To search general register table.
		FNDFR	To search floating register table.
		MEMAC	To get cover for storage reference.
		INSOT	To make code file entry.
		ASAR	To assign a general register.
		LBL	To generate for branch to label.
CALL	Generate CALL.	AGEN	To generate expression.
		INSOT	To make code file entry.
		SADDR	To get local branch cover.
		LINK	To generate load of V/R Adcon pair.
		TEVFL4	To file an address constant.
		TEVFL5	To file a parameter list entry.
		TEVCRL	To create a label for the code file.
		TEVFLL	To make Symbol Table entry for created label.
RTRN	Generate epilogue.	INSOT	To make code file entry.
		FNDAR	To search general register table.
		FNDFR	To search floating register table.
		TEVFL4	To file an address constant.

Table 24. Phase 4 Decision Table (Part 3 of 12)

Routine:----------Phase 4----------Level: 2-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
BL1	Generate removed expressions.	AGEN	To generate expression.
		INSOT	To make code file entry.
		OPND	To process operand.
BL2	Generate recursives and make global assignments.	KEY1	To determine status of single operand.
		OPND	To process operand.
		RSLT	To protect operand.
		COVER	To load specified Adcon into any general register.
		FNDAR	To search general register table.
		FLUSH	To reset and/or transfer register table entry.
		AGEN	To generate expression.
		MEMAC	To get cover for storage reference.
		INSOT	To make code file entry.
		SELSR	To select single general register.
		ASAR	To assign a general register.
		FNDWS	To get next available temporary storage.
		PH4MES	To make table overflow error exit.
BL3	Generate DO loop top.	AGEN	To generate expression.
		INSOT	To make code file entry.
		OPND	To process operand.
		FNDWS	To get next available temporary storage.
		COVER	To load specified Adcon into any general register.
		PH4MES	To make table overflow error exit.
ENDLP	Generate for end of DO loop.	SELSR	To select single general register.
		MEMAC	To get cover for storage reference.
		INSOT	To make code file entry.
		FNDAR	To search general register table.
		OPND	To process operand.
		COVER	To load specified Adcon into any general register.
		ASAR	To assign a general register.
		KEY1	To determine status of single operand.
		EDIT	To set comment item for code file.
		LBL	To generate for branch to label.
		RLSWS	To release temporary storage.
		PH4MER	To make machine/compiler error exit.
RD	Generate for I/O initialization.	INSOT	To make code file entry.
		SELSR	To select single general register.
		SELFR	To select single floating register.
		LINK	To generate load of V/R Adcon pair.
		TEVFL4	To file an address constant.
		TEVFL5	To file a parameter list entry.
		TEVVR	To file a V/R address constant pair.
OLIST	Generate for I/O list.	LINK	To generate load of V/R Adcon pair.
		SELSR	To select single general register.
		INSOT	To make code file entry.
		AGEN	To generate expression.
		MEMAC	To get cover for storage reference.
		OPND	To process operand.
		SELFR	To select single floating register.
		TEVFL4	To file an address constant.
		TEVFL5	To file a parameter list entry.
		TEVVR	To file a V/R address constant pair.

Table 24. Phase 4 Decision Table (Part 4 of 12)

Routine:----------Phase 4----------Level: 2-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
NDLST	Generate for end of I/O list.	LINK	To generate load of V/R Adcon pair.
STOP	Generate for STOP and PAUSE statements.	INSOT	To make code file entry.
		SELSR	To select single general register.
		SELFR	To select single floating register.
		TEVFL4	To file an address constant.
		TEVFL5	To file a parameter list entry.
		TEVVR	To file a V/R address constant pair.

Routine:----------Phase 4----------Level: 3----------

Routine	Usage	Routines Called	Calling Conditions
LINK	Generate load of V/R Adcon pair.	SELFR	To select single floating register.
		SELSR	To select single general register.
		INSOT	To make code file entry.
		TEVFL4	To file an address constant.
		TEVVR	To file a V/R address constant pair.
AGEN	Generate expression.	TRBLD	To convert Polish expression to tree form.
		WGHT	To determine order of computation.
		CSX	To count common expression uses.
		IPLUS	To generate integer addition.
		LADDR	To generate addition with LOAD ADDRESS.
		RPLUS	To generate real addition.
		CPLUS	To generate complex addition or subtraction.
		IMPLY	To generate integer multiplication.
		RMUL	To generate real multiplication.
		CMUL	To generate complex multiplication.
		IDVDE	To generate integer division.
		RDIV	To generate real division.
		CDIV	To generate complex division.
		RLTNL	To generate relational operations.
		ANDOR	To generate logical AND or OR.
		MAX	To generate for MAX operator.
		COMMA	To get function argument in storage with correct sign.
		FUNC	To generate function call.
		DCOM	To select open function module.
ASARS	Assign a general register Synonym.	None	
ASFRS	Assign a floating register synonym.	None	

Routine:----------Phase 4----------Level: 4----------

Routine	Usage	Routines Called	Calling Conditions
TRBLD	Convert Polish expression to tree form.	None	
WGHT	Determine order of computation.	None	
CSX	Count common expression uses.	None	

Table 24. Phase 4 Decision Table (Part 5 of 12)

Routine:----------Phase 4----------Level: 4-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
IPLUS	Generate integer addition.	KEY	To determine status of two operands.
		SLONE	To optimize storage-register operand-situation.
		SELOP	To optimize storage-storage operand-situation.
		SLPOS	To optimize storage-register operand-situation.
		SELSR	To select single general register.
		ASAR	To assign a general register.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		OPND	To process operand.
		RSLT	To protect operand.
		INSOT	To make code file entry.
LADDR	Generate addition with LOAD ADDRESS.	KEY1	To determine status of single operand.
		RSLT	To protect operand.
		OPND	To process operand.
		ASAR	To assign a general register.
		INSOT	To make code file entry.
		SELSR	To select single general register.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
RPLUS	Generate real addition.	KEY	To determine status of two operands.
		SELOP	To optimize storage-storage operand-situation.
		OPND	To process operand.
		INSOT	To make code file entry.
		SLPOS	To optimize register-register operand-situation.
		SELFR	To select single floating register.
		MEMAC	To get cover for storage reference.
		RSLT	To protect operand.
		ASFR	To assign a floating register.
		EDIT	To set comment item for code file.
CPLUS	Generate complex addition or subtraction.	KEY	To determine status of two operands.
		ASFR	To assign a floating register.
		OPND	To process operand.
		INSOT	To make code file entry.
		SELFR	To select single floating register.
		MEMAC	To get cover for storage reference.
		RSLT	To protect operand.
		SELOP	To optimize storage-storage operand-situation.
		EDIT	To set comment item for code file.
		SLPOS	To optimize register-register operand-situation.
		TEVR4	To file a REAL*4 constant.
		TEVR8	To file a REAL*8 constant.
		TEVC8	To file a COMPLEX*8 constant.
		TEVC16	To file a COMPLEX*16 constant.

Table 24. Phase 4 Decision Table (Part 6 of 12)

Routine:----------Phase 4----------Level: 4-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
IMPLY	Generate integer multiplication.	KEY	To determine status of two operands.
		SLONE	To optimize storage-register operand-situation.
		SLPOS	To optimize register-register operand-situation.
		SELOP	To optimize storage-storage operand-situation.
		SELSR	To select single general register.
		EDIT	To set comment item for code file.
		INSOT	To make code file entry.
		OPND	To process operand.
		RSLT	To protect operand.
		MEMAC	To get cover for storage reference.
		ASAR	To assign a general register.
		FLUSH	To reset and/or transfer register table entry.
		SELDR	To select even/odd register pair.
RMUL	Generate real multiplication.	KEY	To determine status of two operands.
		MEMAC	To get cover for storage reference.
		OPND	To process operand.
		RSLT	To protect operand.
		EDIT	To set comment item for code file.
		SLPOS	To optimize register-register operand-situation.
		SELOP	To optimize storage-storage operand-situation.
		SLONE	To optimize storage-register operand-situation.
		INSOT	To make code file entry.
		SELFR	To select single floating register.
		ASFR	To assign a floating register.
CMUL	Generate complex multiplication.	KEY	To determine status of two operands.
		SELFR	To select a floating register.
		ASFR	To assign a floating register.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		INSOT	To make code file entry.
		OPND	To process operand.
		RSLT	To protect operand.
		SELOP	To optimize storage-storage operand-situation.
		SLPOS	To optimize register-register operand-situation.
		SLONE	To optimize storage-register operand-situation.
IDVDE	Generate integer division.	KEY	To determine status of two operands.
		SELOP	To optimize storage-storage operand-situation.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		SLONE	To optimize storage-register operand-situation.
		SELGD	To determine whether to divide in place.
		SELDR	To select even/odd register pair.
		INSOT	To make code file entry.
		OPND	To process operand.
		RSLT	To protect operand.

Table 24. Phase 4 Decision Table (Part 7 of 12)

Routine:----------Phase 4----------Level: 4-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
IDVDE (Cont'd)	Generate integer division.	ASAR	To assign a general register.
		FLUSH	To reset and/or transfer register table entry.
		SELSR	To select single general register.
RDIV	Generate real division.	KEY	To determine status of two operands.
		SELFR	To select single floating register.
		ASFR	To assign a floating register.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		OPND	To process operand.
		RSLT	To protect operand.
		INSOT	To make code file entry.
		SELOP	To optimize storage-storage operand-situation.
		SLONE	To optimize storage-register operand-situation.
CDIV	Generate complex division.	KEY	To determine status of two operands.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		OPND	To process operand.
		RSLT	To protect operand.
		SELOP	To optimize storage-storage operand-situation.
		SLONE	To optimize storage-register operand-situation.
		ASFR	To assign a floating register.
		SELFR	To select single floating register.
		INSOT	To make code file entry.
COMMA	Get function argument in memory with correct sign.	FETCH	To fetch complement and/or store operand.
		MEMAC	To get cover for storage reference.
		SELSR	To select single general register.
		INSOT	To make code file entry.
		ASAR	To assign a general register.
		OPND	To process operand.
		SELFR	To select single floating register.
		FNDWS	To get next available temporary storage.
		ASFR	To assign a floating register.
		TEVI4	To file an INTEGER*4 constant.
		TEVR4	To file a REAL*4 constant.
		TEVR8	To file a REAL*8 constant.
		TEVC8	To file a COMPLEX*8 constant.
		TEVC16	To file a COMPLEX*16 constant.
FUNC	Generate function call.	SELSR	To select single general register.
		SELFR	To select single floating register.
		ASAR	To assign a general register.
		INSOT	To make code file entry.
		MEMAC	To get cover for storage reference.
		OPND	To process operand.
		EDIT	To set comment item for code file.
		COVER	To load specified Adcon into any general register.
		COMMA	To get function argument in storage with correct sign.
		RLSWS	To release temporary storage.
		ASFR	To assign a floating register.
		TEVFL4	To file an address constant.
		TEVFL5	To file a parameter list entry.
		TEVVR	To file a V/R address constant pair.

Table 24. Phase 4 Decision Table (Part 8 of 12)

Routine:----------Phase 4----------Level: 4-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
RLTNL	Generate relational operations.	KEY	To determine status of two operands.
		SLONE	To optimize storage-register operand-situation.
		SLPOS	To optimize register-register operand-situation.
		SELOP	To optimize storage-storage operand-situation.
		SELSR	To select single general register.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		INSOT	To make code file entry.
		OPND	To process operand.
		RSLT	To protect operand.
		SADDR	To get local branch cover.
		LBL	To generate for branch to label.
		ASAR	To assign a general register.
		ASFR	To assign a floating register.
		SELFR	To select single floating register.
ANDOR	Generate logical AND or OR.	KEY	To determine status of two operands.
		SELOP	To optimize storage-storage operand-situation.
		SLONE	To optimize storage-register operand-situation.
		SLPOS	To optimize register-register operand-situation.
		SELSR	To select single general register.
		MEMAC	To get cover for storage reference.
		INSOT	To make code file entry.
		EDIT	To set comment item for code file.
		ASAR	To assign a general register.
		OPND	To process operand.
		RSLT	To protect operand.
		LBL	To generate for branch to label.
DCOM	Select open function module.	OPEN1	To generate selected open functions.
		OPEN2	To generate selected open functions.
		OPEN3	To generate selected open functions.
		OPEN4	To generate selected open functions.
		OPEN5	To generate selected open functions.
		OPEN6	To generate selected open functions.
MAX	Generate for MAX operator.	KEY	To determine status of two operands.
		SELOP	To optimize storage-storage operand-situation.
		SLPOS	To optimize register-register operand-situation.
		SLONE	To optimize storage-register operand-situation.
		MEMAC	To get cover for storage reference.
		EDIT	To set comment item for code file.
		OPND	To process operand.
		RSLT	To protect operand.
		INSOT	To make code file entry.
		SELSR	To select single general register.
		ASAR	To assign a general register.
		ASFR	To assign a floating register.
		SELFR	To select single floating register.
		SADDR	To get local branch cover.

Table 24. Phase 4 Decision Table (Part 9 of 12)

Routine:----------Phase 4----------Level: 5----------

Routine	Usage	Routines Called	Calling Conditions
OPEN1	Generate selected open functions.	INSOT	To make code file entry.
		ASAR	To assign a general register.
		KEY	To determine status of two operands.
		OPND	To process operand.
		SELSR	To select single general register.
		SELFR	To select single floating register.
		ASFR	To assign a floating register.
		MEMAC	To get cover for storage reference.
		RSLT	To protect operand.
		EDIT	To set comment for code file.
		SADDR	To get local branch cover.
		PH4MER	To make machine/compiler error exit.
OPEN2	Generate selected open functions.	OPND	To process operand.
		FLUSH	To reset and/or transfer register table entry.
		KEY	To determine status of two operands.
		EDIT	To set comment item for code file.
		SELFR	To select single floating register.
		SELDR	To select even/odd register pair.
		INSOT	To make code file entry.
		ASFR	To assign a floating register.
		SELSR	To select single general register.
		ASAR	To assign a general register.
		MEMAC	To get cover for storage reference.
		RSLT	To protect operand.
		PH4MER	To make machine/compiler error exit.
OPEN3	Generate selected open functions.	EDIT	To set comment item for code file.
		SELSR	To select single general register.
		KEY1	To determine status of single operand.
		INSOT	To make code file entry.
		ASAR	To assign a general register.
		SELDR	To select even/odd register pair.
		SELFR	To select single floating register.
		MEMAC	To get cover for storage reference.
		ASFR	To assign a floating register.
		OPND	To process operand.
		RSLT	To protect operand.
		PH4MER	To make machine compiler error exit.
OPEN4	Generate selected open functions.	INSOT	To make code file entry.
		ASAR	To assign a general register.
		KEY1	To determine status of single operand.
		FNDFR	To search floating register table.
		SADDR	To get local branch cover.
		SELFR	To select single floating register.
		ASFR	To assign a floating register.
		MEMAC	To get cover for storage reference.
		RSLT	To protect operand.
		OPND	To process operand.
		KEY	To determine status of two operands.
		EDIT	To set comment item for code file.
		SELSR	To select single general register.
		PH4MER	To make machine/compiler error exit.

Table 24. Phase 4 Decision Table (Part 10 of 12)

Routine:----------Phase 4----------Level: 5-(Cont'd)---

Routine	Usage	Routines Called	Calling Conditions
OPEN5	Generate selected open functions.	ASFR	To assign a floating register.
		KEY1	To determine status of single operand.
		OPND	To process operand.
		SELFR	To select single floating register.
		EDIT	To set comment item for code file.
		COVER	To load specified Adcon into any general register.
		SELSR	To select single general register.
		MEMAC	Get cover for storage reference.
		SELDR	To select even/odd register pair.
		INSOT	To make code file entry.
		ASAR	To assign a general register.
		RSLT	To protect operand.
		PH4MER	To make machine/compiler error exit.
		TEVR4	To file a REAL*4 constant.
		TEVR8	To file a REAL*8 constant.
OPEN6	Generate selected open functions.	OPND	To process operand.
		RSLT	To protect operand.
		KEY1	To determine status of single operand.
		EDIT	To set comment item for code file.
		SELSR	To select single general register.
		ASAR	To assign a general register.
		INSOT	To make code file entry.
		SELFR	To select single floating register.
		MEMAC	To get cover for storage reference.
		SELDR	To select even/odd register pair.
		ASFR	To assign a floating register.
		PH4MER	To make machine/compiler error exit.
LBL	Generate for branch to label.	FNDAR	To search general register table.
		SELSR	To select single general register.
		INSOT	To make code file entry.
		ASAR	To assign a general register.
SELGD	Determine whether to divide in place.	None	
SLPOS	Optimize register-register operand-situation.	SELGM	To get multiplicand in proper register.
SLONE	Optimize storage-register operand-situation.	SELGM	To get multiplicand in proper register.
SELOP	Optimize storage-storage operand-situation.	None	
FETCH	Fetch/complement and/or store operand.	SELSR	To select single general register.
		SELFR	To select single floating register.
		ASAR	To assign a general register.
		ASFR	To assign a floating register.
		EDIT	To set comment item for code file.
		COVER	To load specified Adcon into any general register.
		INSOT	To make code file entry.
		MEMAC	To get cover for storage reference.

Table 24. Phase 4 Decision Table (Part 11 of 12)

Routine:----------Phase 4----------Level: 6----------

Routine	Usage	Routines Called	Calling Conditions
SELGM	Get multiplicand in proper register.	SELSR INSOT	To select single general register. To make code file entry.
SELDR	Select even/odd register pair.	SELSR	To select single general register.
SELFR	Select single floating register.	FNDWS COVER INSOT TEVFL4	To get next available temporary storage. To load specified Adcon into any general register. To make code file entry. To file an address constant.
ASFR	Assign a floating register.	None	
EDIT	Set comment item for Code file.	None	
KEY	Determine status of two operands.	FNDAR FNDFR	Two search general register table. To search floating register table.
KEY1	Determine status of single operand.	FNDAR FNDFR	To search general register table. To search floating register table.
MEMAC	Get cover for memory references.	SELSR INSOT COVER FNDAR ASAR	To select single general register. To make code file entry. To load specified Adcon into any general register. To search general register table. To assign a general register.
RSLT	Protect operand.	COVER OPND FNDAR ASAR INSOT FLUSH FNDWS	To load specified Adcon into any general register. To process operand. To search general register table. To assign a general register. To make code file entry. To reset and/or transfer register table entry. To get next available temporary storage.
SADDR	Get local branch cover.	INSOT SELSR ASAR	To make code file entry. To select single general register. To assign a general register.

Routine:----------Phase 4----------Level: 7----------

OPND	Process operand.	RLSWS	To release temporary storage.
FNDFR	Search floating register table.	None	
COVER	Load specified Adcon into any general register.	SELSR FNDAR INSOT ASAR	To select single general register. To search general register table. To make code file entry. To assign a general register.

Table 24. Phase 4 Decision Table (Part 12 of 12)

Routine:----------Phase 4----------Level: 8----------

Routine	Usage	Routines Called	Calling Conditions
ASAR	Assign a general register.	None	
SELSR	Select single general register.	FLUSH	To reset and/or transfer register table entry.
		FNDWS	To get next available temporary storage.
		INSOT	To make code file entry.
		TEVFL4	To file an address constant.
RLSWS	Release temporary storage.	None	
FNDAR	Search general register table.	None	

Routine:----------Phase 4----------Level: 9----------

FNDWS	Get next available temporary storage.	TEVFL4	To file an address constant.
FLUSH	Reset and/or transfer register table entry.	None	
INSOT	Make code file entry.	None	

Routine:----------Phase 4----------Level: Executive Routines

TEVFL4	File an address constant.		
TEVFL5	File a parameter list entry.		
TEVVR	File a V/R address constant pair.		
TEVI2	File an INTEGER*2 constant.		
TEVI4	File an INTEGER*4 constant.		
TEVR4	File a REAL*4 constant.		
TEVR8	File a REAL*8 constant.		
TEVC8	File a COMPLEX*8 constant.		
TEVC16	File a COMPLEX*16 constant.		
TEVRDM	Issue a diagnostic message.		
TEVFLL	Make symbol table entry for created label.		
TEVCRL	Create a label for the code file.		

CEKNX -- Phase 4 Master Control (PHAS4)

The main object of PHAS4 is to perform a serial scan of program file entries and to select the appropriate statement processor for each PF entry. See Figure 30.

ENTRIES: PHAS4 is entered at its external entry point, CEKNX1, from the phase controller via standard linkage. It expects to receive the base of the compiler's intercom as a parameter. There are three additional entry points, PH4MES, PH4MER, and PH4XER, which are used only by Phase 4 subroutines upon a detected compiler error or suspected machine error.

EXITS: PHAS4 has four exits to the phase controller.

OPERATION: Upon entry from the phase controller, PHAS4 copies the compiler's intercom into the phase's PSECT. Three nonvolatile registers are established as phase-wide common registers and initialized to:

1. Symbol Table Base (N1)
2. Expression Tree Base (N2)
3. Name Table Base (N3)

The following areas are cleared:

1. Name Table
2. Temporary Storage Utilization Matrix

Program file entries are processed sequentially. The ID of each PF entry is used to select the statement processor. After processing the last PF entry, PHAS4 restores the compiler's intercom and returns to the phase controller.

CEKOD -- Entry Point Processor (ENT)

Subroutine ENT is used to generate the preamble at an entry point. See Chart DU.

ENTRIES: The entry point is CEKOD1. ENT expects a pointer to the PF item (describing the entry) in register P2.

EXITS: Normal exit only.

OPERATION: For all three types of entries -- entry at the beginning of a main program, main entry of a subprogram, and alternate entry of a subprogram -- ENT generates code necessary to save registers and establish PSECT cover as follows:

```
STM   14,12,12(13)
L     14,72(0,13)
ST    14,8(0,13)
ST    13,4(0,14)
LR    13,14
```

An address constant which covers local temporary storage is then filed in the symbol table, and the instruction

```
L     12,n(0,13)
```

is generated (n is the storage class 4 assignment of the Adcon just filed).

In addition to this "canned" code, which is common to all entries, ENT generates additional instructions depending upon the type of the entry.

In a main program the "canned" code is appended by a call to the Task Initialization Subroutine CHCDB1.

Presence of a parameter list upon an entry into a subprogram requires ENT to generate the necessary code for object time parameter processing. A program file (PF) item describing a subprogram main entry triggers sorting of the Adcon list in the formal argument adcon table (referred to by Phase 4 as FAAL and by Phase 3 as FAAT). The sort arranges the entries in the FAAL in the order of argument numbers (ANO). (Note: ANO is a local abbreviation only, used to define the uses of ANO in associated flowcharts.) Several entries with the same ANO are sorted according to the values of the adcons (A), as given in their corresponding symbol table entries.

Each parameter, as indicated by its argument number (the argument number equals STCL minus 128) in the PF item, is matched against the ANOs in the FAAL, and ENT generates code to combine adcon values with the value of the matching parameter and to store the completed parameter address into the appropriate location (SLOC), as given by the adcon symbol table entry.

A branch around the preamble is generated by ENT for alternate entries not preceded by STOP, RETURN, or branches (GO TO, IF, etc.).

CEKNU -- Referenced Label PF Entry Processor (LABEL)

LABEL is called by the PF scanner to process a referenced label program file entry. See Chart DV.

ENTRIES: Entry is to CEKNU1, with a pointer to a referenced label PF entry in register P2.

EXITS: Normal exit only. The output of LABEL is an updated symbol table entry for the label.

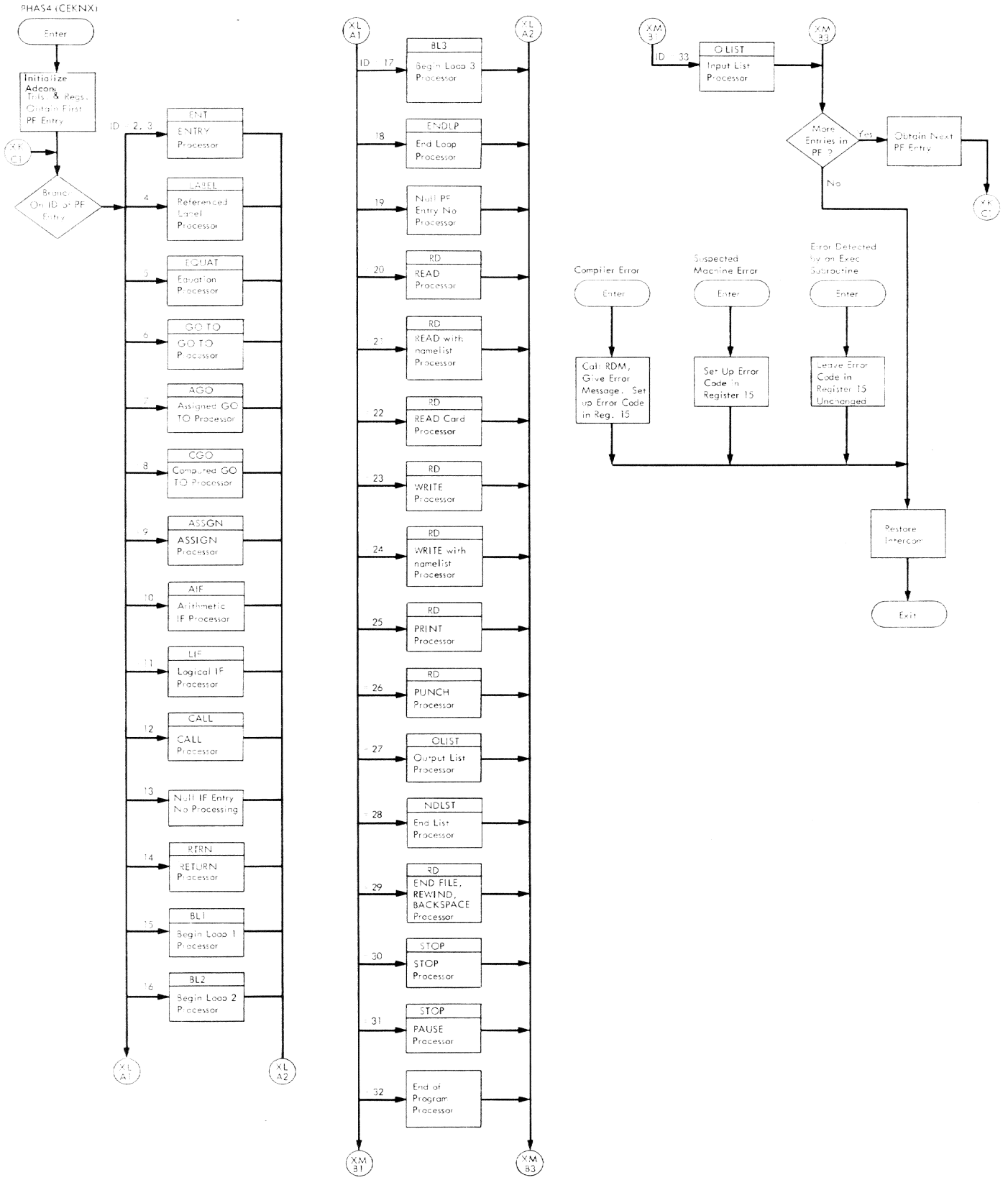


Figure 30. Phase 4 Master Control

OPERATION: LABEL stores the current contents of the location counter in the storage location field of the label's symbol table entry. The symbol table entry's storage class field is set to 1 (code file). Next, the symbol table pointer to the adcon entry is followed, and the storage class field of the name part of the adcon is examined. If it is equal to 1, exit is made; otherwise, the storage class field is set to 1, and the current contents of the location counter is stored in the value 1 field. All entries in the floating-point MRM table are cleared. All nonglobally assigned general register entries are cleared. Control then returns to the PF scanner.

CEKMJ -- Equation PF Entry Processor (EQUAT)

EQUAT is called by the PF scanner to process an equation program file entry. See Chart DW.

ENTRIES: Entry is to CEKMJ1, with a pointer to an equation PF entry in register P2.

EXITS: Normal exit only.

OPERATION: EQUAT first processes the right side of the equation by calling on AGEN and the appropriate lower level subroutines. Code is generated to produce the result, and to load it in a general or floating register. At this point all register table entries for variables and/or subscripted variables, except the right side entry for the current equation, are cleared to prevent potentially conflicting register usage arising through the use of EQUIVALENCE statements or other possible indirect variable definitions. AGEN is then called again to produce code, if necessary, for the left side of the equation. Next, the appropriate store instruction is generated to store the right side into the resultant address specified. OPND is called once for each side, to check for any final usages of common expressions. If the right side is a final usage of a CSX or a noncommon expression, the register containing it is assigned to the left side result. Otherwise, a synonym entry is made for the left side appropriate to its type. Control is then returned to the PF scanner.

CEKNK -- Arithmetic IF PF Entry Processor (AIF)

AIF is called by the PF scanner to process an arithmetic IF program file entry. See Chart DX.

ENTRIES: Entry is to CEKNK1, with a pointer to an arithmetic IF PF entry in register P2.

EXITS: Normal exit only.

OPERATION: AIF first processes the arithmetic expression by invoking AGEN which generates any code necessary to form the expression in an appropriate general or floating register. Next, the three transfer points are checked for fall-through condition (i.e., the label on the following statement matching one or more of the transfer points). No conditional branching code is generated for a transfer point where fall-through exists. Finally, the required conditional branch instructions are generated. Whenever possible, the conditional branch instructions are ordered such that transfer to points which are currently covered are executed first.

CEKNL -- Logical IF PF Entry Processor (LIF)

LIF is called by the PF scanner to process a logical IF program file entry. See Chart DY.

ENTRIES: Entry is to CEKNL1, with a pointer to a logical IF PF entry in register P2.

EXITS: Normal exit only.

OPERATION: LIF first determines the nature of the logical operand by means of a call on AGEN, with the logical IF flag set. If the logical operand is a noncommon or common subexpression just computed, AGEN generates the appropriate code for loading and testing the logical operand and for branching around the logical IF object statement, if necessary. If, however, the logical operand is a variable, constant, residue, or common expression computed previously, LIF generates the required code.

CEKNT -- GO TO PF Entry Processor (GOTO)

GOTO is called by the PF scanner to process to GO TO program file entry.

ENTRIES: Entry is to CEKNT1, with a pointer to a GO TO PF entry in register P2.

EXITS: Normal exit only.

OPERATION: GOTO supplies the required operation code and label symbol table pointer for LBL, which then generates the loading of any necessary adcons and the branching code.

CEKNS -- Assign PF Entry Processor (ASSGN)

ASSGN is called by the PF scanner to process an ASSIGN statement program file entry. See Chart DZ.

ENTRIES: Entry is to CEKNS1, with a pointer to an ASSIGN PF entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: ASSGN first creates an adcon to cover the label being referenced. Next, a load of the adcon is generated. Cover is obtained for the assigned variable, and a store instruction is generated to place the adcon in the assigned variable's location.

CEKNQ -- Assigned GO TO PF Entry Processor (AGO)

AGO is called by the PF scanner to process an assigned GO TO program file entry. See Chart EA.

ENTRIES: Entry is to CEKNQ1, with a pointer to an assigned GO TO PF entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: AGO selects a register for loading of the assigned variable. Cover is obtained for the assigned variable. Code is generated to load the assigned variable into the selected register and to branch unconditionally on the address contained therein. The register table entry for the assigned variable is cleared, and control is returned to the PF scanner.

CEKNR -- Computed GO TO PF Entry Processor (CGO)

CGO is called by the PF scanner to process a computed GO TO program file entry. See Chart EB.

ENTRIES: Entry is to CEKNR1, with a pointer to a computed GO TO PF entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: CGO first generates an adcon to cover the transfer list. The adcon's position in the adcon page is saved for future use in CGO. A label,which will be placed as the first element in the transfer list, is generated. The transfer produced is to the next statement following the computed GO TO. The label is accessed at object time when the computed GO TO index is found to be out of range. After the label has been generated, an adcon is filed to cover each element in the transfer list, including the generated label element. Next, code is generated to load the computed GO TO index variable in a register, if necessary, and test for the out-of-range condition. A register is then selected for the transfer list pointer, and code is generated to accomplish the appropriate transfer. Prior to exit, the register table entries made in CGO are cleared, and the current contents of the location counter and a storage class of 1 (code) are set in the generated label symbol table entry.

CEKOL -- CALL Statement Processor (CALL)

The objective of subroutine CALL is to generate object code for a CALL statement. See Chart EC.

ENTRIES: The entry point is CEKOL1. CALL expects a pointer to the program file entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: For a CALL statement with a parameter list, the object code is generated in subroutine FUNC (via AGEN). Calls to subroutines without parameters are processed by subroutine CALL.

In any case, subroutine CALL generates the code to process optional returns (RETURN i), if applicable.

CEKOE -- RETURN Processor (RTRN)

The objective of subroutine RTRN is to generate the code for a RETURN statement. See Chart ED.

ENTRIES: The entry point is DEKOE1. RTRN expects a pointer to the appropriate program file entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: Subroutine RTRN generates code to reestablish the caller's PSECT cover, reload general registers, set the low-order bit in the caller's forward link to 1, and set general register 15 with a return code. Either of two different instruction sequences is constructed, depending upon the return code being a constant (including no return code) or a variable.

CEKNM -- Begin Loop 1 PF Entry Processor (BL1)

BL1 is called by the PF scanner to process a begin loop 1 program file entry. See Chart EE.

ENTRIES: Entry is to CEKNM1,with a pointer to a begin loop 1 PF entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: BL1 scans the global reservation list, which is part of the BL1 PF entry, looking for CSX entries. For each CSX entry found, the global assignment bit in the CSX's corresponding name table entry

is set. If the loop is unsafe, the global temporary flag is set. The list of removed expressions is then processed, by calling AGEN to generate code for each removed expression in the list. When all list entries have been so processed, control is returned to the PF scanner.

CEKNN -- Begin Loop 2 PF Entry Processor (BL2)

BL2 is called by the PF scanner to process a begin loop 2 program file entry. See Chart EF.

ENTRIES: Entry is to CEKNN1, with a pointer to a begin loop 2 PF entry in parameter register P2.

EXITS: Normal exit is at the termination of begin loop 2 PF entry processing. Exit is via PH4MES in case of loop table overflow.

OPERATION: The BL2 processor generates code for computing all recursive subscript expressions. Processing is divided into two main sections. The first involves the generation of code to produce the initial value and increment expressions for each recursive expression. The second section produces the test value code.

Initial value processing is performed to ensure that code is generated to produce the initial value and load it either into a global register, if required, or into temporary storage. Increment processing performs the same functions for the increment, and, in addition, makes the appropriate entry in the loop table.

After the initial value and increment are processed for each recursive expression, the materialization flag is tested. If materialization is required, code is generated to load the beginning value of the induction variable in storage. The induction variable, increment, and end values are stored in the loop table. If materialization is not required, code is generated to produce the test value and store it or globally assign it, as required.

The final processing in BL2 rescans the global reservation list. All adcons and CSX's listed are globally assigned and loaded into registers, and all remaining active expressions in the general and floating registers are assigned temporary storage. Control then returns to the PF scanner.

CEKNO -- Begin Loop 3 PF Entry Processor (BL3)

BL3 is called by the PF scanner to process a begin loop 3 program file entry. See Chart EG.

ENTRIES: Entry is to CEKNO1, with a pointer to a begin loop 3 PF entry in parameter register P2.

EXITS: Normal exit is at the termination of begin loop 3 PF entry processing. Exit is via PH4MES in case of loop table overflow.

OPERATION: BL3 is responsible for processing all induction-variable-dependent common expressions which can be removed to just inside the loop top. Initially the temp bit matrix is searched locating the lowest available temp byte. BL3 then enters the loop top label's symbol table pointer in the loop table, then updates relevant fields within the symbol table entry for the loop top label. Next, the chain of induction-variable-dependent expressions is processed. Code is generated to produce each expression in the chain. After all expressions in the chain have been processed, control returns to the PF scanner.

CEKNP -- End Loop PF Entry Processor (ENDLP)

ENDLP is called by the PF scanner to process an end loop program file entry. See Chart EH.

ENTRIES: Entry is to CEKNP1, with a pointer to an end loop PF entry in parameter register P2.

EXITS: Normal exit is at the termination of end loop PF entry processing. Error exit is via PH4MER in the event that a globally assigned quantity is found not to be in a register.

OPERATION: The end loop PF entry processor commences by scanning the recursive entries in the loop table. For each entry code is generated to load the recursive expression in a register, if necessary, and increment it. After all recursive entries have been processed, the materialization flag in the loop table is tested. If materialization is not required, code is generated to compare the last recursive expression processed against the test value and to transfer control appropriately. If materialization is required, code is generated to load the induction variable (if necessary), increment it, store the new value back in the induction variable's storage cell, compare the new value of the induction variable against the test value, and transfer control appropriately.

Next, the global assignment list in the loop table for this level is examined. For each CSX entry the global assignment bit in the CSX's name table entry is cleared. The global assignment bit in the register table entry for the CSX is cleared. For each adcon entry, the global assignment bit in the register table entry for the adcon is cleared. Upon completion of this task, the location of the highest assigned temp, saved at exit from BL2, is restored; and, control is returned to the PF scanner.

CEKOH -- I/O Statement PF Entry Processor (RD)

The I/O statement PF entry processor is called by the PF scanner whenever READ, WRITE, PRINT, PUNCH, or file control PF entries are encountered. See Chart EI.

ENTRIES: Entry is to CEKOH1, with a pointer to the pf entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: The I/O statement PF entry processor is responsible for the generation of a standard linkage to the I/O initialization routine. Prior to generation, a parameter list (see Figure 31) is constructed to provide the information required by the initialization routine at object time. After the standard linkage code is generated, all floating register table entries for registers 1, 14, and 15 are cleared. Control then returns to the PF scanner.

CEKOI -- I/O List Element PF Entry Processor (OLIST)

OLIST is called by the PF scanner to process an I/O list program file entry. See Chart EJ.

ENTRIES: Entry is to CEKOI1, with a pointer to an I/O list element PF entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: OLIST is responsible for the generation of a standard linkage to the list item processor. Prior to generation, a parameter list is constructed to provide the information required by the list item processor at object time. In addition, if required, code is generated to compute the effective address of the list item at object time. Upon completion of all required generation, all floating register table entries and general register table entries for registers 1, 14, and 15 are cleared. Control then returns to the PF scanner.

Word	
1	Address of Data Set Reference Number or 0
2	Address of First Control Byte*
3	Address of Second Control Byte*
4	Address of FORMAT or NAMELIST Statement
5	Address of Error Exit
6	Address of End-of-File Exit

*See Figure 32 for format of control bytes.

Parameter 1 is zero for PRINT and PUNCH.

Parameters 3, 4, 5, and 6 are not supplied for REWIND, BACKSPACE, and END FILE.

Any or all of parameters 4, 5, and 6 may be missing for a given I/O statement.

Figure 31. I/O Initialization Parameter List

First Control Byte

23	25	27	28	31
0	R W P PU RW B E 0			

Second Control Byte

23	25	27	29	31
0	F L N ER E F		0	

R	-- READ	E	-- END FILE
W	-- WRITE	F	-- FORMAT
P	-- PRINT	L	-- List
PU	-- PUNCH	N	-- NAMELIST
RW	-- REWIND	ER	-- Error Exit
B	-- BACKSPACE	EF	-- End-of-File Exit
E	-- END FILE		

Control bytes are filed as I*4 constants.

Figure 32. I/O Initialization Control Bytes

CEKOJ -- End List PF Entry Processor (NDLST)

NDLST is called by the PF scanner to process an end list program file entry. NDLST also may be invoked at a second entry point -- LINK -- to generate a standard CALL linkage. See Chart EK.

ENTRIES: This routine has two entry points: NDLIST (CEKOJ1) and LINK (CEKOJ2). Entry to NDLST (CEKOJ1) is made with a pointer to an end list PF entry in parameter register P2. NDLST calls LINK.

Alternate entry, LINK (CEKOJ2), is entered with a symbol table pointer to the subprogram name in register P1.

EXITS: Normal exit only.

OPERATION: NDLST generates a standard linkage to the I/O library List Termination routine (CHCIU).

LINK resets the register tables for all floating-point registers and general registers 1, 14, and 15 and then proceeds to generate the code for a standard linkage.

CEKOK -- STOP and PAUSE Statement PF Entry Processor (STOP)

STOP is called by the PF scanner to process either a STOP or PAUSE program file entry. See Chart EL.

ENTRIES: Entry is to CEKOK1, with a pointer to either a STOP or PAUSE PF entry in parameter register P2.

EXITS: Normal exit only.

OPERATION: STOP is responsible for the generation of a standard linkage to either the STOP or PAUSE library subroutines. No distinction is made between the two cases within STOP, since the necessary distinguishing information is obtained from the same relative PF entry location in either case. Prior to code generation, STOP constructs a 1-entry parameter list with a pointer to the STOP or PAUSE message. Upon completion of code generation, all floating register table entries and general register table entries for registers 1, 14, and 15 are cleared. Control then returns to the PF scanner.

CEKNW -- Arithmetic Expression Generator (AGEN)

AGEN is used to construct expression trees and to resolve the trees by selecting the appropriate generators. See Chart EM.

ENTRIES: The entry point is CEKNW1. The input parameter, in register P2, is a pointer to the major operator of the Polish String in the program file.

EXITS: Normal exit only. The out parameter, in register P2, is a pointer to the major operator of the expression tree.

OPERATION: AGEN is invoked to generate arithmetic expressions. AGEN uses subroutine TRBLD to build the expression tree from program file entries. No more processing is required for a trivial tree where the major operator is either primitive or an already computed common expression. If the tree is more complex, AGEN invokes subroutine WGHT to assign relative weights to the nodes of the tree. These weights determine the sequence in which the tree is resolved by AGEN. Before resolving the tree, AGEN uses subroutine CSX to count the number of occurrences of common expressions in the expression and to record these counts in corresponding name table entries.

To resolve the tree, AGEN inspects the operands at each node, starting with the major operator. If one of the operands is not resolved, the pointer to it is installed as the current node pointer. Then its operands are examined. If both operands at a node are unresolved, the operand with the larger weight is inspected next. Generation occurs when a node is reached where both operands are resolved (primitive or already computed). AGEN selects and invokes the appropriate generator subroutine, based upon the operator in the tree entry. The generator used for plus, multiply, and divide operators depends on the type (integer, real, complex) of operands. After generation, AGEN marks the node and, in the case of a common expression, the corresponding name table entry "computed."

If the result represents a globally assigned common expression of integer type and if it is left in a general register other than 1, 14, or 15, AGEN sets the global assignment flag of the corresponding register entry in the register table.

If the backlink is present, it is installed as the current node pointer and the procedure is repeated by inspecting the operands of the node.

AGEN exits after the major operator has been processed and marked "computed."

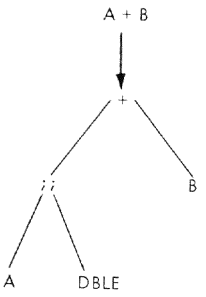
CEKML -- Expression Tree Builder (TRBLD)

TRBLD is entered by the arithmetic expression generator to convert an expression in the PF into tree-form representation in the expression tree. The occurrence of a common expression entry whose last use indicator is set will cause setting of the corresponding bit in the corresponding name table entry. (See Chart EN.)

In addition, when the operators +, * and / combine operands of certain mixed types,

open conversion functions are introduced to eliminate operand type-discrepancy.

For example:



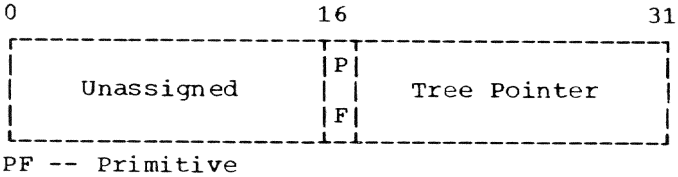
where A is REAL*4 and B is REAL*8.

ENTRIES: The entry point is CELML1. TRBLD expects a pointer to the major operator of the expression in the PF in parameter register P2.

EXITS: Normal exit only. Output consists of the expression tree, the accompanying name table entries, and a pointer to the major operator of the expression in parameter register P2.

OPERATION: The PF is scanned from the major operator until the left-most element of the expression has been located. This is accomplished by initializing a counter to zero, and by adding 1 for each operator encountered and subtracting 1 for each primitive encountered until the count becomes negative. The primitive last encountered, then, is the left-most element of the expression.

Once the left-most element has been found, the tree is built during a left-to-right scan of the PF. The push-down table STACK is used during this process. Whenever a constant or variable is encountered, the PF entry is converted to expression tree form and placed in the next available location in the tree. In addition, a pointer to the tree entry position is made at the top of the stack (see Figure 33).



PF -- Primitive

Figure 33. Stack Table Entry

When an operator or common expression entry is encountered, an entry is made in the next available tree location. At this point, the top two entries in the STACK point to the tree entries for the left and right operands of the operator just entered. (See Tables 25 and 26). The two pointers are inserted into the LOP and ROP fields of the operator entry. In addition, a backlink to the operator is placed in the BLINK field(s) of whichever of the operands is non-primitive. Now the STACK is "popped up" two levels, and the pointer to the current tree entry is "pushed down."

Table 25. Operand Conversion Function Decision Table

Right Operand Types

Left Operand Types (LOP) \ ROP	L*1	L*4	I*2	I*4	R*4	R*8	C*8	C*16
L*1	0	L*1 → L*4	1	1	1	1	1	1
L*4	L*1 → L*4	0	1	1	1	1	1	1
I*2	1	1	0	I*2 → I*4	I*2 → R*4	I*2 → R*8	I*2 → R*4	I*2 → R*8
I*4	1	1	I*2 → I*4	0	I*4 → R*4	I*4 → R*8	I*4 → R*4	I*4 → R*8
R*4	1	1	I*2 → R*4	I*4 → R*4	0	R*4 → R*8	0	R*4 → R*8
R*8	1	1	I*2 → R*8	I*4 → R*8	R*4 → R*8	0	C*8 → C*16	0
C*8	1	1	I*2 → R*4	I*4 → R*4	0	C*8 → C*16	0	C*8 → C*16
C*16	1	1	I*2 → R*8	I*4 → R*8	R*4 → R*8	0	C*8 → C*16	0

The decision table contains 64 two-byte entries.

0 - No conversion necessary (98)
1 - Illegal combination of operands (99)
All Others - Function number of the corresponding conversion function

Table 26. Complex Division Left Operand Conversion Function Decision Table

Right Operand Types

Left Operand Types: LOP \ ROP	C*8	C*16
I*2	I*2 → C*8	I*2 → C*16
I*4	I*4 → C*8	I*4 → C*16
R*4	R*4 → C*8	R*4 → C*16
R*8	C*8 → C*16	R*8 → C*16

This decision table contains 8 two-digit function numbers.

The tree building is terminated after the major operator of the expression has been processed.

CEKNE -- Weight Subroutine (WGHT)

WGHT is used by the arithmetic expression generator to assign to each nonprimitive node of the expression tree a weight (or priority) which will determine order of generation. The weight is such that, in deciding at a given node which branch to generate first (if neither branch has been generated), the branch which has the larger weight will be chosen. See Chart EO.

ENTRIES: The entry point is CEKNE1. WGHT is invoked by AGEN and expects to receive, in parameter register P2, the expression tree pointer to tht major operator.

EXITS: Normal exit only.

OPERATION: The following considerations enter into the assignment of weights:

1. In order to minimize the number of active partial results, branches are given weights according to their complexity.

2. In order to minimize register storing necessitated by function calls, branches containing such calls are given maximum weight.

3. In order to attempt to prevent the storage of a numerator owing to the complexity of the denominator, since the numerator must always exist in a register, the denominator, or right operand, of a division operator is given an arbitrary boost in priority.

When WGHT is entered, the tree pointer is set at the major operator of the expression. First, the left operand is inspected. If it is primitive, a computed common expression, or already weighted, the right operand is inspected. If it, too, is one of the above, the operator at the current node is weighted, as described below. If the left operand is none of the above, the tree pointer is set at the left operand entry, and the above process is repeated. If the left operand is one of the above, but the right operand is not, the tree pointer is set at the right operand, and the above process is repeated.

When, finally, both operands are computed, weighted, or primitive, a weight is assigned to the current node of the tree as follow:

1. If both operands are primitive or computed common expressions, the weight is set to zero.

2. If neither operand is a primitive or a computed common expression, then either WMAX or 1 plus the maximum of the operand weights is chosen, whichever is less. (Note: In the description of the Expression Tree (Appendix A) the WMAX field is identified as WEIGHT.)

3. If only one operand is neither a primitive nor a computed common expression, the weight of that operand is chosen.

When this tentative weight has been established, the type of operator at the current node is determined. If it is a function, the weight is set to WMAX. If not, the backlink (BLINK) is followed, to determine whether the current node is the right operand of a division. If it is, the weight is increased by 5 to a maximum of WMAX. If not and if it is the right operand of a colon, the weight is set to zero.

When the weight computation is complete, the current node is marked "weighted" and the weight is stored in the tree field reserved for this purpose. Then the backlink is tested. If it is empty, the major operator has been weighted, and the generation may proceed. If not, the backlink is installed as the tree pointer, and the entire process is repeated.

CEKOB -- Common Expression Usage Count (CSX)

CSX is used by the arithmetic expression generator (AGEN) to count the number of times each common expression occurs as an operand in a given expression tree. In addition subroutine CSX pushes down signs in the expression tree to the lowest practical level. See Chart EP.

ENTRIES: The entry point is CEKOB1. CSX is invoked by AGEN and expects to receive, in parameter register P2, the expression tree pointer to the major operator.

EXITS: Normal exit only.

OPERATION: CSX starts at the tree base and examines the operands at each node. The left operand is always inspected first. A node is marked "examined," if both operands are either primitive (variables, constants) or "examined" nodes. In addition, if the node represents a CSX, the usage count field in the appropriate name table entry is increased by 1. In all other conditions, the link to the current operand is installed as the node pointer, and the process just described is repeated. If a node represents a computed CSX, or residue, or if the name table entry of an uncomputed CSX shows a nonzero usage count, no further inspection of the operands of such node is performed, but the node is immediately marked "examined" and the usage count is updated. After a node has been examined and so marked, the backlink, if present, replaced the current node pointer. The whole process is terminated when the major operator of the tree has been examined.

CEKMC -- Real Plus Generator (RPLUS)

RPLUS is entered by the expression generator to generate the addition or subtraction of two operands of type REAL*4 or REAL+8. See Chart EQ.

ENTRIES: The entry point is CEKMC1. The input, in parameter register P2, is a pointer to the current node of the expression tree.

EXITS: Normal exit only.

OPERATION: The subroutine KEY is entered to determine the location of both operands (in storage or registers). If both operands are in storage, SELOP is entered to select the better operand to load. SELFR is entered to select a floating-point register. MEMAC is entered to assign B, X, and D fields and to load cover and/or index quantities as necessary. OPND is entered to release temporary storage for the operands when appropriate. An instruction is generated to load the selected operand into the selected register. MEMAC is entered to obtain cover for the nonselected operand. If the operands agree or disagree in sign, an add or subtract instruction, respectively, is generated. Finally a register memory entry is made for the current node of the expression tree.

If only one operand is in a register, that operand is designated as the selected operand. RSLT is entered to store that operand if necessary, and processing continues as above from the point at which MEMAC is entered for the nonselected operand.

If both operands are in registers, SLPOS is entered to select an operand register not requiring storage if altered. After the selection has been made, OPND is entered, and RSLT is entered to store the selected register if required. An RR add or subtract instruction is now generated, and ASFR is entered to make a register memory entry for the current node of the expression tree.

CEKMB -- Real Multiply Generator (RMUL)

RMUL is entered by the expression generator to generate the product of two operands of type REAL*4 or REAL*8. See Chart ER.

ENTRIES: The entry point is CEKMB1. The input is a pointer to the current node of the expression tree, passed in parameter regsiter P2.

EXITS: Normal exit only.

OPERATION: The subroutine KEY is entered to determine the location of both operands (in storage or registers). If both operands are in memory, SELFR is entered to select a floating-point register. MEMAC is entered to assign B, X, and D fields and to load cover and/or index quantities as necessary. OPND is entered to release temporary storage when appropriate. An instruction is generated to load the selected operand into the selected register. MEMAC is entered again to obtain cover for the nonselected operand. A multiply instruction is generated, and finally, ASFR is entered to make a register storage entry for the current node of the tree.

If only one operand is in a register, that operand is designated the selected operand. RSLT is entered to ensure storage of the operand if necessary, and processing continues as above from the point at which MEMAC is entered for the nonselected operand.

If both operands are in registers, SLPOS is entered to select a register not requiring storage if altered. OPND is entered to release storage and register memory for the nonselected operand, and RSLT is entered to ensure preservation of the selected operand. An RR multiply instruction is generated, and finally ASFR is entered to make a register storage entry for the current node of the tree.

CEKMA -- Real Divide Generator (RDIV)

RDIV is entered by the expression generator to generate the quotient of two operands of type REAL*4 or REAL*8. See Chart ES.

ENTRIES: The entry point is CEKMA1. The input is a pointer to the current node of the expression tree, passed in parameter register P2.

EXITS: Normal exit only.

OPERATION: The subroutine KEY is entered to determine the location of both operands (in storage or registers). If both operands are in storage, SELFR is entered to select a floating-point register. MEMAC is entered to assign B, X, and D fields, and to load cover and/or index quantities as necessary. OPND is entered to release temporary storage when appropriate. An instruction is generated to load the dividend into the selected register. MEMAC is entered again to obtain cover for the divisor. A divide instruction is generated, and finally ASFR is entered to make a register storage entry for the current node of the tree.

If the dividend is in a register and the divisor is in storage, RSLT is entered to ensure storage of the dividend if necessary, and processing continues as above from the point at which MEMAC is entered to obtain cover for the divisor.

If the divisor is in a register and the dividend is in storage, SELSR is entered to select a register other than the one containing the divisor. MEMAC is entered to assign B, X, and D fields and to load cover and/or index quantities as necessary. A load is generated to place the dividend into the selected register. OPND is entered to release temporary storage and register assigned to the divisor as appropriate. An RR divide instruction is generated, and ASFR is entered to assign the selected register to the current node of the expression tree.

If both operands are in registers, the register containing the dividend is designated as the selected register. OPND is entered to release the temporary storage and register assigned to the divisor when appropriate. RSLT is entered to store the dividend if necessary. Processing continues as above from the point at which the RR divide instruction is generated.

CEKMF -- Integer Plus Generator (IPLUS)

IPLUS generates the sum of two integer quantities of length two or four. See Chart ET.

ENTRIES: The entry point is CEKMF1. The expression tree address of the plus operator is expected in parameter register P2.

EXITS: Normal exit only.

OPERATION: The code generated depends upon whether the operands are in registers or in storage. Three cases are treated:

1. If neither operand is in a register, a register is selected and SELOP is entered. SELOP determines which operand should be loaded (become the augend or minuend); whether the load should be performed with an L (load) or LA or (load address) command; whether an addition or a subtraction is required; and what sign should be associated with the result.

2. If only one operand is in a register, SLONE is entered. SLONE determines whether an addition or subtraction is required; whether the operation may be performed in the register containing the operand; and what sign should be associated with the result. If the operation may not be performed in the register containing the operand, SLONE indicates whether that operand should be moved to another register before generation of an R-X addition or subtraction, or whether the operand in memory should be loaded into a register before generation of an R-R addition or subtraction. If loading of the operand from storage is indicated, SLONE specifies whether the load should be performed with a load or a load address instruction.

3. If both operands are in registers, SLPOS is entered. SLPOS determines whether an addition or subtraction must be performed; in which, if either, of the operand registers the operation is to be performed; and what sign should be associated with the result. If the operations may be performed in neither of the operand registers, SLPOS indicates which operand should be moved, and whether the move should be performed with a load register or a LCR (load complement) instruction.

IPLUS simply performs the generation indicated by the output from SELOP, SLONE, or SLPOS and assigns the operation result to the selected register.

CEKME -- Integer Multiply Generator (IMPLY)

IMPLY generates the product of two integer quantities of length two or four. See Chart EU.

ENTRIES: The entry point is CEKME1. The expression tree address of the multiplication operator is expected in parameter register P2.

EXITS: Normal exit only.

OPERATION: The code generated depends upon whether the operands are in registers or in storage. Three cases are treated:

1. If neither operand is in a register, SELOP is entered. SLOP determines which operand should be loaded; complements a constant operand and files the new value in the symbol table if such a procedure will result in a product with the desired sign; indicates whether the operand may be loaded by means of a load address instruction; and, indicates whether an operand is a constant power of 2 so that the other operand may be loaded and shifted appropriately.

2. If one operand is in a register, SLONE is entered. SLONE complements the storage operand and files the new value in the symbol table if the operand is a constant and if such a procedure will result in a product with the desired sign; specifies the result sign; indicates whether the storage operand is a constant power of 2 so that the register operand may simply be shifted by an appropriate amount; and, specifies whether the operation may take place in the register or register pair containing the multiplicand or whether that operand must be moved to another register or register pair before the multiplication, and whether the move must be done with a load register or a load complement instruction.

3. If both operands are in registers, SLPOS is entered. SLPOS selects the operand to be used as the multiplicand ("to" register); specifies the result sign; indicates that one operand is a constant power of 2 so that the product may be computed by shifting the other operand; and indicates that the multiplicand must be moved to another register before the multiplication is generated, specifying whether the move should be done with a load register or a load complement instruction.

IMPLY simply generates the multiplication as specified by SELOP, SLONE, or SLPOS and assigns the result to the selected register.

CEKMD -- Integer Divide Generator (IDVDE)

IDVDE is used to generate integer divisions of 2- and 4-byte quantities. See Chart EV.

ENTRIES: The entry point is CEKMD1. IDVDE expects the expression tree address of the division operator in parameter register P2.

EXITS: Normal exit only.

OPERATION: The instructions generated depend upon whether neither, one, or both operands are in registers:

1. If neither operand is in a register, SELOP is entered. SELOP files the complement of a constant operand in the symbol table if this procedure will produce the desired result sign; returns the result sign; and, determines whether the dividend may be loaded by means of a load address instruction.

2. If one operand is in a register, SLONE is entered. SLONE determines the result sign; files the complement of an operand in the symbol table if the operand is a constant and if such a procedure produces the desired result sign; determines whether the operand should be loaded with a load address command; and, determines whether the division may proceed in the register pair containing the numerator or it must be moved to another register pair.

3. If both operands are in registers, SELGD is entered to determine whether the division may proceed in the register pair containing the numerator or the numerator must first be moved to another register pair.

Depending upon the output of SELOP, SLONE, or SELGD, instructions are generated to perform the division, and the result is assigned to the selected register.

CEKOV -- Add by Load Address (LADDR)

LADDR is entered by the AGEN to generate the addition of two quantities (representing a recursive test expression) by means of a load address instruction. See Chart EW.

ENTRIES: The entry point is CEKOV1. LADDR expects restricted linkage convention. LADDR is parameter register P2 to contain the tree address of the major operator.

EXITS: Normal exit only.

OPERATION: LADDR is given the expression tree address of an operator whose two operands are to be added by means of a load address instruction. The operands are loaded into general registers (if not there already) and made to have the correct signs by means of LCR instructions when necessary. The two register numbers are made the X2 and B2 fields of a load address instruction. A third register is selected to contain the sum. This register becomes the R1 field of the LA instruction and may be the same as either B2 or X2, or may differ from both. If one of the operands is a positive constant less than 4096, the constant value will be used as the D2 field, in which case X2 will be zero. Otherwise, when both X2 and B2 differ from zero, D2 will be zero. When all fields have been set, the instruction LA R1,D2(X2, B2) is generated.

CEKMG -- Complex Plus Generator (CPLUS)

CPLUS is entered by the expression generator to generate the complex addition of two operands. See Chart EX.

ENTRIES: The entry point is CEKMG1. CPLUS expects, in parameter register P2, a pointer to the tree node containing the plus operator.

EXITS: Normal exit only.

OPERATION: CPLUS generates code to perform the addition in a manner appropriate to the types of the two operands. Possible operand type combinations are given in Table 27 (CMUL). If both operands, (a+bi) and (c+di), are complex, the following calculation is performed:

(a+bi) + (c+di) = (a+c) + (b+d)i

If one operand r is real, the addition is performed as follows:

r + (a+bi) = [(r + a) + bi] .

The code generated depends upon the operand types, signs, and locations (storage or registers).

CEKOF -- Complex Multiply Generator (CMUL)

CMUL is called by the expression generator to generate a complex multiplication. The combination of operand types processed is given in Table 27. See Chart EY.

Table 27. Operand Types Processed by CMUL

Left \ Right	L*1	L*4	I*2	I*4	R*4	R*8	C*8	C*16
L*1	N	N	N	N	N	N	N	N
L*4	N	N	N	N	N	N	N	N
I*2	N	N	N	N	N	N	I*2 R*4	I*2 R*8
I*4	N	N	N	N	N	N	I*4 R*8	I*4 R*8
R*4	N	N	N	N	N	N	Y	R*4 R*8
R*8	N	N	N	N	N	N	C*8 C*16	Y
C*8	N	N	I*2 R*4	I*4 R*4	Y	C*8 C*16	Y	C*8 C*16
C*16	N	N	I*2 R*8	I*4 R*8	R*4 R*8	Y	C*8 C*16	Y

N - Not processed
Y - Process as given
Other - Indicated conversion function has been supplied by the tree builder (TRBLD)

ENTRIES: The entry point is CEKOF1. CMUL expects, in parameter register P2, a pointer to the tree node which contains the operation to be processed.

EXITS: Normal exit only.

OPERATION: Given complex operands, (a+bi) and (c+di), CMUL performs the computation:

(a+bi) * (c+di) = (ac-bd) and (bc+ad)i.

Given real operand r and complex operand (a+bi), CMUL performs the computation:

r * (a+bi) = (ra+rbi) .

Processing differs according to whether neither, one, or both operands are in floating-point registers, and according to whether or not both operands are complex.

If both operands are in storage and are complex, all four floating-point registers are used in the computation of the product. The product's real and imaginary parts will reside in, and be assigned to, registers 0 and 2, respectively.

If one operand is in a register pair, all four registers will be used, and the result will be assigned to and left in the same register pair.

If each operand is in a register pair, all four registers will be used, and the result, if possible will be left in, and assigned to, the register pair which originally contained an operand already in storage.

CEKOG -- Complex Divide Generator (CDIV)

CDIV is entered by the expression generator to generate code to evaluate a complex quotient. The combination of operand types processed by CDIV is given in Table 28. See Chart EZ.

Table 28. Operand Types Processed by CDIV

Numer. / Denom.	L*1	L*4	I*2	I*4	R*4	R*8	C*8	C*16
L*1	N	N	N	N	N	N	N	N
L*4	N	N	N	N	N	N	N	N
I*2	N	N	N	N	N	N	I*2 R*4	I*2 R*8
I*4	N	N	N	N	N	N	I*4 R*4	I*4 R*8
R*4	N	N	N	N	N	N	Y	R*4 R*8
R*8	N	N	N	N	N	N	C*8 C*16	Y
C*8	N	N	I*2 C*8	I*4 C*8	R*4 C*8	R*8 C*16 C*8 C*16	Y	C*8 C*16
C*16	N	N	I*2 C*16	I*4 C*16	R*4 C*16	R*8 C*16	C*8 C*16	Y

ENTRIES: The entry point is CEKOG1. CDIV expects input in parameter register P2, consisting of a pointer to the tree node containing the divide operator.

EXITS: Normal exit only.

OPERATION: Given complex operands, (a+bi) and (c+di), CDIV generates code to perform the following computation:

$$\frac{a+bi}{c+di} = \frac{(ac+bd)}{c^2+d^2} + \frac{(bc-ad)}{c^2+d^2}\,i$$

Given real divisor r and complex numerator (a+bi), CDIV generates code to perform the evaluation:

$$\frac{a+bi}{r} = \frac{a}{r} + \frac{b}{r}\,i$$

Processing depends upon whether neither, both, or one operand is in floating-point registers. In all but the complex/real case, all four floating registers are used.

CEKMH -- Relational Expression Generator (RLTNL)

RLTNL generates code to evaluate the logical result of the relational operators .GT., .LT., and .EQ., or to generate the conditional branch associated with a logical IF statement. The types of the two operands of any given relational expression must be identical and may be Integer*2, Integer*4, Real*4 or Real*8. The type of the result (when the result is a logical value) is always Logical*4. See Chart FA.

ENTRIES: The entry point is CEKMH1. RLTNL expects the expression tree address of the relational operator in parameter register P2.

EXITS: Normal exit only.

OPERATION: Instructions are generated to compare the left and right operands. As a result of this comparsion a logical value (true or false) is generated in all but one situation: the expression is being used to determine the branch condition of a logical IF statement and the expression is not a common expression. In this case a conditional branch is generated to the label specified in the logical IF PF entry. Otherwise, the same branch type is generated, but in this case a local branch (e.g., BNE*+6) which completes the construction of a logical value by generation of the "false" condition. For example, consider the generation which might result from the following cases:

```
    (1)                    (2)
L=B.GT.C        IF (Y.GT.X) GO TO 20
 LA 3,1
 LE 0,B                 LE 0,Y
 CE 0,C                 CE 0,X
 BH *+6                 BH 20
 SR 3,3
 ST 3,L
```

In case 1 the comparison results in setting general register 3 to true or false. In case 2 the comparison results in conditionally branching to statement number 20.

The code generated further depends upon the types of the operands, the signs of the operands, and whether or not the operands are in registers. SELOP,SLONE, and SLPOS are used to complement constant operands, may be loaded with the load address instruction; determine which, if either, operand must be complemented with a load complement instruction; and, if the operand to be complemented is in a register, whether the operand may be complemented in that register or must be moved to another register (register 0 is always used for this purpose) before the comparison may take place. If the generation results in the computation of a logical result, the result is assigned to the selected register.

CEKMI -- Logical Expression Generator (ANDOR)

ANDOR generates code to evaluate the logical .AND. and .OR. operators or to generate the conditional branch associated with a logical IF statement. The logical operands must have the same type but may be either of type Logical*1 or Logical*4. See Chart FB.

ENTRIES: The entry point is CEKMI1. ANDOR expects the expression tree address of the logical operator in parameter register P2.

EXITS: Normal exit only.

OPERATION: ANDOR generates the logical .AND. or .OR. of two logical operands. Logical*4, and Logical*1 operations are somewhat different. If the operation type is Logical*4, at least one operand is forced to be in a general register by means of a load instruction, and (if necessary) the operand signs are made to match by complementing one of the operands. If the operation type is Logical*1, both operands are forced to be in general registers with the same signs, the loads in this case being performed with the combination subtract register-insert character. The logical operation generated and the result sign depend upon the expression tree operation code and the operand signs (after any complementation required to force operand sign agreement) as summarized in the following table:

Expression Tree Operation Code	Operand Signs	Operation Generated	Result Sign
AND	+	AND	+
AND	-	OR	-
OR	+	OR	+
OR	-	AND	-

After the logical operation has been generated, the result is assigned to the selected register, and if either the backlink to the next higher expression in the expression tree is not zero (indicating that the operation generated is part of a larger expression) or the logical IF flag is not raised (indicating that the expression is not part of a logical IF statement), generation is complete. Otherwise, a conditional branch to the label specified in the IF statement is generated. The condition code has been established by the AND or OR operation generated, and the branch instruction generated depends upon the sign of the result and the sign of the expression tree operator, as follows:

Expression Tree Sign	Result Sign	Branch Operation
+	+	BZ
+	-	BNZ
-	+	BNZ
-	-	BZ

CEKMU -- Maximum Operator Generator (MAX)

MAX generates code to evaluate the maximum of two quantities. The types of the two operands must agree, but may be Integer*2, Integer*4, Real*4, or Real*8. Note: Phase 1 has reduced maximum and minimum operations to maximum according to the transformation

```
MIN (A,B) = -MAX(-A,-B)
```

or

```
MIN (A,B,C) =-MAX(MAX(-A,-B),-C)
```

and has accounted for differences in the types of operands and results by introduction of conversion functions such as

```
AMAX0(I,J)=FLOAT(MAX(I,J))
```

or

```
MAX1 (A,B)=INT(MAX(A,B))
```

See Chart FC.

ENTRIES: The entry point is CEKMU1. MAX expects the expression tree address of the MAX operator in parameter register P2.

EXITS: Normal exit only.

OPERATION: In general the generation of the maximum operations consists of selection of a register to contain the result,

generation of an instruction to obtain one operand in the selected register, generation of a compare instruction to compare the two operands, generation of a conditional local branch (e.g., BH *+8), and generation of a conditionally executed load or load register instruction to obtain the other operand in the same selected register.

Generation varies in the obvious ways according to operand type (integer or real) and in addition depends upon the location and signs of the operands:

1. If neither operand is in a register, SELOP is entered. SELOP determines the result sign; which operand to load first; whether the load may be accomplished with the load address instruction; and, whether the operand to be loaded must be complemented before the comparison is generated.

2. If one operand is in a register, SLONE is entered. SLONE determines the result sign; whether the operation may take place in the register containing the operand; and, whether the operand must be complemented, either in place or while being moved to another register.

3. If both operands are in registers, SLPOS is entered. SLPOS determines the result sign; which operand register is to contain the result or that one of the operands must first be moved to still another register; and, whether the selected operand must be complemented, either in place or while being moved.

Generation of instructions required to position the operands for comparison is performed as indicated by SELOP, SLONE, or SLPOS. The conditional branch instruction generated depends upon the adjusted signs of the operands. If the signs are correct, BNL is generated; otherwise, BNH is.

CEKMK -- External Function Generator (FUNC)

FUNC is entered by the expression generator and by the CALL statement processor to generate a function or subroutine call. Calls upon subroutines with no arguments are not processed by the expression generator and are not treated by this routine. See Chart FD.

ENR

ENTRIES: The entry point is CEKMK1. FUNC expects a pointer to the tree node containing the ; operator to be passed in parameter register P2.

EXITS: Normal exit only.

OPERATION: The tree entry to be processed is a ; operator, whose right operand is the function name and whose left operand is either an expression entry (representing the one argument) or a , operator if there is more than one operand. Upon entry, all arguments are in storage with the desired sign.

The current length of storage class 5 is obtained from the Storage Class Table. The length indicates the next available byte in the parameter list area. A parameter list covering adcon is formed and is entered in the symbol table by TEVFL4. An instruction is generated to load the adcon into general register 1.

The leftmost (first) argument of the function operator is located by following left branches in the tree, until one such branch is not an argument separator operator (comma). Then, starting with the leftmost operand, and following backlinks until the function operator is encountered, each argument is given the following processing, according to its class code.

1. Constant operand

   The SLOC of the covering adcon and the displacement given in the operand item are summed, to form a new adcon covering the same storage class. The new adcon is entered in the parameter list via TEVFL5.

2. Variable operand

   If the variable is not a formal subprogram argument, it is processed in the same manner as a constant. Otherwise, instructions are generated to compute the effective address of argument and to store that address in the next available cell in the parameter list.

3. Operator item

   a. Subscripted variable

      If the operand sign flag is set, the comma processor has previously negated the variable and put it in temporary storage, recording the temporary assignment in the tree. In this case the operand processing is similar to that given a constant. If the flag is not set, instructions are generated to compute the effective address of the variable and to store that address into the next available cell in the parameter list.

b. Other expression

The temporary covering adcon value is added to the displacement given in the tree, and a new adcon is formed and added to the parameter list by TEVFL5.

4. Common expression item

Processing is similar to that given an operator, except that the information concerning temporary assignment is obtained from the name table, rather than from the tree.

5. Function item

a. Function name is a formal subprogram argument.

Code is generated to fetch the parameter location from the adcon page, where it was stored in a subprogram entry preamble, and to store it in the next available parameter list cell.

b. Function name is not a subprogram argument.

A V/R adcon pair is formed and entered in the symbol table by TEVVR. The storage assignment of the V-type adcon is combined with storage class 4, to form a new adcon which points to the V/R pair. The new adcon is entered in the next available cell in the parameter list.

6. Residue

Processing is the same as that given a common expression item.

After all the arguments have been processed, the remainder of the linkage is generated. This consists of the following code:

- Function being called is not a formal argument of the calling routine.

```
L     14,D+4(13)     Load R-adcon
ST    14,72(13)      Store in caller's
                     PSECT
L     15,D(13)       Load V-adcon
BASR  14,15          Branch
```

where D is the displacement with respect to the orgin of the caller's adcon page necessary to cover the first byte of the 8-byte adcon pair.

- Function being called is a formal argument.

```
L     15,C(13)       Load pointer to
                     adcon pair
L     14,4(15)       Load R-adcon
ST    14,72(13)      Store in caller's
                     PSECT
L     15,0(15)       Load V-adcon
BASR  14,15          Branch
```

where C is the displacement from the adcon page origin of the cell into which a pointer to the adcon pair is placed in the subprogram preamble.

CEKNJ -- Comma Operator Processing Subroutine (COMMA)

COMMA is called by the arithmetic expression generator to process the arguments of a comma (argument separator) operator or by the function operator processing subroutine to process the argument of a 1-argument function. Its purpose is to ensure that the operands of the comma operator or the operand of the function operator are in memory with the correct sign. See Chart FE.

ENTRIES: COMMA has two entry points: COMMA (CEKNJ1), entered by the Arithmetic Expression Generator, and COMA2 (CEKNJ2), entered by the Function Operator Processing subroutine. COMMA and COMA2 expect in parameter register P2 the expression tree address of the comma operator and the function operator, respectively.

EXITS: Normal exit only.

OPERATION: The left operand of a function operator and the right operand of a comma operator are always processed. The left operand of a comma operator is processed only if it is not itself a comma operator. The operands processed are treated according to their class:

1. A constant requires no processing if its tree sign is plus. Otherwise, the constant is complemented, the resulting constant is filed in the symbol table, and the constant item in the tree is changed to reflect the new associated symbol table entry.

2. A variable requires no processing if its tree sign flag is plus. Otherwise, a temporary storage location is assigned; and, FETCH is entered

   a. to load, complement, and store the variable in temporary if the variable was not in a register;
   b. to complement and store the variable in temporary if it was in a register with the wrong sign; or,
   c. simply to store the variable in temporary if it was in a register with the desired sign.

The identity of the temporary-covering adcon and displacement are stored in the associated fields in the variable item.

3. A colon operator or colon common expression requires processing if its tree sign flag is not plus. It is treated in the same fashion as a variable. However, the identity of the temporary-covering adcon and displacement are recorded in the name table if the operand is a common expression.

4. Processing of a noncolon operator depends upon whether or not the operand is in storage. If it is in storage and the storage and tree signs agree, no action is taken. If it is in storage and the storage and tree signs differ, FETCH is entered to load the operand from its temporary, to complement it, and to store the complemented value in the same temporary location. If the operand is in a register, a temporary is assigned, and FETCH is entered to complement and store the operand in the temporary if the register and tree signs disagree, or simply to store it in temporary if they agree.

5. If a noncolon common expression is in storage and its storage sign agrees with the tree sign, no further processing is required. If it is in storage and the signs disagree but the Name Table entry indicates that a secondary temporary has been assigned, the tree entry's "use secondary temporary" flag is set.

   If the common expression is in storage, its Name Table storage sign disagrees with its tree sign, and a secondary temporary has not been assigned, then a new temporary is assigned, its assignment is recorded in the secondary temporary assignment field of the name table entry, the name table "secondary temporary assigned" flag is set, and the tree entry's "use secondary temporary" flag is set. FETCH is entered to load, complement, and store the operand if it is not also in a register or to complement and store if it is already in a register with the wrong sign.

   If the common expression is not in storage, a temporary is assigned and the assignment is recorded in the name table. FETCH is entered to complement and store the expression in temporary if its register sign disagrees with its tree sign or simply to store it if the signs agree.

6. A function item requires no processing.

CEKOM -- Open Function Control Routine (DCOM)

This program is called by the arithmetic expression generator (AGEN) when a ;; operator (open function connector) is encountered. The purpose of this routine is to invoke the open function processing module appropriate to the function number which is given in the right-operand of the ;; operator. See Chart FF.

ENTRIES: DCOM has two entry points. The main entry (CEKOM1) is made with the major operator address in parameter register P2. The alternate entry in DCOM (CEKOM2) is physically packaged with CEKOM; but is actually one of the six open function modules called by DCOM, and its description is to be found under "CEKOM2 -- Open Function Processing Routine (OPEN6)."

EXITS: This routine produces no output. If the function number given as input is not among those expected, DCOM makes a machine/compiler error exit via PH4MER.

OPERATION: The function number is obtained from the right-operand of the ;; operator in the expression tree. The appropriate open function module processing routine entry address is obtained by a table look-up operation, indexed by function number. The routine is then invoked. When return is made to DCOM, DCOM immediately returns to AGEN.

CEKOT -- Open Function Processing Routine (OPEN1)

OPEN1 is invoked by DCOM to process any of the open functions, DSIGN, HDIM, IDIM, DIM, SIGN, or DDIM. See Chart FG.

ENTRIES: Entry is to CEKOT1, with the major operator tree address in parameter register P2 and the open function number in parameter register P3.

EXITS: If the function number given as input is not among those expected, OPEN1 makes a machine/compiler error return via PH4MER.

OPERATION: This routine performs the code generation to effect the following functions:

```
SIGN(A,B)=(sign of B) *|A|
DIM(A,B) = Max(A-B,0)
```

CEKOU -- Open function Processing Routine (OPEN2)

OPEN2 is invoked by DCOM to process any of the following open functions:

CMPLX MOD
DCMPLX HMOD

See Chart FH.

ENTRIES: Entry it to CEKOU1, with the major operator tree address in parameter register P2 and the open function number in parameter register P3.

EXITS: If the function number given as input is not among those expected, OPEN2 makes a machine/compiler error exit via PH4MER.

OPERATION: This routine performs the code generation to effect the following functions:

CMPLX (A,B) is the complex quantity whose real and imaginary parts are A and B, respectively.

$$MOD\ (I,J) = I - \frac{I}{j} * J$$

where [] denotes the integral part.

CEKOX -- Open Function Processing Routine (OPEN3)

OPEN3 is invoked by DCOM to process any of the following open functions:
IABS, DABS, ABS
and the following type-conversion functions:

Operand Type	Result Type
I*2	R*4
I*2	R*8
I*2	C*8
I*2	C*16
I*4	C*16
R*4	R*8
R*4	C*8
R*8	C*8
R*4	C*16
C*8	C*16

See Chart FI.

ENTRIES: Entry is to CEKOX1, with the major operator tree address in parameter register P2 and the open function number in parameter register P3.

EXITS: If the function number given as input is not among those expected, OPEN3 makes a machine/compiler error exit via PH4MER.

OPERATION: This routine performs the code generation required to effect the above functions.

CEKOY -- Open Function Processing Routine (OPEN4)

OPEN4 is invoked by DCOM to process any of the following open functions:

AMOD HSIGN ISIGN
DCONJ AINT CONJ
DMOD

See Chart FJ.

ENTRIES: Entry is to CEKOY1, with the major operator tree address in parameter register P2 and the open function number in parameter register P3.

EXITS: If the function number given as input is not among those expected, OPEN4 makes a machine/compiler error exit via PH4MER.

OPERATION: OPEN4 performs the code generation to effect the following functions:

$$MOD\ (A,B) = A - \left[\frac{A}{B}\right] * B$$

AINT (A) = integer part of A
CONJ (A+Bi) = (A-Bi)
SIGN (A,B) = sign of B *|A|

CEKOZ -- Open Function Processing Routine (OPEN5)

OPEN5 is invoked by DCOM to process any of the following functions: Integer and real square, cube, fifth, and seventh powers; and, real reciprocal. See Chart FK.

ENTRIES: Entry is to CEKOZ1, with the tree address of the major operator in parameter register P2 and the open functions number in parameter register P3.

EXITS: If the open function number given as input is not among those expected, OPEN5 makes a machine/compiler error exit via PH4MER.

OPERATION: OPEN5 performs the code generation to effect the above functions.

CEKOM2 -- Open Function Processing Routine (OPEN6)*

OPEN6 is invoked by DCOM to process any of the following open conversion functions:

Operand Type	Result Type
R*4	I*2
R*4	I*4
R*8	R*4

R*8	I*4
C*8	R*4
L*1	L*4
L*4	L*1
I*2	I*4
I*4	I*2
R*8	I*2
C*8	I*2
C*8	I*4
C*16	I*2
C*16	I*4
C*16	R*4
C*16	R*8
C*16	C*8

See Chart FL.

ENTRIES: Entry is to CEKOM2, with the major operator tree address in parameter register P2 and the open function number in parameter register P3.

EXITS: If the function number given as input is not among those expected, OPEN6 makes a machine/compiler error exit via PH4MER.

OPERATION: OPEN6 performs the code generation necessary to effect the above conversion functions.

CEKMV -- Memory Access Routine (MEMAC)

MEMAC is entered to obtain cover for a generated storage reference to an arbitrary expression tree operand and to provide X2, B2, and D2 instruction fields for the reference. See Chart FM.

ENTRIES: The entry point is CEKMV1. MEMAC expects the expression tree address of the operand to be covered to be in parameter register P2. If parameter register P1 is nonzero, it is assumed to contain the number of a general register that must remain undisturbed by MEMAC. If no such protection is desired, the contents of register P1 must be 0.

EXITS: Normal exit only. Parameter registers P1, P2, and P3 contain X2, B2, and D2, respectively.

OPERATION: MEMAC treats two distinct cases: the operand requiring cover either is or is not a subscript connector. If the operand is not a subscript connector, X2 is set to 0, and D2 is obtained directly from the displacement field of tree entry or the name table entry appropriate to the class of the operand. Likewise, the symbol table pointer to the covering adcon is obtained; and, if that adcon is in a general register, B2 is set equal to the register number. However, if the adcon is not in a register, a register is selected, an instruction is generated to load the adcon into the selected register, and B2 is set equal to the number of the selected register.

If the operand is a subscript connector, D2 is set directly to the value given in the displacement field of the left operand of the subscript connector, and one of two cases exists: the right operand of the subscript connector either is or is not a ? operator. If it is not, X2 is set to 0, the right operand of the subscript connector is obtained in a general register (by loading it if necessary), and B2 is set equal to the corresponding register number.

If the right operand of the subscript connector is a ? operator, each of the two operands of the ? operator is treated (first right and then left) as if it were the single non-? operator operand of the subscript connector. The right operand determines B2, and the left operand (taking care that loading of the left operand does not disturb register B2) determines X2.

CEKOP -- Load Covering Adcon Routine (COVER)

COVER is called to obtain adcon cover for generation of storage reference that does not require an index field.

ENTRIES: Entry is to CEKOP1, with the symbol table pointer to the desired adcon in parameter register P2. Parameter register number P1 must have either the number of a general register whose contents must be left undisturbed by the potential adcon load or zero.

EXITS: Normal exit only. The number of the register which contains the adcon is returned in parameter register P2.

OPERATION: COVER is given the symbol table pointer to an adcon. It determines whether the adcon is in a general register. If so, the register number is returned. If not, a register is selected, the adcon is loded into that register, and the selected register number is returned.

CEKMZ -- Local Branch Generator (SADDR)

SADDR is called by the relational expression generator and by other routines requiring generation of a forward branch relative to the current value of the location counter. See Chart FN.

---

*CEKOM2 is physically an alternate in mode CEKOM.

ENTRIES: Entry is to CEKMZ1, with the desired displacement in parameter register P3. Up to two registers may be specified as unavailable for use as branch cover. These are input in parameter registers P1 or P1 and P2.

EXITS: Normal exit only. SADDR returns the effective displacement and a register number in parameter registers P1 and P2 respectively.

OPERATION: SADDR expects as input a parameter whose value is between 0 and 4095, inclusive. The value represents the desired destination in bytes, relative to the location counter. It is assumed that if SADDR must generate a load to cover the destination address, the parameter value will apply to the location counter setting in effect after that load has been generated.

The sum (S) of the current location counter value and the parameter value is computed. The register table is searched to find a register containing an adcon or a code cover quantity whose value (V) is such that:

$$V \leq S < V+4096.$$

If such a register is found, the register number and displacement are returned to the caller. The displacement, in this case, is equal to S-V. If no such register exists, SELSR is called to select a register (R) into which cover may be loaded. INSOT is called to generate the instruction BASR R,0. ASAR is entered to make a register table entry for the code cover quantity now in R. R, together with the same displacement given as input, is output to the caller, and exit is made.

CEKNV -- Labeled Branch Generator (LBL)

LBL is called to output code to branch to a statement label and provide any necessary cover prior to branching. See Chart FO.

ENTRIES: Entry is made to CEKNV1, with one of the types of branch operation codes in parameter register P1 and a symbol table label entry pointer in parameter register P2.

EXITS: Normal exit only. The output of LBL is via INSOT, a generated branch instruction in the code file.

OPERATION: LBL is provided with two input parameters: the type of branch required to be generated, and a pointer to the symbol table entry for the label to which the branch is to be made. LBL then searches the registers to determine if an appropriate adcon is present to cover the branch about to be generated. If not, a register is selected and assigned, and a load of the adcon generated. Next, a special ID item is created for Phase 5, to flag this as a branch reference whose address is to be filled in. Finally, the incomplete branch instruction is generated.

CEKOS -- Operand Fetch Complement/Store Routine (FETCH)

FETCH is called by the comma operator processor to ensure that each argument of a function or subroutine is in storage with the desired sign. See Chart FP.

ENTRIES: The entry point is CEKOS1. Input parameters are as follows:

Parameter Register	Contents
P1	Request Key 2^0 = Fetch 2^1 = Complement 2^2 = Store
P2	Temp-covering adcon pointer if store requested
P3	Displacement relative to temp-cover if store requested.
P4	Register number if operand is in a register
P5	Operand tree address

EXITS: Normal exit only.

OPERATION: FETCH generates instructions as indicated by input options to load an operand into a general or floating register, to complement it in that register, and to store it in a specified temporary cell. The usage of floating versus general registers and related instructions is dictated by the operand type given in the expression tree.

CEKND -- Select Operand Routine (SELOP)

SELOP is a general purpose operand optimizing routine which is used by all the routines RPLUS, CPLUS, IPLUS, RMUL, CMUL, IMPLY, RDIV, CDIV, IDVDE, RLTNL, ANDOR, and MAX when both operands are in storage. See Chart FQ.

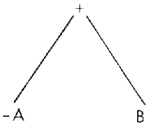
ENTRIES: The entry point is CEKND1. Registers P1 and P2 contain the output parameters from subroutine KEY (i.e., the signs of both operands), and parameter register P3 contains the expression tree address of the operator whose operands are being considered.

Register	Operation	Value	Significance
P5	maximum, relational, or logical	0	Do not complement after loading operand.
P5	maximum, relational, or logical	1	Complement after loading operand.
P5	multiplication multiplication multiplication	0 1 -4095	Standard multiplication. Multiply with shift. Multiplication by 1.
P6	maximum, relational, or logical	irrelevant	
P6	multiplication	-	Amount of shift if shift is indicated.

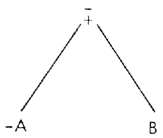
EXITS: Normal exit only. Parameter registers P1 and P2 contain the result sign (0 for plus and 1 for minus) and the expression tree address of the operand to be loaded, respectively. If bit 0 of parameter register P3 is 0, the operand is not to be loaded with a load address instruction; otherwise, bits 20 through 31 contain the immediate value to be loaded by means of a load address instruction. If the operation is plus, the value in parameter register P4 (0 or 1) indicates that an addition or a subtraction, respectively, is to be performed; otherwise, the register contains the expression tree address of the second operand. Parameter registers P5 and P6 have meaning only for the maximum, relational, logical, and multiplication operators as summarized in the above table:

OPERATION: The output parameters from SELOP specify a procedure for generating code in an optimal fashion for the given operation and combination of operand signs. In particular, SELOP attempts, in one of two ways, to allow for generation of a result whose sign matches that of the expression tree node being processed. The first way takes similar forms for the plus and logical operators and simply involves a choice of which operand is to be loaded. For trees

1.

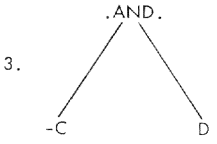


2.

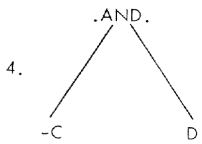


the right and left operands, respectively, would be chosen to be loaded. Similarly, for trees

3.



4.



the left and right operands, respectively, would be chosen. The code generated for the cases might be

```
1.  LE   0,B          2.  LE   0,A
    SE   0,A              SE   0,B

3.  L    5,C          4.  L    5,D
    BCTR 5,0              BCTR 5,0
    LCR  5,5              LCR  5,5
    N    5,D              O    5,C
```

For the multiply and divide operators, the operand loaded is irrelevant in determining the result sign. For these operators, if a direct product would not produce the desired sign, the operands are examined. If one is a constant, the constant is complemented and filed in the symbol table, and the tree entry for the constant item is modified to reflect the change.

For any of the arithmetic operators if the operand selected for loading is a positive integer constant less then 4096 (indeed, if there is no other basis for choosing which operand to load, such a constant operand is chosen), a parameter is returned in the form of a flag to indicate that the selected operand may be loaded with a load address instruction.

In only one situation is the production of the desired result sign ignored. If the

operation is an integer multiplication, and one of the operands is an integer power of two, the other operand is selected for loading and multiplication by shifting (or by loading only, if the operand is 1) is indicated, regardless of the result sign.

CEKNF -- Select Position for Operation (SLPOS)

SLPOS is a general purpose operand optimizing routine used by all the routines RPLUS, CPLUS, IPLUS, RMUL, CMUL, IMPLY, RLTNL, ANDOR, and MAX when both operands are in registers. See Chart FR.

ENTRIES: The entry point is CEKNF1. Parameter registers P1 and P2 contain the output from the KEY subroutine (numbers of registers containing the two operands and the corresponding register signs), and parameter register P3 contains the expression tree address of the operator whose operands are being considered.

EXITS: Normal exit ionly. Parameter register P1 contains the selected operand register in the following form:

$0 \le P1 \le 15$	P1 = selected register; use as is.
$-15 \le P1 \le -1$	|P1| = selected register; move operand before using.
$16 \le P1 \le 31$	P1-16 = selected register; complement register before using.
4095	Floating-point register 0 is selected; complement before using.
$-31 \le P1 \le -16$	|P1|-16 = selected register; move and complement before using.

Parameter register P2 contains the result sign: 0 for plus, 1 for minus. Parameter register P3, applicable only for plus operator, contains 0 to indicate addition or 1 to indicate subtraction. For a multiplication operator, the content of P3 has the following meaning:

```
    0 -- perform standard multiplication.
    1 -- multiply by shifting.
-4095 -- multiply by 1
         (no multiplication).
```

Parameter register P4 is of significance only if P3 contains a 1, in which case the contents of register P4 indicate the number of places of shift to be generated.

OPERATION: The two registers containing the operands are given weights according to their contents as determined from the ID field (MRM2) of the initial entries of the corresponding MRM or MRMFR table entries. Following is a tabulation of the weights assigned:

Weight	Contents
9	Operator
8	Common Expression (last use)
6	Variable
5	Constant Less than 4096
4	Constant (Greater than 4095)
3	Address Constant
2	Stored Common Expression
1	Unstored Common Expression

If the operand signs are identical, or if the two weights are not identical and either operand is an unstored common expression which is not now being used for the last time, no refinement of the preliminary weights is made. Otherwise, an increase of 2 is made to the weight of the operand whose register sign matches the tree sign of the operator being generated. This tends to increase the probability that the result sign will be the desired sign.

For all floating-point and logical operators, the weights are compared, the operand with the highest weight is chosen as the "to" operand, and the corresponding determination of result sign is made. Additionally, if the floating-point operation is plus, the agreement or disagreement of the operand signs determines the operation, add or subtract, respectively.

The integer operators require further tests. If the register occupied by either of the operands is globally assigned, the corresponding weight is complemented. Then if the maximum weight is negative, SLPOS specifies that one of the operands must be moved to another register which will become the "to" register. If moving one of the operands with a load complement instruction will produce the desired sign, SLPOS specifies that this be done.

CEKOW -- Select One Operand in a Register (SLONE)

SLONE is a general purpose operand optimizing routine used by RMUL, CMUL, IMPLY, RDIV, CDIV, IDVDE, RLTNL, ANDOR, and MAX when only one operand is in a register. See Chart FS.

ENTRIES: The entry point is CEKOW1. Parameter registers P1 and P2 contain the output from the KEY subroutine (the register number and register sign of the operand that is in a register and the storage sign of the other operand) and parameter regis-

ter P4 contains the expression tree address of the operator being processed.

EXITS: Normal exit only. Parameter register P1 contains the result sign; P2 contains the expression tree address of the operand in storage; and, P3 contains 0 if the storage operand need not be loaded, 1 if it must be loaded from storage or X'80000nnn', indicating that it should be loaded with a load address instruction whose D2 field has the value nnn. The contents of parameter registers P4, P5, and P6 vary with the operator being processed:

Register	Operation	Value
P4	Maximum, logical, or relational	Expression Tree address of register operand
P4	Plus	0 -- Add 1 -- Subtract
P4	Multiply	0 -- Multiply 1 -- Shift -1 -- Multiply by 1
P5	Maximum, logical or relational	-1 -- Move with Load Complement FFFF0000 -- Move with Load Register
P5	Add, multiply, or divide	0 -- Do not move operand 1 -- Move with Load Register -1 -- Move with Load Complement
P6	Maximum, logical, or relational	Desired result sign
P6	Plus	0 -- Add or subtract in register 1 -- Load storage operand into a second register and add or subtract there
P6	Multiply	Length of shift if P4 = 1; otherwise, irrelevant if P4 = 1; otherwise, irrelevant

OPERATION: For a multiplication operation SLONE determines whether the operation may be performed by a shift instruction. For all operations SLONE determines whether the operation may be performed in the register (or register pair) containing one operand or if the operand must be moved to another register (or register pair). SLONE also determines the result sign and attempts to arrive at the desired sign either by filing the complement of a constant storage operand in the symbol table or by determining, when the register operand must be moved to another register, whether it should be moved with a load complement instruction.

If the storage operand must be loaded into a register, SLONE determines whether the load may be performed with a load address instruction.

CEKNB -- Determine Availability of Register for Multiplication (SELGM)

SELGM is entered by SLONE to determine if an integer multiplication may be performed in a register pair which contains a given register, and by SLPOS to determine whether a multiplication may be performed in either of two register pairs. See Chart FT.

ENTRIES: The entry point is CEKNB1. Either parameter registers P1 and P2 contain the two given registers, or P1 contains the only given register and P2 contains -1.

EXITS: Normal exit only. For each of the inputs in parameter registers P1 and P2, the output contained in the same register is either the input value if the corresponding register member may be used as the R1 field of an integer multiply instruction, or is negative if the corresponding register number may not be so used.

OPERATION: If either register contains a quantity of type Integer*2, the register is disqualified only if it is globally assigned. If two registers are specified, neither is globally assigned, and they are an even/odd pair (e.g., 2/3 but not 7/8), neither is disqualified.

If the register number is even, it is disqualified. If the register number is odd, and the even numbered member of the corresponding even/odd register pair neither is globally assigned nor contains an unstored common expression, the odd numbered register is qualified. In the remaining case if the register number is odd and the even numbered member of the corresponding even/odd register pair is not globally assigned, but is an unstored common expression, the qualification of the odd numbered register is postponed.

After the registers presented as input to SELGM have been processed as above, each register will have been qualified or disqualified or will have had its qualification postponed.

If no postponement of the qualification of a register has occurred, exit is made from SELGM. If qualification of only one of the two registers has been postponed and the other register has been qualified, the former register is disqualified and exit is made.

In the remaining case, qualification of both registers has been postponed, or there has been one postponement and one disqualification. If there is at least one remaining general register which is not globally assigned and does not contain an unstored common expression, both input registers are disqualified, and exit is made from SELGM. Otherwise, SELSR is entered to select one of the registers which has not been disqualified (thereby storing its contents); if the other register has had its qualification postponed, it is now disqualified, and exit is made from SELGM.

If only one register is to be considered by SELGM, its treatment may be found from the above discussion by assuming that it is one of two registers to be considered, the other of which has already been disqualified.

CEKNA -- General Register Availability for Integer Divide (SELGD)

SELGD is called by IDVDE when the numerator of a quotient is in a general register to determine whether the division may take place in the register pair containing the numerator. See Chart FU.

ENTRIES: The entry is CEKNA1. Parameter register P1 contains the register number of the numerator. P2 contains the register number of the denominator if the denominator, too, is in a register; otherwise, it contains -1.

EXITS: Normal exit only. Parameter register P1 contains:

1. The register number of the numerator if division may take place in that register, or
2. The complement of that number if division may not proceed there, or
3. Four times that number if the divisor may proceed and numerator and denominator are in the same register.

OPERATION: If the numerator and denominator are in the same register, exit is made from SELGD. If they are not in the same register pair, the numerator is in an even register that is not globally assigned, and the next higher numbered register is not globally assigned and does not contain an unstored common expression, then SELGD indicates that division may proceed in the register pair containing the numerator. Otherwise, SELGD indicates that the numerator must be moved to another register pair before the division may take place. See Chart NF.

CEKOC -- Operand Status Routine (KEY)

KEY is called by the real and complex +, *, and / generators to determine the location (storage or register) of both operands of the current operation. See Chart FV.

ENTRIES: The entry point is CEKOC1. Parameter register P2 contains a pointer to the operator tree node whose operand locations are to be determined.

EXITS: Normal exit only. KEY returns in parameter registers the indicated switching code, the register number of each operand in a register, and the associated register sign indicator. The left operand register number is right-justified in the left half of parameter register P1. The right operand number is right-justified in the left half of parameter register P2. The register signs of the left and right operands are right-justified in the right halves of registers P1 and P2, respectively.

OPERATION: KEY first processes the left operand. If it is a variable or constant, FNDAR or FNDFR is called to determine whether the operand is in a register. If the operand is a partial result, the determination is made by examining its tree entry. If the operand is a common expression or residue, the determination is made by examining the name table. With this information, KEY constructs a 2-bit switching code with the first bit indicating the absence or presence of the left operand in a register and the second indicating similar information for the right operand. For each operand in a register, KEY determines the register and sign from the corresponding MRM table.

CEKOR -- Single Operand Locating Routine (KEY1)

KEY1 is entered to establish whether an operand is in a register or in storage, and to determine the operand's register/storage sign. See Chart FW.

Quantity	P1	P2	P3	P4
Operator	1	Expression Tree Pointer	N/A	N/A
Common Expression	2	Name Table Pointer	N/A	N/A
Adcon	3	Symbol Table Pointer	N/A	N/A
Variable	5	Symbol Table Pointer to Covering Adcon	Associated Displacement	Type Code
Constant	6	Symbol Table Pointer	N/A	N/A
Residue	8	Name Table Pointer	N/A	N/A

ENTRIES: Entry is to CEKOR1, with the operand tree address in parameter register P2.

EXITS: Upon exit parameter register P2 contains the operand's register number if any; parameter register P3 contains the register/memory sign; and parameter register P1 contains a 1 if the operand is in a register, a zero otherwise.

OPERATION: KEY1 locates the operand by means of the FNDAR of FNDFR routines if the operand is a constant, variable, or adcon. If the operand is an operator, the information is obtained from the expression tree. If the operand is a common expression or residue, the information is obtained from the name table. If an operand is found to be in a register, the register number and register sign are returned, whether or not the operand may also be in storage. If an operand is not found to be in a register, it is assumed to be in storage; and, the storage sign is returned. A variable, constant, or adcon in storage is assumed to have storage sign plus.

CEKMR -- Search General Registers (FNDAR)

FNDAR is used either to determine whether a given quantity is in one of the general registers or to determine whether there is at least one empty general register. See Chart FX.

ENTRIES: The entry point is CEKMR1. If an empty register is to be found, parameter register P1 must contain 0; otherwise, input is required in the parameter register as above:

EXITS: Normal exit only. If parameter register P1 is positive, its content is the register number containing the desired quantity or the number of an empty register. If P1 contains the value -40, the desired quantity is not in a register or there is no empty register. Otherwise, if the content of P1 is negative, the absolute value of the content is the register containing the desired quantity, and the expression sign of the quantity is negative.

CEKMS -- Search Floating Registers (FNDFR)

FNDFR is used either to determine whether a given quantity is in one of the floating registers or to determine whether there is at least one empty floating register. See Chart FY.

ENTRIES: The entry point is CEKMS1. If an empty register is to be found, parameter register P1 must contain 0; otherwise, input is required in the parameter registers as shown in the diagram.

EXITS: Only normal exit is made. If the content of P1 is not negative and less than 8, it represents the register number containing the desired quantity or the number of an empty register. If P1 contains the value -40, the desired quantity is not in a register or there is no empty register. Otherwise, if P1 contains a negative number, the absolute value of that number is the number of the register containing the desired quantity, and the expression sign is negative. In the System/360 computers 0 has no distinct complement; therefore, if the register containing the negative expression sign is register 0, parameter register P1 will contain plus 8.

CEKMW -- Operand Processing Routine (OPND)

OPND is called by the various arithmetic generators, or by the subroutine RSLT, at the statement processing level. It is called immediately after each reference to an expression tree entry, as the operand of a larger expression or as an operand from the statement level. Its purpose is to clear register storage and to release tem-

porary storage assigned to a partial result or to a common expression which will not again be referenced. If the given operand is a subscript connector (:), processing is somewhat different. A reference to the subscripted variable is treated as a reference to the subscript expression which is said to be the operand. Further, for this purpose there are actually two kinds of subscript expressions -- depending upon whether the major operator of the subscript is a split (?) operator. The split operator represents an addition (performed by use of hardware base-index addition) of two operands. In this case, OPND must treat both operands of the ? operator individually, rather than the operand of the : operator by itself. See Chart FZ.

ENTRIES: The entry points are CEKMW1, CEKMW2, and CEKMW3. The input is a pointer to the first byte of the operand representation in the expression tree in parameter register P2.

EXITS: Normal exit only.

OPERATION: Processing varies with the class of operand presented -- variable or constant, operator, common expression, and residue.

Variable and constants require no processing, and exit is made immediately.

Operators (partials) which are not subscript connectors are processed as follows. If the tree "in storage" bit is set, the subroutine RLSWS is entered to reassign the associated temporary storage. If the "in register" bit is set, the specified register table entry is cleared. After this has been accomplished, exit is made.

If the operator is a subscript connector, its right operand is examined. If the right operand of the : is not a ?, and if it is an operator, it is processed as above, and exit is made; if it is a common expression, it is treated as described in the following paragraph; then exit is made. If the right operand of the : is a ?, each of its operands is processed as described immediately above before exit is made.

If the given operand is a common expression or residue, the "last use" bit of the corresponding name table entry is tested. If it is not set, no further action is taken, since the last use of this expression does not occur in the present statement. If, however, the bit is set, the count field of the name table entry is reduced by 1. If it has not reached zero, no further action is taken. Otherwise, register storage and temporary storage are released by use of the name table "in storage" and "in register" bits, in the same manner as they were used in the operator processing. When this is complete, the name table entry is cleared, and exit is made.

CEKMY -- Result-Register Operand Processing Subroutine (RSLT)

RSLT is entered by the various arithmetic generators as a substitute for entering OPND. RSLT is called when the operand participating in an operation is in the register which is destined to contain the result of the operation; that is, the operand is in the "to" register. The purpose of this routine is to ensure that if the "to" register contains an active common expression, that expression will either be moved to another register or stored, provided it is not already in storage. Since RSLT calls upon COVER, which may be required to load a temporary-covering adcon, the status of the noselectable bits of the result register and any other register specified as input parameters will be saved on entry to RSLT, then set to nonselectable, and restored immediately before exit. This is done to ensure that the register selected for temporary-cover, if any, will not be one containing either of the operands of the current operation. See Chart GA.

ENTRIES: The entry point is CEKMY1. RSLT expects two parameters: The first is a tree pointer to the operand to be protected from the operation about to be performed, in parameter register P2. The second is the register number of another register to be protected against loading with temp cover, in parameter register P1. If R is the given register number, R is interpreted as follows:

> If $1 \leq R \leq 15$, then R is the corresponding general register.
>
> If R = 0 (applicable only to the second parameter), then no protection is required.

EXITS: Normal exit only.

OPERATION: On entry, the status of the operand nonselectable indicator is manipulated as described above. Now OPND is called. Upon return, if the operand is not a common expression, it is a nonactive common expression, or if it is an active common expression already in storage, the nonselectable indicators are restored, and exit is made. Otherwise, if the operand is not of complex or real type, FNDAR is entered to determine if there is an empty general register. If there is, the common expression is moved to that register by an RR-load instruction, and ASAR is entered to assign the expression to the register. If, however, the type is real or complex or

there is no empty register, FNDWS is entered to assign temporary storage; the assignment is entered in the name table; COVER is entered to cover the temp; and, the operand is stored. Now, in either case, the status of the nonselectable bits is restored, and exit is made.

CEKNG -- Select Single General Register (SELSR)

SELSR is entered to select a specified general register, to select any register in an optimal manner, or to select in an optimal manner any register from a specified restricted set of registers. See Chart GB.

ENTRIES: The entry point is CEKNG1. One or two parameters may be entered in parameter registers P1 and P2. However, if only one parameter is entered, it must be in P1, and P2 must contain 0. Specification of the register to be selected must be designated only in register P1. If no specification of or restriction on the register to be selected is desired, register P1 must contain 0, in which case the contents of register P2 are irrelevant. Otherwise, specifications and restrictions are made in either of or both registers P1 and P2 as follows:

Value (V)	Meaning
1≤V≤15	V = Reg. No. Do not select this register.
16	Do not select register 1, 14, or 15, and in the course of selection, do not move any quantity into register 1, 14, or 15.
-15≤V≤-1	V = -(Reg. No.) Select register number |V|. In the course of selection, do not move any quantity into register 1, 14, or 15.
-30≤V≤-16	V = -15- (Reg. No.) Select register number |V|-15.

EXITS: Normal exit only. Parameter register P1 contains the number of the register selected.

OPERATION: SELSR associates with each register a weight which is determined from the corresponding MRM table entry as follows:

Weight Assigned	Register Contents
9	Register empty
8	Constant smaller than 4096
7	Constatn greater than 4095
6	Variable
5	Stored common expression
4	BASR-generated address
3	Adcon
2	Unstored common expression
1	Operator (partial result)
0	Register not selectable (either because the MSL or MGBL flag of the MRM Table entry is raised, or because an input parameter to SELSR had disqualified the register).

If there has been no request for selection of a specific register, the register with the greatest weight is chosen. If two or more registers have the same weight and that weight is greater than 6, the register with the smallest register number is chosen; however, if the smallest register number among the two or more equally weight registers is 1, then the next higher numbered register with the same weight is taken. If a request for a specific register has been made, that register is chosen, regardless of its relative weight.

If the chosen register has a weight greater than 2, the routine FLUSH is called to reset the register storage for all quantities in the chosen register and to initialize the corresponding MRM Table entry. The initialization causes the entire MRM table entry to be cleared except for the MSL and BUSY flags which are left raised.

If the chosen register has a weight equal to either 1 or 2, the entire set of register weights is examined to determine if there is an empty register (weight = 9). If there is an empty register, FLUSH is called to transfer the register contents to storage, to transfer the corresponding MRM table entry contents from the chosen register to the empty register, and to initialize the chosen register. An LR instruction is generated to move the contents of the chosen register. If there is not an empty register, FNDWS is called to assign a temporary to the quantity in the chosen register. The MRM Table is searched to find an adcon which covers the temporary assigned. If there is such an adcon, an instruction is generated to store the chosen register. If there is no temporary-covering adcon in a register, the register weights are examined, the largest weight which exceeds 2 is chosen, and the temporary-covering adcon is loaded into

that register and assigned to the corresponding MRM Table entry. The chosen register is then stored and its MRM Table entry initialized. If, however, no register contains such an adcon and no register weight exceeds 2, an escape mechanism is used.

A Load Register instruction is generated to move the contents of the chosen register to register 0. The temporary-covering adcon is loaded into the chosen register and used from there to cover the store of register 0 into temporary. Finally, FLUSH is called to initialize the chosen register.

CEKNH -- Select Even/Odd General Register Pair (SELDR)

SELDR is entered to select optimally an even/odd pair of general registers. See Chart GC.

ENTRIES: The entry point is CEKNH1. Parameter registers P1 and P2 may be used to specify as many as two registers that must be excluded from consideration in the selection process. Each of the parameter registers must either contain 0, indicating no exclusion, or a number greater than 0 and less than 16, indicating that the corresponding general register should be excluded.

EXITS: Normal exit only. Parameter register P1 contains the number of the even member of the selected even/odd register pair.

OPERATION: A weight is assigned to each register in accordance with its contents. The weights are then combined in pairs to give a combined weight for each even/odd register pair. If a member of any pair has been excluded from consideration, the pair is given weight 0. The register pair with the largest combined weight is selected. SELSR is invoked to select specifically each member of the pair, thereby making the pair available for use.

CEKMQ -- Select Floating Register (SELFR)

SELFR is used to select a floating-point register or to select a pair of floating-point registers (either registers 0 and 2 or registers 4 and 6). See Chart GD.

ENTRIES: The entry point is CEKMQ1. Parameter register P1 contains a number which indicates the kind of selection desired:

Value in P1 (V)	Meaning
$0 \le V \le 6$	Select register number V.
$10 \le V \le 14$	Select register pair (V-10)/(V-8).
V = 16	Select either register pair.
V = 18	Select any one register.

EXITS: Normal exit only. Parameter register P1 contains the number of the register selected or the number of the lower-numbered member of the register pair selected.

OPERATION: If a specific register is requested, SELFR determines whether that register, or the register pair containing that register, must be stored. If storing is not necessary, the register storage is cleared, and the MRMFR table entry is initialized by raising its MSL and BUSY flags and clearing the rest of the entry. If the registers must be stored, temporary storage is assigned to the quantity in the register and instructions are generated to perform the storage.

If no choice of registers has been specified, SELFR assigns weights to each register according to its contents and either chooses the single register having the highest weight or the register pair having the highest combined weight, as appropriate. The selected registers are then treated as if they were specifically requested, as described above.

CEKMM -- Make Initial Assignment to General Register (ASAR)

ASAR is used to record in an MRM table entry the assignment of a quantity to the corresponding general register. See Chart GE.

ENTRIES: The entry point is CEKMM1. Parameter registers P1 through P6 contain the following:

- P1 Register number to be assigned.
- P2 Expression tree ID of quantity being assigned (or 4 if the quantity being assigned is an address generated with a BASR instruction).
- P3 Expression sign.
- P4 Pointer to name table, expression tree, or symbol table, according to ID.
- P5 Displacement -- applicable only if quantity is a variable.
- P6 Type -- applicable only if quantity is a variable.

EXITS: Normal exit only. Parameter register P6 contains the address of the MRM table entry that ASAR made.

OPERATION: ASAR sets the MRM table initial entry fields to the ID of the quantity to be assigned and clears the MSL flag. In addition, the register storage entry is made in the name table or expression tree if the quantity being assigned is either a common expression or an operator, respectively.

CEKMN -- Make Synonym Assignment to General Register (ASARS)

ASARS is used to record the assignment of a quantity as a synonym entry for a given general register. See Chart GF.

ENTRIES: The entry point is CEKMN1. Parameter registers P1 through P6 contain the following:

- P1 Register member to be assigned.
- P2 Expression tree ID of quantity being assigned (or 4 if the quantity being assigned is an address generated with a BASR instruction).
- P3 Pointer to name table, expression tree, or symbol table, according to ID.
- P4 Expression sign.
- P5 Displacement -- applicable only if quantity is a variable.
- P6 Type -- applicable only if quantity is a variable.

EXITS: Normal exit only. Parameter register P6 contains the address of the MRM table entry that ASARS made.

OPERATION: ASARS sets the fields of an MRM table synonym entry to the ID of the quantity to be assigned. If there is an inactive synonym entry, the first synonym entry is used. In the latter case, the register storage is cleared for the quantity being replaced.

CEKMO -- Make Initial Assignment to Floating-Point Register (ASFR)

ASFR is used to record in an MRMFR table entry the assignment of a quantity to the corresponding floating-point register.

ENTRIES: The entry point is CEKMO1. Parameter registers P1 through P6 contain the following:

- P1 Register number to be assigned.
- P2 Expression Tree ID of quantity being assigned (or 4 if the quantity being assigned is an address generated with a BASR instruction).
- P3 Pointer to name table, expression tree, or symbol table, according to ID.
- P4 Expression sign.
- P5 Displacement -- applicable only if quantity is a variable.
- P6 Type -- applicable only if quantity is a variable.

EXITS: Normal exit only. Parameter register P6 contains the address of the MRMFR table entry that ASFR made.

OPERATION: ASFR sets the MRMFR table initial entry fields to the ID of the quantity to be assigned and clears the MSL flag. In addition, the register storage entry is made in the name table or expression tree if the quantity being assigned is either a common expression or an operation, respectively.

CEKMP -- Make Synonym Assignment to Floating Register (ASFRS)

ASFRS is used to record the assignment of a quantity as a synonym entry for a given floating-point register. See Chart GG.

ENTRIES: The entry point is CEKMP1. Parameter registers P1 through P6 contain the following:

- P1 Register number to be assigned.
- P2 Expression tree ID of quantity being assigned (or 4 if the quantity being assigned is an address generated with a BASR instruction).
- P3 Pointer to name table, expression tree, or symbol table, according to ID.
- P4 Expression sign.
- P5 Displacement -- applicable only if quantity is a variable.
- P6 Type -- applicable only if quantity is a variable.

EXITS: Normal exit only. Parameter register P6 contains the address of the MRMFR table entry that ASFRS made.

OPERATION: ASFRS sets the fields of an MRMFR table synonym entry to the ID of the quantity to be assigned. If there is an inactive synonym entry, the first such entry is used. If there is no inactive synonym entry, the first synonym entry is used. In the latter case, the register storage is cleared for the quantity being replaced.

CEKMT -- Find Temporary Storage (FNDWS)

FNDWS is entered to locate and reserve a given number of bytes of object program temporary storage. In n bytes are requested, the first of the n contiguous bytes assigned will be located at a byte address which is an integral multiple of n. See Chart GH.

ENTRIES: The entry point is CEKMT1. Parameter register P2 must contain a 1, 2, 4, 8, or 16 to indicate the number of bytes of temporary storage needed.

EXITS: Normal exit only. Parameter register P2 contains a symbol table pointer to the adcon which must be used to cover a reference to the assigned temporary storage, and parameter register P3 contains the displacement to be used in conjunction with the adcon.

OPERATION: Available temporary storage is found by searching the temporary storage allocation matrix for an n-bit field of 0's which starts on a bit boundary location which is the smallest integral multiple of n. The field is filled with 1's, and the distance in bits from the origin of the matrix is computed. This distance represents the relative byte address of the temporary storage location assigned. The relative address is converted to a page address and a displacement within that page. An address constant is filed to cover the page. The symbol table pointer for this adcon and the displacement are returned to the caller.

The preceding discussion describes generally the process for assignment of temporary storage. Actually one of two kinds of temporary storage may be assigned depending upon the status of the flag GLMODE. If the flag is raised, global temporary storage is assigned; otherwise, local temporary storage is assigned.

The GLMODE flag is tested by FNDWS which simply decides whether to search the local or global half of the temporary storage allocation matrix and whether the address constant created should cover the local or global temporary storage class.

CEKMX -- Release Temporary Storage (RLSWS)

RLSWS is entered to make a temporary storage location available for reuse. See Chart GI.

ENTRIES: The entry point is CEKMX1. Parameter register P1 contains a 0 if the temporary location being released was occupied by an operator, or a 1 if it was occupied by a common expression. Accordingly, parameter register P2 contains the associated expression tree address or name table address.

EXITS: Normal exit only.

OPERATION: The adcon pointer, its associated displacement, and the operand type are extracted from the name table or expression tree. The adcon pointer is followed to the associated symbol table entry from which the storage class covered by the adcon and the location within that storage class are extracted.

If the storage class covered is global temporary, no further action is taken, and RLSWS exits. If the storage class is local temporary, a group of bits in the temporary storage allocation matrix is cleared. The first of these bits is determined from the sum of the displacement and address covered by the adcon. The number of bits cleared is determined from the operand type.

Two temporary locations are released in the above manner if the operand is a common expression and the name table flag "secondary temp assigned" is raised; otherwise, only one temporary location is released.

CEKON -- Register Storage Clear Routine (FLUSH)

FLUSH is invoked to initialize a given MRM table entry or to move it from one entry to another before initializing it. See Chart GJ.

ENTRIES: The entry point is CEKON1. The contents of parameter register P3 are treated as follows:

Contents	Action
0	Initialize a general register.
1	Move, then initialize general register.
2	Initialize a floating-point register.
3	Move, then initialize a floating-point register.

Parameter register P2 may contain either the number of the register whose MRM/MRMFR

Table entry is to be initialized or the address of the entry, as desired by the caller. Parameter register P1 is irrelevant if P3 = 0 or 2; otherwise, it contains the number of the register to which a move is required, or the associated MRM/MRMFR address, again at the option of the caller.

EXITS: Normal exit only. If only initialization is required, parameter register P6 contains the MRM/MRMFR address of the register initialized. If a move and initialize are required, P6 contains the MRM/MRMFR address of the register to which the movement will be made.

OPERATION: The distinction between register number and MRM/MRMFR address for the input quantities is made according to whether their values are less than or greater than 16, respectively. Each, which is a register number, is converted to the corresponding MRM/MRMFR address.

If a move is specified, the entire MRM/MRMFR entry is moved as specified. If the register from which the move is made contains operators or common expressions, the corresponding register number fields in the name table or expression tree entries are altered to reflect the new register location.

If no move is specified and the register which is to be initialized contains operators or common expressions, the corresponding "in register" flags in the expression tree or name table are lowered.

Processing is completed by initializing the specified MRM/MRMFR entry.

CEKNI -- Code File Output Subroutine (INSOT)

INSOT is called whenever an entry is made in the code file. See Chart GK.

ENTRIES: The entry point is CEKNI1. The input parameters to INSOT are

1. ID in parameter register P2.
2. OP (Line Number or SYMT with IDs 7 or 8) in parameter register P1.
3. All other parameters in a 6-word area at INSOTP (in phase's PSECT) in the order listed in Figure 34. Each parameter occupies one word and is right-justified.

ID	Other Parameters Required	Usage
0	OP, R1, R2	RR Instruction
1	OP, R1, X2, B2, D2, SYMT	RX Instruction
2	OP, R1, X2, B2, D2, DISPL	Local Branck (*+)
4	OP, R1, X2, B2, D2, SYMT	Temporary
5	OP, M1, X2, B2, D2, SYMT, ADCON	Branch-Displacement Supplied by Phase 5
6	OP, R1, R3, B2, D2	RS Instruction
7	Line Number	Statement Header
8	SYMT	Label Definition
9	None	End Program

SYMT - Symbol Table Pointer for label or primitive
Adcon - Symbol table pointer for Adcon
Line Number - Source line number from PF entry
All other symbols have the accepted System/360 meaning.

Figure 34. INSOT Input Parameters

If storage class is 7 or 8, the symbol table pointer points to the adcon covering the temp; otherwise, storage class field is 0, and Phase 5 obtains actual storage class from the symbol table.

EXITS: INSOT has two exits: one normal return and one error exit. The INSOTP area is undisturbed except for INSOTP +17.

OPERATION: Based upon the contents of input parameters, INSOT generates the appropriate code file entry (given in Appendix A). INSOT also maintains the size of Storage Class 1 in the storage class table, and updates the code file top in intercom. Label definitions are not entered in the code file if the ISD, memory map, and object listing options are all off.

CEKOQ -- Edit for Code File (EDIT)

EDIT may be used in preparation for generation of an RX-type instruction via INSOT. Its purpose is to centralize the preparation of some of the input parameters required by INSOT.

ENTRIES: The entry point is CEKOQ1. Entry is made with the tree address of the operand in register P2.

EXITS: Exit is made with the operation type code in parameter register P2. The ISYM entry of the INSOT parameter list is set according to the following table:

OPERATION: EDIT sets input parameters to INSOT according to the following table:

Operand ID	Op-Type	ISYM Contents
Constant	RX	Symbol table pointer
Variable	RX	Symbol table pointer
Adcon	RX	Symbol table pointer
Operator	temp. ref.	Temp-covering adcon pointer
CSX	temp. ref.	Temp-covering adcon pointer
Subscript Connector	RX	Variable symbol table pointer

INTRODUCTION

Phase 5 generates the output of the FORTRAN compiler. This output can broadly be divided into two parts: the object program module (OPM) and the external listings. The object program module consists of:

1. Loading information (relocation factors, external names, etc.). This information comprises the program module dictionary (PMD) and is used at load time by the dynamic loader.

2. Object text (code, constants, etc.).

3. An optional list, called the internal symbol dictionary (ISD), of the internal symbols in the FORTRAN program for use in checkout with PCS.

4. A list of external names (entry points, subroutine calls, etc.).

The external listings are produced in accordance with options selected by the user. It should be noted that the first option (of the five listed below) must be requested in order to receive any of the other four. The selections are:

1. A basic output listing which consists mainly of the names and sizes of control sections in the object program module.

2. An expanded listing of the above to include items such as the relative locations of labels and variables.

3. A comprehensive output that, in addition to 1 and 2 (above), lists object code and give an assembly-like listing of the object code along with comments identifying what type item is being referenced. This selection also produces a listing of adcons, parameter lists, and numeric and alphameric constants.

4. A symbol table list which gives all the variable names in alphabetical order with important attributes.

5. A cross reference list which gives all the variables listed in alphabetical order, followed by the labels, in numeric order, showing the line numbers where each is defined and referenced.

Thus, the function of Phase 5 can be stated as: generating the OPM (i.e., constructing the PMD, building the object program, and producing the optionally selected ISD) and producing various selections of external listings. These functions are itemized below, together with the routines that contribute to their development.

1. Generating the object program module

   a. The program module dictionary. The program module dictionary consists of heading information and one control section dictionary (CSD) for each control section in the object program.

      BUILD — Processes the heading information in the PMD.

      COSEC — Constructs the control section dictionary for the code control section.

      PRSEC — Builds the control section dictionary for the prototype control section.

      CMSEC — Constructs one control section dictionary for each COMMON.

      SPECS — Makes entries in the control section dictionary of the control section containing any preset data present.

   b. The Object Program

      BUILD — Determines the type of program under construction and initializes, communicates with other needed routines, and post-processes the object program.

      CMSEC — Builds one control section for each COMMON (named or blank) present.

      SPECS — Inserts preset data (when present) into the text of the appropriate control section.

      COSEC — Constructs the code control section which consists of code and numeric constants.

      PRSEC — Constructs the prototype control section. (The module PRSEC should be consulted if additional information is desired.)

c. Internal Symbol Dictionary

COSEC — Determines statement number entries and the number of control sections in the object program module and enters them into the ISD.

ASSIST — Responsible for all entries in the ISD except for those made by COSEC.

2. Producing various selections of external listings

EDIT — Optionally produces up to three levels of output documentation. The routine itself and Figure 35 should be consulted if more information is desired.

SYMSRT — Sorts and outputs an alphabetical listing of items in the symbol table along with important attributes relating to the symbol.

CRFSRT — Produces a listing of all variables in alphabetical order, followed by the labels in numeric order, indicating the statement numbers where each is defined and referenced.

PHEAD — Ejects to a new page and outputs the page heading, which consists of module name, data, and page number. PHEAD is called by the above three routines.

OBJECT PROGRAM MODULE (OPM)

The output from the compiler is known as an object program module. This is composed of a program module dictionary (PMD), text, and internal symbol dictionary (ISD), and external name list (ENL).

PROGRAM MODULE DICTIONARY (PMD)

Each PMD consists of one PMD heading plus as many control section dictionaries (CSD) as there are control sections in the module. Address pointers in the PMD are initially relative to the beginning of the PMD itself (not the PMD preface), except where otherwise specified. Some fields in the PMD are filled in by the loader. These are not set by the compiler. The PMD format is shown in Figure 35.

PMD Heading

1. Length of PMD in bytes

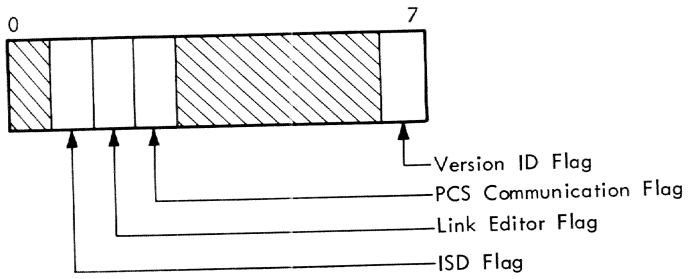
   This length does not include the PMD preface.

2. Diagnostic Code (1 byte)

   The Diagnostic code indicates the highest level diagnostic encountered during generation of the module by the language processor that created it.

3. PCS Communication Indicator (1 byte)

   This is a 1-byte field which is used by the program checkout subsystem (PCS). Currently defined settings are as follows (bits are numbered from left to right starting with 0):



Bit 0 - Module has been altered by a non-source modification. That is, a language processor did not make the change.

Bit 1 - Module has an ISD associated. This bit is set by the processor creating the PMD.

Bit 2 - Module was produced by link editing. This bit is set by the Link Editor.

Bit 3 - PCS is to be called before module is dynamically unlinked. This bit is set by PCS.

Bit 5 - Module was produced by the FORTRAN compiler.

Bit 6 - FORTRAN module is a main program, not a SUBROUTINE, FUNCTION, or BLOCK DATA subprogram.

Bit 7 - Version ID indicator. If this bit is set, the module version ID is to be interpreted as a 64-bit binary number which is the creation date of the module. If this bit is not set, the version ID is eight alphameric EBCDIC characters.

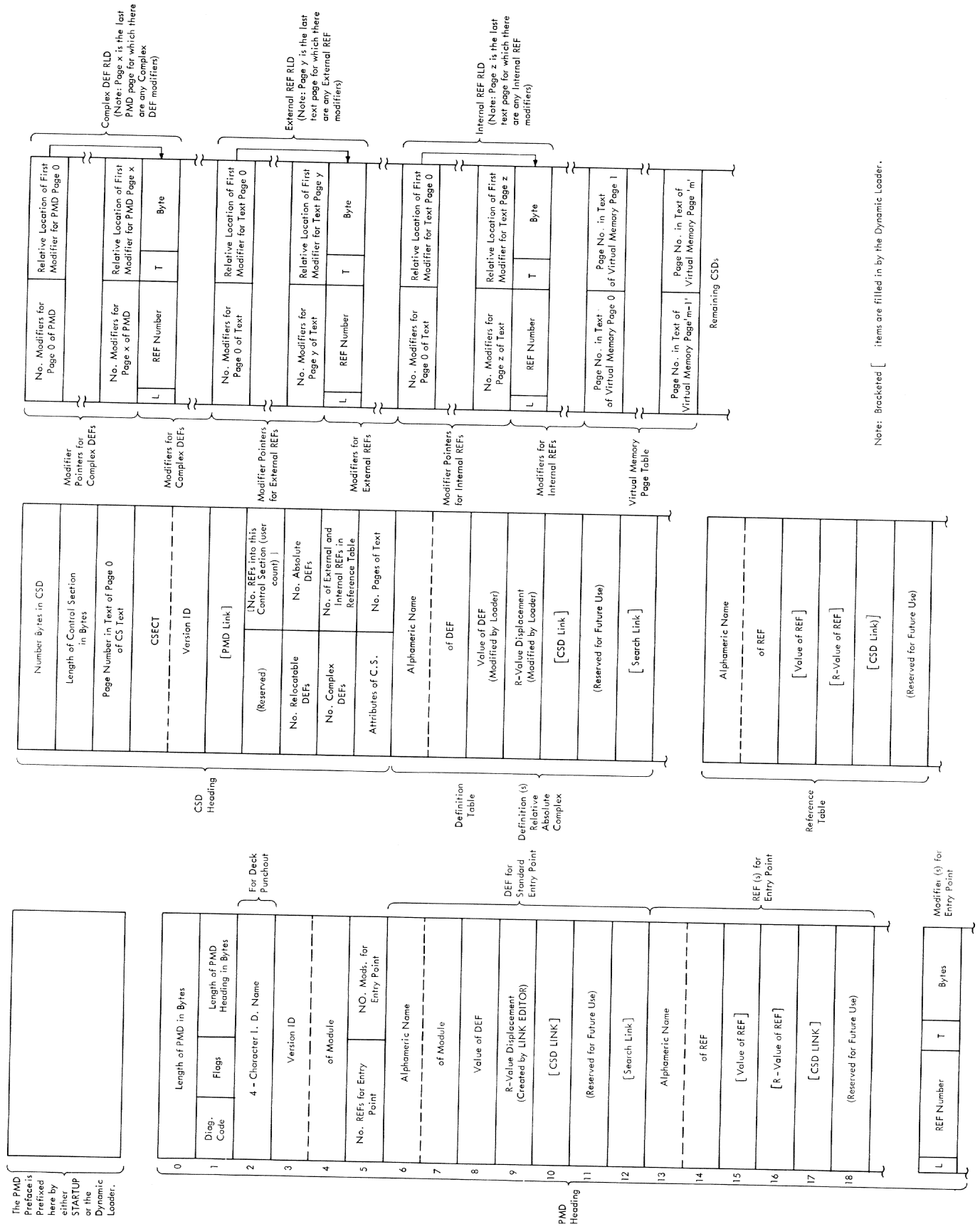


Figure 35. Format of PMD Entry

4. Length of PMD Heading

   This is the length in bytes of the PMD heading.

5. 4-Character ID Name

   The 4-character ID name is supplied by the user to serve as deck identification if the module is punched into cards. This field is currently unused.

6. Version ID

   See item 3 (bit 7 description) for interpretation of version ID.

7. Number of REFs for the Standard Entry Point

   The DEF for the standard entry point is always treated as a complex DEF. This field contains the number of REFs. It may be zero.

8. Number of Modifiers for the Standard Entry Point

   This field contains the number of modifiers that are to be used to compute the DEF for the standard entry point.

9. DEF for Standard Entry Point

   This 7-word entry describes the DEF for the standard entry point of the module. It has the same form as the individual DEF entries within the CSDs. The standard entry point DEF for the module is considered to belong to the first PSECT of the module and is treated the same as a complex DEF whose ENTRY statement appears within that PSECT. If no PSECT is declared, the standard entry point will be associated with the first CSECT instead. This DEF entry contains the following subfields which are described under "Control Section Dictionary."

   a. Alphameric name of module

   b. Value of DEF

   c. R-Value displacement

   d. CSD link

   e. Reserved for future use

   f. Search link

   The alphameric name is also the name of the module.

10. REF(s) for Entry Point

    These correspond to the REF(s) for complex DEFs within a CSD.

11. Modifier(s) for Entry Point

    These correspond to the modifier(s) for complex DEFs within a CSD.

Control Section Dictionary (CSD)

The control section dictionary (see Figure 35 comprises the following components:

1. CSD Heading
2. Definition Table
3. Reference Table
4. Relocation Dictionaries (RLDs)
5. Virtual Memory Page Table

CSD Heading

1. Number Bytes in CSD

   This field specifies the length of the control section dictionary in bytes.

2. Length of Control Section in Bytes

   This specifies the virtual storage span of the control section. The length of the virtual storage page table is derived from this length. For example, if the length of the control section is 8192, the virtual storage page table will contain two pages; but if the length is 8193 bytes, the virtual storage page table will contain three pages. This value will be equal to the highest location counter value assigned by the language processor, plus 1.

3. Page Number in Text of Page 0 of CS Text

   The text for each control section in the module occupies an integral number of pages in its resident data set. The text pages for all control sections in a module are contiguous. This number is the page number, relative to the first page of text for this module, of the first page of text for this control section. (Numbering begins with 0.)

4. Version ID

   This is a 64-bit binary number which is the creation date of the control

section expressed as the number of microseconds that have elapsed from March 1, 1900, until the time of control section creation.

5. PMD Link

   The PMD link is filled in by the loader. It points to the beginning of the PMD preface.

6. Number of Implicit References to this Control Section (User Count)

   This is a count of the number of REF entries which refer to this control section and are linked to this CSD through their CSD link. It is computed by the loader. It includes both external and internal references.

7. Number of Relocatable Definitions

   This is the number of relocatable definitions in the definition table. It is always at least 1, namely, the control section DEF.

8. Number of Absolute Definitions

   This is the number of absolute definitions in the definition table. It may be zero.

9. Number of Complex Definitions

   This is the number of complex definitions in the definition table. It may be zero.

10. Number of References from this CSD

    This is the sum of external and internal references in the reference table. It may be zero.

11. Attributes

    This word has one bit set for each attribute possessed by the control section. Currently defined attributes are shown below. Bits are numbered from left to right starting with 0. Bit 15 is not used:

    a. Fixed-Length (Bit 14 off)

       A fixed-length control section will be allocated a fixed number of pages at load time.

    b. Variable-Length (Bit 14 on)

       A variable-length control section is a section of indeterminate length and will be allocated pages in excess of the length stated in the CSD heading.

    c. Read-only (Bit 13 on)

       Read-only specifies that the control section may not be stored into. It causes storage protection by means of a storage class B assignment to all pages of the control section. Non-read-only and nonprivileged control sections are assigned storage class A.

    d. Public (Bit 12 on)

       Control sections are not shared by control section name alone. A PUBLIC control section of a module residing in a given data set (library) is shared if another user has access to the same data set and module. Control sections of a given module need not all be PUBLIC or non-PUBLIC. Fixed-length PUBLIC control sections with the same attributes are assigned storage in the same assignment. A PUBLIC control section should never contain relocatable adcons (A, V, or R type).

    e. PSECT (Bit 11 on)

       If this bit is set, the dynamic loader overrides the system packing indicator and inserts this control section as packed.

    f. COMMON (Bit 10)

       A COMMON section is a control section common to all modules in which it is declared. COMMON sections are more fully discussed in the linkage editor manual and the assembler manual.

       COMMON sections are of two types:

       (1) Named COMMON sections (those with a name not all blanks). These are treated as fixed-length sections.

       (2) Blank COMMON sections, whose name consists of eight blanks.

       FORTRAN blank COMMON is assigned the VARIABLE and COMMON attributes by the FORTRAN compiler.

       The treatment of blank COMMON sections differs from that of blank non-COMMON sections. Control section rejection is instituted between blank COMMON sections of different modules whereas blank non-COMMON sections of different modules are treated as independent

control sections. The latter are called unnamed control sections.

g. Privileged (Bit 9 on)

A control section with a privileged attribute is assigned storage key C which provides fetch as well as store protect. This attribute overrides read-only.

Anything in a privileged control section may be referenced only when the PSW key is zero.

h. SYSTEM (Bit 8 on)

Any external symbol that appears in a control section which has the SYSTEM attribute cannot be referenced by a user program unless the symbol begins with "SYS". Conversely, no reference from a control section with a system attribute may be to a "user" symbol.

i. Public Name (Bit 0 on)

This is used only by the dynamic loader to specify nonblank control sections whose names appear in the SDST (shared data set table). The first such control section will appear in the SDST under the module name. A control section may be indicated as both having a public name and rejected.

12. Number of Pages of Text

This specifies the number of pages of text for this control section in the data set. It should be noted that this generally does not correspond to the number of pages in the virtual storage page table. It cannot, of course, be larger.

Definition Table

The definition table is made up of 7-word entries, one for each external definition in the current control section. Definitions are grouped as relocatable, absolute, and complex in that order. The first definition in the table is the name of the current control section.

Relocatable definitions are external definitions whose values may be computed as the sum of the origin of the control section wherein they appear, and a constant.

An absolute definition is an EQU item with an absolute value whose name has been declared an entry point in the control section in which the name is defined.

A complex definition is either an EQU item with a complex relocatable value; i.e., containing external symbol(s), or a simple relocatable definition whose ENTRY statement appeared within a named section other than the section in which it is defined. The definition entry appears within the CSD of the control section which contains the ENTRY statement. (Note that the origin of the same control section is the R-value for the DEF.) The complex DEF is required in this case, with one REF entry that names the control section in which the DEF symbol is actually defined.

Each DEF in the definition table contains the following entries:

1. Alphameric Name of DEF

This field contains the 8-character alphameric name of the DEF.

2. Value of DEF

The value of the DEF is set by FORTRAN and is modified by the loader in the case of complex and relocatable definitions. For relocatable DEFs the value portion of the definition entry contains the displacement value of the symbol relative to the base of its control section.

For absolute DEFs this entry contains the absolute value; for complex DEFs it contains the absolute portion of the DEF value, which may be zero.

3. R-Value Displacement

The "displacement for R-value" word contains the displacement of the original defining control section origin with respect to the head of the control section within the definition now appears. This is required to compute valid R-values for control sections which have been COMBINED by Linkage Editing. In creating the PMD, only the Link Editor will ever produce a nonzero value in this word.

4. CSD Link

The CSD link is initially zero. It is filled in by the loader when the control section is loaded as a pointer to the beginning of the CSD in which this DEF appears, providing neither the DEF nor the control section has been rejected.

5. For future use.

6. Search Link

   This field is filled by the hash search routine of the loader. It contains the address of the beginning of the next DEF entry which hashes to the same value. It contains zero if thereare no more DEFs with the same hash value in this chain.

Reference Table

The reference table is made up of 6-word entries, one for each external symbol referenced within the control section. Each entry contains the following:

1. Alphameric Name of REF

   This field contains the 8-character alphameric name of the REF.

2. Value of REF

   This is filled in by the loader. It contains the value of the DEF to which the REF refers. If the DEF is undefined, it contains the address of a portion of virtual storage wherein reference is illegal.

3. R-Value REF

   This is filled in by the loader. It contains the virtual storage address of the beginning of the control section wherein the DEF appears. This value is obtained from the "R-value displacement" word of the satisfying DEF entry.

   If the DEF is undefined, this word contains the address of a portion of virtual storage wherein reference is illegal.

4. CSD Link

   This pointer, initially zero, is filled by the dynamic loader. It points to the beginning of the CSD wherein the DEF which defines this REF appears. If a corresponding DEF could not be found upon the appearance of a REF, the CSD link is to the beginning of the CSD wherein the REF itself appears.

5. Reserved for future use.

Relocation Dictionary (RLD)

Three RLDs appear in each control section dictionary:

1. RLD for complex definitions
2. RLD for internal references
3. RLD for external references

Each RLD has the same format consisting of modifier pointers and modifiers. The RLD for complex definitions differs in that pages mentioned in this table are pages of the PMD rather than the text.

Modifier Pointer

Modifier pointers are used to designate the application of modifiers to adcons on appropriate pages of text (or of the PMD for complex DEFs). The first modifier pointer applies to the first page; the second modifier pointer, to the second page; etc. For an RLD there always exists at least one modifier pointer. However, there need not necessarily be a modifier pointer for each page of text; the modifier pointers may be ended at the last text page for which there exists any modifier.

The modifier pointers consists of two fields, in the left and right halfwords.

Left-half - Number of modifiers for page

   This field contains the number of modifiers that apply in this page.

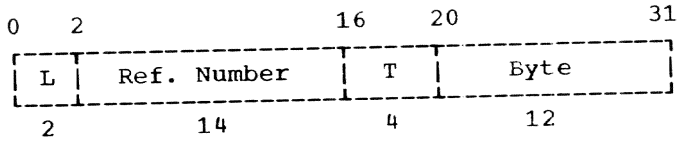
Right-half - Location of first modifier for this page

   This contains the location in bytes relative to the right half of the pointer itself for the first modifier for this page. If there are none, it points to the location where one would have appeared if there were any.

   A special note should be made of the technique for determining the length of an RLD. In the right half of the first pointer for an RLD, is the location of the first modifier for this page. In the word preceding the first modifier word is the last modifier pointer for the RLD. By adding the location of the right half (of the last pointer) to the contents of the right half (of the last pointer), one gets the beginning of the last set of modifiers. Adding to this four times the number of modifiers in the last set, one gets the end of the RLD.

Modifier

The modifiers are each a fullword and are divided into four fields:



L

L (2 bits) is the length in bytes of the adcon to be modified. A value of zero indicates a fullword (4 bytes).

Ref Number

Reference number (14 bits) is the ordinal number in this CSD's reference table of the reference whose definition value is to be used in modifying the adcon. References are numbered starting with zero.

T

T (4 bits) is the operation to be performed in modifying the adcon by the reference value.

The values of T currently defined are as follows:

a. Addition (T = 1)

The definition value of the reference at "Reference Number" is added to the field of L bytes starting at the indicated byte of the page to which the modifier applies.

b. Subtraction (T = 2)

Same as addition, except read "subtracted from" for "added to."

c. R-value (T =3)

The value from the "R-value" word of the REF is stored into the field of length L according to the "Byte" field.

Byte

Byte (12 bits) is the displacement in bytes (from the origin of its original containing page) of the adcon to be modified. It should be noted that since PMDs are packed to word boundaries, this displacement will be added to an address for complex DEFs which generally is not a page boundary.

RLD for Complex Definitions

The format of these modifiers is as described above. These modifiers apply to the values of complex definitions; that is, the byte addresses in the modifier will be to the value words of complex DEF entries in the definition table, and the page numbers in the modifier pointers are for pages of the program module dictionary itself.

RLD for Text External Reference

This relocation dictionary is in the same form as described above. It has one pointer for each page of program text up to that text page which is the last to contain an adcon, and appropriate modifiers for each adcon in the text which refers to a symbol defined externally to this module. The page numbers are based on the first page for this control section, beginning with 0.

RLD for Text Internal Reference

This is identical to RLD for text external reference above, except that the modifiers are to adcons in the text which reference symbols defined within this module, such as control section names. This permits communication between control sections of the same module that may be allocated noncontiguous virtual storage.

Virtual Memory Page Table (VMPT)

This table has a halfword for each page of virtual storage beginning with page 0 and continuing upward in order.

The contents of each entry will be either:

1. All bits if the corresponding page is empty as a result of a DS or ORG statement.

2. The number of the page in the text relative to the beginning of text for this control section if the page contains code or data. This value multiplied by 4 becomes an index into both the external and internal RLDs and is used to select the correct modifier pointer word for adcon relocation.

This table is the means by which the text of the control section is related to the virtual storage assigned the control section. This is necessitated by the fact that language processors do not necessarily output a byte of text for each byte of virtual storage assigned; that is, large ORG and DS statements may result in pages of text being skipped.

If, for example, a source program were to begin with

```
ORG   10000
```

there would be no text output for the first two pages of virtual storage, and the first page of text would correspond to the third page of the user's virtual storage. The first two VMPT entries would be all bits, and the third would contain zero. Within a page, however, the bytes of text correspond directly to the bytes of virtual storage. Thus, in the example above, the first page of text would represent virtual storage locations 8192-12287, and the first 1808 bytes of the page of text would be vacant (10000-8192 = 1808). The pages of text will always begin on page boundaries within the text module.

INTERNAL SYMBOL DICTIONARY (ISD)

The ISD (see Figure 36) has four sections: a heading, section name table, statement number table, and a symbol table.

Heading

Word 1 - Bits 0-15 contain the indicator (8) identifying the ISD as FORTRAN produced.

Word 2 - The length of the ISD in bytes.

Word 3 - Contains a link to the start of the symbol table.

Word 4 - The number of entries in the section name table.

Word 5 - The number of entries in the statement number table.

Word 6 - The number of entries in the symbol table.

Section Name Table

All control section names and their version identifications (CSECT, PSECT, labeled and blank COMMONs) are listed here. The last two entries are the CSECT and the PSECT.

Statement Number Table

For each executable statement in the program, FORTRAN inserts an entry containing the statement number and the offset from the CSECT base. Entries for unnumbered statements contain a statement number of zero. The entries are arranged in source order.

Symbol Table

The FORTRAN compiler inserts into the symbol table a defining item for all variables, section names, and FORMAT state-

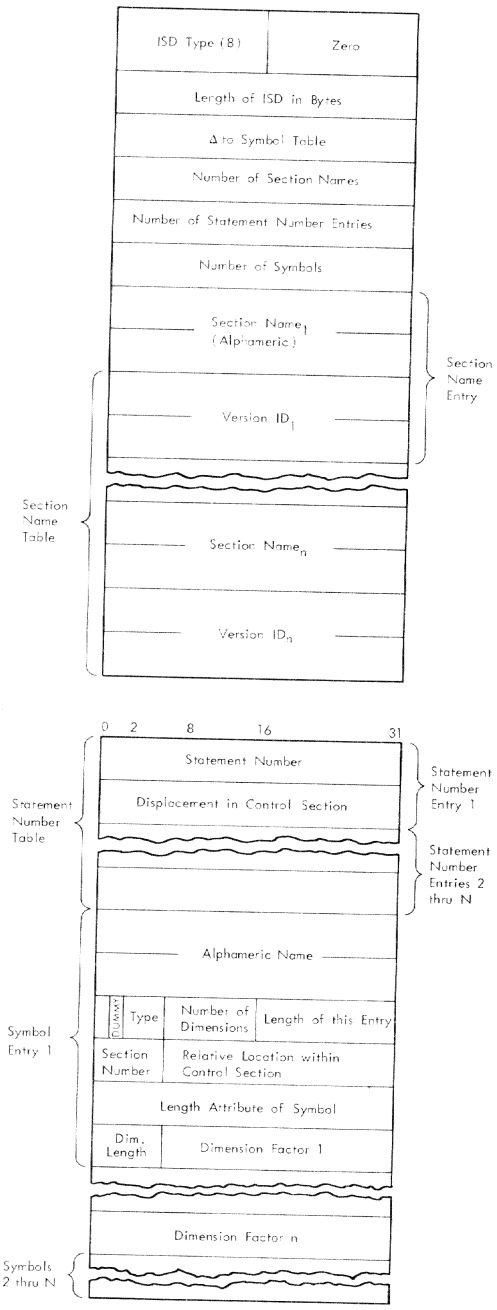


Figure 36. FORTRAN Internal Symbol Dictionary

ment numbers. Entries are grouped according to control section and are ordered within each group by ascending location counter value.

Name - Two words containing the alphameric name of the variable.

Type - Identifies the type of variable as:

Type	Code
Section Name	3
Integer	4
Real Number	5
Character Constant (FORMAT)	6
Complex Number	13
Logical	14

In addition the second high-order list in the TYPE field indicates if the variable is a formal argument. Thus, a type code '45' designates the appearance of a real variable as a formal argument.

Number of Dimensions - The number of dimensions of a dimensioned variable (0 for nondimensioned variables).

Length of Entry - Length in bytes for this symbol entry.

Section Number - A number corresponding to the order of the names in the section table of the ISD.

Displacement - The offset in bytes from the control section base.

Length - Length attribute of the variable.

Dimension Type - '00' for constant dimension '02' for adjustable dimension of type Integer*2
'04' for adjustable dimension of type Integer*4

Dimension - For each dimension of an array the dimension product value is listed. The value of the nth dimension factor is the byte length times the product of the sizes of dimensions 1 through n. For arrays that are formal arguments the dimension factor is:

1. The dimension itself, if constant.
2. Offset in bytes of the appropriate adcon from the base of object program's PSECT, if adjustable.

ROUTINE DESCRIPTIONS

Phase 5 routines bear mnemonic titles as well as coded labels. The 5-character coded labels begin with the letters CEKS; the fifth letter identifies a specific routine. Each of the routines in Phase 5 has a single entry point. When reference is made to a compiler executive routine or entry point, the mnemonic title is used, followed immediately by the corresponding coded label enclosed with parentheses.

There are no hardware configuration requirements for any of the Phase 5 routines. They are all reenterable, nonresident, nonprivileged, and closed. PHASE5 (Output Generator -- CEKSA) is entered by standard linkage; all other Phase 5 routines are entered by restricted linkage.

The relationships of routines in this phase are shown in the following nesting chart (Figure 37) and decision table (Table 29). The relationships are shown in terms of levels; a called routine is considered to be one level lower than the calling routine. Output Generator is considered to be level 1.

CEKSA -- FORTRAN Compiler Output Generator (PHASE5)

PHASE5 consists of terminal operations of the FORTRAN compiler. Its purpose is to

1. Build the object program text and the associated program module dictionary (PMD).
2. Construct an internal symbol dictionary (ISD).
3. Procuce user-selected documentation.
4. Generate entry point table.

ENTRIES: The only entry into PHASE5 is from the Compiler Executive, via standard linkage, at entry point CEKSA1. PHASE5 expects to receive the base of the Intercom as a parameter.

EXITS: Before exiting back to the Exec, PHASE5 checks the upper limit for the following items:

1. Object program text.
2. Program module dictionary.
3. Internal symbol dictionary.
4. External name table.

If the upper limit has been exceeded, PHASE5 will set the error code (in the intercom) to "Fatal" and output a diagnostic message.

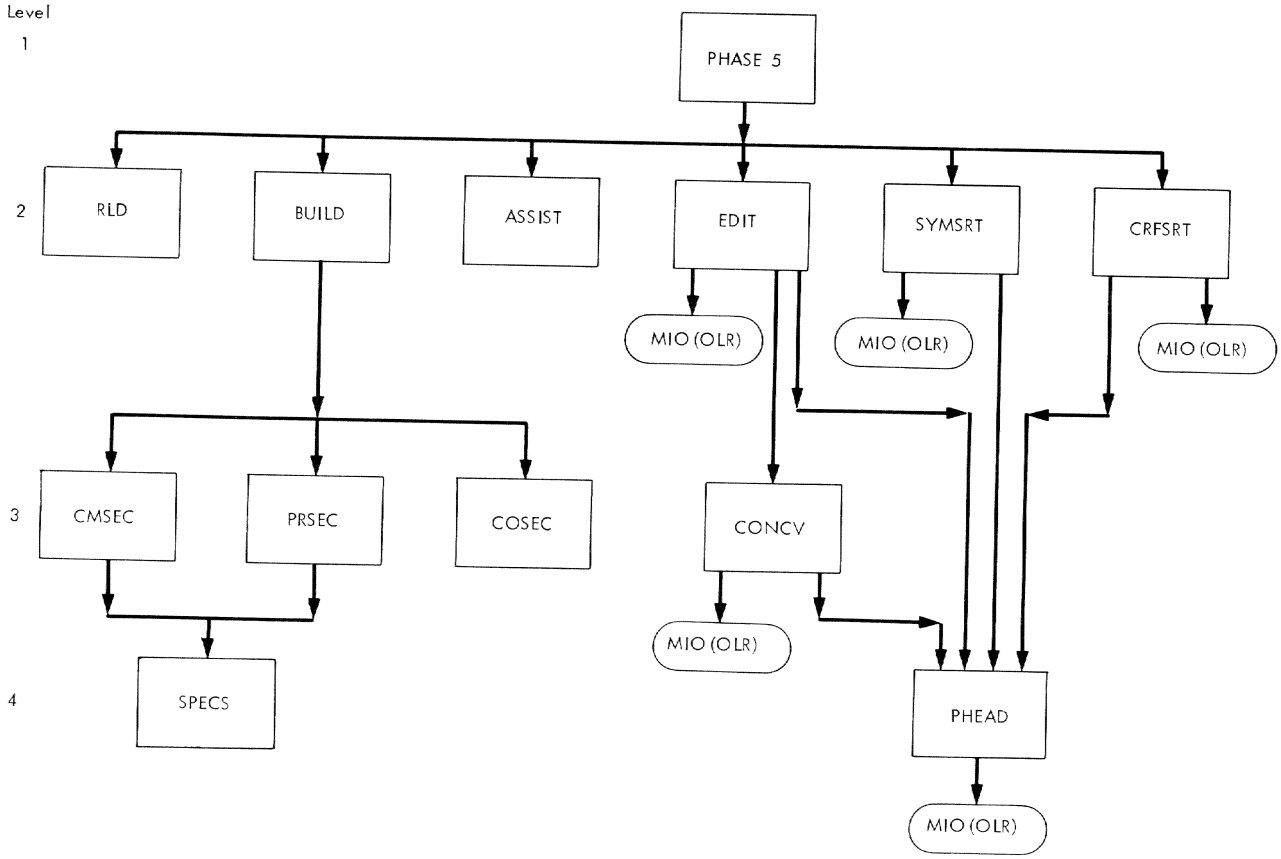


Figure 37. Phase 5 Nesting Chart

Table 29. Phase 5 Decision Table (Part 1 of 3)

Routine:----------Phase 5----------Level: 1----------

Routine	Usage	Called Routines	Calling Conditions
PHASE5	To control the overall operations of Phase 5.	BUILD	Entered for every compilation to produce text and the Program Module Dictionary (PMD).
		ASSIST	Entered when the option is requested by user to generate the Internal Symbol Dictionary (ISD).
		EDIT	Entered when the option is requested by user to produce and output the object program module information, the memory map, and the object code listing.
		SYMSRT	Entered when the option is requested by the user to produce and output the listing of the sorted Symbol Table information.
		CRFSRT	Entered when the option is requested by the user to produce and output the Cross Reference Listing.
		RDM (CEKTE)	Entered when an error condition is encountered.
		PHEAD	To eject a page and print the heading for the Table of Initialized Variables.
		CONCV	Entered to convert integer, real, or complex values for printing.
		OLR (CEKTHE)	An Exec routine entered to print each line of the Table of Initialized Variables.

Table 29. Phase 5 Decision Table (Part 2 of 3)

Routine:----------Phase 5----------Level: 2

Routine	Usage	Called Routines	Calling Conditions
BUILD	To produce the object program module (OPM) and Program Module Dictionary (PMD) for each compilation.	CMSEC	Entered for each COMMON block to create a Control Section in the OPM.
		PRSEC	Entered for each compilation to build a prototype control section (PSECT) for the OPM.
		COSEC	Entered for each compilation to create a Control Section for the object code.
ASSIST	To produce the Internal Symbol Dictionary (ISD) when the option is selected by the user.	None	
EDIT	To produce and output several user-selected listings concerning the OPM.	CONCV	Entered when a constant is encountered in the object code listing to convert to EBCDIC.
		PHEAD	Entered when a new page of the listing is needed, ejects the page and prints the heading.
		OLR (CEKTHE)	An Exec routine entered to print each line of the output listing.
SYMSRT	To produce and output an alphabetical listing of all the items in the Symbol Table.	PHEAD	Entered when a new page of the listing is needed, ejects the page and prints the heading.
		OLR (CEKTHE)	An Exec routine entered to print each line of the output listing.
CRFSRT	To produce and output a Cross Reference List for all symbols and statement labels.	PHEAD	Entered when a new page of the listing is required, ejects the page and prints the heading.
		OLR (CEKTHE)	An Exec routine entered to print each line of the listing.

Routine:----------Phase 5----------Level: 3

CMSEC	To create a Control Section (CSECT) corresponding to a COMMON block definition.	SPECS	Entered for BLOCK DATA subprograms to store the pre-set data.
COSEC	To produce the Control Section for the object code and the numeric constants and enter some information into the PMD and ISD.	None	
PRSEC	To produce the prototype control section (PSECT) for the OPM.	SPECS	Entered if any pre-set data is present to store it in the noncommon variables.
CONCV	To convert constants to EBCDIC for the output listing.	PHEAD	Entered when a new page is needed to eject a page and print the heading.
		OLR (CEKTHE)	An Exec routine entered to print each output line.

Table 29. Phase 5 Decision Table (Part 3 of 3)

Routine:----------Phase 5----------Level: 4

Routine	Usage	Called Routines	Calling Conditions
SPECS	To place the values given in pre-set data statements into the text of the appropriate control section.	RDM (CEKTE)	Entered if an error condition encountered.
PHEAD	To eject a page, update the page number, form and print the page heading, and initialize the line count.	OLR (CEKTHE)	An Exec routine entered to print the heading.

OPERATION: PHASE5 (Figure 38), activated by a call from the exec, combines and edits outputs from the earlier phases, to produce the object program and the various optional program documentation. The symbol table, the code file from Phase 4, and several special lists from Phase 1 are the main sources of material.

PHASE5 initializes nonvolatile register N1 with the symbol table base and thus establishes a phase-wide common register. It also copies the intercom into the Phase's PSECT.

PHASE5 proceeds to construct the object program text and the program module dictionary (PMD). If the ISD option has been selected and the program is not a BLOCK DATA subprogram, PHASE5 will build the related internal symbol dictionary. After successfully building the OPM and its related ISD(if requested), PHASE5 edits any user-selected documentation.

Finally, PHASE5 restores the compiler's intercom and returns to the phase controller.

CEKSB -- Object Program Module Builder (BUILD)

The purpose of subroutine BUILD is to construct the object program module and to BUILD a part of the internal symbol dictionary. See Chart GL.

ENTRIES: BUILD has only a single entry (CEKSB1) from PHASE5. BUILD expects no input parameters other than those contained in the phasewide register assignments.

EXITS: Normal exit only to PHASE5. No output parameters.

OPERATION: The object program module is a part of the edited end product of the TSS/360 FORTRAN compiler. More specifically, it is that part which ultimately participates in the routine execution of a task.

The object program module consists of executable object code, and other control and reference parameters necessary for the relocation and execution of the control sections contained within the OPM. From the viewpoint of the compiler, the OPM is the result of one complete pass through the compiler, and thus represents a unit of source code terminating with an END statement.

The OPM is organized into several control sections (CS), each of which has a dictionary part (CSD) and an optional text part. This material of the OPM is divided into two data sets, with one set containing the module heading and CSDs (also called PMD), and the other set containing the text part. The text of each control section starts on a page boundary. Named COMMON control sections may or may not contain text. The OPM is designed to be compatible with the assembler output and is suitable for processing by the dynamic loader and link editor.

Initialization of Module: Upon entry BUILD initializes the object program module. The initialization consists of:

1. Pre-processing of the PMD heading (see Figure 35) which includes the following of:

   a. OPM name (six characters) is obtained from the intercom region and inserted in the 8-byte name field (left-justified) of the standard entry point (SEP).

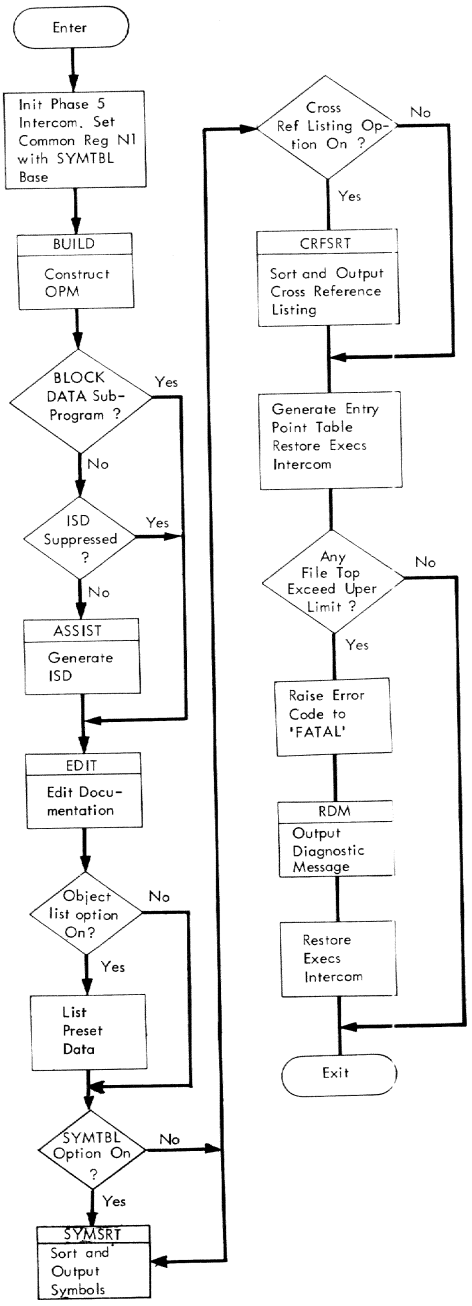


Figure 38. Phase 5 General Flow

b. The length of the PMD heading, in bytes is inserted in the PMD.

c. The diagnostic code field is set to the contents of maximum error code in the intercom region.

d. The deck ID name is obtained from the intercom region and inserted into the 4-character ID name field.

e. The PCS communication indicator is inserted, based upon the contents of the ISD flag and whether the module is a main program. The FORTRAN module bit is set on.

f. The version ID is retrieved from the intercom and inserted in the 8-byte field allocated. The PCS communications indicator is set when the version ID is a time-stamp.

g. Number of references and number of modifiers are both set to 1.

h. The name field in the reference from the SEP is set to:

(1) Name of the CSECT, if other than block data.

(2) CHCIW5, if block data.

i. The fields in the modifier are

(1) Reference number and T field are set to 1.

(2) L and byte are set to 0.

2. Establishment of the number of COMMON control sections.

3. Initializing two parameter registers, one to the base of the first CSD in the PMD and the other to the base of the OPM.

Processing of COMMON Control Sections: The CMSEC subroutine is called by BUILD repeatedly. CMSEC generates a control section corresponding to a COMMON block. In a BLOCK DATA subprogram BUILD terminates the module immediately after processing COMMON control sections.

Code and PSECT Control Sections: Subroutines COSEC and PRSEC produce the CSECT control section and PSECT control section, respectively.

Termination of a Module: Before returning to PHASE5, BUILD inserts the length (in bytes) of the PMD into the appropriate call in the PMD heading.

CEKSC -- Common Control Section Generator (CMSEC)

The purpose of CMSEC is to create a control section corresponding to a COMMON definition. CMSEC is used by BUILD. See Chart GM.

ENTRIES: CMSEC has a single entry (CEKSC1) from BUILD.

CMSEC expects the following input parameters, in addition to those specified in the phase-wide register assignments:

1. A pointer to the current COMMON block entry in the storage class table.
2. The location for building the CSD in the PMD.
3. The base for making text entries into the current COMMON control section.

EXITS: Before exiting, CMSEC

1. Sets the base for the next CSD entry in the PMD. Specifically, this is the first word boundary after the last entry made (in the PMD) by the current call on CMSEC. This value will be passed back to BUILD as a register parameter and will also be stored in Phase 5's intercom (as the current PMD top).
2. If any text entries were made into the current COMMON control section, CMSEC sets the base for the next entries in the OPM. The base will be the first page boundary following the last page of text of the current control section. This parameter will be passed back to BUILD in a register and will also be stored in the intercom as the (current) OPM top.

OPERATION: Upon entry, CMSEC clears the CSD heading and definition. The number of relocatable definitions is set to 1, and the attributes are set to indicate the type of COMMON control section (blank or named COMMON). The name of the COMMON block, found in the storage class table, is inserted (left-justified) in the alphameric name of definition field. The storage class table entry also supplies the size of the COMMON block. The number of pages of virtual storage is now calculated

$$N_P = \frac{\text{Size} + 4095}{4096}$$

and inserted in the CSD. The virtual storage page table of the CSD is then constructed and consists of N_P halfwords, containing only 1 bits.

In the case of a BLOCK DATA subprogram, control sections corresponding to named COMMON blocks undergo additional processing. CMSEC prepares to process preset data and calls for the SPECS subroutine. SPECS performs the following functions:

1. Does the actual scan of the preset data file.
2. Selects the appropriate DATA items.
3. Generates the necessary text pages.
4. Moves the preset data items into the text.
5. Indicates in the virtual storage page table those pages that contain text.

SPECS returns to CMSEC, which terminates the control section. Termination of CMSEC includes entering the number of bytes in the CSD into the first word of the CSD and the number of bytes of control section into the second word of the CSD.

CEKSF -- Code Control Section Generator (COSEC)

COSEC produces one of the component control sections of the object program module (OPM). In particular, it produces the control section that consists of the object code, including numeric constants. In addition, COSEC enters information into the program module dictionary (PMD) and generates part of the internal symbol dictionary if the ISD flag is on. See Chart GN.

ENTRIES: COSEC has a single entry (CEKSF1) from BUILD.

Input Parameters:

1. Origin of CSECT's CSD.
2. Origin of CSECT's text.

EXITS: COSEC makes a single exit to BUILD.

Output Parameters:

1. Instructions that needed their displacement fields filled in before being stored into CSECT's text will be returned to the code file in completed form. (This is done to facilitate the work done by EDIT, whose function is to produce an output listing).
2. As an additional service to EDIT, COSEC will store in Phase 5's PSECT the number of bytes of text of the code control section and the origin of the OPM's PSECT.
3. The top of the code control section will be stored in a specified location in Phase 5's PSECT and will also be passed as a parameter in a register.

4. The register that contains the base of the CSD for CSECT will be set to the first word boundary following the last entry COSEC made. This value will also be stored in a specified location in Phase 5's PSECT.

OPERATION:

Code Control Section: The code control section (CSECT) consists of code and numeric constants.

1. Object Code. The code entries into the CSECT are determined by an investigation of the code file (generated by Phase 4). The items of the code file are sequentially scanned and may be ignored or entered into the CSECT, or they may give information for determining an entry in the CSECT. Entries into the CSECT are placed sequentially, as they are encountered (or determined). Three registers are used during this operation; one register marks the item to be investigated in the code file, and two registers are used to indicate the available byte for storage in the CSECT. The first byte (operation field) of each item in the code file determines the action to be taken. Following are the actions taken for the various possible values.

   Let X represent the value of the first byte of the item being investigated. Then:

   a. For X less than or equal to 2:

      No entry is made into the CSECT.

      If X is equal to 1, the location counter of the code file is incremented by eight bytes; otherwise, it is incremented by four bytes.

   b. For X strictly less than 40_{16} and strictly greater than 2:

      The item being investigated is a halfword instruction and is placed into the CSECT. The index register for CSECT is incremented by two bytes, and the address register of the code file by four.

   c. For X strictly less than $D0_{16}$ and greater than or equal to 40_{16} :

      The item being investigated is a fullword instruction and is placed in the CSECT. CSECT's index register and the code file's address register are both incremented by four bytes.

   d. For X greater than or equal to $D0_1$ and strictly less than FE_{16}:

      The item being investigated is a 6-byte instruction and is placed in the CSECT. CSECT's index register is incremented by six bytes, and the location counter for the code FILE scan is incremented by eight bytes.

   e. For X exactly equal to FE_{16}:

      The word being investigated, plus the following two words, contains information for determining a fullword entry into the CSECT.

      The second word (investigated word plus 1) is to be placed into the CSECT after its displacement field has been completed.

      Both the first and third words (second half) contain pointers to the symbol table. The first-word pointer is to an adcon entry, and the third-word pointer is to a statement label entry.

      The displacement is calculated by subtracting the offset (in the value field of the adcon) from the assignment of the label.

      The address register of the code file is updated by twelve bytes, and the index counter of CSECT by four.

   f. For X exactly equal to FF_{16}:

      This is the termination code to inform COSEC that the scan of the code file is completed.

2. Numeric Constants. Upon completion of the code file scan, COSEC will check storage Class 2 in the storage class table for the presence of numeric constants. In the event that storage Class 2 is not empty, COSEC will execute the following.

   a. The first available byte for a numeric entry into the CSEC will be set on the first quadruple-word boundary following the last code entry (these zero to twelve bytes will be filled with zero bits).

   b. The numeric constants will be retrieved and placed in CSECT's text, by following pointers to the symbol table from the constant header table.

Control Section Dictionary Entries: (See Figure 35) Each control section has a control section dictionary (CSD) associated with it in the program module dictionary (PMD). The origin of CSECT's CSD will be passed as a parameter in a register for COSEC to make the following entries:

1. Initialize the CSD heading to all zero bits.

2. Retrieve from the intercom the time stamped version ID of the module and insert it into the CSD.

3. Set the appropriate word in the CSD to indicate CSECT's attributes, which are reentrant and read only, plus PUBLIC if indicated by the user's options.

4. Retrieve from the intercom the module name, add a suffix (#C) to this name, and place it in the CSD.

5. Calculate and insert the length (in pages) of CSECT into the CSD. This is determined by subtracting CSECT's origin from its next available storage byte, and dividing the result by 4096. The integral portion (plus 1 if the remainder is nonzero) is the number of pages.

6. Complete the virtual storage page table for the CSD. A halfword page number entry will be made for each text page of CSECT. The page number entries will be ordinal numbers of the form I, where I = 0, ..., n-1 ($0 \leq n-1$) and n equals the integer portion of

$$\frac{\text{no. bytes of text} + 4094}{4096}$$

7. The number of bytes in the CSD will be placed in the first word of the CSD.

8. In the second word of the CSD, COSEC will store the number of bytes of text of the code control section.

ISD Entries: (See Figure 36) Concurrent with the building of CSECT, certain entries will be made in the ISD, if the ISD option is not suppressed. Initially, the number of control sections will be calculated and stored (right-justified) in the fourth word of the ISD. The number of control sections is equal to the number of COMMON control sections plus 2 (one for PSECT and one for CSECT).

The number of COMMON control sections is directly obtainable from the storage class table. The first two bytes of the storage class table give the number of named COMMONs, and the tenth word (Storage Class 9) indicates if there is a blank COMMON.

The number of control sections, multiplied by 16, plus 24, gives the offset (in bytes) for the beginning of statement number storage in the ISD.

During the scan of the code file, while CSECT is being built, if the first byte of the item being investigated has a value of 1 or 2, it may cause a statement number entry to be made into the ISD.

1. If the value of the item is 1, the item is a statement header and is eight bytes long. The last four bytes of the item contain a line number. This line number is checked against the last line number encountered. If it is the same, no entry is made into the ISD. However, if it is different, an entry will be made into the ISD. The statement number field in the ISD will be set to all zero bits, and the following word (offset field) will have the current value of CSECT's offset counter stored in it. In addition, the new line number will replace the old line number (for testing), the location counter for the ISD (marking the next available word for a statement number entry) is incremented by eight bytes, and the code file's address counter is incremented by eight bytes.

   Note: The next paragraph outlines an additional condition on the handling of statement headers.

2. If the value of the byte being investigated is 2, the item is four bytes long, with the last two bytes of the item containing a pointer to a statement label entry in the symbol table. A test is conducted (on the class field of the label) to see if it is a source label. If it is, the binary value of the label is retrieved (from the symbol table) and placed in the statement number field of the ISD. The word following the label entry will have the contents of CSECT's offset counter stored in it. The ISD's location counter is incremented by eight bytes and code file's address counter is incremented by four bytes. Further, no entry into the ISD will be made for the next statement header encountered.

   Note: Upon completion of statement number entries into the ISD, COSEC will place the number of statement number entries made into the fifth word of the ISD. Also, at this time the ISD's location counter will contain the beginning location for symbol entries (in the ISD). The offset from the base of the ISD to where the sym-

bol entries are to start will be calculated and stored in the third word of the ISD.

CEKSG -- PSECT Builder (PRSEC)

The purpose of subroutine PRSEC is to build a prototype control section for the object program module. See Chart GO.

ENTRIES: PRSEC has a single entry (CEKSG1) from BUILD.

Parameters expected by PRSEC, in addition to phase wide register parameters, are the base of its CSD and its text base.

EXITS: Parameters passed upon exit back to BUILD are the PMD top and the OPM top.

OPERATION: The building of the PSECT may be divided into the procedures discussed below. Concepts and terminology with respect to the program module dictionary (PMD) are closely related to their usage in TSS/360 Dynamic Loader, Internal Programming Specifications.

1. Initialization of PSECT's control section dictionary (See Figure 35)

   The CSD heading is cleared. The version ID time stamp is inserted in the fourth and fifth words of the CSD. The number of relocatable definitions field is set equal to 1. The attributes field is set to indicate a fixed-length prototype control section. The name of the module, with a suffix #P, is inserted in the name field of the first definition.

2. Processing of Entry Points

   A special entry in the intercom region points to the beginning of a chain in the symbol table containing (exclusively) descriptions of entry points. PRSEC follows this chain and processes each entry as follows:

   a. The name of the entry is inserted in the definition table.

   b. The offset (SLOC) is inserted as the value of DEF.

   c. The number of complex definition is increased by 1.

   d. The CSD index is incremented appropriately.

   e. Unusual fields are cleared to binary zeros.

3. Construction of Reference Table

   The reference table of the PSECT's CSD contains entries similar to the entries in the definition table. There are two types of entries:

   a. Names of control sections of this OPM.

   b. Names of external references (entry points defined in other modules).

   Subroutine PREC first processes references to control sections within the module. The procedure is similar to the processing of entry points, except that the number-of-references field is incremented by 1 for each entry in the reference table. Control section names are entered in the order in which the control sections appear in the OPM. External references are obtained by following a chain in the symbol table. Reference numbers, which are the ordinal numbers of the entries in the reference table, are retained for subsequent use in constructing the relocation directories. Each storage class is represented by one word starting at WORK + 100 in the phase's intercom, and each word contains an LA reg, d instruction, where d is preset with the appropriate reference number. The reference number of an external reference is saved in the DMLST field of the corresponding symbol table entry.

4. Building of Relocation Directory (RLD) for Entry Points

   Each item in the definition table, that describes an entry point, has a 1-word relocation item in the RLD for complex definitions. The RLD itself starts with a list of 1-word modifier-pointers. The following steps are performed to establish the modifier-pointer list.

   For each PMD page, up to and including complex definitions, a 1-word modifier-pointer is inserted into the CSD. The value of each modifier-pointer is

   4 (N - n) + 2

   where:

   N is the number of pages in the PMD.

n is the page number to which the modifier-pointer refers.

Thus, initially, each pointer indicates the first RLD item.

Each RLD item has zero in its L field and 1 in its T field, indicating a length of four bytes for the adcon to be modified, and specifying addition as the modification operation. The reference number is the ordinal number of CSECT's reference in the reference table. The byte address is set to the displacement of the value field of the complex definition within the appropriate PMD page.

At the completion of the RLD, or at page boundaries within the complex definition list, the number of RLD items pertaining to that page is inserted, in the left half of the corresponding modifier-pointer. The pointer of the next modifier-pointer, if there is one, is increased by the current number of bytes in the RLD.

5. Initialization of PSECT Text

The first 84 bytes of the PSECT text are cleared. Before offsets in PSECT of the various storage classes can be computed, the size of NAMELIST information must be determined.

6. Calculation of NAMELIST Size

NAMELIST size is accumulated as follows:

Symbol table entries in the non-variable name chain are scanned and searched for NAMELIST names. Each NAMELIST name contributes 12 bytes. Each variable in a NAMELIST increases the size by 16 + 8N bytes, where N is the number of dimensions. The number of variables is obtained from the NAMELIST name entries, and dimension information is obtained by following symbol table pointers from the preset data file NAMELIST entries.

The offsets from PSECT's text base are computed and stored temporarily in the PSECT's register save area.

7. Processing of External References

Address constants for external references may be present in the adcon page, as well as parameter lists. If the combined size of register save area, local working area, address constants, and parameter lists does not exceed one page of PSECT's text, the RLD modifier-pointer is set to 4 (4-byte integer) and the adcons are immediately processed. Otherwise, the portion of parameter lists which is in excess of one page is examined for the presence of external references. If none are encountered, processing takes place as above. If, however, there are external references, the number of additional text pages containing such references is determined, and the appropriate number of words in the RLD modifier-pointer list is cleared.

Processing of external reference adcons and the corresponding relocation items in the RLD is as described in the following paragraphs:

Adcon entries in the Symbol Table are scanned. The value of the adcon is tested for storage class 254 and, if the test is successful, the corresponding non-variable name entry (in symbol table) is inspected for its class. If the class indicates external reference, the name is matched against the list of reference names in the CSD, to obtain the ordinal number of such matching reference. The relocation item is composed to contain 0 in the L field, and the ordinal number of the reference in the external reference number field. The T field is set to 2, and the byte field is set equal to the offset of the adcon + 84. Four bytes in the PSECT, starting at the location indicated by byte field are cleared, the number of text modifiers field in the appropriate modifier-point is increased by 1, and a pointer to the RLD is advanced by four bytes so that the next relocation item may be received.

External reference adcons in parameter lists are processed in similar manner, except that the byte field is increased by the size of storage class 4 to give the proper offset within PSECT.

If the external reference adcons are distributed over several pages of PSECT's text, a different procedure is followed. After processing adcon page, a counter is initialized to determine page boundaries within parameter lists. At the start of each page, the pointer in the modifier-pointer for that page is set with the location, relative to the modifier-pointer, of the next relocation item. The byte field in the relocation item contains the displacement within the corresponding text page.

8. Processing of Internal References and NAMELIST Items

The number of modifier-pointers for the RLD for internal references is determined by the number of pages in PSECT's text, from base to Namelists inclusive. Processing of the RLD entries is very similar to the procedure outlined under Processing of External References. There are, however, several differences. In the relocation item, the reference number field is obtained as follows:

a. For adcons pertaining to storage classes 1 and 2, the references number is

   (Number of COMMON blocks) + 1

b. For adcons covering storage classes 3, 4, 5, 6, 7, and 8, it is

   (Number of COMMON blocks) + 2

c. Adcons referring to storage classes 9 through 127 have a reference number = (storage class) - 8, if blank COMMON is present, or (storage class) - 9, if blank COMMON is absent.

d. Adcons that belong to one of the storage classes from 128 through 253 have no RLD entry made in the CSD; they will, however, cause a word to be set to zero-bits in PSECT's text.

In general, the value of an adcon is

   Storage Class + Offset

If for any adcon (both storage classes 4 and 5) the storage class of its value is 254, the offset part of the value field contains a symbol table pointer, and PRSEC obtains the value of the offset from the symbol table entry. In any case, the content of the 4-byte adcon is computed as follows:

   Offset in Storage Class + Offset of Storage Class in CS

If an adcon that points to a NAMELIST name entry in the symbol table is encountered, the contents of the adcon are made equal to the SLOC of the NAMELIST name, if assigned, or to the next available NAMELIST entry. The NAMELIST entries are processed to give information in the format specified by FORTRAN I/O. The location field in each variable description of the NAMELIST entry is set with a symbol table pointer to that variable for later processing. The SLOC field of the NAMELIST name entry is assigned as offset in PSECT.

Adcons in parameter lists and namelists are processed as described before. A page count and special handling of RLD, similar to the procedure mentioned in (7) above, may take place. Adcons in NAMELIST (locations of variables) are computed from the appropriate symbol table entries.

9. Processing of Alphameric Information

The alphameric information will be retrieved from the preset data file and stored in the PSECT, beginning at the first doubleword boundary after the last NAMELIST entry.

10. Insertion of Virtual Storage Page Table

A virtual storage page table (VMPT) is constructed and contains halfword entries of ordinal numbers from 0 to n-1, where n is the number of pages that contain text (up to and including NAMELIST). Pages corresponding to noncommon variables, global and local temps are allocated by setting the corresponding number of halfwords in the VMPT with 1-bits.

11. Processing Preset Data Stems in non-COMMON Variables

If preset data is present, subroutine SPECS is called to select and insert any preset data items of storage class 6 into the area of nonCOMMON variables.

12. Termination of PSECT

Termination of PSECT consists of:

a. Storing the number of bytes in the PSECT's text into the second word of the CSD.

b. Setting the OPM top in Phase 5's intercom to the first page boundary following the last text entry in the PSECT (also passed in a register).

c. Storing the number of bytes in PSECT's CSD into the first word of the CSD.

d. Setting the PMD top in Phase 5's intercom to the first doubleword boundary following the last CSD

entry made by PRSEC (also passed in a register).

e. The last six words of the first 100 bytes of the PSECT will be used as masks and should contain values (left-justified) as shown

Bytes	Value	Usage
76-79	80_1	1-word sign mask
80-87	$4E_1$	2-word float mask
88-95	00	2-word temporary
96-99	46_1	1-word float mask

CEKSD -- Preset Data Processor (SPECS)

For variables whose initial values are given, subroutine SPECS places these values into the text of the appropriate control section. See Chart GP.

ENTRIES: Subroutine SPECS has a single entry (CEKSD1) from either subroutine CMSEC or subroutine PRSEC.

Input Parameters:

1. Location of virtual storage page table in current Control Section Dictionary.
2. Base of storage class in current control section.
3. Storage class of items to be filed in current control section.

EXITS: Normal exit only. SPECS returns control to the calling program with input parameters unchanged.

OPERATION: For the purpose of the following discussion, the following conventions are established:

Variable - A variable is either a simple variable or an array variable.

Variable - The value of a variable is either the initial value of a simple variable or the initial value of an element of an array.

Preset Data - Preset data are those entries in the preset data file (PDF) that originated from a DATA statement or from data specified in a type specification statement.

The preset data file is a prime source of information used by SPECS. The preset data entries in the PDF contain, among other information, a link to the next variable in the preset data file, a pointer to the symbol table for each variable, one or more value entries for each variable (i.e., elements of an array), a replication factor for each value, and an offset entry that indicates where the values are to be placed from the base of the variable's storage class.

For descriptive purposes, SPECS is characterized in terms of retrieval and storage.

Retrieval:

1. Subroutine SPECS is called during the generation of a PSECT if any preset data is present in the PDF or during the generation of a named COMMON control section of a BLOCK DATA subprogram.
2. SPECS locates the first preset data entry in the PDF by following a pointer given in the intercom.
3. For each variable within each preset data entry, SPECS follows the variable's pointer to the symbol table and tests its storage class.

Storage:

1. The values for variables of appropriate storage class are stored into the text of the current control section.
2. For each page that is to have information placed on it, SPECS enters the virtual storage page number in the CSD.

CEKSH -- Internal Symbol Dictionary Generator (ASSIST)

Phase 5 of the FORTRAN compiler will test to see if the user has chosen to have an internal symbol dictionary (ISD) generated. In the event the user has indicated his desire to do so, it will be the function of ASSIST to generate the ISD (see Figure 36).

It should be noted that no ISD will be built for a BLOCK DATA subprogram and that the ISD is a prerequisite for utilizing the program control system (PCS). See Chart GQ.

RESTRICTIONS: The first word at the beginning of a FORTRAN-generated ISD must have a 1 in bit 20. The rest of the word will be zeros.

ENTRIES: Subroutine ASSIST has a single entry (CEKSH1) from PHASE5.

ASSIST has the following input parameters:

1. Offset (in bytes) in PSECT of nonCOMMON variables.

2. The third word of the ISD contains the offset from the base of the ISD for symbol entries and is filled in by COSEC.

EXITS: Normal exit only, to PHASE5. Before exiting, ASSIST stores the ISD top into the intercom.

OPERATION: The entries in the ISD primarily are subdivided into control section name entries, statement number entries, and symbol entries. Statement number entries are made (in the ISD) during the time that subroutine COSEC is operating and will not be discussed here.

Control Section Names: All control section Names (CSECT, PSECT, labeled and blank COMMON) are listed in the ISD. Immediately following each control section name entry in the ISD is the alphameric version ID of the module.

Method: COMMON names (including blank COMMON) are among the items listed in the storage class table. ASSIST will retrieve these names from the storage class table and insert them in the ISD. The name used for the CSECT name will be the name of module with a #C suffix, and the name used for the PSECT name will be the module name with a #P suffix. (If the module name is seven or eight characters long, it will be truncated to six characters before the suffix -- #C for the CSECT name and #P for the PSECT name -- is added.)

Symbols: Each symbol, either COMMON or nonCOMMON (integer, real, complex, or logical) will have information about it placed in the ISD. Most of the information is obtained from the symbol table.

The information in the ISD will consist of

Alphameric Name	Obtained from the symbol table
Type	Obtained from the symbol table
Number of Dimensions	Obtained from the dimension table
Length of Entry	Calculated
Length Attribute of Symbol	Obtained from the symbol table
Section Number	Refers to the order in which the symbol's control section name appears in the ISD.
Offset	For variables in COMMON control sections, the offset is equal to the contents of the SLOC field assigned to the symbol in the symbol table. For symbols belonging to the PSECT, the offset will be the contents of the SLOC (from the symbol table plus the offset of nonCOMMON variables in the PSECT. (This additional offset is an input to ASSIST.)
Dimension Constants	Obtainable from the dimension table, or none.

The symbol table has all symbols linked according to their respective storage classes. Initial entry to the first variable in the chain is accomplished by following the pointers listed in the storage class table.

CEKSI -- Object Program Documentation (EDIT)

The purpose of EDIT is to edit the object program module at a user-selected documentation level. See Chart GR.

ENTRIES: EDIT has a single entry (CEKSI1), from PHASE5.

Upon entry, EDIT expects the following information:

The number of bytes of the code control section (first word of Phase 5's work area).

Origin of the OPM's PSECT (second word of Phase 5's work area).

Parameters passed in the phase-wide assignment of registers.

EXITS: EDIT exits to PHASE5. No output parameters.

OPERATION:

General Discussion: There are three documentation levels at which an object program module may be edited.

1. The basic level is always edited. The user receives this documentation even if he fails to specify one of the higher documentation levels. The basic level consists of:

a. Program header, including the OPM name and the combined size of the CSECT and PSECT.

b. Names and offsets of entry points.

c. Names of external references.

d. Names and sizes of control sections.

e. Names, offsets, and sizes of storage classes, if other than control sections.

The basic documentation level is included in both of the higher levels.

2. The second level is generated if the user has selected the MAP option. At the MAP level, the basic level is augmented to include (in storage order):

   a. Names and locations of labels.

   b. Names, offsets and sizes of COMMON and nonCOMMON variables.

3. The third level of OPM documentation is obtained by exercising the LIST option. The LIST documentation level expands the MAP level to a full representation of the object program module. This level adds:

   a. A listing of the CSECT text, including object code and numeric constants.

   b. A listing of the PSECT text, including tabulation of address constants, available information about parameter lists, and alphameric constants.

Throughout the documentation, locations and offsets are given as the number of bytes from the corresponding control section base, in hexadecimal. All sizes, in decimal, indicate the number of bytes.

Method: (See Figure 39) After initialization, EDIT prepares the program header, consisting of the OPM name and the size of CSECT + PSECT, as follows:

```
NAME                    SIZE
```

All entry points are now listed, one entry per line:

```
ENTRY NAME              OFFSET IN CSECT
```

External references are represented by name only:

```
EXTERNAL REFERENCES NAME1, NAME2, NAME3,
                    NAME4, ...
```

This completes the header information, and EDIT now processes the CSECT, if it is present. At the basic documentation level, the CSECT is described as follows:

```
CSECT NAME                          SIZE
  CODE                              SIZE
  NUMERIC CONSTANTS   OFFSET        SIZE
```

The necessary information is obtained from the storage class table.

If the MAP level is selected, EDIT appends a list of labels to the CODE entry.

```
LINE NO.     LABEL      LOC HEX
XXXXX.XX     LABEL1     OFFSET
XXXXX.XX     LABEL2     OFFSET
                .          .
                .          .
                .          .
```

The labels are listed in the order of their assignments and are obtained by scanning the code file and utilizing the label entries in the symbol table.

With the LIST documentation level, EDIT generates a full expansion of the object code, as well as constants. The general format, appropriately arranged within an assumed line of 132 print positions, is as follows:

```
LINE NO.  LABEL  LOCATION  INSTRUCTION
COMMENTS
```

The LINE NO, represented in statement header entry of the code file as a packed decimal, is edited as follows:

1. All leading zeros are suppressed.

2. In the case of integers, the decimal point and fractional digits are suppressed.

LABEL entries may consist of:

1. Source statement numbers.

2. Internally created labels (>99999).

3. Entry point names.

EDIT processes LABEL entries as they are encountered in the code file.

LOCATION contains the location of the instruction with respect to the base of the CSECT. It is obtained from a register, which keeps a cumulative total of bytes in the object code.

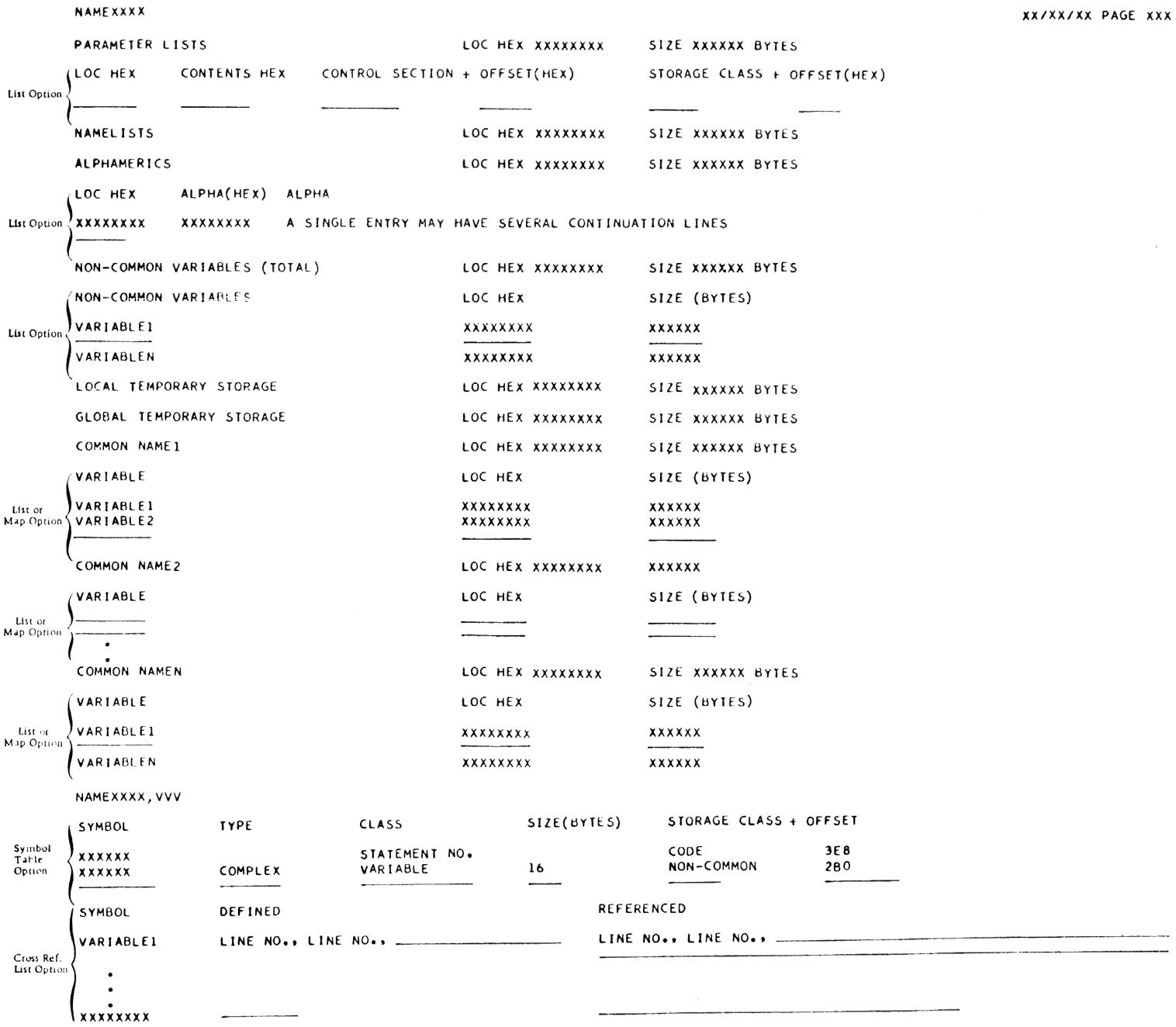
```
NAMEXXXX,VVVVVVVV[1]                                                    XX/XX/XX PAGE XXX

NAMEXX          SIZE XXXXXXXX BYTES

ENTRY NAME      LOC HEX

ENTRY1          XXXXXX
ENTRY2          XXXXXX
  .               .
  .               .
ENTRYN          XXXXXX

EXTERNAL REFERENCES

EXTER1      EXTER2      EXTER3      EXTER4      EXTER5      EXTER7      EXTER8
                                    EXTERN

NAME#C          SIZE XXXXXXXX BYTES         MM/DD/YY            HH:MM/SS[2]

CODE                                        LOC HEX 000000      SIZE XXXXXXXX BYTES

List Option
LINE NO.    LABEL   LOC HEX     INST HEX    INST ASSEMBLER      COMMENTS
XXXXX.XX    XXXXXX  XXXXXXXX    XXXXXXXX    XXXX XX,XXXX(XX,XX) (±.XXXXXXXXXXXXXXXXE±XX,±.XXXXXXXXXXXXXXXXE±XX)

Map Option
LINE NO.    LABEL   LOC HEX
XXXXX.XX    XXXXXX  XXXXXXXX

NUMERIC CONSTANTS                           LOC HEX XXXXXXXX    SIZE XXXXXX BYTES

List Option
TYPE                LOC HEX                 VALUE
C*16                XXXXXXXX                (±.XXXXXXXXXXXXXXXXE±XX,±.XXXXXXXXXXXXXXXXE±XX)
I*2                 XXXXXXXX                482
R*4                 XXXXXXXX                ±.982E+1

NAME#P          SIZE XXXXXXXX BYTES         MM/

REGISTER SAVE AREA                          LOC HEX 00000000    SIZE      76 BYTES

CONVERSION CONSTANTS                        LOC HEX 0000004C    SIZE      24 BYTES

ADDRESS CONSTANTS                           LOC HEX XXXXXXXX    SIZE XXXXXXXX BYTES

List Option
LOC HEX     CONTENTS HEX    CONTROL SECTION + OFFSET(HEX)       STORAGE CLASS + OFFSET(HEX)
XXXXXXXX    XXXXXXXX        NAME#C          47                  CODE            37
XXXXXXXX    XXXXXXXX        NAME#P          4E7                 ALPHAMERIC      E87

                                    54 Lines per Page
```

[1] VERSION ID

[2] TIME STAMP (FOR CONTROL SECTIONS ONLY)

Figure 39. Output Listing Format (Part 1 of 2)

Figure 39. Output Listing Format (Part 2 of 2)



An INSTRUCTION entry is edited in two parts. The hexadecimal representation of the instruction, as given in the code file, is edited first. The second part contains an assembly-like entry of the instruction. The operation field is translated to the corresponding mnemonic; the operand fields are converted to decimal integers and are rearranged with the appropriate punctuation.

A COMMENTS entry consists of a description (if available) of the second operand. It may contain one of the following:

1. Name of a variable, where all subscripted variables are represented by the array name only.
2. Label which appears as a statement number.
3. Constant, shown as a literal.
4. Address constant, edited to give results in the form

STORAGE CLASS NAME + OFFSET

EDIT uses a descriptor entry in the code file to obtain information for the COMMENTS field. The descriptor trails the respective instruction and points to the appropriate symbol table entry.

The representation of numeric constants has entries similar to the object code, with the following exceptions:

1. LINE NO. and LABEL entries do not apply.
2. The COMMENTS entry is omitted.

The symbol table provides EDIT with the necessary information to list numeric constants.

After completing the editing of the CSECT, EDIT processes the PSECT. At the basic documentation level, PSECT is edited as follows:

```
PSECT NAME                              SIZE

  REGISTER SAVE AREA                    SIZE
  ADDRESS CONSTANTS          OFFSET     SIZE
  PARAMETER LISTS            OFFSET     SIZE
  NAME LISTS                 OFFSET     SIZE
  ALPHAMERIC CONSTANTS       OFFSET     SIZE
  NONCOMMON VARIABLES        OFFSET     SIZE
  LOCAL TEMPORARY STORAGE    OFFSET     SIZE
  GLOBAL TEMPORARY STORAGE   OFFSET     SIZE
```

At the MAP level, the NONCOMMON VARIABLES entry is expanded to include individual variable names.

```
NONCOMMON VARIABLES   OFFSET   SIZE

  VARIABLE1           OFFSET   SIZE
  VARIABLE2           OFFSET   SIZE
      .                  .       .
      .                  .       .
      .                  .       .
```

The LIST documentation level includes description of alphameric constants, address constants, and parameter lists. For each address constant, EDIT supplies the following information:

LOCATION/CONTENTS/CONTROLSECTION/STORAGE CLASS

LOCATION gives the offset of the adcon from the PSECT base. CONTENTS is the value of the adcon. CONTROL SECTION gives the name of the referenced control section, and STORAGE CLASS refers to the particular class in that control section, such as code, numeric, alphameric, etc.

For any COMMON control section, EDIT produces, at the basic documentation level, the name of the section and its size. The MAP and LIST levels expand each COMMON entry with a listing of variables. The format is similar to the representation of NONCOMMON VARIABLES in the PSECT.

The editing of COMMON cntrol sections completes the work of EDIT on the OPM. Symbol table and cross reference Index option flags are examined, and, if selected, the appropriate table is edited.

CEKSJ -- Symbol Table Sort (SYMSRT)

SYMSRT produces, upon request, an alphabetical listing of items in the symbol table, as shown in Figure 39. See Chart GS.

ENTRIES: SYMSRT has a single entry (CEKSJ1) from PHASE5. PHASE5 uses the INVOKE macro to call SYMSRT.

EXITS: Normal exit only. No output parameters.

OPERATION: PHASE5 inspects an entry in the intercom to determine if the user desires an output listing of items in the symbol table. Should this option be selected, PHASE5 invokes SYMSRT, which provides an alphabetical listing of the following items:

1. Entry names.
2. NonCOMMON variable names.
3. COMMON variable names.

4. External names.

5. NAMELIST names.

In addition to writing the name of the item, SYMSRT writes its type, class, storage class, and offset from base of storage class, where these items are applicable.

Type - I*2, I*4, C*8, R*4, etc.

Class - Array variable, simple variable, entry name, external name, etc.

Storage Class - Code, nonCOMMON, named COMMON, etc.

Immediately following the listing of the items (explained in the preceding discussion), a listing of labels, both source-generated and compiler-generated, are output in ascending order.

CEKSE -- Output Page Heading (PHEAD)

CEKSE produces the page heading for each page of output listing generated by PHASE5.

ENTRIES: PHEAD has asingle entry (CEKSE1) and is invoked by:

EDIT
SYMSRT
CRFSRT
CONCV

CEKSE expects one input parameter in register P1 for initializing the line-count counter. The value must be 0, 1, or 2. The value passed is subtracted from some-constant, and the line count counter is initialized to the result.

EXITS: Normal exit only. No output parameters.

OPERATION: Upon entry, CEKSE does the following:

1. Initializes a cell in the phase's work area that is used as a line count counter (i.e., the number of available lines left on the current output page).

2. Skips to a new page.

3. Forms and writes a page heading for the new page. The heading consists of the module name, the date, and the page number. All three items may be found in the phase's copy of the intercom.

CEKSL -- Constant Conversion (CONCV)

CONCV converts constants (integer, real, or complex) to EBCDIC code for output in documentation of a FORTRAN-compiled program. See Chart GT.

ENTRIES: CONCV has a single entry (CEKSL1) and is invoked by subroutine EDIT.

Input Parameters

A parameter register passed to subroutine CONCV contains the address of a constant's descriptive entry in the symbol table.

EXITS: Normal exit only. No output parameters.

OPERATION: While EDIT edits the code file to generate the output listing, any comment entries that refer to constants cause EDIT to invoke subroutine CONCV. CONCV converts, formats, and writes the constant and its offset in the code control section.

A parameter register passed to subroutine CONCV contains the address of the constant's descriptive entry in the symbol table. From the descriptive entry, CONCV determines the following:

1. The constant type - integer, real or complex.

2. The length of the constant - two, four, eight, or sixteen bytes.

3. The address of the constant itself.

4. The constant's offset from the base of the CSECT.

The first three items are used to convert and write the constant in the correct format, and the fourth item is written in hexadecimal, as additional information for the user.

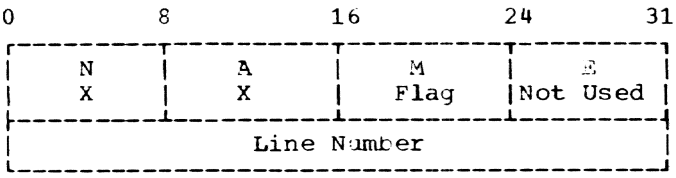
CEKSK -- Cross Reference List Routine (CRFSRT)

CEKSK prints a cross reference listing for the variables (symbols) and labels, from the list generated by Phase 1. See Figure 39 and Chart GU.

ENTRIES: This routine has one entry point (CEKSK1).

EXITS: Only the normal exit is made, with no output parameters.

OPERATION: Upon entry, CRFSRT makes a pass across the cross reference list which was generated by Phase 1. Each entry is examined. If it is a symbol reference, the symbol table pointer is used to get the symbol name. A table of symbol references is built in the code file with the following format:



where

"NAMEXX" is the symbol name in EBCDIC (or BCD), lef-justified with trailing blanks.

"Flag" indicators a reference (X'02') or a definition (X'01').

"Line Number" is the line number of the reference or definition, in packed decimal.

If the cross reference listing (CRL) entry is a label reference, a table is built in the CRL, with the format unchanged.

After all the original entries in the CRL have been sorted into symbols and labels, the sorting of each list is begun. First the symbol list is sorted in alphabetical order, with definitions first in increasing line numbers, followed by the references, also by increasing line number. The logic involved in the sorting of the list is best explained by the accompanying flowchart (Chart ON). In this type of sort scheme, a "delta" is calculated which is one-half the list size. Each entry (i) is compared with the corresponding "i+delta" entry. If a switch is necessary, it is made. If a switch of two entries is made below a certain point in the list, more comparisions are made before stepping to the "i+1" entry. When the bottom of the list is reached, the delta is reduced to half its size and the list scanned again. When delta is zero, the sort is completed.

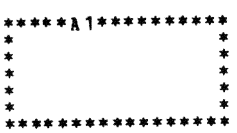
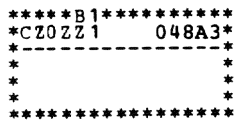
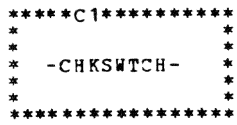
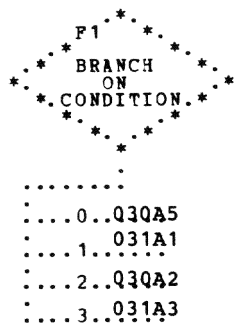
After the symbol list has been put into proper order, the heading for the symbol cross reference listing is printed. The symbol "name" is printed on the first line of its group only. If more than one line of printing is necessary, the rest are single-spaced. When a new name is found, the first line is double-spaced to separate the groups. The line numbers are printed with two spaces between the last digit of one and the first significant digit of the next. Leading zeros of line numbers are not printed.

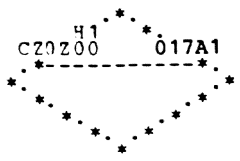
After all the symbol list entries have been printed, the label list is sorted and printed in the same manner.

SECTION 8: FLOWCHARTS

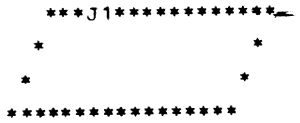
Each chart in this section is referenced from an associated routine described in an earlier section of this manual. The charts in this section are presented in the same order as are the routine descriptions. Not all routines are illustrated by charts.

The flowcharts in this manual have been produced by the IBM System/360 Flowchart Program (Flowchart/360), using USASI symbols. These descriptions of the USASI symbols and the Flowchart/360 conventions will simplify interpretation of the flowcharts in this manual:

SYMBOL	DEFINITION
	The processing block indicates any processing function, or a defined operation that causes change in value, form, or location of information.
	When this block is striped, it indicates the entry point of a subroutine or module that is included in the flowcharts in this manual. System/360 Flowchart automatically generates a page-and-block locator (048A3 in this illustration) that specifies the page with this entry point. See below for an explanation of page-and-block locators.
	When a call is made to a subroutine that is not in a flowchart, but is described in this manual, the call is shown in a processing block without the stripe and the subroutine's entry point is shown.
	The library or predefined process block indicates a module or subroutine that is in the flowcharts of another PLM. Whenever possible the entry point of the module or subroutine is listed. Refer to the Flowchart Directory in IBM System/360 Time Sharing System: System Logic Summary, Form Y28-2009.
	The decision block indicates a decision- or switching-type operation that determines which of a number of alternate paths should be followed.
	When there are more than three alternatives, a branch table is generated.

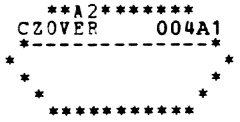


When the decision process is so involved that a detailed flowchart is required, the decision block is striped, and the page-and-block locator is generated as for the processing block.

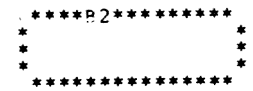


The I/O block indicates the general I/O functions, which include (but are not limited to) the GET, PUT, READ, WRITE, or device-control macro instructions, and the SIO instruction. Wherever possible, the entry point of a macro instruction processor is shown.

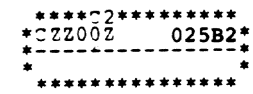
The modification block indicates an instruction or process that changes program operation, e.g., sets a switch, modifies an index register, or initializes a routine.



If the modification is performed by a subroutine that is included in a flowchart in this PLM, the modification block is striped and a page-and-block locator is generated.



The terminal or interrupt block indicates a terminal point in a flowchart. It is used to show start, stop, halt, delay, or interrupt. The terminal block is always used for either entry to a routine or for exit when a routine has completed its processing, and will not be reentered for the same service request. This block is also used for macro instructions such as ABEND, EXIT, and RETURN.



The terminal block will be striped if the exit is to a routine that is included in a flowchart in this PLM. A page-and-block locator is automatically generated when this block is striped.

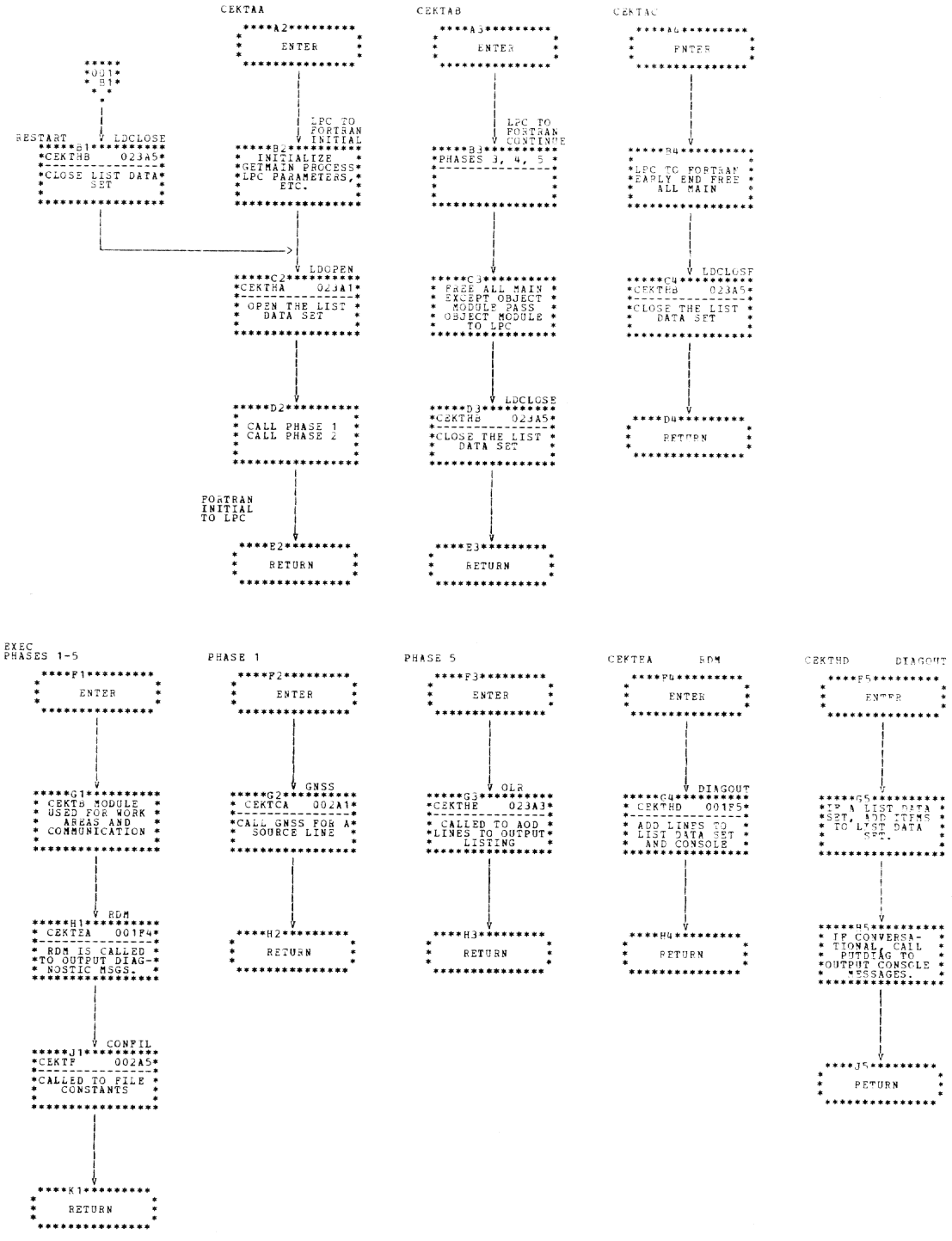
The on-page connector indicates exit to or entry from a block on the same flowchart page.

The off-page connector indicates entry from or exit to a block on another page of the same flowchart. Note: Exit to another flowchart in this PLM is indicated by a striped block.

Page-and-Block Locators

The page-and-block locator generated for off-page connectors and striped flowchart blocks is a five-character string. The first three characters indicate the flowchart page on which the transfer address is located. This page number is relative to the first flowchart page and appears in the upper right corner of each flowchart. The last two characters indicate the block at which the entry point is defined on the referenced page.

Chart AA. Executive Overall Flow -- CEKUA (Page 1 of 2)



Program Logic Manual

GY28-2019-1

FORTRAN IV Compiler

Flowcharts on pages 217-630 were not scanned.

PROGRAM REPRESENTATION FILE (PRF)

Program Representation File entry identification values are as follows:

Entry Name	Identification Code (16)
Begin Program	1
Subprogram Entry	2
Alternate Entry	3
Label Definition	4
Equation	5
Unconditional GO TO	6
Assigned GO TO	7
Computed GO TO	8
ASSIGN	9
Arithmetic IF	A
Logical IF	B
CALL	C
Argument Definition Point	D
RETURN	E
Begin Loop 1	F
Begin Loop 2	10
Begin Loop 3	11
End Loop	12
CONTINUE	13
READ	14
READ with Namelist	15
READ without Namelist	16
WRITE	17
WRITE with Namelist	18
PRINT	19
PUNCH	1A
Output List Element	1B
End List	1C
File Control	1D
STOP	1E
PAUSE	1F
End Program	20
Input List Element	21

Field Identifiers

All fields marked "not used" contain zeros. ID appears in every item and identifies the kind of item. Other fields, the IDs of the items in which they appear, and explanations of the fields, are:

Field	IDs	Explanation
ABN	5,A,B	Abnormal Function flag.
ALAB	9	Symbol Table pointer to label descriptive part entry.
ASTX	2,3	Symbol Table pointer to formal argument descriptive part.
BL3PT	12	PRF pointer to corresponding Begin Loop 3.
BEG	10,12	Symbol table pointer to lower loop limit descriptive part pointer.
BL1PT	12	PRF pointer to corresponding Begin Loop 1.
BL2PT	12	PRF pointer to corresponding Begin Loop 2.
CDP	5,A,B, C,11, 21	In Phase 2, zero; or a link to the previous PRF entry in which COMMON was redefined. Redefinition is either by a call on an abnormal subprogram or by definition of a formal argument called by name. Or, a link to the previous PRF entry in which a COMMON variable was defined.
CEX	C	EF pointer to subroutine entry, in call (;) expression, or alone if no parameters.
CNT	1E	No. of characters in message.
EDLNK	11	PRF pointer to corresponding End Loop entry.
END	10,12	Symbol Table pointer to upper loop limit descriptive part entry.
EOF	14,15	In Phase 1, label value in binary of End-of-file return. Negative if reference is to next statement. In Phase 2, Symbol Table pointer to label descriptive part entry.
ERR	14,15	In Phase 1, label value in binary of Error return. Negative if reference is to next statement. In Phase 2, Symbol Table Pointer to label descriptive part entry.
ESLOC	4	Estimated location in object program.

ETRA	A	Zero branch.
EXITLB		Symbol Table pointer to label outside a loop branched to from within a loop -- Set by Phase 2 for determining Materialize on Exit optimization in Phase 3.
FLAGS	4	Bit 7-Statement is labeled, but must be referenced.
FLAGS	14,16, 17,19, 1A	Bit 1 - Non-zero indicates no list with I/O statement (X'80'). Bit 2 - Non-zero indicates LABF is an expression file Pointer (X'40').
FLAGS	11,12	(Left to Right) X'80' - Labels in Loop X'40' - Unsafe Loop X'20' - Materialize X'10' - Parameter X'08' - Global Flag (inner, no external calls) X'04' - BXLE on Rec X'02' - ONEASN (remove floating load) X'01' - IOFLAG (Phase 1)
FNSW	1D	0 for END FILE 1 for REWIND 2 for BACKSPACE
GLAB	11,12	Symbol Table pointer to created loop top label descriptive part entry.
GLNK	6,7,8, A,B, C,14, 15	Link to previous PRF entry containing a label reference
GTRA	A	Positive branch.
ILNK	all	Link to previous PRF entry.
INC	10,11	Symbol Table pointer to loop increment descriptive part entry.
IVAR	10,11	Symbol Table pointer to loop variable descriptive part entry.
LABF	14,16 19,1A	Symbol Table pointer to FORMAT label descriptive part entry or Expression File pointer to Format variable name.
LABN	15,18	Symbol Table pointer to Namelist descriptive part entry.
LEV	11	Level of nesting of this loop.
LINO	all but 1,4,D, 10,11, 12,1B, 1C,20, 21	Line number in packed decimal.
LLNK	2,3,4	In Phase 1, link to previous PRF Entry or Label Definition entry. In Phase 2, link to previous PRF Entry or referenced Label Definition entry. In Phase 3, link to previous PF entry or referenced label definition entry.
LLNO	6,7,8 C	In Phase 1, label number is binary. Negative if reference is to next statement. In Phase 2, Symbol Table pointer to label descriptive part entry.
LTRA	A	In Phase 1, label value in binary for negative branch. Negative if reference to next statement. In Phase 2, Symbol Table pointer to label descriptive part entry. X'00008000' if reference to next statement.
MSG	1E,1F	Symbol Table pointer to alphanumeric message descriptive part entry.
NARG	2,3	Number of ASTX fields.
NOEL	6,7,8, A,B,C	Number of LLNO fields. Set by phase 2 for IDs 10,11.
ODLNK	11	PRF pointer to Begin Loop 3 entry for next outer loop.
ODP	7,8,9, 1B,21	EF pointer to variable or subscripted variable.
OPD1	5	EF pointer to left side variable or subscripted variable expression.
OPD2	5	EF pointer to right side expression.
PDLNK	11,12	Link to previous PRF Begin Loop 3 or End Loop entry.

PLAB	3	Pointer to created Label descritpive part entry if flow into Entry statement. Set to X'8000' if no flow.
PLIB	1F	Symbol Table pointer to PAUSE subroutine descriptive part entry.
PNAM	2,3	Symbol Table pointer to Entry Name descriptive part.
RIND	E	Return indicator.
RVAR	E	If RIND = 0, 0 for unindexed return, or constant index. If RIND ≠ 0, Expression File pointer to variable.
SLIB	1E	Symbol Table pointer to STOP subroutine descriptive part entry.
STNO	4	In Phase 1, label value in binary. In Phase 2, Symbol Table pointer to label descriptive part entry.
TTRA	B	In Phase 1, label value in binary for true branch. Negative if reference to next statement. In Phase 2, Symbol Table pointer to label descriptive part entry.
TVAL	A,B	EF pointer to text expression.
UNIT	14,15 17,18, 1D	Expression File pointer to unit designator descriptive part entry.
VAR	5	Symbol Table pointer to variable on left side.
VAR	D	Symbol Table pointer variable descriptive part entry.
VDP	5,D, 10,21	Link to previous PRF entry in which variable was defined.

Entry Formats

Begin Program (4 bytes)

ID = 1	Not Used	ILNK = X'8000'

Subprogram Entry (Variable Length)

ID = 2	NARG	ILNK
PNAM		X'8000'
LLNK = X'8000'		ASTX
:		:
ASTX		ASTX or Not Used
LINO		

Alternate Entry (Variable Length)

ID = 3	NARG	ILNK
PNAM		PLAB
LLNK		ASTX
:		:
ASTX		ASTX or Not Used
LINO		

Label Definition (12 bytes)

ID = 4	FLAGS	ILNK
STNO		
LLNK		ESLOC

Equation (10 bytes)

ID = 5	ABN	ILNK
OPD1		VDP
OPD2		CDP/VAR
LINO		

Unconditional GO TO (16 bytes)

ID = 6	NOEL = 1	ILNK
GLNK		Not Used
LLNO		
LINO		

Assigned-GO-TO-(Variable Length)

ID = 7	NOEL	ILNK
GLNK		OPD
LLNO		
:		:
LLNO		
LINO		

Computed GO TO (Variable Length)

ID = 8	NOEL	ILNK
GLNK		OPD
LLNO		
:		:
LLNO		
LINO		

ASSIGN (12 bytes)

ID = 9	Not Used	ILNK
OPD		ALAB
LINO		

Arithmetic IF (28 bytes)

ID = A	ABN	ILNK
GLNK		CDP
TVAL		Not Used
LTRA		
ETRA		
GTRA		
LINO		

Logical IF (20 bytes)

ID = B	ABN	ILNK
GLNK		CDP
TVAL		Not Used
TTRA		
LINO		

CALL (Variable Length)

ID = C	NOEL	ILNK
GLNK		CDP
CEX		Not Used
LLNO		
:		:
LLNO		
LINO		

Argument Definition Point (8 bytes)

ID = D	Not Used	ILNK
VAR		VDP

RETURN (12 bytes)

ID = E	RIND	ILNK
Not Used		RVAR
LINO		

Begin Loop 1 (8 bytes)

ID = F	Not Used	ILNK
LINO		

Begin Loop 2 (16 bytes)

ID = 10	Not Used	ILNK
IVAR		VDP
BEG		INC
END		EXITLB

Begin Loop 3 (16 bytes)

ID = 11	Not Used	ILNK	
EDLNK		PDLNK	
ODLNK		Flags	LEV
CDP		GLAB	

End Loop (24 bytes)

ID = 12	Flags	ILNK
BL3PT		PDLNK
IVAR		GLAB
BEG		INC
END		Not Used
BL1PT		BL2PT

CONTINUE (8 bytes)

ID = 13	Not Used	ILNK
LINO		

READ (24 bytes)

ID = 14	Flags	ILNK
LABF		UNIT
ERR		
EOF		
GLNK		Not Used
LINO		

READ with Namelist (24 bytes)

ID = 15	Not Used	ILNK
LABN		UNIT
ERR		
EOF		
GLNK		Not Used
LINO		

READ without Unit (12 bytes)

ID = 16	FLAGS	ILNK
LABF		Not Used
LINO		

WRITE (12 bytes)

ID = 17	FLAGS	LINK
LABF		UNIT
LINO		

WRITE with Namelist (12 bytes)

ID - 18	Not Used	ILNK
LABN		UNIT
LINO		

PRINT (12 bytes)

ID = 19	FLAGS	ILNK
LABF		Not Used
LINO		

PUNCH (12 bytes)

ID = 1A	FLAGS	ILNK
LABF		Not Used
LINO		

Output List Element (8 bytes)

ID = 1B	Not Used	ILNK
Not Used		OPD

End List (4 bytes)

ID = 1C	Not Used	ILNK

File Control (12 bytes)

ID = 1D	FNSW	ILNK
Not Used		UNIT
LINO		

STOP (12 bytes)

ID = 1E	CNT	ILNK
SLIB		MSG
LINO		

PAUSE (12 bytes)

ID = 1F	CNT	ILNK
PLIB		MSG
LINO		

End Program (4 bytes)

ID = 20	Not Used	ILNK

Input List Element (12 bytes)

ID = 21	Not Used	ILNK
OPD		VDP
CDP/VAR		Not Used

STORAGE SPECIFICATION TABLES

Common Variable Table Format

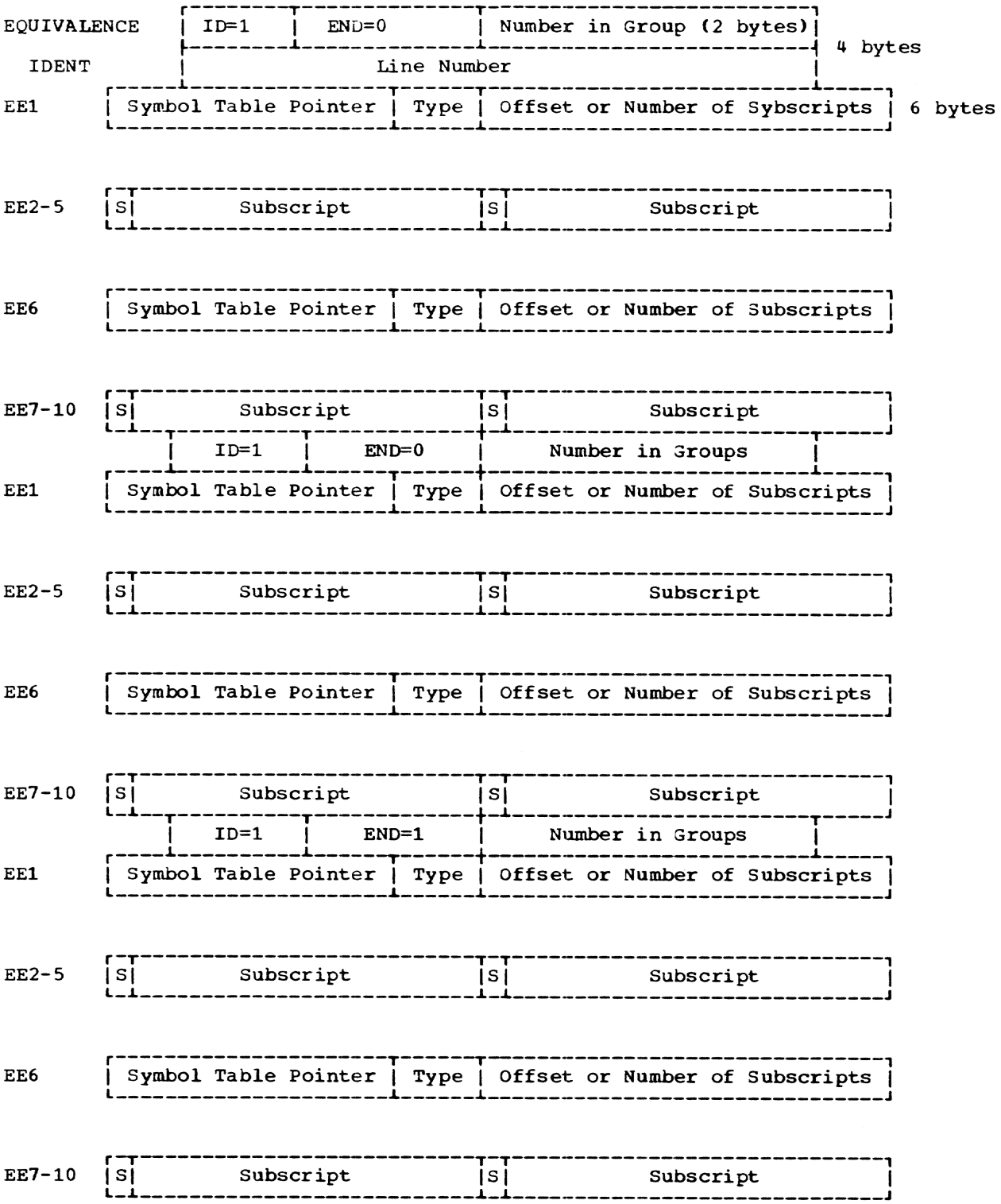
One entry per COMMON statement starts on a word boundary.

ID = 0	Not Used	
Line Number		
TERM*	Storage Class	Symbol Table Index
:		:
TERM*	Storage Class	Symbol Table Index

*Non-Zero denotes terminal entry.

EQUIVALENCE Entry

One entry per EQUIVALENCE statement. Entries start on a word boundary.



EQUIVALENCE Entry

Field Identifiers

Field	Explanation
ID	1 = EQUIVALENCE entry 0 = COMMON entry
END	1 = end of group 0 = not end of group
TYPE	Identifier type may be: 1, 2 Unknown (00) Integer *2 (1,2) Integer *4 (3,2) Automatic (0,7) Real *4 (3,3) Real *8 (7,3) Complex *8 (7,4) Complex *16 (F,4) Logical *4 (3,1) Logical *1 (0,1)
Number in Group	Number of variables in a specified EQUIVALENCE entry
Offset/Number of Subscripts	In the event that dimension information for a particular variable (from a DIMENSION, COMMON or TYPE statement) precedes the EQUIVALENCE statement, or that a subscripted variable in the EQUIVALENCE statement contains only a single subscript, the offset in EE1 or EE6 is computed. When dimension information does not precede the EQUIVALENCE statement and a subscripted variable in the EQUIVALENCE statement contains more than one subscript, EE1 or EE6 contains the number of subscripts. In this case EE2 or EE7 are required, and EE3-5 or EE8-10 may be required.
S	High-order bit: 1 = negative subscript 0 = positive subscript

PRESET DATA TABLES

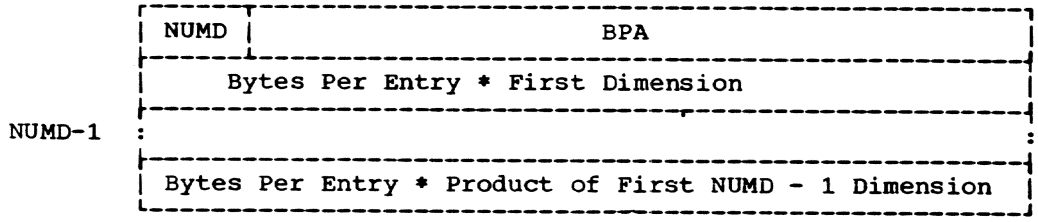
Field Identifiers

Field	Explanation
NUMB	Number of dimensions
BPA	Bytes per array
VC	Variable/constant switch
DM	Integer constant value if VC = 0 Variable,Symbol,Table pointer if VC = 1
Flag (Cross Reference List)	1 - symbol table pointer definition 2 - symbol table pointer reference 3 - label value definition 4 - label value reference

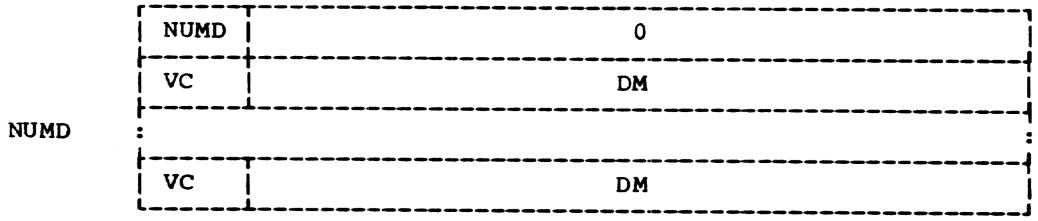
Entry Formats

Dimension Table

Declared array not a formal argument



Declared array a formal argument



Alphameric Table

Continuation Link	Not Used		
Line Number			
Descriptive Part Pointer		Size (Bytes)	
:			:

Namelist Table

Symbol Table Pointer of Variable$_1$	Symbol Table Pointer of Variable$_2$
:	:
Symbol Table Pointer of Variable$_{n-1}$	Symbol Table Pointer of Variable$_n$

Data Table

Continuation Link	Symbol Table Pointer	
Offset (3 Bytes)		Number of Values
Repetitions (3 Bytes)		Length
Constant Type	Number of Values	
Value		
:		:
Constant Type		
Value		

Cross Reference List

Line Number	
Flag	S.T.P. or L.V.

EXPRESSION FILE (ERF OR EF)

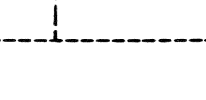
The expression file is formed of individual strings of entries, each with the following general format:

EFID	EF Flags	Content
Content (Continued)		

As described below, both the ID and the content may take one of two forms. The strings are the usual right-hand Polish notation.

EFID - FF = Null entry

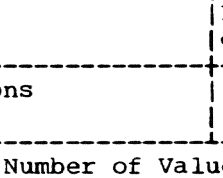
Form 1:



Code: 0 Variable
1 Constant
2 Function
3 Residue of removed expression
4 Operator, general
5 Operator, common or removed expression
6 Adcon

Type: 1 Logical (1)
2 Logical (4)
3 Integer (2)
4 Integer (4)
5 Real (4)
6 Real (8)
7 Complex (8)
8 Complex (16)
9 Literal

Form 2 (loop variables or parameters only):



IV Flag = Induction variable
Level Number = Loop level

EF Flags -- Both forms use EF Flags as follows (left to right, beginning with high-order bit):

X'80'	Sign indicator (EFSIGNF)
X'40'	Subscript indicator (EFSUBS)
X'20'	Last use flag
X'10'	Short form notation in I/O list
X'80'	Split recursive constant 0 (Phase 3 only)
X'40'	Global floating point register quantity

Content

Form 1 (variables, functions, and constants):

	QUANT
ADCON	DISPL

QUANT	Symbol Table (Descriptive Part Pointer) reference to variable or constant
ADCON DISPL	During Phases I and II this is the offset; during Phase III ADCON is made the Symbol Table reference to Adcon and DISPL is the immediate displacement

Note:

ADCON = 0 for subscripted variable
ADCON = FF for Adcon page reference

Form 2 (operators):

	OP	NOT USED
TRIAD	NAME	

OP Operator code:

Code (16)		Operator
00	+	add (- by negation)
01	*	Multiply
02	/	Divide
04	**	Exponentiate
05	,	Argument of closed function
06	,,	Argument of intrinsic function
07	;	Closed function
08	.EQ.	Equivalence (.NE. by negation)
09	.GT.	Greater than (.LE. by negation)
0A	.AND.	Logical AND (.NOT. by negation)
0B	;;	Intrinsic function
10	:	Subscript
19	.LT.	Less than (.GE. by negation)
1A	.OR.	Logical OR (.NOT. by negation)
1B	MAX	Maximum
11	!	recursive add -- Phases 3 and 4
12	?	index add -- Phases 3 and 4
13	@	dummy (to distinguish variables) -- Phase 3 only

TRIAD Expression file reference (Phase 3)

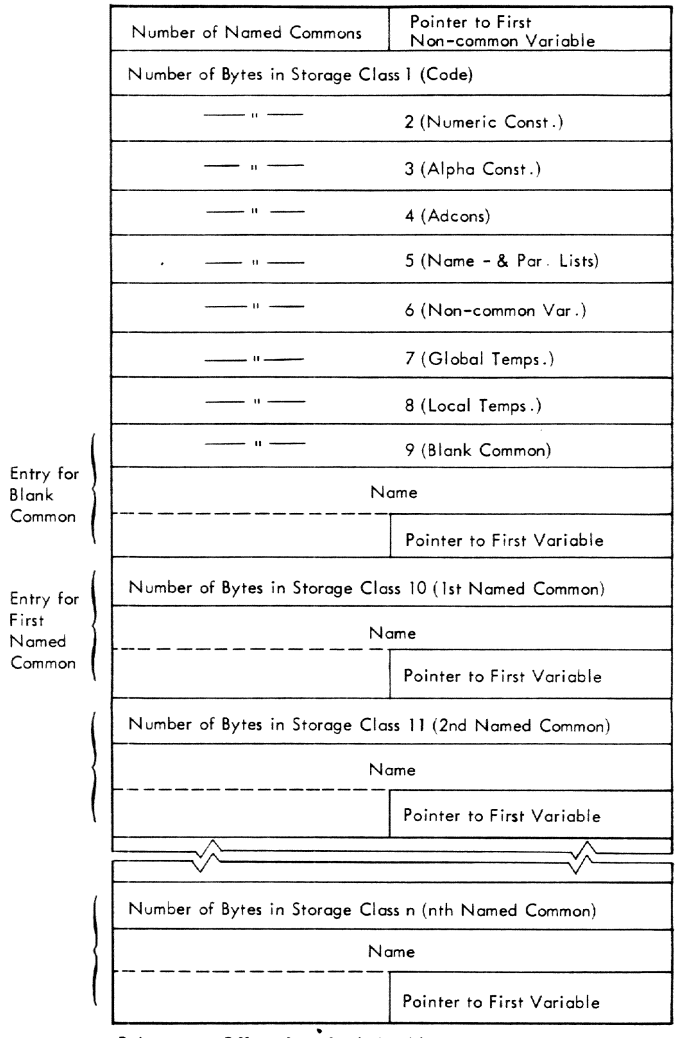
NAME For : operator, displacement (Phase 3)

For common or removed expression, identifying number (Phase 3 and 4)

Storage classes 128 to 253 correspond to formal arguments called by name. Storage class 254 is used for locations in the code covered by a special class of address constants, whose values are filled in by Phase 5.

Storage class 255 is used for estimated locations in the code and is converted to storage class 1 when the correct location is entered.

STORAGE CLASS TABLE (STCLTB)



Pointers are Offsets from Symbol Table Base and Indicate Origins of Each Chain of Variable Entries. n ≤ 127

PROGRAM FILE (PF) FORMATS OUTPUT BY PHASE 3

All the program representation file entries, except begin program (identification code 1), are modified during Phase III and put into the program file.

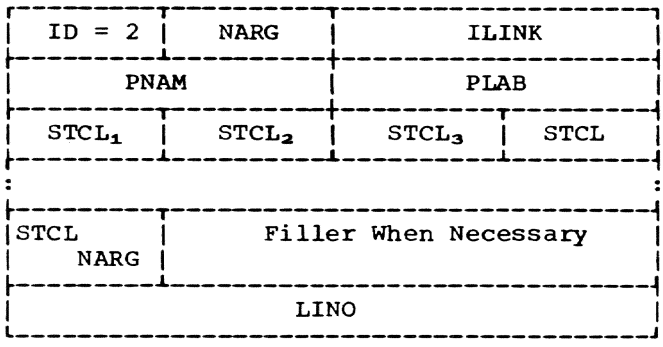
Field Identifiers

The fields have the same meaning as for the program representation file entries, with the addition of the following:

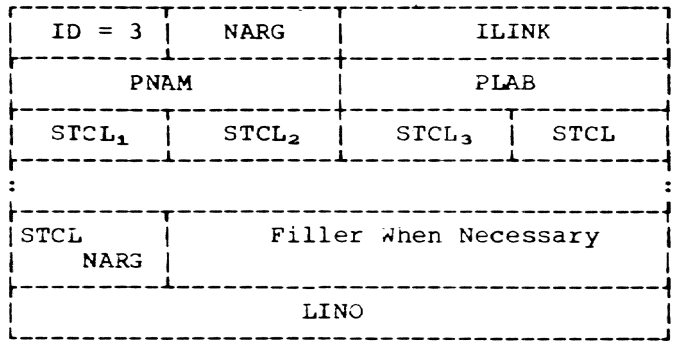
Field	IDs	Explanation
BEG	10	Link to Polish string, INITIAL VALUE
END	10	Link to Polish string, FINAL VALUE
EXITLB	10	Symbol Table pointer to label outside the loop, where the induction variable must be materialized. If loop has no exit, then EXITLB= X'8000'.
FLAGS	F,10,(Left to Right)	X'80' Labels in loop X'40' Unsafe loop X'20' Materialize X'10' Parameter X'08' Global Flag (inner loop, no external calls) X'04' BXLE on recursive X'02' ONEASN (globally assign FP register 6) X'01' I/O implied loop
GLBL 1 to GLBL8	F	Names of global expressions for this loop or Symbol Table reference, if Adcons. The names will have 7000_1 added to them to distinguish them from Symbol Table references.
GBLREAL	11	When the ONEASN flag is set, this field contains a PF pointer to the left side of an assignment statement which can be globally assigned to floating-point register 6 by Phase 4. When ONEASN flag is off, this field contains either a symbol table pointer to a simple real variable or constant which can be loaded into floating-point register 6 outside of the loop, or it contains X'8000'.
INC	10	Link to Polish string, increment value
IVAR	10	Link to Polish string. If materialize flag is 1, this is the induction variable; otherwise it is a test expression for the first variable in RMVAL chain.
IVARSVE	10	Contains the PF pointer for IVAR saved for materialization on exit loop.
LABL	4	Symbol Table reference
LLNK	4	Link to previous referenced label definition entry.
PNAM	2	Symbol Table entry of the ENTRY name. This field is '8000' if this is a main program.
RMVAL (Begin Loop 1)	F	Link to chain of removed expression in Polish, with the format: Insert R
RMVAL (Begin Loop 2)	10	Link to chain recursive removed expressions.
RMVAL (Begin Loop 3)	11	Link to chain of common expressions which would be removable but for the loop variable.
STCL	2	Storage Class Number of the respective argument.

Entry Formats

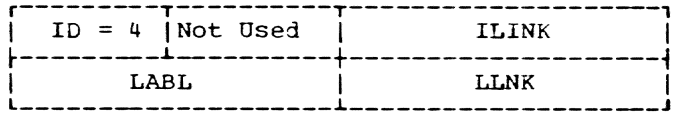
Subprogram Entry (Variable Length)



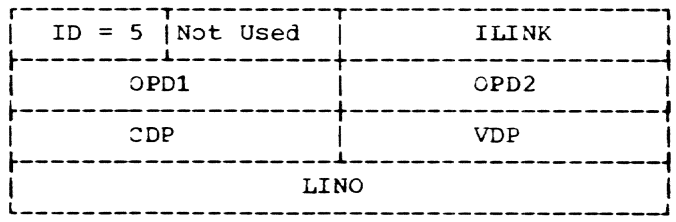
Alternate Entry (Variable Length)



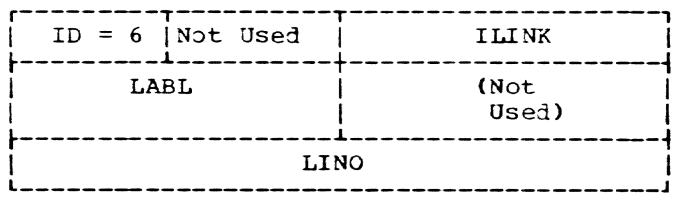
Label (8 bytes)



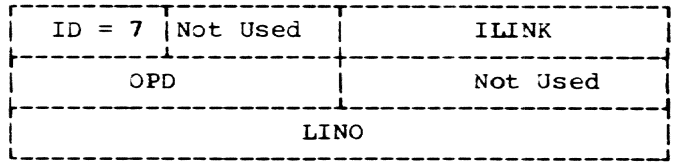
Equation (16 bytes)



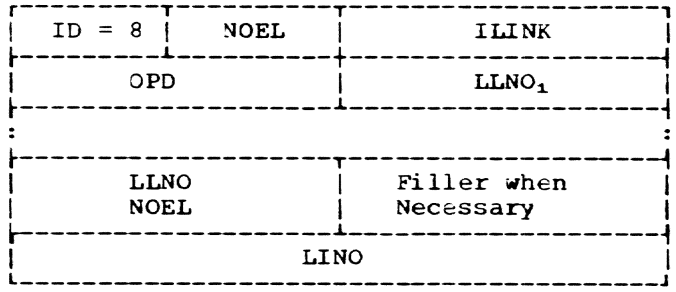
Unconditional GO TO (12 bytes)



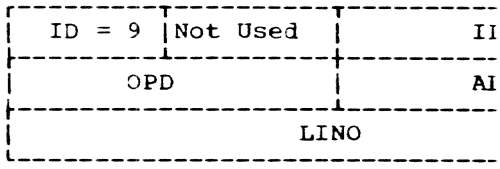
Assigned GO TO (12 bytes)



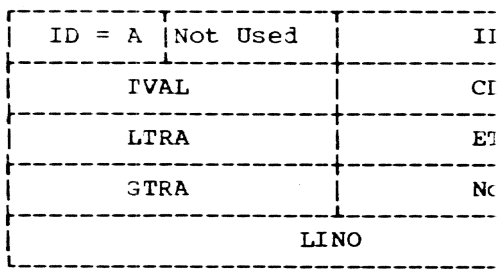
Computed GO TO (Variable Length)



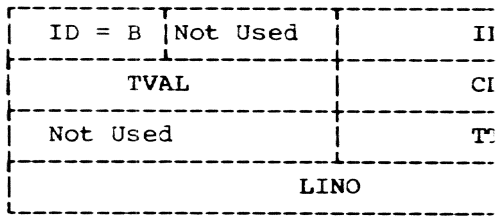
ASSIGN (12 bytes)



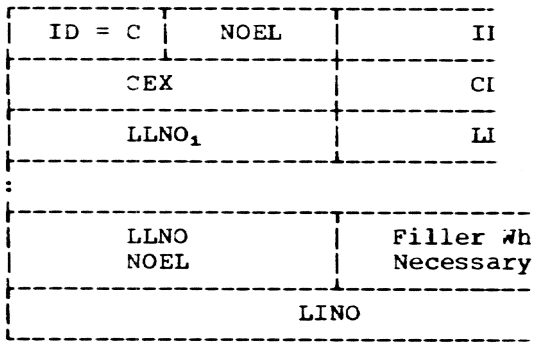
Arithmetic IF (20 bytes)



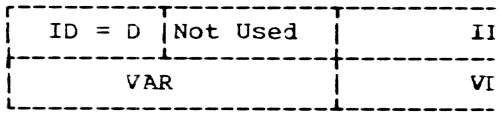
Logical IF (16 bytes)



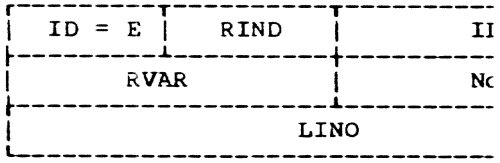
CALL (Variable Length)



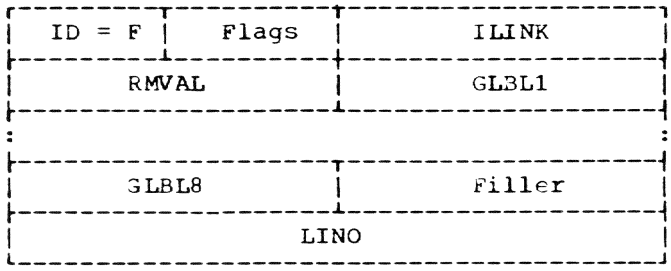
Argument Definition Point (8 byt



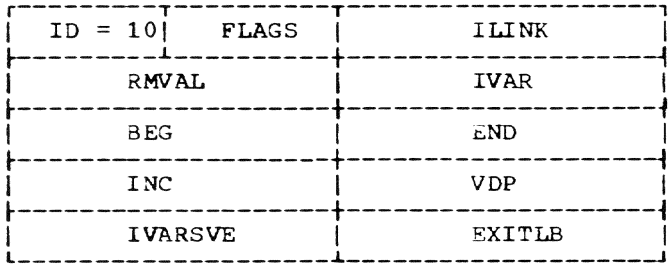
RETURN (12 bytes)



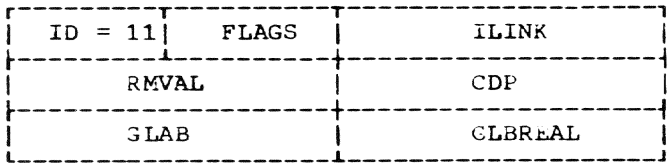
Begin Loop 1 (28 bytes)



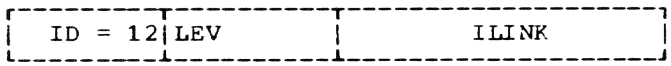
Begin Loop 2 (16 bytes)



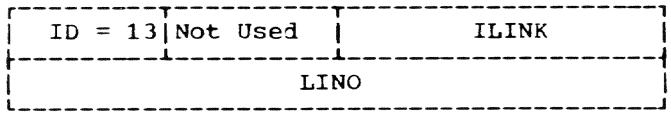
Begin Loop 3 (12 bytes)



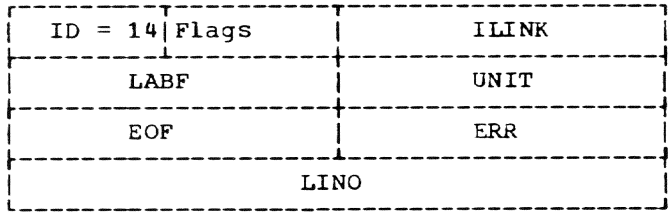
End Loop



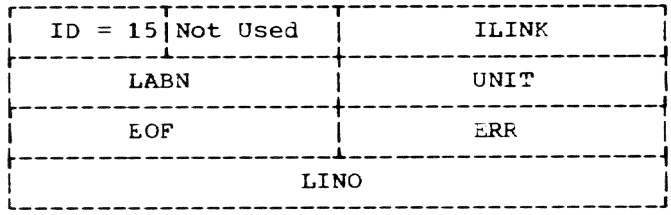
CONTINUE (8 bytes)



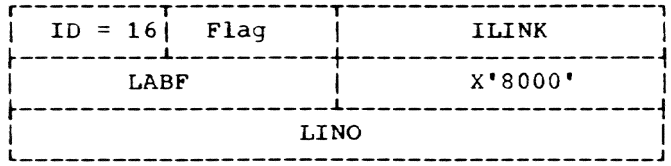
READ (16 bytes)



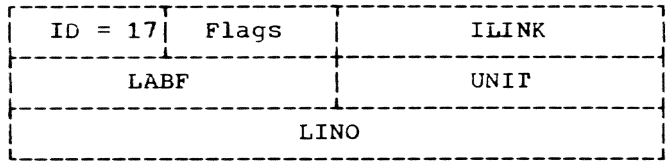
READ with Namelist (16 bytes)



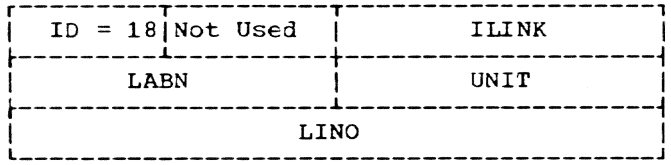
READ without Unit (12 bytes)



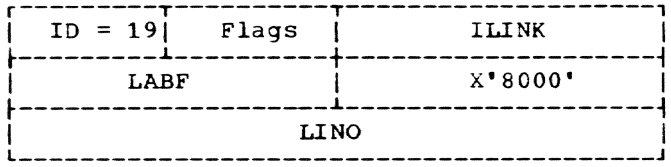
WRITE (12 bytes)



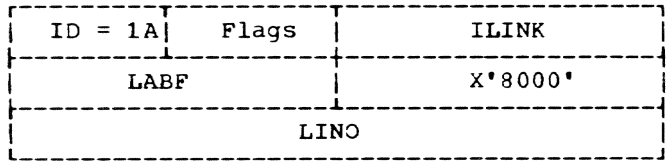
WRITE with Namelist (12 bytes)



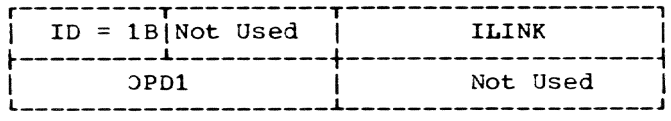
PRINT (12 bytes)



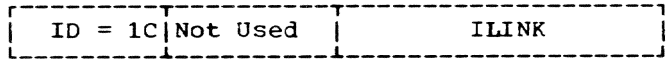
PUNCH (12 bytes)



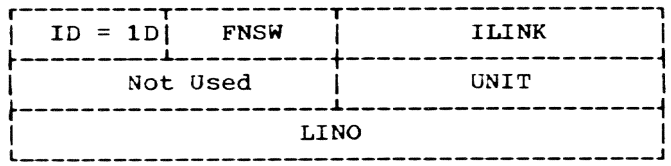
Output List Element (8 bytes)



End List (4 bytes)



File Control (12 bytes)



STOP (12 bytes)

ID = 1E	CNT	ILINK
SLIB		MSG
LINO		

PAUSE (12 bytes)

ID = 1F	CNT	ILINK
PLIB		MSG
LINO		

End Program (4 bytes)

ID = 20	Not Used	ILINK = 8000

Input List Element (12 bytes)

ID = 21	Not Used	ILINK
OPD		VDP
CDP		Not Used

CODE FILE FORMAT

Statement Header

01	0
Line Number	

Label Definition

02	0	Symbol T. Pointer

RR Instruction

OP	R1	R2	0

RX Instruction

OP	R1	X2	B2	D2
0	ST. Class		Symbol T. Pointer	

RS Instruction

OP	R1	R3	B2	D2
0	ST. Class		Symbol T. Pointer	*

*The second word - Descriptor - of an RS instruction is optional.

Label Reference
(Displacement Supplied by Phase 5)

FE	0		Symbol T. Pointer	[1]
OP	R1 M1	X2 R3	B2	0
0			Symbol T. Pointer	[2]

[1](ADCON Entry)
[2](LABEL Entry)

End of Code

FF	0

SYMBOL TABLE

The Name Part and Descriptive Part of Symbol Table entries are placed in the same storage area but are separated from each other; the two parts are therefore shown individually.

General Format

Name Part (at higher address portion of table)

Name	
Name(Cont'd.)	DPP
LINK	DMLST

Field	Description	Setting Phase
NAME	Identifier name in EBCDIC	1
LINK	Link to next identifier entry in chain, otherwise X'80--','END CHAIN'.	1
DPP	Descriptive part pointer.	1
DMLST	Dimension list pointer.	1

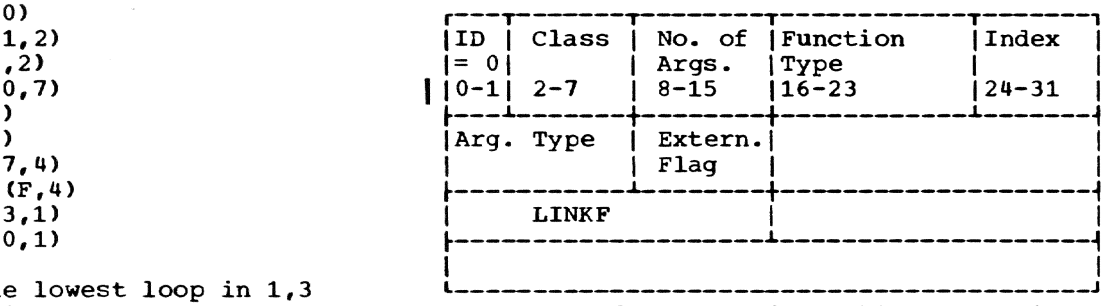
Descriptive Part (at lower address portion of table)

ID = 0 0-2	not used 2-3	Class 4-7	Flags 8-15	Type 16-23	ULEV 24-31
SLOC					STCL
LINKF			NUMENT		
FDP			LSTBDP		

Field	Setting Phase	Description
ID	Variable Name = 0	1
CLASS	Identifier class may be: 0 = Unknown 1 = Simple variable 2 = Array variable 3 = Statement function 4 = External subprogram reference 5 = Open (Intrinsic function reference) 6 = LIB (Library function reference) 7 = Namelist 8 = Label - primary and secondary subprogram entry except primary function name 9 = Statement function argument 10 = OPENA (Intrinsic Fn with automatic typing) 11 = LIBA (Library Fn with automatic typing) 12 = MAX (MAX MIN Function) 13 = Function Name of Function Subprogram 14 = Unknown Function	1
FLAGS	One-bit indicators which are: (Left to Right) X'80' Type Frozen X'40' Formal argument name X'20' Not Used X'10' Defined X'08' Active induction variable X'04' Common X'02' Equivalence or Interfering* X'01' Nonredefinable In Phase 1 X'02' indicates equivalenced variable. In Phase 2 X'02' indicates variables which may interfere with each other.	
TYPE	Identifier type may be: Unknown (0,0) Integer*2 (1,2) Integer*4(3,2) Automatic (0,7) Real*4 (3,3) Real*8 (7,3) Complex*8 (7,4) Complex*16 (F,4) Logical*4 (3,1) Logical*1 (0,1)	1,2
ULEV	Level of the lowest loop in which this is a loop variable or parameter.	1,3
SLOC	Storage location. Byte 1 used in Phase 1 for following flags: (Left to Right) Initial data Must not be dimensioned Function name Common block name Induction Var. in Namelist Byte 2 used in Phase 1 for: External Flag	2
STCL	Storage Class	2
LINKF	Link from descriptive part to the name part.	1
NUMENT	During Phase 1, is class if NAMELIST, the number of list elements. During Phase 2, the number of words of storage required for the array. During Phase 3, the forward compute point.	1,2,3
FDP	Forward Definition Point. During Phase 1, contains the latest PRF entry in which the variable was defined. During Phase 3 contains the current forward definition point.	1,3
LSTBDP	If class is Namelist Label, link to the chain of elements of the Namelist. During Phase 3, contains the backward definition point.	1,3

Specific Descriptive Part Formats of Intrinsic and Library Functions

1. LIB (Class 5) and LIBA (Class 11)



Index - Used by LIBA for table lookup in Phase 1.

2. OPEN (Class 5)
   Assemble in

ID = 0	Class	No. of Args.	Function Type	Function Number
0-1	2-7	8-15	16-23	24-31
Arg. Type				

The 'OPEN' class functions and their function numbers are listed in the following table. If the function does not have a name, a description is provided.

Function No.	Name	Description
1	FLOAT	
2	DFLOAT	
3	HFIX	
4	IFIX,INT	
5	DBLE	
6	SNGL	
7	IDINT	
8	REAL	
9	--	Convert L*1 to L*4
10	--	Convert L*4 to L*1
11	--	Convert L*2 to I*4
12	--	Convert I*2 to R*4
13	--	Convert I*2 to R*8
14	--	Convert I*2 to C*8
15	--	Convert I*2 to C*16
16	--	Convert I*4 to I*2
17	--	Convert I*4 to C*8
18	--	Convert I*4 to C*16
19	--	Convert R*4 to C*8
20	--	Convert R*4 to C*16
21	--	Convert R*8 to I*2
22	--	Convert R*8 to C*8
23	--	Convert R*8 to C*16
24	--	Convert C*8 to I*2
25	--	Convert C*8 to I*4
26	--	Convert C*8 to R*8
27	--	Convert C*8 to C*16
28	--	Convert C*16 to I*2
29	--	Convert C*16 to I*4
30	--	Convert C*16 to R*4
31	--	Convert C*16 to R*8
32	--	Convert C*16 to C*8
33	AMOD	
34	DMOD	
35	IABS	
36	DABS	
37	AINT	
38	ISIGN	
39	DSIGN	
40	IDIM	
41	AIMAG	
42	CMPLX	
43	DCMPLX	
44	DCONJG	
45	HMOD	MOD Function with Arg Type I*2, Fn. Type I*2
46	--	MOD Function with Arg Type I*4, Fn. Type I*4
47	--	ABS Function with Arg Type I*2, Fn. Type I*2
48	--	ABS Function with Arg Type R*4, Fn. Type R*4
49	HSIGN	SIGN Function with Arg Type I*2, Fn. Type I*2
50	--	SIGN Function with Arg Type R*4, Fn. Type R*4
51	HDIM	DIM Function with Arg Type I*2, Fn. Type I*2
52	--	DIM Function with Arg Type R*4, Fn. Type R*4
53	DDIM	DIM Function with Arg Type R*8, Fn. Type R*8
54	--	CONJG Function with Arg Type C*8, Fn. Type C*8

A list of special 'OPEN' class functions for exponentiation and their function numbers is given below:

Function Number	Function Name
72	ISQ
73	ICUBE
74	IFIFTH
75	ISEVEN
76	SQ
77	CUBE
78	FIFTH
79	SEVEN
80	RECIP

3. OPENA (Class 10).
Assemble in

ID = 0	Class	No. of Args.	Function Type	Index
0-2	2-7	8-15	16-23	24-31
Arg. Type				

Index is used for table lookup in Phase 1.

A list of "OPENA" class functions and their function numbers is given below:

Function Number	Function Name
55	MOD
56	ABS
57	SIGN
58	DIM
59	CONJG

4. MAX (Class 12).
Assemble in

ID = 0	Class	MIN Flag	Function Type	Function Number
0-2	2-7	8-15	16-23	24-31
Arg. Type				

MIN Flag
raised if function is from MIN family.

Function Number
is either zero or the number of conversion function needed.

A list of "MAX" class functions and their function numbers is given below:

Function Number	Function Name
60	AMAX0
61	AMAX1
62	MAX0
63	MAX1
64	DMAX1
65	AMIN0
66	AMIN1
67	MIN0
68	MIN1
69	DMIN1

Constant Format

Name Part (at higher-address portion of table)

Value
Value
Value
Value
LINK / DPP

Variable Length Maximum of 16 Bytes

Field	Description
VALUE	Binary value of the constant.
	Value of logical constants
	1 = true
	0 = false
LINK	Link to next Constant entry in chain, otherwise X'80--', 'END CHAIN'
DPP	Descriptive Part Pointer

Descriptive Part (at lower-address portion of table)

ID = 1	FLAGS	TYPE LENGTH, ATYPE	LINKF
0-1	2-7	8-15	16-31
SLOC			STCL

Field	Description
ID	Constant = 1
FLAGS	Available if needed

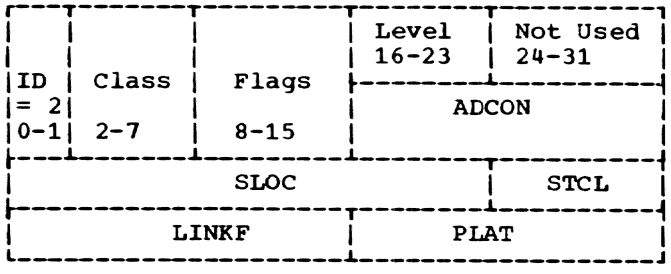
TYPE	Type of constant, which may be: Null (0,0) Logical*1 (0,1) Logical*4 (3,1) Integer*2 (1,2) Integer*4 (3,2) Real*4 (3,3) Real*8 (7,3) Complex*8 (7,4) Complex*16 (F,4)	
LINKF	Link to name part	
SLOC	Storage Location (offset with respect to Storage Class base)	
STCL	Storage Class	

Label Format

Name Part (at higher-address portion of table)

Field	Description	Setting Phase
LABEL	Binary value of the label	1,2
LINK	Link to next Label table entry in chain, otherwise X '80--', 'END CHAIN'	1,2
DPP	Descriptive part pointer	1

Descriptive Part (at lower-address portion of table)



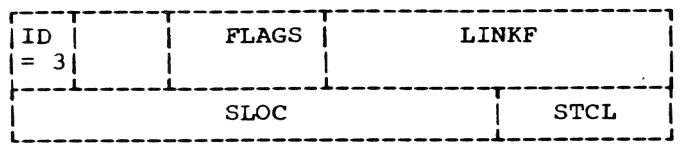
Field	Description	Setting Phase
ID	Label = 2	1,2
CLASS	Class of label, which may be: 0 = Unknown 1 = Source number 2 = Format number 3 = Compiler generated	1,2
FLAGS	One-bit indicators which are: (Left to Right) Not Used Referenced Defined	1,2
LEVEL	Loop level at which the label was defined	
ADCON	Reference to ADCON entry in Symbol Table Applicable to labels only	3
SLOC	During Phase 1, the storage location is assigned for Format Labels. During Phase 4, the storage location is assigned for statement labels when first referenced or defined	1,4
STCL	Storage class, set during Phase 1 for Format Labels or during Phase 4 for Statement Labels when first referenced or defined	1,4
LINKF	Link to name part	
PLAT	PRF entry of the Begin or End Loop item preceding the statement number	2

Address Constant Format

Name Part (at higher-address portion of table)

Field	Description
VALUE 1	If value 2 = 254, pointer to symbol table entry for entity to be addressed. If value 2 = 255, estimated location addressed, ELSE location addressed.
VALUE 2	Storage class of location or entity addressed.
LINK	Link to next entry in chain, else X'80--', 'END CHAIN'.
DPP	Descriptive part pointer

Descriptive Part (at lower-address portion of table)



Field	Description
ID	Address Constant = 3
FLAGS	One bit indicators (as yet unassigned)
LINKF	Link to name part
SLOC	Location (on ADCON page)
STCL	Storage class
	4 = shared address constants
	5 = unshared address constants

INTERCOM TABLE

The area called "Intercom" (Interphase Communication) is the most widely used interphase file in the compiler. This area is 512 storage locations in length and contains information used by all modules in the compiler. The structure and contents of Intercom are shown in Figures 40 and 41. Figure 40 is a DSECT listing for the area, giving the Intercom items in increasing storage location order. Figure 41 is a listing of all items in Intercom in order of increasing alphameric labels. In Figure 41 the relative displacement of each item is given following the item description.

A 512-byte area for Intercom is reserved in the Phase Controller's PSECT and in the PSECT for each of the compiler phases. Intercom is initialized by the Phase Controller for each compilation. When the Phase Controller calls a phase, the location of Intercom in the Phase Controller's PSECT is passed to the phase. Each phase copies the 512 bytes into its own PSECT, updates the area (in its own PSECT) as required during processing, and copies the 512-byte area back into the Phase Controller's PSECT before returning to the Phase Controller.

Many phase modules call Exec modules during their processing. All such calls provide the Exec module called with the location of Intercom in the phase. The Exec modules change the phase's copy of Intercom as necessary, thus insuring that changes make their way into the copy of Intercom passed to later phases.

```
LOCATN  OBJECT CODE   ADDR1 ADDR2 STMNT   SOURCE STATEMENT
                                  0433*CEKTEX   DS    0D EXEC INTERCOM
   CC0                            0434****                                  OBJECT PROGRAM NAMES
                                  0435*TEMCDN   DS    8C MODULE NAME
   C00                            0436*TEMEP    DS    8C MAIN ENTRY POINT
   008                            0437*TEDKID   DS    4C DECK IDENTIFICATION
   C10                            0438*TEVID    DS    8C VERSION IDENTIFICATION
   014                            0439****
                                  0440***                                   ENTRY POINTS
                                                                            PHASE CONTROLLER ENTRY     &

   01C                            0441*TEGNS2   EQU   *
                                  0442*TEVGNS   ADCON IMPLICIT GNSS
   01C                            0443*         DS    OF
   010                            0444*TEVGNS   EQU   *-12
   01C                            0445*         DS    A(0)
   020                            0446*         DS    A(0)
                                  0447*TEVRDM   ADCON IMPLICIT RDM
   024                            0448*         DS    OF
   018                            0449*TEVRDM   EQU   *-12
   024                            0450*         DS    A(0)
   028                            0451*         DS    A(0)
                                  0452*TEVI2    ADCON IMPLICIT CONI2
   02C                            0453*         DS    OF
   020                            0454*TEVI2    EQU   *-12
   02C                            0455*         DS    A(0)
   030                            0456*         DS    A(0)
                                  0457*TEVI4    ADCON IMPLICIT CONI4
   034                            0458*         DS    OF
   028                            0459*TEVI4    EQU   *-12
   034                            0460*         DS    A(0)
   038                            0461*         DS    A(0)
                                  0462*TEVR4    ADCON IMPLICIT CONR4
   03C                            0463*         DS    OF
   030                            0464*TEVR4    EQU   *-12
   03C                            0465*         DS    A(0)
   040                            0466*         DS    A(0)
                                  0467*TEVR8    ADCON IMPLICIT CONR8
   044                            0468*         DS    OF
   038                            0469*TEVR8    EQU   *-12
   044                            0470*         DS    A(0)
   048                            0471*         DS    A(0)
                                  0472*TEVC8    ADCON IMPLICIT CONC8
   04C                            0473*         DS    OF
   040                            0474*TEVC8    EQU   *-12
   04C                            0475*         DS    A(0)
   050                            0476*         DS    A(0)
                                  0477*TEVC16   ADCON IMPLICIT CONC16
   054                            0478*         DS    OF
   048                            0479*TEVC16   EQU   *-12
   054                            0480*         DS    A(0)
   058                            0481*         DS    A(0)
                                  0482*TEVFL4   ADCON IMPLICIT FLAD4
   05C                            0483*         DS    OF
   050                            0484*TEVFL4   EQU   *-12
   05C                            0485*         DS    A(0)
   060                            0486*         DS    A(0)
                                  0487*TEVFL5   ADCON IMPLICIT FLAD5
   064                            0488*         DS    OF
   058                            0489*TEVFL5   EQU   *-12
   064                            0490*         DS    A(0)
   068                            0491*         DS    A(0)
```

Figure 40. CEKTD, Compiler Exec Process Terminal Modifications (Part 1 of 4)

```
LOCATN  OBJECT CODE   ADDR1 ADDR2 STMNT   SOURCE STATEMENT

                                  0492*TEVVR    ADCON IMPLICIT FLADVR
 06C                              0493*         DS    0F
 06C                              0494*TEVVR    EQU   *-12
 06C                              0495*         DS    A(0)
 070                              0496*         DS    A(0)
                                  0497*TEVFLL   ADCON IMPLICIT FLL
 074                              0498*         DS    0F
 068                              0499*TEVFLL   EQU   *-12
 074                              0500*         DS    A(0)
 078                              0501*         DS    A(0)
                                  0502*TEVCRL   ADCON IMPLICIT CRL
 07C                              0503*         DS    0F
 07C                              0504*TEVCRL   EQU   *-12
 07C                              0505*         DS    A(0)
 080                              0506*         DS    A(0)
                                  0507*TEVCLR   ADCON IMPLICIT CLR
 084                              0508*         DS    0F
 078                              0509*TEVCLR   EQU   *-12
 084                              0510*         DS    A(0)
 088                              0511*         DS    A(0)
                                  0512****                                  SOURCE LINE INFORMATION
 08C                              0513*TESLNO   DS    PL4 SOURCE LINE NO.
 090                              0514*TESTNO   DS    6C SOURCE STATEMENT NO.
 096                              0515*TEFORG   DS    X FORGET FLAG
 097                              0516*TECXR    DS    X CONVERSATION/BATCH SWITCH
 098                              0517*TEVSTR   DS    F SOURCE STATEMENT TEXT BASE
 09C                              0518*TEEND    DS    X END STATEMENT FLAG
 09D                              0519*TEP3CH   DS    C
 09E                              0520*TEP4CH   DS    C
 09F                              0521*TEP5CH   DS    C
                                  0522****                                  TABLES BASES,TOPS,ANCHORS
 0A0                              0523*TEVSYM   DS    V SYMBOL TABLE BASE
 0A4                              0524*TENAMT   DS    H SYMBOL TABLE NAME PART TOP
 0A6                              0525*TEDEST   DS    H SYMBOL TABLE DESC.TOP
 0A8                              0526*TEVHTB   DS    A VARIABLE HASH TABLE BASE
 0AC                              0527*TELHTB   DS    A LABEL HASH TABLE BASE
 0B0                              0528*TECHTB   DS    A CONSTANT HEADER TABLE BASE
 0B4                              0529*TESCTB   DS    A STORAGE CLASS TABLE BASE
 0B8                              0530*TEITTB   DS    A IMPLICIT TYPE TABLE BASE
 0BC                              0531*TEPSEB   DS    A EXEC'S PSECT BASE (FOR
 0C0                              0532*TESTEA   DS    H ANCHOR FOR SYMBOL TABLE          &
                                                      ENTRY CHAIN
 0C2                              0533*TEXRAN   DS    H ANCHOR FOR SYMBOL TABLE          &
                                                      XREF CHAIN
                                  0534****                                  FILE AND LIST BASES, TOPS &
                                                                            AND ANCHORS
 0C4                              0535*TESPLB   DS    V STORAGE SPECIFICATION LIST       &
                                                      BASE
 0C8                              0536*TESPLT   DS    V STORAGE SPECIFICATION LIST       &
                                                      TOP
 0CC                              0537*TESPLU   DS    F STORAGE SPECIFICATION LIST       &
                                                      UPPER LIMIT
 0D0                              0538*TEPRFB   DS    V PRF BASE
 0D4                              0539*TEPRFT   DS    H PRF TOP
 0D6                              0540*TEKEYT   DS    H VISAM 'PUT' KEY FOR CEKVU MACRO
 0D8                              0541*TEEFB    DS    V EF BASE
```

Figure 40. CEKTD, Compiler Exec Process Terminal Modifications (Part 2 of 4)

```
LOCATN  OBJECT CODE   ADDR1 ADDR2 STMNT   SOURCE STATEMENT

  0DC                          0542*TEEFT    DS    H EF TOP
  0E0                          0543*TEPFB    DS    V PF BASE
  0E4                          0544*TEPFT    DS    H PF TOP
  0E6                          0545*TEPFU    DS    H PF UPPER LIMIT
  0E8                          0546*TECFB    DS    V CODE FILE BASE
  0EC                          0547*TECFT    DS    F CODE FILE TOP
  0F0                          0548*TECFU    DS    V CODE FILE UPPER LIMIT
  0F4                          0549*TEPSB    DS    V PRESET DATA BASE
  0F8                          0550*TEPST    DS    A PRESET DATA TOP
  0FC                          0551*TECRLB   DS    V CROSS REFERENCE LIST BASE
  10C                          0552*TECRLT   DS    F CROSS REFERENCE LIST TOP
  104                          0553*TEFAAB   DS    V FORMAL ARGUMENT ADCON LIST                    &
                                                     BASE
  108                          0554*TEFAAT   DS    F FORMAL ARGUMENT ADCON LIST                    &
                                                     TOP
  10C                          0555*TEPMDB   DS    V PMD BASE
  110                          0556*TEPMDT   DS    F PMD TOP
  114                          0557*TECPMB   DS    V CPM BASE
  118                          0558*TECPMT   DS    F CPM TOP
  11C                          0559*TERBAS   DS    A EXTERNAL NAMELIST BASE
  120                          0560*TEISDU   DS    F ISD U.L. FOR PHASE 5
  124                          0561*TEISDB   DS    V ISD BASE
  128                          0562*TEISDT   DS    F ISD TOP
  12C                          0563*TEGLAN   DS    H GLNK CHAIN ANCHOR
  12E                          0564*TELLAN   DS    H LLNK CHAIN ANCHOR
  130                          0565*TEPDAN   DS    H PDLNK CHAIN ANCHOR
  132                          0566*TECPAN   DS    H CPD CHAIN ANCHOR
  134                          0567*TEADAN   DS    H ADP CHAIN ANCHOR
  136                          0568*TEDAAN   DS    H DATA CHAIN ANCHOR
  138                          0569*TEALFA   DS    H ALPHANUMERIC CHAIN ANCHOR
  13A                          0570*TESTAN   DS    H SYMBOL TABLE NAMELIST                         &
                                                     CHAIN ANCHOR
                               0571****                                  FLAGS AND OPTIONS
  13C                          0572*TEMEC    DS    X MAX ERROR CODE
  13D                          0573*TECPUT   DS    X LIST DATA SET EXISTS FLAG
  13E                          0574*TESLO    DS    X SOURCE LISTING OPTION
  13F                          0575*TEMMC    DS    X MEMORY MAP OPTION
  13F                          0576*TEMNO    EQU   TEMMO
  140                          0577*TECCLO   DS    X OBJECT CODE LISTING OPTION
  141                          0578*TESTEC   DS    X SYMBOL TABLE EDIT OPTION
  142                          0579*TECRLO   DS    X CROSS-REFERENCE LISTING                       &
                                                     OPTION
  143                          0580*TEISDO   DS    X ISD OPTION
  144                          0581*TEPTYP   DS    X PROGRAM TYPE
  145                          0582*TENEP    DS    X NO. OF ENTRY POINTS
  146                          0583*TENAR    DS    X NO. OF ALTERNATE RETURNS
  147                          0584*TEBCD    DS    X EBCDIC/BCD INDICATOR
  148                          0585*TENFA    DS    X NEXT FORMAL-ARG-BY-NAME                       &
                                                     NO.
                               0586****                                  MISCELLANEOUS
  14A                          0587*TEIFLP   DS    H SYMBOL TABLE POINTER TO                       &
                                                     INTRINSIC FUNCTION LIST
                               0588*TEFEU    ADCON IMPLICIT
  14C                          0589*         DS    0F
  140                          0590*TEFEU    EQU   *-12
  14C                          0591*         DS    A(0)
```

Figure 40. CEKTD, Compiler Exec Process Terminal Modifications (Part 3 of 4)

```
LOCATN  OBJECT CODE   ADDR1 ADDR2 STMNT   SOURCE STATEMENT

  150                          0592*          DS    A(0)
  154                          0593*TETRUE    DS    H SYMBOL TABLE POINTER TO                                  &
                                                      .TRUE.
  156                          0594*TEFALS    DS    H SYMBOL TABLE POINTER TO                                  &
                                                      .FALSE.
  158                          0595*TERSAV    DS    16F REGISTERS SAVE AREA                                    &
                                                        WHEN SUSPECTED SYSERR
  198                          0596*TEPAGE    DS    F CURRENT OUTPUT PAGE NO.
  19C                          0597*TEDATE    DS    CL8 DATE--MM/DD/YY
  1A4                          0598*TECFLC    DS    F O'FLOW FLAG(EXEC ONLY)
  1A8                          0599*TENCSX    DS    F NO.OF COMMON SUB-EXP.NAMES
                               0600**
  1AC                          0601*TEFVAL    DS    H POINTER TO FUNCTION VALUE
  1AE                          0602*TEP1DB    DS    C
  1AF                          0603*TEP2DB    DS    C
                               0604****                                          CONSTANT FILING AREA
  1B0                          0605*TECCNS    DS    0D
  1B0                          0606*TECNS1    DS    F LENGTH 4 CONSTANTS
  1B4                          0607*TECNS2    DS    F LENGTH 8 AND 16
  1B8                          0608*TECNS3    DS    F LENGTH 16
  1BC                          0609*TECNS4    DS    F LENGTH 16
  1C0                          0610*TEPNTR    DS    F SYMBOL TABLE POINTER
  1C4                          0611*TEGNU     DS    X CONSTANT ALREADY FILED FLG
  1C8                          0612*TELINO    DS    F BINARY LINE NO. FROM CRL
  1CC                          0613*TEWAAH    DS    A ADDR OF PRF 1ST WORD
  1D0                          0614*TEDESB    DS    A ADDR OF DESC.PART 1ST WORD
  1D4                          0615*TENAMB    DS    A ADDR OF NAME PART 1ST WORD
  1D8                          0616*TEDCB     DS    F DATA SET DCB
  1DC                          0617*TECXBA    DS    F
  1E0                          0618*TEICIN    DS    H CHCIA1
  1E2                          0619*TEICTR    DS    H CHCIE1
  1E4                          0620*TEICEN    DS    H CHCIH1
  1E6                          0621*TEINTR    DS    H INTERRUPT ROUTINE PNTR
                               0622*TEVGAT    ADCON IMPLICIT
  1E8                          0623*          DS    0F
  1DC                          0624*TEVGAT    EQU   *-12
  1E8                          0625*          DS    A(0)
  1EC                          0626*          DS    A(0)
  1F0                          0627*TESCB     DC    A(TEGNS2-TEVGNS)
  1F4                          0628*TEIOSM    DS    C
  1F5                          0629*TEPUPR    DS    C
  1F6                          0630*TEDUMP    DS    H
  1F8                          0631*TEPDMP    DS    H
```

Figure 40. CEKTD, Compiler Exec Process Terminal Modifications (Part 4 of 4)

```
CEKTEX CS OD     EXEC INTERCOM BASE                                          000
TEADAN DS H      ADP CHAIN ANCHOR                                            134
TEALFA DS H      ALPHANUMERIC CHAIN ANCHOR                                   138
TEBCD DS X      EBCDIC/BCD INDICATOR                                         147
TECFB DS V      CODE FILE BASE                                               0E8
TECFT DS F      CODE FILE TOP                                                0EC
TECFU DS V      CODE FILE UPPER LIMIT                                        0F0
TECHTB DS A      CONSTANT HEADER TABLE BASE                                  080
TECNS1 DS F      LENGTH 4 CONSTANTS                                          180
TECNS2 DS F      LENGTH 8 AND 16                                             184
TECNS3 DS F      LENGTH 16                                                   188
TECNS4 DS F      LENGTH 16                                                   18C
TECONS CS OD                                                                 180
TECPAN DS H      CPD CHAIN ANCHOR                                            132
TECRLB DS V      CROSS REFERENCE LIST BASE                                   0FC
TECRLO DS X      CROSS-REFERENCE LISTING OPTION                        X     142
TECRLT DS F      CROSS REFERENCE LIST TOP                                    100
TECXB DS X      CONVERSATION/BATCH SWITCH                                    097
TECXBA   DS     F                          ADDR IN LPC OF BATCH/CONV.IND     1DC
TEDAAN DS H      DATA CHAIN ANCHOR                                           136
TEDATE DS CL8      DATE--MM/DD/YY                                            19C
TEDCB    DS     F                          DCB ADDR FROM LPC                 1D8
TEDESB DS A      ADDR OF DESC.PART 1ST WORD                                  1DC
TEDEST DS H      SYMBOL TABLE DESC.TOP                                       0A6
TEDIAG   DS     C                          IF Y, ALLOW MAINT OUTPUT          1C5
TEDKID DS 4C      DECK IDENTIFICATION                                        01C
TEDUMP   DS     XL2                                                         1F6
TEEFB DS V      EF BASE                                                      0D8
TEEFT DS H      EF TOP                                                       0DC
TEEND DS X      END STATEMENT FLAG                                           0EC
TEFAAB DS V      FORMAL ARGUMENT ADCON LIST BASE                       X     104
TEFAAT DS F      FORMAL ARGUMENT ADCON LIST TOP                        X     108
TEFALS DS H      SYMBOL TABLE POINTER TO .FALSE.                       X     156
TEFEU    DS     2A                         CEKTG MACRO USES                  140
TEFORG DS X      FORGET FLAG                                                 069
TEFVAL DS H      POINTER TO FUNCTION VALUE                                   1AC
TEGLAN DS H      GLNK CHAIN ANCHOR                                           12C
TEGNS2   EQU    *                                                            01C
TEGNU DS X      CONSTANT ALREADY FILED FLG                                   1C4
TEIFLP DS H      SYMBOL TABLE POINTER TO INTRINSIC FUNCTION LIST       X     144
TEINTR   DS     H                          INTERRUPT PROG PNTR               1E6
TEIOEN DS H      CHCIH1                                                      1E4
TEIOIN DS H      CHCIA1                                                      1E0
TEIOSM   DS     C                                                           1E4
TEIOTR DS H      CHCIE1                                                      1E2
TEISDB DS V      ISD BASE                                                    124
TEISDO DS X      ISD OPTION                                                  143
TEISDT DS F      ISD TOP                                                     128
TEISDU DS F      ISD U.L. FOR PHASE 5                                        120
TEIITB   EQU    TEITTB                                                       0B8
TEITTB DS A      IMPLICIT TYPE TABLE BASE
TEKEYT   DS     H                          VISAM 'PUT' KEY FOR CEKVU MACRO   0D6
TELHTB DS A      LABEL HASH TABLE BASE                                       0AC
TELINO DS F      BINARY LINE NO. FROM CRL                                    1C8
TELLAN DS H      LLNK CHAIN ANCHOR                                           12E
TEMEC DS X      MAX ERROR CODE                                               13C
TEMEP DS 8C      MAIN ENTRY POINT                                            008
TEMMO DS X      MEMORY MAP OPTION                                            13F
TEMNO EQU TEMMO                                                              13F
TEMODN DS 8C      MODULE NAME                                                000
```

Figure 41. Alphabetically Sorted Listing of Intercom Items, With Displacements (Part 1 of 2)

```
TENAMB DS A     ADDR OF NAME PART 1ST WORD                          1D4
TENAMT DS H     SYMBOL TABLE NAME PART TOP                          0A4
TENAR DS X     NO. OF ALTERNATE RETURNS                             146
TENCSX DS F     NO.OF COMMON SUB-EXP.NAMES                          1A8
TENEP DS X     NO. OF ENTRY POINTS                                  145
TENFA DS X     NEXT FORMAL-ARG-BY-NAME  NO.                    X    148
TEOCLO DS X     OBJECT CODE LISTING OPTION                          140
TEOFLO DS F     O'FLOW FLAG(EXEC ONLY)                              1A4
TEOPMB DS V     OPM BASE                                            114
TEOPMT DS F     OPM TOP                                             118
TEOPUT DS X     LIST DATA SET EXISTS FLAG                           130
TEPAGE DS F     CURRENT OUTPUT PAGE NO.                             198
TEPDAN DS H     PDLNK CHAIN ANCHOR                                  130
TEPDMP   DS     XL2                                                 1F8
TEPFB DS V     PF BASE                                             0E0
TEPFT DS H     PF TOP                                              0E4
TEPFU DS H     PF UPPER LIMIT                                      0E6
TEPMDB DS V     PMD BASE                                            10C
TEPMDT DS F     PMD TOP                                             110
TEPNTR DS F     SYMBOL TABLE POINTER                                1C0
TEPRFB DS V     PRF BASE                                            0D0
TEPRFT DS H     PRF TOP                                             0D4
TEPSB DS V     PRESET DATA BASE                                    0F4
TEPSEB DS A     EXEC'S PSECT BASE                                   0BC
TEPST DS A     PRESET DATA TOP                                     0F8
TEPTYP DS X     PROGRAM TYPE                                        144
TEPUPR   DS     C                                                   1F5
TEP1DB DS C                          PHASE 1--5 DIAGNOSTIC BYTES    1AE
TEP2DB DS C                                                         1AF
TEP3DB DS C                                                         09D
TEP4DB DS C                                                         09E
TEP5DB DS C                                                         09F
TERBAS DS A     EXTERNAL NAMELIST BASE                              11C
TERSAV DS 16F     REGISTERS SAVE AREA WHEN SUSPECTED SYSERR      X    158
TESCTB DS A     STORAGE CLASS TABLE BASE                            0B4
TESLNO DS PL4     SOURCE LINE NO.                                   090
TESLO DS X     SOURCE LISTING OPTION                                13E
TESOB    DC     A(TEGNS2-TEVGNS)                                    1F0
TESPLB DS V     STORAGE SPECIFICATION LIST BASE                 X   0C4
TESPLT DS V     STORAGE SPECIFICATION LIST TOP                  X   0C8
TESPLU DS F     STORAGE SPECIFICATION LIST UPPER LIMIT          X   0CC
TESTAN DS H     SYMBOL TABLE NAMELIST CHAIN ANCHOR              X   13A
TESTEA DS H     ANCHOR FOR SYMBOL TABLE ENTRY CHAIN             X   0C0
TESTEO DS X     SYMBOL TABLE EDIT OPTION                            141
TESTNO DS 6C      SOURCE STATEMENT NO.                              090
TETRUE DS H     SYMBOL TABLE POINTER TO .TRUE.                  X   154
TEVCRL   CEKTX                       EXEC MODULES ACCON PAIRS       070
TEVC16   CEKTX                                                      048
TEVC8    CEKTX                                                      040
TEVFLL   CEKTX                                                      068
TEVFL4   CEKTX                                                      050
TEVFL5   CEKTX                                                      058
TEVGAT   CEKTX                                                      1DC
TEVGNS   CEKTX                                                      010
TEVHTB DS A     VARIABLE HASH TABLE BASE                            0A8
TEVID DS 8C     VERSION IDENTIFICATION                              014
TEVI2    CEKTX                                                      020
TEVI4    CEKTX                                                      028
TEVOLR   CEKTX                                                      078
TEVRDM   CEKTX                                                      018
TEVR4    CEKTX                                                      030
TEVR8    CEKTX                                                      038
TEVSTB DS F     SOURCE STATEMENT TEXT BASE                          098
TEVSYM DS V     SYMBOL TABLE BASE                                   0A0
TEVVR    CEKTX                                                      060
TEWAAH DS A     ADDR OF PRF 1ST WORD                                1CC
TEXRAN DS H     ANCHOR FOR SYMBOL TABLE XREF CHAIN              X   0C2
```

Figure 41. Alphabetically Sorted Listing of Intercom Items, With Displacements (Part 2 of 2)

INTRODUCTION

Linkage conventions are those conventions which govern communication among programs. Basically, there are two types of communication: that defined as standard linkage, and that defined as restricted linkage.

Restricted linkage provides highly efficient communication among programs which satisfy a definitive set of requirements. Such linkage is intended for use within a "black box" (precisely defined by the requirements given below) which has little or no interface with the rest of the system (or user), while standard linkage is the vehicle for all other communication.

In TSS/360 all interfaces among CSECTs which reside in virtual memory (execute with dynamic relocation turned on), whether or not the CSECTs are in the same assembly, conform to either standard or restricted linkage. Furthermore, all linkage within a CSECT conforms to one of these conventions. No other mechanism for communication is recognized. It is emphasized, however, that the restricted linkage never has to be used. A standard linkage is always acceptable and it is expected that the majority of program linkage will follow the standards.

The following paragraphs describe the linkage conventions to which all compiler modules must (and do) adhere. Reference is made in this description, and elsewhere in this PLM, to "Type I linkages." Type I linkage is one of several types defined for use in compiler and other system modules. For a complete description of all linkage types see the Systems Programmer's Guide.

For a precise description of the macros used for standard linkages (the CALL, SAVE, and RETURN macros) see Assembler Users Macro Instructions. Macro support for restricted linkages (INVOKE, STORE, and RESUME) is described later in this appendix.

CONVENTIONS FOR TYPE I LINKAGES (STANDARD)

Register Conventions

General Register	Usage	Mnemonic
15,0	Supervisor Parameter Register	SP
1	Parameter List Register, Supervisor Parameter Register, or Parameter List Register	PL
13	Save Area Register	SA
14	Return Register	R
15	Entry Point Register, Return Code Register	E

It is the responsibility of the called program to maintain the integrity of general registers 2-12 so that their contents are the same at exit as they were at entry to the called program. It is the calling program's responsibility to maintain the floating-point registers around a call. General registers 0, 1, and 13-15 must conform to the indicated conventions.

Save Area

Whenever one program calls another, the calling program provides a save area for use by the called program; the calling program is known as the owner of the save area. This save area is addressed by the save area register (SA) on entry to a called program and is described in detail in Assembler Users Macro Instructions; in general the format is:

Word 1 - Contains the length in bytes of the save area and any appendages to it. This field is set by the calling program in its own save area, and always contains the integer 76 (in TSS/360).

Word 2 - Contains a pointer to the save area of the calling program. This field is set by the called program in its own save area. This procedure allows all save area of active programs to be linked in a reverse chain.

Word 3 - Contains a pointer to the save area of a called program after its invocation. This field is set by the called program in the calling program's save area. This allows all save areas of active programs to be linked in a forward chain. When the called program is complete, if trace forward has been specified, it sets the low order bit of this field to 1 to stop the forward chain.

Word 4 - Contains the return linkage for use by the called program when it is complete. This field is set by the called program in the calling program's save area.

Word 5 - Contains the entry point address to the called program. This field is set by the called program in the calling program's save area.

Words 6-18 - Register save area. These fields are set by the called program in the calling program's save area as necessary to preserve registers 0 through 12.

Word 19 - Contains the address of the PSECT belonging to the called program. This field is set by the calling program in its own save area.

It is clear that a program may use its own PSECT for a save area provided that the head of the PSECT is formatted as indicated above. However, if PSECTs are used for save areas, a called program will not use its own PSECT to save registers, but rather will use the save area (PSECT) of the calling program.

Parameter List, Type I Linkage

Whenever it is necessary for one program to explicitly communicate information to another program using Type I linkage, it must do so by using a parameter list. A parameter list is an ordered list of addresses of information. At the time of the CALL, the calling program places the address of the parameter list in the parameter list register (PL). (This list is most probably a list of address constants residing in the PSECT.) See the CALL macro expansion.

It should be noted that an active parameter of the CALL (a parameter which is set by the called program) is also passed in this manner.

Type I Linkage, Return and Entry Linkage and Return Code

It is the calling program's responsibility to establish the return and entry linkage at the time of a call. This will normally be accomplished by placing a V-type Adcon in the entry point register using a load instruction and then executing a branch and store instruction which will establish the return location in the return register and pass control to the indicated program.

RESTRICTED LINKAGE CONVENTIONS

Scope and Applicability of Restricted Linkage

A restricted linkage may only occur between two programs if all the following conditions are met.

1. The two programs involved must have the same PSECT in common and it must be contiguous; this area must be covered at all times by the PSECT cover register.
2. The invocation may not use or require explicit dynamic linkage.
3. The invoked program may not be enterable at the same point by way of a standard linkage.
4. The invoked program may not establish a non-volatile register as a common register (see below).
5. The programs involved must reside in virtual (as opposed to real) storage.
6. The invoking and the invoked programs must both be privileged or non-privileged.

Register Usage and Assignment in Restricted Linkage

There are four classes of restricted linkage registers: parameter, volatile, non-volatile (including common), and linkage. These classes are now described individually.

1. Parameter Registers and Parameter List Registers

   These registers are used explicitly to pass information in a restricted linkage both from the invoking to the invoked program and from the invoked to the invoking program. A parameter, in this context, may be by name or value. These registers may also be used to address parameter lists. Clearly, the content of these registers must be known by implicit agreement among the programs involved.

   Parameter registers are of the pseudo-volatile class where it is generally the responsibility of the invoking program to insure their integrity as necessary.

2. Volatile Registers

These registers may always be changed at will by the invoked program. The invoking program may never assume that they contain any specific values after an invocation is complete.

3. Non-Volatile (and Common) Registers

These registers are generally transparent around an invocation; that is, a non-volatile register must be STOREd and RESUMEd in the invoked program as necessary to preserve their contents (or the equivalent of STORE, RESUME). It is possible, however, that some, or all of the non-volatile registers may be established as common registers. A common register is a pseudo-parameter which is known to invoked programs and may change from invocation to invocation; thus a common register need not maintain constant value but must retain constant function. When a non-volatile register is established as a common register, all programs invocable by the establishing program (directly or indirectly) must be implicitly aware of this assignment.

In this context "Establish" means to set aside or dedicate a register for a particular use; this does not mean that the register must be initialized or modified. Common register usage must conform to tne following rules:

a. A common register may not be established by a program invoked with a restricted linkage; thus, a common register may be established only by a program CALLed with a standard linkage.

b. Common registers may not be STOREd and RESUMEd by a program invoked with a restricted linkage unless that program contains no linkage to other programs.

c. The scope of common registers shall extend only to those programs invoked while the establishing program is active; furthermore, no common register definition is known to any program CALLed with a standard linkage from a program invoked with a restricted linkage.

4. Linkage Registers

The linkage registers are those registers which must contain specific information during an invocation. There are three such registers which are now described individually.

a. PSECT cover register. This register must cover the common PSECT at all times (point to the origin of the PSECT).

b. Return register. This register is initialized by the invoking program to the proper return location in the invoking program. This register must be used by the invoked program upon completion to return control (to the indicated return location).

c. The entry point and return code register. This register must be initialized by the invoking program to the entry point address in the invoked program before the linkage takes place; it may also be used by the invoked program to pass a "return code" (see standard linkage) to the invoking program.

The entry and return registers are volatile in the sense that the act of invocation will cause their values to change.

The following register assignments have been made for the above classes:

Function	Assignment
Parameter (list) registers	0-5
Volatile registers	6-7
Non-Volatile registers	8-12
Common registers sequentially from	8
PSECT cover register	13
Return and entry registers	14-15

The floating-point register conventions, PICA, and program mask requirements are as described for standard linkage. No special save area format or location is committed; this function is left to the user.

MACRO INSTRUCTION SUPPORT

The following macro instructions are defined as support for restricted linkage. These macro instructions are used exclusively when effecting a restricted linkage.

INVOKE Macro Instruction

The INVOKE macro instruction transfers control from one program to another with a restricted linkage. It is not possible to specify parameters of a linkage in the INVOKE macro as in the CALL macro. This function is left to the user to accomplish however he sees fit.

General Form

[symbol] INVOKE addr

symbol - Any programmer-defined label.

addr - Specifies the address of a full-word which contains the address of the program to be invoked.

Expansion

The expansion of this macro instruction causes linkage register 15 to be loaded with the address at addr, linkage register 14 to be set to the return location in the invoking program, and control to be passed to that location specified in linkage register 15.

Examples:

1. INVOKE A
   in line
   L 15,A
   BASR 14,15

2. INVOKE B(3)
   in line
   L 15,B(3)
   BASR 14,15

STORE MacroInstruction

The STORE macro instruction causes the indicated non-volatile registers to be stored in the specified area.

General Form

[symbol] STORE addr, (reg$_1$-integer [,reg$_2$ - integer])

symbol - Any programmer-defined label.

addr - Specifies the address of an area sufficient to contain the indicated registers.

reg$_1$,reg$_2$ - Integers specifying a range of registers to be stored at addr. If reg$_2$ is not specified, only reg$_1$ will be stored.

Expansion

A STM (or ST, in the event that reg$_2$ is not specified) instruction is generated to store the indicated register(s) at addr.

Notes:

1. It is possible to specify reg$_2$ as 14 or 15 thus causing the linkage registers to be stored.

2. Reg$_1$ must be greater than or equal to 8.

3. If reg$_2$ is 14 or 15, a single STM will be generated spanning register 13. Thus, the area at addr must be of such size as to contain this register, and it is stored redundantly.

Examples:

1. STORE A, (9,10)
   in line
   STM 9, 10, A

2. STORE B, (11)
   in line
   ST 11,B

RESUME Macro Instruction

This macro instruction causes the indicated non-volatile registers to be restored from the specified area and control to be passed via the return register.

General Form

[symbol] RESUME [addr, (reg$_1$-integer [, reg$_2$-integer])]
[, RC=integer]

symbol - Any programmer-defined label.

addr - Specifies the address of an area from which the indicated registers are to be restored.

reg$_1$, reg$_2$ - Integers specifying the range of registers to be restored. If reg$_2$ is not specified, only reg$_1$ will be restored.

RC - Specifies an integer to be used as a return code ($0 \leq RC \leq 4092$). RC must be a multiple of 4.

Expansion

A LM (or L, if reg$_2$ is not specified) instruction is generated to load the indicated register(s) from addr, the return code register is loaded as necessary with the specified integer (using a LA instruction), and a BR 14 instruction is generated to return control to the invoking program.

Notes:

1. If the addr and register fields are not specified, only the BR 14 instruction is generated.

2. It is possible to specify reg_2 as 14 or 15 causing the linkage registers to be loaded.

3. Reg_1 must be greater than or equal to 8.

4. If reg_2 is 14 or 15, a LM instruction is generated spanning register 13 (thus loading it redundantly). The user must use a certain amount of caution when allowing this to happen in order that the contents of register 13 are not destroyed.

Examples:

1. RESUME A, (9)
   in line
   L 9, A
   BR 14

2. RESUME RC=4
   in line
   LA 15,4
   BR 14

3. RESUME
   in line
   BR 14

APPENDIX C: FORTRAN INTERNAL MACRO INSTRUCTION USAGE

The table below contains a brief description of the user macro instructions required for assembly of compiler modules. These macro instructions are contained on the second macro library provided the TSS/360 Assembler when assembling compiler modules. The first macro library provided is that containing the system macro instructions, described in the Assembler User Macro Instructions.

Name	Description
EXECUTIVE Macro Instructions	
CEKTO	Performs all operations concerned with calling a phase, including checking the return code, giving diagnostic option dumps if requested, and logging the phases on and off.
CEKT7	Moves an option from the F-option table passed from LPC to FORTRAN at the 'Initial' call to the Executive PSECT. If LPC passes neither an 'N' nor a 'Y', the default value assembled into the phase controller module is used.
CEKT8	Executive intercom macro.
CEKT9	Identical to CEKT8, but contains DS rather than DC. Used by other Executive programs, and the 5 phases.
CEKTG	Diagnostic option macro instruction. Sets up calling list for module CEKTS, and calls that module.
CEKTT	Null.
CEKTX	Forms V-R con pairs, with the aid of the ADCON macro instruction.
CEKTY	Checks batch/conversation switch in the Executive, and LPC if in conversion. Resets switch as appropriate.
CEKU1	Converts a binary number to zoned format, edits it under control of a mask, and moves the result to a user specified area.
CEKU3	Used to determine, from the limits of a main storage file, the base and two- or four-byte offsets from the base to the limits.
CEKU4	Null.
CEKU5	Branches to one of five places, depending upon the return code in register 15.
CEKU6	Establishes intercom cover in register N4 and cover for the Executives PSECT page 1 in general register N3.
CEKU7	Macro instruction for the Phase Controller PSECT. Used to generate a DSECT in other Exec modules.
CEKU8	Used where the test must be made of a list data set exists, and, if so, OLR is called.

Name	Description
EXECUTIVE Macro Instructions (Continued)	
CEKU9	Used to edit the line number of a line for which a diagnostic message is to be given, move the number to a message area, call DIAGOUT to output the message, test the return code, and branch accordingly.
CEKUX	Sets up FREEMAIN calls (using CEKV9) for those parts of Phase 5 files not containing the output module.
CEKV1	Moves addresses and lengths of Phase 5 created tables to LPC prior to the 'Continue' call return.
CEKV2	Determines, for GNSS, if an END statement has been encountered.
CEKV3	Performs all operations concerned with checking to see if the diagnostic mode is allowed, checking for the existence of the diagnostic line, and, if it is present, processing the two diagnostic request lines. Also sets up the interruption entries.
CEKV5	Used in producing diagnostic output. Checks the 10-column fields provided for each phase on diagnostic card 1 and gives dumps accordingly.
CEKV6	Used in producing diagnostic output. Loads a register with a two- or three-character code, then invokes a sequence of instructions that will dump this code plus contents of all general registers, if so requested on diagnostic card 1.
CEKV7	Used to establish a CSECT (for the initial entry to a module), USING statement, PSECT cover, and code cover in register N5.
CEKV8	Similar to CEKV7, but establishes the PSECT and ENTRY statements, plus the 19-word save area.
CEKV9	Used for all FREEMAIN operations. In addition to the actual FREEMAIN, records are kept in the PSECT of main storage areas freed, and their size.

Name	Description
EXECUTIVE Macro Instructions	
CEKVA	Obtains the version of this compilation and moves it to appropriate edit areas.
CEKVB	Used in preparing diagnostic information. Reached by an INVOKE in macro instruction CEKV6.
CEKVC	Used for all GETMAIN operations. Obtains parameters from the PSECT giving the number of pages to obtain, obtains them, and keeps records in the Phase Controller PSECT of pages obtained and their location.
CEKVD	Produces a complete virtual storage dump. Used in diagnostic mode processing.
CEKVI	Contains code related to control of unexpected interruptions during the compilation process. Entered only in diagnostic mode.
CEKVU	The data management operations OPEN, SETL, PUT, and CLOSE are all embedded in the macro instruction CEKVU.
CEKZD	Establishes PSECT cover, and backward and forward PSECT chains.
Phase 1 Macro Instructions	
CEKHB	Saves a one- or two-byte Symbol Table field.
CEKHC	DSECT describing the Symbol Table fields for a variable entry.
CEKHD	Phase 1 PSECT.
CEKHF	Updates the Cross-Reference List.
CEKHG	Creates a two-byte, signed Adcon for the error routine.
CEKHH	Sets up INVOKE, tests the return code, branches conditional, and sets up RESUME.
CEKHI	Defines field used in internal tables and flags with EQU cards.
CEKHJ	Several DSECTs defining some internal tables.
CEKHL	Makes an RDM entry for the error routine.
CEKHM	Saves a one-byte field of the Symbol Table.
CEKHN	Uses CEKHG to create parameter lists for the error routine.
Phase 2 Macro Instructions	
CEKJM	Defines the fields of internal tables by EQU cards.
CEKJO	Phase 2 PSECT.
CEKJ1	Checks for a barrier between the plateau values of a reference and a definition of a statement label.
CEKJ2	Marks the materialization list.

Name	Description
Phase 2 Macro Instructions (Continued)	
CEKJ3	Checks the inner loop table for a reference or a definition of a specified statement label.
CEKJ4	Marks the return list.
CEKJ5	Relinks a specified chain in the opposite direction.
CEKJ6	Tests for a D-Loop or an E-Loop table entry between ILINK and PDLINK.
CEKJ7	Defines the fields of internal tables by EQU cards.
CEKJ8	Creates the parameter list for the error routine, DX.
CEKJ9	Generates an invoke to the error routine, DX.
Phase 3 Macro Instructions	
CEKKS	Phase 3 PSECT.
CEKKK	DSECTs describing the internal tables.
CEKZB	Sets the DSECT or PSECT control and specifies print options.
LDPNT	Converts a pointer to an address and loads it into a register.
STPNT	Converts an address into a pointer and saves the result.
EKKSB	Assembles an address as a pointer.
EKKSC	Sets an origin to a specified boundary.
CEKKD	diagnostic mode, provides dumps during Phase 3 processing.
Phase 4 Macro Instructions	
CEKNY	Phase 4 PSECT.
CEKN2	Tests the return code, and branches conditionally to the error routine (PH4MER).
CEKN3	Generates the calling sequence to find a constant and its associated Adcon in the Symbol Table.
CEKN4	Generates the calling sequences to obtain the location of an operand in the expression tree, make the appropriate entries in the internal tables, and insert the associated instructions into the Code File.
CEKN5	Inserts previously specified sequence of instructions (canned code) into the Code File.
Phase 5 Macro Instructions	
CEKSY	Phase 5 PSECT.
CEKSZ	EQU cards for the standard register symbolic names.
CEKS2	Picks up a specified number of characters from a table.
CEKS3	Decrements the print control line count.

APPENDIX D: LIST OF MAJOR TABLES REFERENCED BY FORTRAN ROUTINES

Phase 1 (X = Set; O = Referenced Only)

Name	Program Representation File (PRF)	Symbol Table	Cross Reference Table	Expression Representation File (ERF)	Dimension Table	Storage Class Table	Alphametric Table	Implicit Type Table	HSTCK	SXS	Do Loop Stack
CEKAB (ESC)											
CEKAC (SID)	X	X									
CEKAD (PHIM)		X						O			
CEKAE (ACOMP)		X					X				
CEKAF (ARDIM)		X			O						
CEKAG (SUBS)		X		X							
CEKAH (IDATA)		X	X						X	X	
CEKAI (EXPR)	X	X	X	X							
CEKAJ (LABL)	X	X									O
CEKAK (EQUA)	X	X									
CEKAL (END)	X										
CEKAM (EXTE)	X										
CEKAN (CNVRT)		O		X						X	
CEKAQ (GOTO)	X	X	X	X							
CEKAR (IF)	X	X			X						
CEKAS (TYPE)		X			X						
CEKAT (CONT)	X										
CEKAU (DIMN)					X	X					
CEKAV (COMM)		X									
CEKAW (DCL2)						X					
CEKAX (EXEC2)	X										
CEKAY (EQUI)		X	X								
CEKAZ (DO)											X O
CEKBA (BGNLP)	X	X									
CEKBB (ENDLP)		X									
CEKBC (ASSI)	X	X		X							
CEKBD (FCON)	X	X		X							
CEKBE (RWIO)	X	X		X							
CEKBF (FORM)	X	X		X			X				
CEKBG (PSR)		X				X	X				
CEKBH (NAML)		X									
CEKBI (BLDA)											
CEKBJ (BLDA2)											
CEKBK (SFDEF)		X		X					X	X	
CEKBL (SFEXP)											
CEKBM (DATA)											
CEKBN (IMPL)								O			
CEKBP (IMPL2)								X			
CEKBQ (FALTH)	X										
CEKBR (BLNK)											
CEKBS (SUBE)		X				X					
CEKBT (SUBE2)	X										
CEKBU (CALL)	X	X	X								O
CEKBV (CALL2)	X	X		X							
CEKBW (IOLST)		X									
CEKBX (FNCLS)		X		X							
CEKBY (LIBN)		X								O	
CEKBZ (STEN2)		X									
CEKCA (ERR)		O									O
CEKCB (ARITH)											
CEKCC (LBSTR)	X	X	X								
CEKCD (FLABL)	X	X		X							
CEKCE (RTRAN)	X	X									
CEKCF (FNAME)	X	X									
CEKCG (TRMPRO)		X									
CEKCH (FLRC)			X								
CEKCI (IVST)		X		X							
CEKCJ (CKLIM)	X	X									
CEKCK (CLLIM)											
CEKCL (IVAL)		X									
CEKCN (ICNV)											
CEKCP (FCNV)											
CEKCQ (FLIC)											
CEKCR (AARC)	X	X		X					X	X	
CEKCS (CHKINT)											

Phase 2 (X = Set; O = Referenced Only)

Name	PRF	Symbol Table	Storage Class table	Storage Specification List	GLINK	LINK	PDLNK
CEKJA (PHASE2)	X	X	X	O			
CEKJB (FSCAN)	X	X			X	X	
CEKJC (VSCAN)		X	X	O			
CEKJD (RTN1)	X	X					
CEKJE (LAB)	O	O					
CEKJF (ISP)	O	O					
CEKJG (FORMAT)	O	O					
CEKJH (DX)	O	O					

Phase 3 (X = Set; O = Referenced Only)

Name	Program Representation File (PRF)	Symbol Table	Expression Representation File (ERF)	Parallel Loop Table (PLP)	Current Loop Level (LEV)	Current Next Table (CNT)	Compute and Remove Table (CRT)	Program File (PF)	Triad Table	Operand Push-down Table (OPTT)
CEKKA	X	X	X	X	X	X			O	
CEKKB	X	X		X					X	
CEKKC		X		X	X	X	X	X	X	X
CEKKE		X	X							
CEKKF		X								
CEKKG			X	X					O	
CEKKH			X			O			O	
CEKKI					O				O	
CEKKJ			O	O					O	
CEKKK			X	X	O	O	X			
CEKKL										
CEKKM										
CEKKN				X					O	
CEKKO									O	
CEKKP				X	O	O				
CEKKR	O									
CEKKS										
CEKKU	X	X		X	O			X		
CEKKV	X	X			X					
CEKKW	X		X	O	O			X	X	
CEKLA		X	X				X			
CEKLB			X							
CEKLD			X							
CEKLE			X				X	X	X	
CEKLF			X							
CEKLI	O	X								

Phase 4 (X - Set; O - Referenced)

Name	Symbol Table	Storage Class Table	Program File (PF)	General Register Table (MRM)	Name Table	Expression Table	Loop Table	Floating-Point Register Table (MRMFR)
CEKMA (RDIV)					O	O		
CDKMB (RMUL)					O	O		
CEKMC (RPLUS)					O	O		
CEKMD (IDVDF)				O	O	O		
CEKME (IMPLY)				O	O	O		
CEKMF (IPLUS)				O	O	O		
CEKMG (CPLUS)					O	O		
CEKMH (RLTNL)					O	O		O
CEKMI (ANDOR)				O	O	O		
CEKMJ (EQUAT)	O			X				
CEKMK (FUNC)			X					
CEKML (TRBLD)					X	X		
CEKMM (ASAR)				O	O	O		
CEKMN (ASARS)				O	O	O		
CEKMO (ASFR)				O	O	O		X
CEKMP (ASFRS)					O	O		O
CEKMQ (SELFR)					O	O		O
CEKMR (FNDAR)				O				
CEKMS (FNDFR)								O
CEKMT (FNDWS)								
CEKMU (MAX)				O	O	O		O
CEKMV (MEMAC)				X	X	O		
CEKMW (OPND)				X	X	O		
CEKMX (RLSWS)					O	O		
CEKMY (RSLT)					X	O		
CEKMZ (SADDR)				O				
CEKNA (SELGD)				O	O	O		
CEKNB (SELGM)				O	O	O		
CEKND (SELOP)					O	O		
CEKNE (WGHT)						X		
CEKNF (SLPOS)					O	O		O
CEKNG (SELSR)				O	O	O		
CEKNH (SELDR)				O	O	O		
CEKNI (INSOT)		X						
CEKNJ (COMMA)					X	X		
CEKNK (AIF)				O				
CEKNL (LIF)				O				

Phase 4 (X - Set; O - Referenced)

Name	Symbol Table	Storage Class Table	Program File (PF)	General Register Table (MRM)	Name Table	Expression Table	Loop Table	Floating-Point Register Table (MRMFR)
CEKNM (BLI)								
CEKNN (BL2)				X	X	X	X	
CEKNO (BL3)	X				X		O	
CEKNP (ENDLP)					X	X	X	
CEKNQ (AGO)				X				
CEKNR (CGO)	X			X				
CEKNS (ASSGN)	X			X				
CEKNT (GOTO)	O							
CEKNU (LABEL)	X							
CEKNV (LBL)	O							
CEKNW (AGEN)					X	X		X
CEKNX (PHAS4)			O					
CEKOB (CSX)					X	X		
CEKOC (KEY)				O				
CEKOD (ENT)	O		O		O	O		
CEKOE (RTRN)	O		O					
CEKOF (CMUL)				O	O	O		
CEKOG (CDIV)				O	O	O		
CEKOH (RD)	O			X				X
CEKOI (OLIST)	O			X				X
CEKOJ (NDLST)	O			X				X
CEKOK (STOP)	O			X				X
CEKOL (CALL)								
CEKOM (DCOM)								
CEKOM2 (OPEN6)				X				
CEKON (FLUSH)				O	O	O		O
CEKOP (COVER)	O							
CEKOQ (EDIT)								
CEKOR (KEY1)					O			
CEKOS (FETCH)				X	X			
CEKOT (OPEN1)				X	X			
CEKOU (OPEN2)				X				
CEKOV (LADDR)				X	X			
CEKOW (SLONE)	O			O		O	O	
CEKOX (OPEN3)				X				
CEKOY (OPEN4)				X				
CEKOZ (OPEN5)				X				

Phase 5 (X = Set; O = Referenced Only)

Name	Symbol Table	Storage Class Table	Code File	Constant Header Table	Internal Symbol Dictionary (ISD)	Program Module Dictionary (PMD)
CEKSA (PHASE5)						
CEKSB (BUILD)		O			X	O
CEKSC (CMSEC)		O				O
CEKSD (SPECS)	O					
CEKSE (PHEAD)						
CEKSF (COSEC)	O	O	O	O	X	X
CEKSG (PRSEC)	O	O		O		
CEKSH (ASSIST)	O	O			O	
CEKSI (EDIT)	O	O	O	O		
CEKSJ (SYMSRT)	O	O				
CEKSK (CRFSRT)	O		O			
CEKSL (CONCV)	O					

Executive

All Exec routines set the Intercom and Excom.
CEKTF also sets the Symbol Table.

The FORTRAN routines bear coded labels as well as mnemonic titles. The 5-character coded label begins with the letters CEK; the fourth and fifth identify the specific routine. The entry points to a routine are specified by a sixth character appended to the coded label; for example, the coded label for the Master Input/Output routine is CEKTH, and there are entry points CEKTHA, CEKTHB, etc.

FORTRAN ROUTINES LISTED BY CODED LABELS (Part 1 of 6)

Routine ID Label	Entry Point	Purpose	Mnemonic	Phase	Chart ID
CEKAB		Obtain-next-character	ESC	1	BV
	CEKAB1	Get next non-blank character	ESC		
	CEKAB2	Get next character (including blanks)	ESCB		
CEKAC	CEKAC1	Statement identification	SID	1	CM
CEKAD	CEKAD1	Phase 1 main loop	PHIM	1	AM
CEKAE	CEKAE1	Assemble components	ACOMP	1	BW
CEKAF	CEKAFA	Process array dimension specification	ARDIM	1	CK
CEKAG	CEKAG1	Process subscripts	SUBS	1	AQ
CEKAH		Process initial value data specification	IDATA	1	CI
	CEKAH1	Entry for Type statements	TDATA		
	CEKAH2	Entry for DATA statements	DDATA		
CEKAI	CEKAI1	Process expression	EXPR	1	BL
CEKAJ	CEKAJA	Process statement label	LABL	1	CN
CEKAK	CEKAK1	Process assignment statement	EQUA	1	AN
CEKAL	CEKAL1	Process END statement	END	1	BG
CEKAM	CEKAMA	Process EXTERNAL statement	EXTE	1	AO
CEKAN		Conversion	CNVRT	1	BM
	CEKAN1	Converts constants to new type and checks legal type mixes for expressions	CNVRT		
	CEKAN2	Converts constants to new type	CNVRTD		
CEKAQ	CEKAQA	Process GO TO statement	GOTO	1	AP
CEKAR	CEKARA	Process IF statement	IF	1	AQ
CEKAS		Process Type statement	TYPE	1	AR
	CEKAS1	Entry for INTEGER statements	INTE		
	CEKASR	Entry for REAL statements	REAL		
	CEKASC	Entry for COMPLEX statements	COMP		
	CEKASL	Entry for LOGICAL statements	LOGL		
	CEKASD	Entry for DOUBLE PRECISION statements	DOBP		
CEKAT	CEKAT1	Process CONTINUE statement	CONT	1	None
CEKAU	CEKAUA	Process DIMENSION Statement	DIMN	1	AS
CEKAV	CEKAV1	Process COMMON statement	COMM	1	AT
CEKAW		Process declatation statements in Pass 2	DCL2	1	None
	CEKAW1	Entry for COMMON statements	COMM2		
	CEKAW2	Entry for other declaration statements	DCL2		
CEKAX		Process executable statements in Pass 2	EXEC2	1	BH
	CEKAX1	Entry for no-flow statements	NF2		
	CEKAX2	Entry for flow-thru statements	FL2		
CEKAY	CEKAYA	Process EQUIVALENCE statement	EQUI	1	CA
CEKAZ	CEKAZ1	Process DO statement	DO	1	CC
CEKBA	CEKBA1	Analyzes and encodes begin loop information	BGNLP	1	FM
CEKBB	CEKBB1	Encodes the end loop entries	ENDLP	1	CC
CEKBC	CEKBCA	Process ASSIGN statement	ASSI	1	AW
CEKBD		Process file control statements	FCON	1	AX

FORTRAN ROUTINES LISTED BY CODED LABELS (Part 2 of 6)

Routine ID Label	Entry Point	Purpose	Mnemonic	Phase	Chart ID
	CEKBD1	Entry for BACKSPACE statement	BXSP		
	CEKBD2	Entry for END FILE statement	ENDF		
	CEKBD3	Entry for REWIND statement	REWI		
CEKBE		Process input/output statements	RWIO	1	AY
	CEKBE1	Entry for READ statement	READ		
	CEKBE2	Entry for WRITE statement	WRIT		
	CEKBE3	Entry for PRINT statement	PRNT		
	CEKBE4	Entry for PUNCH statement	PUNCH		
CEKBF	CEKBF1	Process FORMAT statement	FORM	1	AZ
CEKBG		Process PAUSE, STOP, RETURN statements	PSR	1	BA
	CEKBG1	Entry for PAUSE statement	PAUS		
	CEKBG2	Entry for STOP statement	STOP		
	CEKBG3	Entry for RETURN statement	RETU		
	CEKBG4	Stop when execution flows into END	ESTOP		
CEKBH	CEKBH1	Process NAMELIST statement	NAML	1	BB
CEKBI	CEKBI1	Process BLOCK DATA statement	BLDA	1	BC
CEKBJ	CEKBJ1	Sets the program type for the BLOCK DATA statement	BLDA2	1	None
CEKBK	CEKBK1	Enables EXPR to translate a statement function expression	SFDEF	1	BN
CEKBL		Expand-statement-functions	SFEXP	1	BO
	CEKBL1	Statement function expansion initialization	SFEXPI		
	CEKBL2	Statement function expansion continuation	SFEXPC		
CEKBM	CEKBM1	Process DATA statement	DATA	1	BD
CEKBN	CEKBN1	Process IMPLICIT statement	IMPL	1	BE
CEKBP	CEKBP1	Perform final IMPLICIT statement housekeeping	IMPL2	1	None
CEKBQ	CEKBQA	Determine if a label reference refers to the current statement	FALTH	1	CO
CEKBR	CEKBR1	Process a blank source statement	BLNK	1	None
CEKBS		Process subprogram entry statements	SUBE	1	BF
	CEKBS1	Process ENTRY statement	ENTR		
	CEKBS2	Process FUNCTION statement	FUNC		
	CEKBS3	Process SUBROUTINE	SUBR		
CEKBT		Process subprogram entry statements in Pass 2	SUBE2	1	BI
	CEKBT1	Process ENTRY statement	ENTR2		
	CEKBT2	Process FUNCTION statement	FUNC2		
	CEKBT3	Process SUBROUTINE statement	SUBR2		
CEKBU	CEKBU1	Process CALL statement	CALL	1	None
CEKBV	CEKBV1	Process CALL statement in Pass 2	CALL2	1	BJ
CEKBW	CEKBW1	Analyze and encode list elements for READ, WRITE, PRINT, and PUNCH statements	IOLST	1	CE
CEKBX	CEKBX1	Determine the proper class of a function	FNCLS	1	BP
CEKBY		Select-library-functions	LIBN	1	BQ
	CEKBY1	Functions with automatic typing	LIBN		
	CEKBY2	Functions used as arguments	LIBNA		
	CEKBY3	Exponential library function selection	LIBNX		
CEKBZ	CEKBZ1	Determine statement function in Pass 2	STFN2	1	None
CEKCA		Generate diagnostic messages	ERR	1	CP
	CEKCAA	Warning messages (error level 1)	ERR1		
	CEKCAB	Serious messages (error level 2)	ERR2		
	CEKCAC	Serious messages associated with statement deletion (error level 2)	ERRD		

FORTRAN ROUTINES LISTED BY CODED LABELS (Part 3 of 6)

Routine ID Label	Entry Point	Purpose	Mnemonic	Phase	Chart ID
	CEKCAD	Fatal messages (error level 3)	ERR3		
CEKCB	CEKCB1	Perform constant arithmetic	ARITH	1	BR
CEKCC	CEKCCA	Process label string	LBSTR	1	CL
CEKCD	CEKCD1	Process format label for input/output statements	FLABL	1	CF
CEKCE	CEKCE1	Process ERR and END labels for input/output statements	RTRAN	1	CG
CEKCF	CEKCF1	Process FORMAT or NAMELIST name	FNAME	1	CH
CEKCG	CEKCG1	Process subscript term	TRMPRO	1	BS
CEKCH	CEKCH1	File real and complex constants in Symbol Table	FLRC	1	BX
CEKCI	CEKCI1	Insert variable in Symbol Table	IVST	1	BY
CEKCJ	CEKCJ1	Check loop limits	CKLIM	1	CD
CEKCK	CEKCK1	Clear loop limits	CLLIM	1	None
CEKCL		Process initial value	IVAL	1	
	CEKCL1	First value in type statement group	IVAL		CJ
	CEKCL2	Other than first value in type statement group	IVAL1		AI
CEKCN	CEKCN1	Convert decimal to binary integer	ICNV	1	BZ
CEKCP	CEKCP1	Convert decimal to floating binary	FCNV	1	CA
CEKCQ	CEKCQ1	File integer constant	FLIC	1	None
CEKCR	CEKCR1	Provide service in processing actual argument	AARG	1	BT
CEKCS		Provide for treatment of interruptions	CHKINT	1	BU
	CEKCS1	Divide check	CEKCS1		
	CEKCS2	Exponent overflow	CEKCS1		
	CEKCS3	Return flags	CHKINT		
	CEKCS4	Enable 3 interruptions	CHKINT		BU
	CEKCS5	Disable 3 interruptions	CEKINT		BU
CEKJA	CEKJA1	Control Phase 2	PHASE2	2	Fig.19
CEKJB	CEKJB1	Process label references and definitions	FSCAN	2	CR
CEKJC	CEKJC1	Make storage assignments for all variables	VSCAN	2	CQ
CEKJD	CEKJD1	Process label references	RTN1	2	CS
CEKJE	CEKJE1	Process label references	LAB	2	CT
CEKJF	CEKJF1	Inspect statement label references	ISP	2	None
CEKJG	CEKJG1	Inspect format reference	FORMAT	2	None
CEKJH		Generate diagnostic messages	DX	2	CU
	CEKJH1	Warning messages	DXW	2	
	CEKJH2	Serious messages	DXF	2	
	CEKJH3	Abort messages	DXA	2	
CEKKA	CEKKAA	Acquire entry from Compute and Removal Table	-	3	DP
CEKKB	CEKKBA	Generate Polish expression	-	3	DQ
CEKKC	CEKKCA	Process End Loop entries in PRF Table	-	3	CX
CEKKE	CEKKEA	Scan entry in the Expression File	-	3	DA
CEKKF	CEKKFA	Pushdown primitive operator	-	3	DC
CEKKG	CEKKGA	Determine the forward compute point and removal level for a variable	-	3	DD
CEKKH	CEKKHA	Manipulate Triad File	-	3	DF
CEKKI	CEKKIA	Determine commonality or removability of an expression	-	3	DH
CEKKJ	CEKKJA	Determine whether entries in tne PRF are common	-	3	DK
CEKKK	CEKKKA	Establish common expression	-	3	DJ
CEKKL	CEKKLA	Form operand list expression	-	3	DE
CEKKM	CEKKMA	Revise subscript expression	-	3	DO
CEKKN	CEKKNA	Put expression into cannonical form	-	3	DH
CEKKO	CEKKOA	Save popularity counts for register assignment	-		

FORTRAN ROUTINES LISTED BY CODED LABELS (Part 4 of 6)

Routine ID Label	Entry Point	Purpose	Mnemonic	Phase	Chart ID
CEKKP	CEKKPA	Search and insert triad	-	3	DG
CEKKR		Control Phase 3		3	CV
	CEKKRA	standard entry by EXEC	-	3	
	CEKKRE	Error exit by all Phase 3 routines	-	3	
CEKKS	-	Phase 3 storage	PSECT	3	None
CEKKU	CEKKUA	Manipulates the PRF entry into its proper PF format	-	3	CW
CEKKV	CEKKVA	Process the Begin Loop 1 entries in PRF	-	3	CY
CEKKW	CEKKWA	Process the Begin Loop 2 entries in PRF	-	3	CZ
CEKLA	CEKLAA	Label common expression	-	3	DL
CEKLB	CEKLBA	File a constant, compute and file its covering Adcon, and compute displacement	-	3	IW
CEKLD	CEKLDA	Expunge a removable subexpression	-	3	DN
CEKLE	CEKLEA	File CRT entries	-	3	DM
CEKLF	CEKLFA	Copy and edit an expression	-	3	DB
CEKLI	CEKLIA	Generate loop text-expression	-	3	DT
CEKMA	CEKMA1	Generate real divide	RDIV	4	ES
CEKMB	CEKMB1	Generate real multiply	RMUL	4	ER
CEKMC	CEKMC1	Generate real plus	RPLUS	4	KM
CEKMD	CEKMD1	Generate integer divide	IDVDE	4	EV
CEKME	CEKME1	Generate integer multiply	IMPLY	4	EW
CEKMF	CEKMF1	Generate integer plus	IPLUS	4	ET
CEKMG	CEKMG1	Generate complex plus	CPLUS	4	EX
CEKMH	CEKMH1	Generate relational expression	RLTNL	4	FA
CEKMI	CEKMI1	Generate logical expression	ANDOR	4	FB
CEKMJ	CEKMJ1	Process equation program file entry	EQUAT	4	DW
CEKMK	CEKMK1	Generate external function	FUNC	4	FD
	CEKML2	Generate tree entries for a conversion function		4	None
CEKML	CEKML1	Build expression tree	TRBLD	4	EN
CEKMM	CEKMM1	Make initial assignment to General Register	ASAR	4	GE
CEKMN	CEKMN1	Assign to Arithmetic Register	ASARS	4	GF
CEKMO	CEKMO1	Make initial assignment to Floating-Point Register	ASFR	4	None
CEKMP	CEKMP1	Make synonym assignment to Floating Register	ASFRS	4	GG
CEKMQ	CEKMQ1	Select Floating Register	SELFR	4	GD
CEKMR	CEKMR1	Search General Registers	FNDAR	4	FX
CEKMS	CEKMS1	Search Floating Registers	FNDFR	4	FY
CEKMT	CEKMT1	Find temporary storage	FNDWS	4	GH
CEKMU	CEKMU1	Generate maximum operator	MAX	4	FC
CEKMV	CEKMV1	Access storage	MEMAC	4	FM
	CEKMV2		LSUB	4	None
CEKMW	CEKMW	Process operands	OPND		FZ
	CEKMW1	Process variable or constant	OPND	4	
	CEKMW2	Process operator	TROP	4	NK
	CEKMW3	Process common expression	CSOP	4	NK
CEKMW	CEKMW1	Process operand	OPND	4	FZ
CEKMX	CEKMX1	Release temporary storage	RLSWS	4	GI
CEKMY	CEKMY1	Process Result-Register operand	RSLT	4	GA
CEKMZ	CEKMZ1	Generate local branch	SADDR	4	FN
CEKNA	CEKNA1	Determine whether division may take place in register pair containing numerator	SELGD	4	FU
CEKNB	CEKNB1	Determine availability of register for multiplication	SELGM	4	FT
CEKND	CEKND1	Select operand	SELOP	4	FQ

FORTRAN ROUTINES LISTED BY CODED LABELS (Part 5 of 6)

Routine ID Label	Entry Point	Purpose	Mnemonic	Phase	Chart ID
CEKNE	CEKNE1	Assign a weight to each non-primitive mode of the expression tree	WGHT	4	EO
CEKNF	CEKNF1	Select position for operation	SLPOS	4	FR
CEKNG	CEKNG1	Select single General Register	SELSR	4	GB
CEKNH	CEKNH1	Select even/odd General Register pair	SELDR	4	GC
CEKNI	CEKNI1	Output code file	INSOT	4	GK
	CEKNI2	Error processing for code file overflow	*	4	None
CEKNJ		Process comma operator	COMMA	4	FE
	CEKNJ1	Entry for other cases	COMMA		
	CEKNJ2	Entry for CEKMK when function has only one argument	COMA2		
CEKNK	CEKNK	Process arithmetic IF PF entry	AIF	4	DX
CEKNL	CEKNL1	Process logical IF PF entry	LIF	4	DY
CEKNM	CEKNM1	Process begin loop 1 PF entry	BL1	4	EE
CEKNN	CEKNN1	Process begin loop 2 PF entry	BL2	4	EF
CEKNO	CEKNO1	Process begin loop 3 PF entry	BL3	4	EG
CEKNP	CEKNP1	Process end loop PF entry	ENDLP	4	EH
CEKNQ	CEKNQ1	Process assigned GO TO PF entry	AGO	4	EA
CEKNR	CEKNR1	Process computed GO TO PF entry	CGO	4	EB
CEKNS	CEKNS1	Process ASSIGN PF entry	ASSGN	4	DZ
CEKNT	CEKNT1	Process GO TO PF entry	GO TO	4	None
CEKNU	CEKNU1	Process referenced label PF entry	LABEL	4	DV
CEKNV	CEKNV1	Generate labeled branch	LBL	4	FO
CEKNW	CEKNW1	Generate arithmetic expression	AGEN	4	EM
CEKNX	CEKNX1	Control Phase 4	PHAS	4	Fig.30
CEKOB	CEKOB1	Count common expression usage	CSX	4	EP
CEKOC	CEKOC1	Determine the location of both operands of the current operation	KEY	4	FV
CEKOD	CEKOD1	Process entry point	ENT	4	DU
CEKOE	CEKOE1	Process RETURN	RTRN	4	ED
CEKOF	CEKOF1	Generate complex multiply	CMUL	4	EY
CEKOG	CEKOG1	Generate complex divide	CDIV	4	EZ
CEKOH	CEKOH1	Process input/output statement PF entry	RD	4	EI
CEKOI	CEKOI1	Process input/output list element PF entry	IOLIST	4	EJ
CEKOJ		Process and list PF entry	NDLST	4	EK
	CEKOJ1	Process an end list program file entry	NDLST		
	CEKOJ2	Generate a standard call linkage	LINK		
CEKOK	CEKOK1	Process STOP and PAUSE statement PF entry	STOP	4	EL
CEKOL	CEKOL1	Process CALL statement	CALL	4	EC
CEKOM	DCOM	Control selection of open function processing	DCOM	4	FF
CEKOM2	OPEN6	Process function numbers 3, 4, 6-11, 16, 21, 24, 25, 28-32	OPEN6	4	FL
CEKON	CEKON1	Clear register	FLUSH	4	GS
CEKOP	CEKOP1	Obtain Adcon cover for generation of a reference	COVER	4	None
CEKOQ	CEKOQ1	Edit for code file	EDIT	4	None
CEKOR	CEKOR1	Locate single operand	KEY1	4	FW
CEKOS	CEKOS1	Ensure that each argument of a function or subroutine is in memory with desired sign	FETCH	4	FP
CEKOT	CEKOT1	Process open function	OPEN1	4	FG

FORTRAN ROUTINES LISTED BY CODED LABELS (Part 6 of 6)

Routine ID Label	Entry Point	Purpose	Mnemonic	Phase	Chart ID
CEKOU	CEKOU1	Process open function	OPEN2	4	FH
CEKOV	CEKOV1	Add by load address	LADDR	4	EW
CEKOW	CEKOW1	Select one operand in a register	SLONE	4	FS
CEKOX	CEKOX1	Process open function	OPEN3	4	FI
CEKOY	CEKOY1	Process open function	OPEN4	4	FJ
CEKOZ	CEKOZ1	Process open function	OPEN5	4	FK
CEKSA	CEKSA1	Generate FORTRAN compiler output	PHASE5	5	None
CEKSB	CEKSB1	Build object program module	BUILD	5	GL
CEKSC	CEKSC1	Generate common control section	CMSEC	5	GM
CEKSD	CEKSD1	Process preset data	SPECS	5	GP
CEKSE	CEKSE1	Produce page headings for each Phase 5 output page	PHEAD	5	None
CEKSF	CEKSF1	Generate code control section	COSEC	5	GN
CEKSG	CEKSG1	Build PSECT	PRSEC	5	GO
CEKSH	CEKSH1	Generate Internal Symbol Dictionary	ASSIST	5	GQ
CEKSI	CEKSI1	Document object program	EDIT	5	GR
CEKSJ	CEKSJ1	Sort symbol table	SYMSRT	5	GS
CEKSK	CEKSK1	List cross-reference	CRFSRT	5	GU
CEKSL	CEKSL1	Convert constant	CONCV	5	GT
CEKTA		Control compiler phases	PHC	Exec	AB
	CEKTAA	LPC to FORTRAN initial			
	CEKTAB	LPC to FORTRAN continue			
	CEKTAC	LPC to FORTRAN early-end			
CEKTC	CEKTCA	Get next source statement (contains CEKTI and CEKTJ)	GNSS	Exec	AC
CEKTD	CEKTDA	Process terminal modification	MOD	Exec	AD
CEKTE	CEKTEA	Receive diagnostic messages	RDM	Exec	AE
CEKTF		Constant filers	CONFIL	Exec	AF
	CEKTFB	File I*2 constants	CONI2		
	CEKTFC	File I*4 constants	CONI4		
	CEKTFD	File R*4 constants	CONR4		
	CEKTFE	File R*8 constants	CONR8		
	CEKTFF	File C*8 constants	CONC8		
	CEKTFG	File C*16 constants	CONC16		
	CEKTFI	File storage Class 4 other than R-cons	FLAD4		
	CEKTFJ	File storage Class 5 constants	FLAD5		
	CEKTFK	File V-con, R-con pairs	FLADVR		
	CEKTFL	File labels	FLL		
	CEKTFM	Create and file labels	CRL	AN	
CEKTH		Provide communication between interface programs (contains CEKTK, CEKTL, and CEKTM)	MIO	Exec	AG
	CEKTHA	Open the list data set	LDOPEN		AW
	CEKTHB	Close the list data set	LDCLOSE		
	CEKTHC	Obtain a line for GNSS	LINEIN		
	CEKTHD	Output diagnostic lines	DIAGOUT		
	CEKTHE	Add lines to list data set	OLR		
	CEKTHF	Flush a statement buffer	BFLUSH		
CEKTI	CEKTIA	Analyze console source line	ANALYZ	Exec	AH
CEKTJ	CEKTJA	Inspect a console character	INSCON	Exec	AI
CEKTK	CEKTKA	Move a line to a list data set	LDMOVE	Exec	AJ
CEKTL	CEKTLA	Build a list data set buffer	BUILD	Exec	AK
CEKTM	CEKTMA	Flush a list data set buffer	FLUSH	Exec	AL
CEKTQ	CEKTQ1	Prepare hexadecimal dumps of internal files	COMDUMP	Exec	None
CEKTS	CEKTSA	Form lines and issue them via PUT macro instruction	LINDUMP	Exec	None

FORTRAN ROUTINES LISTED BY MNEMONICS (Part 1 of 6)

Mnemonic	Entry Point	Purpose	Routine ID Label	Phase	Chart ID
AARG	CEKCR1	Provide service in processing actual argument	CEKCR	1	BT
ACOMP	CEKAE1	Assemble components	CEKAE	1	BW
AGEN	CEKNW1	Generate arithmetic expression	CEKNW	4	EM
AGO	CEKNQ1	Process assigned GO TO PF entry	CEKNQ	4	EA
AIF	CEKNK1	Process arithmetic IF PF entry	CEKNK	4	DX
ANALYZ	CEKTIA	Analyze console source line	CEKTI	Exec	AH
ANDOR	CEKMI1	Generate logical expression	CEKMI	4	FB
ARDIM	CEKAFA	Process array dimension specification	CEKAF	1	CK
ARITH	CEKCB1	Perform constant arithmetic	CEKCB	1	BR
ASAR	CEKMM1	Make initial assignment to General Register	CEKMM	4	GE
ASARS	CEKMN1	Assign to Arithmetic Register	CEKMN	4	GF
ASFR	CEKMO1	Make initial assignment to Floating-Point Register	CEKMO	4	None
ASFRS	CEKMP1	Make synonym assignment to Floating Register	CEKMP	4	GG
ASSGN	CEKNS1	Process ASSIGN PF entry	CEKNS	4	DZ
ASSI	CEKBCA	Process ASSIGN statement	CEKBC	1	AW
ASSIST	CEKSH1	Generate Internal Symbol Dictionary	CEKSH	5	GQ
BGNLP	CEKBA1	Analyzes and encodes begin loop information	CEKBA	1	CB
BLDA	CEKBI1	Process BLOCK DATA statement	CEKB1	1	BC
BLDA2	CEKBJ1	Sets the program type for the BLOCK DATA statement	CEKBJ	1	None
[illegible]	[illegible]	Process a [illegible] source statement	CEKBR	1	None
BL1	CEKNM1	Process begin loop 1 PF entry	CEKNM	4	EF
BL2	CEKNN1	Process begin loop 2 PF entry	CEKNN	4	EF
BL3	CEKNO1	Process begin loop 3 PF entry	CEKNO	4	EG
BUILD	CEKTI	[illegible] list data set buffer	CEKTL	Exec	K
BUILD	CEKSB1	Build object program module	CEKSB	5	GL
CALL	CEKBU1	Process CALL statement	CEKBU	1	None
CALL	CEKOL1	Process CALL statement	CEKOL	4	EC
CALL2	CEKBV1	Process CALL statement in Pass 2	CEKBV	1	BJ
CDIV	CEKOG1	Generate complex divide	CEKOG	4	EZ
CGO	CEKNR1	Process computed GO TO PF entry	CEKNR	4	EB
CHKINT		Provide for treatment of interrupts	CEKCS	1	BU
	CEKCS1	Divide check			
	CEKCS2	Exponent overflow			
	CEKCS3	Return flags			
CKLIM	CEKCJ1	Check loop limits	CEKCJ	1	CD
CLLIM	CEKCK1	Clear loop limits	CEKCK	1	None
CMSEC	CEKSC1	Generate common control section	CEKSC	5	GM
CMUL	CEKOF1	Generate complex multiply	CEKOF	4	EY
CNVRT		Conversion	CEKAN	1	BM
CNVRT	CEKAN1	Converts constants to new type and checks legal type mixes for expressions			
CNVRT2	CEKAN2	Converts constants to new type			
COMDUMP	CEKTQ	Dump Compiler file	CEKTQ	Exec	None
COMM	CEKAV1	Process COMMON statement	CEKAV	1	AT
COMMA		Process comma operator	CEKNJ	4	FE
	CEKNJ1	Entry for other cases			
	CEKNJ2	Entry for CEKMM when function has only one argument			

FORTRAN ROUTINES LISTED BY MNEMONICS (Part 2 of 6)

Mnemonic	Entry Point	Purpose	Routine ID Label	Phase	Chart ID
CONCV	CEKSL1	Convert constant	CEKSL	5	GT
CONFIL		Constant filers	CEKTF	Exec	AF
	CEKTFB	File I*2 constants			
	CEKTFC	File I*4 constants			
	CEKTFD	File R*4 constants			
	CEKTFE	File R*8 constants			
	CEKTFF	File C*8 constants			
	CEKTFG	File C*16 constants			
	CEKTFI	File storage Class 4 constants other than R-cons			
	CEKTFJ	File storage Class 5 constants			
	CEKTFK	File V-con, R-con pairs			
	CEKTFL	File Labels			
	CEKTFM	Create and file labels			
CONT	CEKAT1	Process CONTINUE statement	CEKAT	1	None
COSEC	CEKSF1	Generate code control section	CEKSF	5	GN
COVER	CEKOP1	Obtain adcon cover for generation of a storage reference	CEKOP	4	None
CPLUS	CEKMG1	Generate complex plus	CEKMG	4	EX
CRFSRT	CEKSK1	List cross-reference	CEKSK	5	GU
CSX	CEKOR1	Count common expression usage	CEKOB	4	EP
DATA	CEKBM1	Process DATA statement	CEKBM	1	BD
DCL2		Process declaration statements in Pass 2	CEKAW	1	
	CEKAW1	Entry for COMMON statements			None
	CEKAW2	Entry for other declaration statements			None
	DCOM	Control selection of open function processing	CEKOM	4	FF
	CEKOM2	OPEN6 function			
DIMN	CEKAUA	Process DIMENSION statement	CEKAU	1	AS
DO	CEKAZ1	Process DO statement	CEKAZ	1	AV
DX		Generate diagnostic messages	CEKJH	2	CU
	CEKJH1	Warning messages		2	
	CEKJH2	Serious messages		2	
	CEKJH3	Abort messages		2	
EDIT	CEKOQ1	Edit for code file	CEKOQ	4	None
EDIT	CEKSI1	Document object program	CEKSI	5	GR
END	CEKAL1	Process END statement	CEKAL	1	BG
ENDLP	CEKBB1	Encodes the end loop entries	CEKBB	1	CC
ENDLP	CEKNP1	Process and loop PF entry	CEKNP	4	EH
ENT	CEKOD1	Process entry point	CEKOD	4	DU
EQUA	CEKAK1	Process assignment statement	CEKAK	1	AN
EQUAT	CEKMJ1	Process equation PF entry	CEKMJ	4	DW
EQUI	CEKAYA	Process EQUIVALENCE statement	CEKAY	1	AU
ERR		Generate diagnostic messages	CEKCA	1	CP
	CEKCAA	Warning messages (error level 1)			
	CEKCAB	Serious messages (error level 2)			GO
	CEKCACC	Serious messages associated with statement deletion (error level 2)			
	CEKCAD	Fatal messages (error level 3)			
ESC		Obtain next character	CEKAB	1	BV
	CEKAB1	Get next non-blank character			
	CEKAB2	Get next character (including blanks)			

FORTRAN ROUTINES LISTED BY MNEMONICS (Part 3 of 6)

Mnemonic	Entry Point	Purpose	Routine ID Label	Phase	Chart ID
EXEC2		Process executable statements in Pass 2	CEKAX	1	BH
	CEKAX1	Entry for no-flow statements			
	CEKAX2	Entry for flow-thru statements			
EXPR	CEKAI1	Process expression	CEKAI	1	BL
EXTE	CEKAMA	Process EXTERNAL statement	CEKAM	1	AO
FALTH	CEKBQA	Determine if a label reference refers to the current statement	CEKBQ	1	CO
FCNV	CEKCP1	Convert decimal to floating binary	CEKCP	1	
FCON		Process file control statements	CEKBD	1	AX
	CEKBD1	Entry for BACKSPACE statement			
	CEKBD2	Entry for END FILE statement			
	CEKBD3	Entry for REWIND statement			
FETCH	CEKOS1	Ensure that each argument of a function or subroutine is in storage with desired sign	CEKOS	4	FP
FLABL	CEKCD1	Process format label for input/output statements	CEKCD	1	CF
FLIC	CEKCQ1	File integer constant	CEKCQ	1	None
FLRC	CEKCH1	File real and complex constants in Symbol Table	CEKCH	1	BX
FLUSH	CEKTMA	Flush a list data set buffer	CEKTM	Exec	AL
FLUSH	CEKON1	Clear register storage	CEKON	4	GJ
FNAME	CEKCF1	Process FORMAT or NAMELIST name	CEKCF	1	CH
FUNCLS	CEKBX1	Determine the proper class of a function	CEKBX	1	BP
FNDAR	CEKMR1	Search General Register	CEKMR	4	FX
FNDFR	CEKMS1	Search Floating Register	CEKMS	4	FY
FNDWS	CEKMT1	Find temporary storage	CEKMT	4	GH
FORM	CEKBF1	Process FORMAT statement	CEKBF	1	AZ
FORMAT	CEKJG1	Inspect FORMAT reference	CEKJG	2	None
FSCAN	CEKJB1	Process label references and definitions	CEKJB	2	CR
FUNC	CEKMK1	Generate external function	CEKMK	4	FD
GNSS	CEKTCA	Get next source statement (contains CEKTI and CEKTJ)	GEKTC	Exec	AC
GOTO	CEKAQA	Process GO TO statement	CEKAQ	1	AP
GOTO	CEKNT1	Process GO TO PF entry	CEKNT	4	None
ICNV	CEKCN1	Convert decimal to binary integer	CEKCN	1	BZ
IDATA		Process initial value data specificiation	CEKAH	1	CI
	CEKAH1	Entry for Type statements			
	CEKAH2	Entry for DATA statements			
IDVDE	CEKMD1	Generate integer divide	CEKMD	4	EV
IF	CEKARA	Process IF statement	CEKAR	1	AQ
IMPL	CEKBN1	Process IMPLICIT statement	CEKBN	1	BE
IMPLY	CEKME1	Generate integer multiply	CEKME	4	
IMPL2	CEKBP1	Perform final IMPLICIT statement housekeeping	CEKBP	1	None
INSCON	CEKTJA	Inspect a console character	CEKTJ	Exec	AI
INSOT	CEKNI1	Output code file	CEKNI	4	GK
	CEKNI2	Error processing for code file overflow			
IOLIST	CEKOI1	Process input/output list element PF entry	CEKOI	4	EJ

FORTRAN ROUTINES LISTED BY MNEMONICS (Part 4 of 6)

Mnemonic	Entry Point	Purpose	Routine ID Label	Phase	Chart ID
IOLST	CEKBW1	Analyze and encode list elements for READ, WRITE, PRINT, and PUNCH statements	CEKBW	1	CE
IPLUS	CEKMF1	Generate integer plus	CEKMF	4	ET
ISP	CEKJF1	Inspect statement label references	CEKJF	2	None
IVAL		Process initial value	CEKCL	1	
	CEKCL1	First value in type statement group			CJ
	CEKCL2	Other than first value in type statement group			AI
IVST	CEKCI1	Insert variable in Symbol Table	CEKCI	1	BY
KEY	CEKOC1	Determine the location of both operands of the current operation	CEKOC	4	FV
KEY1	CEKOR1	Locate single operand	CEKOR	4	FW
LAB	CEKJE1	Process label references	CEKJE	2	CT
LABEL	CEKNU1	Process referenced label PF entry	CEKNU	4	DV
LABL	CEKAJA	Process statement label	CEKAJ	1	CN
LADDR	CEKOV1	Add by load address	CEKOV	4	EW
LBL	CEKNV1	Generate labeled branch	CEKNV	4	FO
LBSTR	CEKCCA	Process label string	CEKCC	1	CL
LDMOVE	CEKTKA	Move a line to a list data set	CEKTK	Exec	AJ
LIBN		Select library functions	CEKBY	1	BQ
	CEKBY1	Functions with automatic typing			
	CEKBY2	Functions used as arguments			
	CEKBY3	Exponential library function selection			
LIF	CEKNL1	Process logical IF PF	CEKNL	4	DY
LINDUMP	CEKTSA	Dump compiler module entry	CEKTS	Exec	None
MAX	CEKMU1	Generate maximum operator	CEKMU	4	FC
MEMAC	CEKMV1	Access storage	CEKMV	4	FM
	CEKMV2				
MIO		Provide communication between interface programs (contains CEKTK, CEKTL and CEKTM)	CEKTH	Exec	AG
	CEKTHA	Open the list data set			
	CEKTHB	Close the list data set			
	CEKTHC	Obtain a line for GNSS			
	CEKTHD	Output diagnostic lines			
	CEKTHE	Add lines to list data set			
	CEKTHF	Flush a statement buffer			
MOD	CEKTDA	Process terminal modification	CEKTD	Exec	AD
NAML	CEKBH1	Process NAMELIST statement	CEKBH	1	BB
NDLST		Process and list PF entry	CEKOJ	4	
	CEKOJ1	Process an end list program file entry			EK
	CEKOJ2	Generate a standard call linkage			EK
OPEN1	CEKOT1	Process open function	CEKOT	4	FG
OPEN2	CEKOU1	Process open function	CEKOU	4	FH
OPEN3	CEKOX1	Process open function	CEKOX	4	FI
OPEN4	CEKOY1	Process open function	CEKOY	4	FJ
OPEN5	CEKOZ1	Process open function	CEKOZ	4	FK
CEKOM2	OPEN6	Process function numbers 3, 4, 6-11, 16, 21, 24, 25, 28-32	CEKOM2	4	FL

FORTRAN ROUTINES LISTED BY MNEMONICS (Part 5 of 6)

Mnemonic	Entry Point	Purpose	Routine ID Label	Phase	Chart ID
OPND	CEKMW1	Process operand	CEKMW	4	FZ
	CEKMW2	Operator processing			
	CEKMW3	Common expression operand processing			
PHAS4	CEKNX1	Control Phase 4	CEKNX	4	Fig.30
PHASE2	CEKJA1	Control Phase 2	CEKJA	2	Fig.19
PHASE5	CEKSA1	Generate FORTRAN compiler output	CEKSA	5	None
PHC		Control compiler phases	CEKTA	Exec	AB
	CEKTAA	LPC to FORTRAN initial			
	CEKTAB	LPC to FORTRAN continue			
	CEKTAC	LPC to FORTRAN early-end			
PHEAD	CEKSE1	Produce page headings for each Phase 5 output page	CEKSE	5	None
PHIM	CEKAD1	Phase 1 main loop	CEKAD	1	AM
PSR		Process PAUSE, STOP, and RETURN statements	CEKBG	1	BA
PAUS	CEKBG1	Entry for PAUSE statement			
STOP	CEKBG2	Entry for STOP statement			
RETU	CEKBG3	Entry for RETURN statement			
ESTOP	CEKBG4	Stop when execution flows into END			
PRSEC	CEKSG1	Build PSECT	CEKSG	5	GO
PSECT	-	Phase 3 storage	CEKKS	3	None
RD	CEKOH1	Process input/output statement PF entry	CEKOH	4	EI
RDIV	CEKMA1	Generate real divide	CEKMA	4	ES
RDM	CEKTEA	Receive diagnostic messages	CEKTE	Exec	AE
RLSWS	CEKMX1	Release temporary storage	CEKMX	4	GI
RLTNL	CEKMH1	Generate relational expression	CEKMH	4	FA
RMUL	CEKMB1	Generate real multiply	CEKMB	4	ER
RPLUS	CEEQC1	Generate real plus	CEEQC	4	KM
RSLT	CEKMY1	Process Result - Register operand	CEKMY	4	GA
RTN	CEKJD1	Process label references	CEKJD	2	CS
RTRAN	CEKCE1	Process ERR and END labels for input/output statements	CEKCE	1	CG
RTRN	CEKOE1	Process RETURN	CEKOE	4	ED
RWIO		Process input/output statements	CEKBE	1	AY
	CEKBE1	Entry for READ statement			
	CEKBE2	Entry for WRITE statement			
	CEKBE3	Entry for PRINT statement			
	CEKBE4	Entry for PUNCH statement			
SADDR	CEKMZ1	Generate local branch	CEKMZ	4	FN
SELDR	CEKNH1	Select even/odd General-Register pair	CEKNH	4	GC
SELFR	CEKMQ1	Select Floating Register	CEKMQ	4	GD
SELGD	CEKNA1	Determine whether division may take place in register pair containing numerator	CEKNA	4	FU
SELGM	CEKNB1	Determine availability of register for multiplication	CEKNB	4	FT
SELOP	CEKND1	Select operand	CEKND	4	FQ
SELSR	CEKNG1	Select single General Register	CEKNG	4	GB
SFDEF	CEKBK1	Enables EXPR to translate a statement function expression	CEKBK	1	BN
SFEXP		Expand statement functions	CEKBL	1	
	CEKBL1	Statement function expansion initialization			BO

FORTRAN ROUTINES LISTED BY MNEMONICS (Part 6 of 6)

Mnemonic	Entry Point	Purpose	Routine ID Label	Phase	Chart ID
	CEKBL2	Statement function expansion continuation			BO
SID	CEKAC1	Statement identification	CEKAC	1	CM
SLONE	CEKOW1	Select one operand in a register	CEKOW	4	FS
SLPOS	CEKNF1	Select position for operation	CEKNF	4	FR
SPECS	CEKSD1	Process preset data	CEKSD	5	GP
STFN2	CEKBZ1	Determine statement function in Pass 2	CEKBZ	1	None
STOP	CEKOK1	Process STOP and PAUSE statement PF entry	CEKOK	4	EL
SUBE		Process subprogram entry statements	CEKBS	1	BF
	CEKBS1	Process ENTRY statement			
	CEKBS2	Process FUNCTION statement			
	CEKBS3	Process SUBROUTINE statement			
SUBE2		Process subprogram entry statements in Pass 2	CEKBT	1	BI
	CEKBT1	Process ENTRY statement			
	CEKBT2	Process FUNCTION statement			
	CEKBT3	Process SUBROUTINE statement			
SUBS	CEKAG1	Process subscripts	CEKAG	1	AQ
SYMSRT	CEKSJ1	Sort Symbol Table	CEKSJ	5	GS
TRBLD	CEKML1	Build expression tree	CEKML	4	EN
	CEKML2	Generate tree entries for conversion function			
TRMPRO	CEKCG1	Process subscript term	CEKCG	1	BS
TYPE		Process Type statement	CEKAS	1	AR
	CEKAS1	Entry for INTEGER statements			
	CEKASR	Entry for REAL statements			
	CEKASC	Entry for COMPLEX statements			
	CEKASL	Entry for LOGICAL statements			
	CEKASD	Entry for DOUBLE PRECISION statements			
VSCAN	CEKJC1	Make storage assignments for all variables	CEKJC	2	CQ
WGHT	CEKNE1	Assign a weight to each non-primitive mode of the expression tree	CEKNE	4	EO

APPENDIX F: LINKAGE EDITED COMPILER ROUTINES LISTED BY CODED LABELS (MODULE NAMES)

Linkage Edited Routine ID Label (Module Name)	Description	Modules Included	Rename Information	
			External Symbol Definitions and References Prior to Linkage Editing	Names Following Linkage Editing
CEKWX* (EXECFTN)	Compiler Executive	CEKTA CEKTC CEKTD CEKTE CEKTF CEKTH CEKTQ CEKTS		
CEKW1* (PHASE1)	Compiler Phase 1	CEKAB CEKAC CEKAD CEKAE CEKAF CEKAG CEKAH CEKAI CEKAJ CEKAK CEKAL CEKAM CEKAN CEKAO CEKAR CEKAS CEKAT CEKAU CEKAV CEKAW CEKAX CEKAY CEKAZ CEKBA CEKBB CEKBC CEKBD CEKBE CEKBF CEKBG CEKBH CEKBI CEKBJ CEKBK CEKBL CEKBM CEKBN CEKBP CEKBQ CEKBR CEKBS CEKBT CEKBU CEKBV		

(Part 2 of 5)

Linkage Edited Routine ID Label (Module Name)	Description	Modules Included	Rename Information: External Symbol Definitions and References Prior To Linkage Editing	Rename Information: Names Following Linkage Editing
CEKW2 (Phase 23)	Compiler Phases 2 and 3	CEKBW CEKBX CEKBY CEKBZ CEKCA CEKCB CHCBMA CEKCC CEKCD CEKCE CEKCF CEKCG CEKCH CEKCI CEKCJ CEKCK CEKCL CEKCN CEKCP CEKCQ CEKCR CEKCS CEKJA CEKJB CEKJC CEKJD CEKJE CEKJF CEKJG CEKJH CEKKA CEKKB CEKKC CEKKE CEKKF CEKKG CEKKH CEKKI CEKKJ CEKKL CEKKM CEKKN CEKKO CEKKP CEKKU CEKKV CEKKW CEKLA CEKLB CEKLD CEKLE CEKLF CEKLI	CHCBGA CHCBIA CHCBKA CHCBKC	CEKUGA CEKUIA CEKUKA CEKUKC CEKUMA

(Part 3 of 5)

Linkage Edited Routine ID Label (Module Name)	Description	Modules Included	Rename Information	
			External Symbol Definitions and References Prior To Linkage Editing	Names Followiing Linkage Editing
CEKW4* (PHASE4)	Compiler Phase 4	CEKMA		
		CEKMB		
		CEKMC		
		CEKMD		
		CEKME		
		CEKMF		
		CEKMG		
		CEKMH		
		CEKMI		
		CEKMJ		
		CEKMK		
		CEKML		
		CEKMM		
		CEKMN		
		CEKMO		
		CEKMP		
		CEKMQ		
		CEKMR		
		CEKMS		
		CEKMT		
		CEKMU		
		CEKMV		
		CEKMW		
		CEKMX		
		CEKMY		
		CEKMZ		
		CEKNA		
		CEKNB		
		CEKND		
		CEKNE		
		CEKNF		
		CEKNG		
		CEKNH		
		CEKNI		
		CEKNJ		
		CEKNK		
		CEKNL		
		CEKNM		
		CEKNN		
		CEKNO		
		CEKNP		
		CEKNQ		
		CEKNR		
		CEKNS		
		CEKNT		
		CEKNU		
		CEKNV		
		CEKNW		
		CEKNX		
		CEKOB		
		CEKOC		
		CEKOD		
		CEKOE		
		CEKOF		
		CEKOG		
		CEKOH		
		CEKOI		

(Part 4 of 5)

Linkage Edited Routine ID Label (Module Name)	Description	Modules Included	Rename Information: External Symbol Definitions and References Prior To Linkage Editing	Rename Information: Names Following Linkage Editing
		CEKOJ		
		CEKOK		
		CEKOL		
		CEKOM		
		CEKON		
		CEKOP		
		CEKOQ		
		CEKOR		
		CEKOS		
		CEKOT		
		CEKOU		
		CEKOV		
		CEKOW		
		CEKOX		
		CEKOY		
		CEKOZ		
CEKW5* (CEKPH5)	Compiler Phase 5	CEKSA		
		CEKSB		
		CEKSC		
		CEKSD		
		CEKSE		
		CEKSF		
		CEKSG		
		CEKSH		
		CEKSI		
		CEKSJ		
		CEKSK		
		CEKSL		
CEKUX* (PHASE6)	Contains all mathematical library modules required by the compiler.	CHCAD	CHCADI	CEKUQI
			CHCADR	CEKUQR
			CHCADW	CEKUQW
			CHCBD4	CEKUD4
			CHCBD5* (BD005)	CEKBD5
			CHCBZA	CEKUZA
			DEXP	CEKUX1* (DEXPU)
		CHCAF	CHCAFR	CEKUFR
			CHCAFW	CEKUFW
			CHCBZA	CEKUZA
			DLOG	CEKUX2* (DLOGU)
			DLOG10	CEKUX3* (DLOG10U)
		CHCBD	CHCBDR	CEKUDR
			CHCBDW	CEKUDW
			CHCBD1	CEKUD1
			CHCBD2	CEKUD2
			CHCBD3	CEKUD3
			CHCBD4	CEKUD4
			CHCBD5	CEKUD5
			CHCBD5* (BD005)	CEKBD5
			CHCBE1	CEKUE1
			CHCBZA	CEKUZA

(Part 5 of 5)

Linkage Edited Routine ID Label (Module Name)	Description	Modules Included	Rename Information: External Symbol Definitions and References Prior To Linkage Editing	Rename Information: Names Following Linkage Editing
			DVCHK	CEKUX5 (DVCHKU)
			OVERFL	CEKUX4* (OVERFLU)
			SLITE	CEKUX6 (SLITEU)
			SLITET	CEKUX7* (SLITETU)
		CHCBE	CHCBER	CEKUER
			CHCBEW	CEKUEW
			CHCBE1	CEKUE1
		CHCBG	CHCBGA	CEKUGA
			CHCBGB	CEKUGB
			CHCBGC	CEKUGC
			CHCBGD	CEKUGD
			CHCBGR	CEKUGR
			CHCBGW	CEKUGW
			CHCBZA	CEKUZA
		CHCBI	CHCBIA	CEKUIA
			CHCBIB	CEKUIB
			CHCBIR	CEKUIR
			CHCBIW	CEKUIW
			CHCBZA	CEKUZA
		CHCBK	CHCBKA	CEKUKA
			CECBKB	CEKUKB
			CHCBKC	CEKUKC
			CHCBKD	CEKUKD
			CHCBKE	CEKUKE
			CHCBKR	CEKUKR
			CHCBKW	CEKUKW
			CHCBZA	CEKUZA
			DEXP	CEKUX1* (DEXPU)
			DLOG	CEKUX2* (DLOGU)
		CHCBM	CHCBMA	CEKUMA
			CHCBMB	CEKUMB
			CHCBMR	CEKUMR
			CHCBMW	CEKUMW
			CHCBZA	CEKUZA
		CHCBZ	CHCBZA	CEKUZA
			CHCBZR	CEKUZR
			CHCBZW	CEKUZW

*Names given in parentheses are temporary names and will be replaced by the preceding name as soon as is feasible. Thus, module EXECFTN will become module CEKWX, module PHASE6 will become module CEKUX, entry point DEXPU will become entry point CEKUX1, etc.

When more than one page reference is given, the major reference is first.

AARG
(see: Actual Argument Service Routine)
ACOMP
(see: Assemble Components)
Acquire Entry from Compute and Removal Table
decision table 104
flowchart 458
routine description 125
Actual Argument Service Routine
flowchart 334
routine description 71-72
Add by Load Address
decision table 145
flowchart 533-534
routine description 104-165
AGEN
(see: Arithmetic Expression Generator)
AGO
(see: Assigned GO to PF Entry Processor)
AIF
(see: Arithmetic IF PF Entry Processor)
Alphameric constant processing 44
ANALYZ
(see: Analyze Console Source Line)
Analyze Console Source Line
flowchart 240-241
overview 11
routine description 35
ANDOR
(see: Logical Expression Generator)
ARDIM
(see: Array Dimension Specification Processor)
Argument Definition Point Entry
in PF 643
in PRF 42
ARITH
(see: Constant Arithmetic Subroutine)
Arithmetic Expression Generator
decision table 144
flowchart 513-514
routine description 159
Arithmetic IF entry
in PF 643
in PRF 634,42
Arithmetic IF PF Entry Processor
decision table 142
flowchart 476-480
routine description 155
Array Dimension Specification Processor
flowchart 368-369
routine description 77
ASAR
(see: Make Initial Assignment to General Register)
ASARS
(see: Make Synonym Assignment to General Register)
ASFR
(see: Make Initial Assignment to Floating-Point Register)
ASFRS
(see: Make Synonym Assignment to Floating Register)
ASSGN
(see: Assign PF Entry Processor)
Assemble Components
character table 74
decision table 75,54
flowchart 337-344
routine description 73-74
ASSI
(see: ASSIGN statement processor)
ASSIGN entry
in PF 643
in PRF 634,41
ASSIGN PF Entry Processor
flowchart 482
routine description 155-156
ASSIGN Statement Processor
decision table 142
flowchart 269-270
routine description 57-58
Assigned GO TO entry
in PF 643
in PRF 634,41
Assigned GO TO PF Entry Processor
decision table 142
flowchart 483
routine description 156
Assignment Character Table 78
Assignment Precedence Table 78
Assignment Statement Processor
flowchart 250
routine description 55-56
ASSIST
(see: Internal Symbol Dictionary Generator)

BACKSPACE entry in PRF 43
Begin Loop Processor
flowchart 349-350
routine description 75
Begin Loop 1 entry
in PF 644
in PRF 634,42
Begin Loop 1 PF Entry Processor
decision table 143
flowchart 489
routine description 156-157
Begin Loop 1 PRF Processor
decision table 103
flowchart 421-423
routine description 113-114
Begin Loop 2 Entry
in PF 644
in PRF 634

Begin Loop 2 PF Entry Processor
decision table 143
flowchart 490-498
routine description 157
Begin Loop 2 PRF Processor
decision table 103
flowchart 424-426
routine description 114-115
Begin Loop 3 Entry
in PF 644
in PRF 634
Begin Loop 3 PF Entry Processor
decision table 143
flowchart 499-501
routine description 157
Begin Program Entry in PRF 633,41
BGNLP
(see: Begin Loop Processor)
Blank Statement Processor
decision table 51
routine description 60
BLDA
(see: BLOCK DATA Statement Processor)
BLDA2
(see: BLOCK DATA Statement, Pass 2)
BLNK
(see: Blank Statement Processor)
BLOCK DATA Statement Processor
decision table 50,51
flowchart 291
routine description 60-62
BL1
(see: Begin Loop 1 PF Entry Processor)
BL2
(see: Begin Loop 2 PF Entry Processor)
BL3
(see: Begin Loop 3 PF Entry Processor)
Build a List Data Set Buffer
flowchart 244
overview 11
routine description 36
BUILD
(see: Object Program Module Builder, Build a List Data Set Buffer)

CALL
(see: CALL Statement Processor)
CALL Entry in PF 643
Call
function 67-68
subroutine 67-68
CALL Entry in PRF 634,42
CALL Statement Final Processing
decision table 51
CALL Statement, Pass 2
flowchart 304
routine description 62
CALL Statement Processor
decision table 51,142
flowchart 485
routine description 62,156
CALL2
(see: CALL Statement Pass 2)
Canonical Form Routine
decision table 105
flowchart 443-444
routine description 121
CDIV
(see: Complex Divide Generator)
CEKAB
(see: Extract Source Character)
CEKAC
(see: Statement of Identification)
CEKAD
(see: Phase 1 Main Loop)
CEKAE
(see: Assemble Components)
CEKAF
(see: Array Dimension Specification Processor)
CEKAG
(see: Subscript Processor)
CEKAH
(see: Initial Value Data Specification Processor)
CEKAI
(see: Expression Processor)
CEKAJ
(see: Statement Label Processor)
CEKAK
(see: Assignment Statement Processor)
CEKAL
(see: END Statement Processor)
CEKAM
(see: EXTERNAL Statement Processor)
CEKAN
(see: Conversion subroutine)
CEKAQ
(see: GO TO Statement Processor)
CEKAR
(see: IF Statement Processor)
CEKAS
(see: Type Statements Processor)
CEKAT
(see: CONTINUE Statement Processor)
CEKAU
(see: DIMENSION Statement Processor)
CEKAV
(see: COMMON Statement Processor)
CEKAW
(see: Declaration Statement, Pass 2)
CEKAX
(see: Executable Statements, Pass 2)
CEKAY
(see: EQUIVALENCE Statement Processor)
CEKAZ
(see: DO Statement Processor)
CEKBA
(see: Begin Loop Processor)
CEKBB
(see: End Loop Processor)
CEKBC
(see: ASSIGN Statement Processor)
CEKBD
(see: File Control Statement Processor)
CEKBE
(see: Input/Output Statement Processor)
CEKBF
(see: FORMAT Statement Processor)
CEKBG
(see: PAUSE, STOP, RETURN Statement Processor)
CEKBH
(see: NAMELIST Statement Processor)
CEKBI

(see: BLOCK DATA Statement Processor)
CEKBJ
(see: BLOCK DATA Statement, Pass 2)
CEKBK
(see: Statement Function Definition)
CEKBL
(see: Statement Function Expansion)
CEKBM
(see: DATA Statement Processor)
CEKBN
(see: IMPLICIT Statement Processor)
CEKBP
(see: IMPLICIT Statements, Pass 2)
CEKBQ
(see: Fall Through Determination)
CEKBR
(see: Blank Statement Processor)
CEKBS
(see: Subprogram Entry Statements Processor)
CEKBT
(see: Subprogram Entry Statements, Pass 2)
CEKBU
(see: CALL Statement Processor)
CEKBV
(see: CALL Statement, Pass 2)
CEKBW
(see: I/O List Processor)
CEKBX
(see: Function Classifier)
CEKBY
(see: Library Function Selector)
CEKBZ
(see: Statement Function Definition, Pass 2)
CEKCB
(see: Constant Arithmetic Subroutine)
CEKCC
(see: Label String Processor)
CEKCD
(see: Format Label Processor for I/O Statements)
CEKCE
(see: Read Transfer Processor for I/O Statements)
CEKCF
(see: FORMAT or NAMELIST Name Processor)
CEKCG
(see: Term Processor)
CEKCH
(see: File Real Constant)
CEKCI
(see: Insert Variable in Symbol Table)
CEKCJ
(see: Check Limits)
CEKCK
(see: Clear Limits)
CEKCL
(see: Initial Value Processor)
CEKCN
(see: Decimal to Binary Integer Conversion)
CEKCP
(see: Decimal to Floating Binary Conversion)
CEKCQ
(see: File Integer Constant)
CEKCR
(see: Actual Argument Service Routine)
CEKCS
(see: Constant Arithmetic Interrupt)
CEKHB macro instruction 663
CEKHC macro instruction 663
CEKHD macro instruction 663
CEKHF macro instruction 663
CEKHG macro instruction 663
CEKHH macro instruction 663
CEKHI macro instruction 663
CEKHJ macro instruction 663
CEKHL macro instruction 663
CEKHM macro instruction 663
CEKHN macro instruction 663
CEKJA
(see: PHASE2)
CEKJB
(see: Process Label References and Definitions)
CEKJC
(see: Storage Assignments for Variables)
CEKJD
(see: Label Reference Processor)
CEKJE
(see: Label Reference Processor)
CEKJF
(see: Statement Label Reference Inspection)
CEKJG
(see: Format Reference Inspection)
CEKJH
(see: Diagnostic Message Generator)
CEKJM macro instruction 663
CEKJO macro instruction 663
CEKJ1 macro instruction 663
CEKJ2 macro instruction 663
CEKJ3 macro instruction 663
CEKJ4 macro instruction 663
CEKJ5 macro instruction 663
CEKJ6 macro instruction 663
CEKJ7 macro instruction 663
CEKJ8 macro instruction 663
CEKJ9 macro instruction 663
CEKKA
(see: Acquire Entry from Compute and Removal Table)
CEKKB
(see: Polish Expression Generation Routine)
CEKKC
(see: End Loop PRF Entry Routine)
CEKKD macro instruction 663
CEKKE
(see: Expression Scan Routine)
CEKKF
(see: Pushdown Primitive Operand Routine)
CEKKG
(see: Variable Compute Point and Removal Level Routine)
CEKKH
(see: Triad File Manipulation Routine)
CEKKI
(see: Expression Removal and Commonality Determination Routine)

CEKKJ
(see: Check Commonality)
CEKKK
(see: Establish Common Expression Routine)
CEKKK macro instruction 663
CEKKL
(see: Operand List Expression Formation Routine)
CEKKM
(see: Subscript Expression Revision Routine)
CEKKN
(see: Canonical Form Routine)
CEKKO
(see: Save Popularity Counts for Register Assignment)
CEKKP
(see: Search and Insert Triads)
CEKKR
(see: Phase 3 Master Control Routine)
CEKKS
(see: Phase 3 storage PSECT)
CEKKS macro instruction 663
CEKKU
(see: PRF Processing Routine)
CEKKV
(see: Begin Loop 1 PRF Processor)
CEKKW
(see: Begin Loop 2 PRF Processor)
CEKLA
(see: Label Common Expressions)
CEKLB
(see: File Constant and Covering Adcon)
CEKLD
(see: Expunge a Removable Expression)
CEKLE
(see: File CRT Entries)
CEKLF
(see: Copy and Edit an Expression)
CEKLI
(see: Loop Test-Expression Generator)
CEKMA
(see: Real Divide Generator)
CEKMB
(see: Real Multiply Generator)
CEKMC
(see: Real Plus Generator)
CEKMD
(see: Integer Divide Generator)
CEKME
(see: Integer Multiply Generator)
CEKMF
(see: Integer Plus Generator)
CEKMG
(see: Complex Plus Generator)
CEKMH
(see: Relational Expression Generator)
CEKMI
(see: Logical Expression Generator)
CEKMJ
(see: Equation PF Entry Processor)
CEKMK
(see: External Function Generator)
CEKML
(see: Expression Tree Builder)
CEKMM
(see: Make Initial Assignment to General Register)
CEKMN
(see: Make Synonym Assignment to General Register)
CEKMO
(see: Make Initial Assignment to Floating-Point Register)
CEKMP
(see: Make Synonym Assignment to Floating Register)
CEKMQ
(see: Select Floating Register)
CEKMR
(see: Search General Registers)
CEKMS
(see: Search Floating Registers)
CEKMT
(see: Find Temporary Storage)
CEKMU
(see: Maximum Operator Generator)
CEKMV
(see: Memory Access Routine)
CEKMW
(see: Operand Processing Routine)
CEKMX
(see: Release Temporary Storage)
CEKMY
(see: Result-Register Operand Processing Subroutine)
CEKMZ
(see: Local Branch Generator)
CEKNA
(see: General Register Availability for Integer Divide)
CEKNB
(see: Determine Availability of Register for Multiplication)
CEKND
(see: Select Operand Routine)
CEKNE
(see: Weight Subroutine)
CEKNF
(see: Select Position for Operation)
CEKNG
(see: Select Single General Register)
CEKNH
(see: Select Even/Odd General Register Pair)
CEKNI
(see: Code File Output Subroutine)
CEKNJ
(see: Comma Operator Processing Subroutine)
CEKNK
(see: Arithmetic IF PF Entry Processor)
CEKNL
(see: Logical IF PF Entry Processor)
CEKNM
(see: Begin Loop 1 PF Entry Processor)
CEKNN
(see: Begin Loop 2 PF Entry Processor)
CEKNO
(see: Begin Loop 3 PF Entry Processor)
CEKNP
(see: End Loop PF Entry Processor)
CEKNQ
(see: Assigned GO TO PF Entry Processor)

CEKNR
(see: Computed GO TO PF Entry Processor)
CEKNS
(see: Assign PF Entry Processor)
CEKNT
(see: GO TO PF Entry Processor)
CEKNU
(see: Referenced Label PF Entry Processor)
CEKNV
(see: Labeled Branch Generator)
CEKNW
(see: Arithmetic Expression Generator)
CEKNX
(see: Phase 4 Master Control)
CEKNY macro instruction 663
CEKN2 macro instruction 663
CEKN3 macro instruction 663
CEKN4 macro instruction 663
CEKN5 macro instruction 663
CEKOB
(see: Common Expression Usage Count)
CEKOC
(see: Operand Status Routine)
CEKOD
(see: Entry Point Processor)
CEKOE
(see: RETURN Processor)
CEKOF
(see: Complex Multiply Generator)
CEKOG
(see: Complex Divide Generator)
CEKOH
(see: I/O Statement PF Entry Processor)
CEKOI
(see: I/O List Element PF Entry Processor)
CEKOJ
(see: End List PF Entry Processor)
CEKOK
(see: STOP and PAUSE Statement PF Entry Processor)
CEKOL
(see: CALL Statement Processor)
CEKOM
(see: Open Function Control Routine)
CEKOM2
(see: Open Function Processing Routine)
CEKON
(see: Register Storage Clear Routine)
CEKOP
(see: Load covering Adcon Routine)
CEKOQ
(see: Edit for Code File)
CEKOR
(see: Single Operand Locating Routine)
CEKOS
(see: Operand Fetch Complement/Store Routine)
CEKOT
(see: Open Function Processing Routine)
CEKOU
(see: Open Function Processing Routine)
CEKOV
(see: Add by Load Address)
CEKOW
(see: Select One Operand in a Register)
CEKOX
(see: Open Function Processing Routine)
CEKOY
(see: Open Function Processing Routine)
CEKOZ
(see: Open Function Processing Routine)
CEKSA
(see: FORTRAN Compiler Output Generator)
CEKSB
(see: Object Program Module Builder)
CEKSC
(see: Common Control Section Generator)
CEKSD
(see: Preset Data Processor)
CEKSE
(see: Output Page Heading)
CEKSF
(see: Code Control Section Generator)
CEKSG
(see: PSECT Builder)
CEKSH
(see: Internal Symbol Dictionary Generator)
CEKSI
(see: Object Program Documentation)
CEKSJ
(see: Symbol Table Sort)
CEKSK
(see: Cross Reference List Routine)
CEKSL
(see: Constant Conversion)
CEKSY macro instruction 663
CEKSZ macro instruction 663
CEKS2 macro instruction 663
CEKS3 macro instruction 663
CEKTA
(see: Phase Controller)
CEKTAA
(see: LPC to FORTRAN Initial)
CEKTAB
(see: LPC to FORTRAN Continue)
CEKTAC
(see: LPC to FORTRAN Early End Call)
CEKTC
(see: Get Next Source Statement)
CEKTD
(see: Process Terminal Modification)
CEKTE
(see: Receive Diagnostic Message)
CEKTF
(see: Constant Filers)
CEKTG macro instruction 662
CEKTH
(see: Master Input/Output)
CEKTI
(see: Analyze Console Source Line)
CEKTJ
(see: Inspect a Console Character)
CEKTK
(see: Move a Line to the List Data Set)
CEKTL
(see: Build the List Data Set Buffer)
CEKTO macro instruction 602
CEKTQ
(see: Compiler Dump)
CEKTS
(see: Dump Line Preparation and Output)

CEKTT macro instruction 662
CEKTX macro instruction 662
CEKTY macro instruction 662
CEKT7 macro instruction 662
CEKT8 macro instruction 662
CEKT9 macro instruction 662
CEKUY macro instruction 662
CEKU1 macro instruction 662
CEKU3 macro instruction 662
CEKU4 macro instruction 662
CEKU5 macro instruction 662
CEKU6 macro instruction 662
CEKU7 macro instruction 662
CEKU8 macro instruction 662
CEKU9 macro instruction 662
CEKVA macro instruction 663
CEKVB macro instruction 663
CEKVC macro instruction 663
CEKVD macro instruction 663
CEKVI macro instruction 663
CEKVU macro instruction 663
CEKV1 macro instruction 662
CEKV2 macro instruction 662
CEKV3 macro instruction 662
CEKV5 macro instruction 662
CEKV6 macro instruction 662
CEKV7 macro instruction 662
CEKV8 macro instruction 662
CEKV9 macro instruction 662
CEKZB macro instruction 663
CEKZD macro instruction 663
CGO
(see: Computed GO TO PF Entry Processor)
Check Commonality
decision table 105
flowchart 451
routine description 122-123
Check for Arithmetic Interrupt during Expression Processing 53
Check Limits
decision table 54
flowchart 352
routine description 76
CHKINT
(see: Constant Arithmetic Interrupt)
CKLIM
(see: Check Limits)
Classify Function Name 53
Clear Limits
decision table 54
routine description 76
CLLIM
(see: Clear Limits)
CMSEC
(see: Common Control Section Generator)
CMUL
(see: Complex Multiply Generator)
CNVRT
(see: Conversion Subroutine)
Code Control Section Generator
decision table 197
flowchart 614
routine description 200-203
Code File 645
Code File Output Subroutine
decision table 152
flowchart 611
routine description 184
COMDUMP
(see: Compiler Dump)
COMM
(see: COMMON statement processor)
Comma Operator Processing Subroutine
decision table 147
flowchart 547-549
routine description 169-170
COMMA
(see: Comma Operator Processing Subroutine)
Common Control Section Generator
decision table 197
flowchart 613
routine description 199-200
Common Expressions 96
Common-Expression Storage 132
Common Expression Usage Count
decision table 144
flowchart 518-520
routine description 162
COMMON Statement Processor
decision table 49
flowchart 263-265
routine description 57
compiler origanization 3
compiler executive 8,2
description 8,2
flowchart 216-217
Compiler File Dump 36
Compiler Line Dump 36-38
Compiler Dump 11
Complex Divide Generator
decision table 147
flowchart 540-541
routine description 166
Complex Multiply Generator
decision table 146
flowchart 538-539
routine description 165-166
Complex Plus Generator
decision table 145
flowchart 535-537
routine description 165
Compute and Remove Item Table 108,109
Computed GO TO entry
in PF 643
in PRF 634,41
Computed GO TO PF Entry Processor
decision table 142
flowchart 484
routine description 156
CONCV
(see: Constant Conversion)
CONFIL
(see: Constant Filers)
Constant Arithmetic Interrupt
decision table 54
flowchart 335
routine description 72
Constant Arithmetic Subroutine
flowchart 332
routine description 71
Constant Conversion
decision table 197
flowchart 624
overview 68-69

routine description 212
Constant Filers (CONFIL)
decision table 20
flowchart 230-237
overview 10
routine description 28-32
CONT
(see: CONTINUE Statement Processor)
CONTINUE entry
in PF 644
in PRF 635,42
CONTINUE entry to Compiler Executive 2
CONTINUE Statement Processor
decision table 49
routine description 56-57
Control Section Dictionary 189-191,1
conversion
decimal to binary integer 74
decimal to floating binary 74
Conversion Subroutine
decision table 54
flowchart 324-326
routine description 68-69
Copy and Edit on Expression
decision table 104
flowchart 430-434
routine description 116-117
COSEC
(see: Code Control Section Generator)
COVER
(see: Load Covering Adcon Routine)
CPLUS
(see: Complex Plus Generator)
CRFSRT
(see: Cross Reference List Routine)
Cross Reference Index List 45
Cross Reference List Routine
decision table 197
flowchart 625-629
routine description 212-213
CSD
(see: control section dictionary)
CSX
(see: Common Expression Usage Count)

DATA
(see: DATA Statement Processor)
data management interface 2
DATA statement processor
decision table 50
flowchart 292
overview 44
routine description 60
DCL2
(see: Declaration Statements, Pass 2)
DCOM
(see: Open Function Control Routine)
Decimal to Binary Integer Conversion
decision table 55
flowchart 347
routine description 74
Decimal to Floating Binary Conversion
decision table 55
flowchart 348
routine description 74
Declaration statements final processing 51
Declaration statements, Pass 2 61
Definition table 191-192
Determine Availability of Register for Multiplication
decision table 151
flowchart 588-589
routine decription 176-177
Delete the Undefined Level Subroutine 115
Determine Fall-Through on GO TO and IF Statements 53
diagnostic information 17-19,21
Diagnostic Message Generator
decision table 83
flowchart 375-378,401-402
routine description 91
DIMENSION Statement Processor
decision table 49
flowchart 262
routine description 57
Dimension Table 44
DIMN
(see: DIMENSION statement processor)
DO
(see: DO statement processor)
DO loop processing 136
DO Statement Processor
decision table 50
flowchart 268
routine description 57
documentation, object program 2
Dump Line Preparation and Output 11
DUNL Subroutine 115
DX
(see: Diagnostic Message Generator)

EDIT
(see: Edit for Code File; Object Program Documentation)
EDIT for Code File
decision table 151
routine description 184-185
edit lines 17,16
EF
(see: Expression File)
EKKSB macro instruction 663
EKKSC macro instruction 663
END
(see: END Statement Processor)
END FILE entry in PRF 43
End List entry
in PF 644
in PRF 635,43
End List PF Entry Processor
decision table 144
flowchart 511
routine description 158-159
End Loop entry
in PF 644
in PRF 635,42
End Loop PF Entry Processor
decision table 103
flowchart 502-507
routine description 157-158
End Loop PRF Entry Routine
flowchart 419-420
routine description 113
End Loop Processor
decision table 143

flowchart 351
routine description 75-76
ENDLP
(see: End Loop Processor)
End Program Entry
in PF 645
in PRF 636,43
END statement processor
decision table 51
flowchart 301
routine description 61
ENDLP
(see: ENDLOOP PF Entry Processor)
ENT
(see: Entry Point Processor)
Entry Point Processor
decision table 141
flowchart 472-473
routine description 153
EQUA
(see: Assignment Statement Processor)
EQUAT
(see: Equation PF Entry Processor)
Equation entry
in PF 642
in PRF 633,41
Equation PF Entry Processor
decision table 141
flowchart 475
routine description 155
Equation Statement Processor 49
EQUI
(see: EQUIVALENCE Statement Processor)
EQUIVALENCE Statement Processor
decision table 49
flowchart 266-267
routine description 58
ERF
(see: Expression File)
ESC
(see: Extract Source Character)
Establish Common Expression Routine
decision table 105
flowchart 450
routine description 123
EXCOM
(see: Phase Controller PSECT)
EXEC
(see: Compiler Executive)
EXEC2
(see: Executable Statement, Pass 2)
Executable statements final processing 52
Executable Statements, Pass 2
flowchart 302
routine description 61-62
Executive
flowchart 216-217
routine description 8,2
Expand Statement Function Reference 53
EXPF entries 69
Explicit Type Statement Processor 49
EXPR
(see: Expression Processor)
Expression File 43,640-641
Expression Processor
flowchart 311-323
routine description 64-68
Expression Removal and Commonality Determination Routine
decision table 104
flowchart 445-449
routine description 121-122
Expression Scan Routine
decision table 103
flowchart 427-429
routine description 115-116
Expression Storage 132
Expression Tree 131
Expression Tree builder
flowchart 515-516
routine description 159-161
Expunge a Removable Expression
decision table 105
flowchart 454
routine description 124
EXTE
(see: EXTERNAL Statement Processor)
External Function Generator
decision table 147
flowchart 544-546
routine description 168
EXTERNAL statement processor
decision table 49
flowchart 251
routine description 56
Extract Source Character
decision table 54
flowchart 336
routine description 72-73

Fallthrough Determination
flowchart 374
routine description 79
FALTH
(see: Fallthrough Determination)
FCNV
(see: Decimal to Floating Binary Conversion)
FCON
(see: File Control Statement Processor)
FEEP Subroutine 115
FETCH
(see: Operand Fetch Complement/Store Routine)
File Constant and Covering Adcon
flowchart 465
File Constant and Covering Adcon
decision table 105
flowchart 465
routine description 126-127
File Control entry
in PF 644
in PRF 635
File Control Statement Processor
flowchart 271-272
routine description 58
File CRT Entries
decision table 105
flowchart 453
routine description 123-124
File EF and Point Subroutine 115
File Integer Constant
decision table 54
routine description 74-75

File Real Constant
decision table 54
flowchart 345
routine description 74
Find Temporary Storage
decision table 152
flowchart 608
routine description 183
FLABL
(see:Format Label Processor for I/O Statements)
FLIC
(see: File Integer Constant)
FLRC
(see: File Real Constant)
FLUSH
(see: Flush the List Data Set Buffer; Register Storage Clear Routine)
Flush the List Data Set Buffer
decision table 152
flowchart 245
overview 11
routine description 36
FNAME
(see: FORMAT or NAMELIST Name Processor)
FNCLS
(see: Function Classifier)
FNDAR
(see: Search General Registers)
FNDFR
(see: Search Floating Registers)
FNDWS
(see: Find Temporary Storage)
FORM
(see: FORMAT Statement Processor)
Formal Argument Adcon Table 109
FORMAT
(see: Format Reference Inspection)
Format Label Processor for I/O Statements
flowchart 357
routine description 76
FORMAT or NAMELIST Name Processor
flowchart 359
routine description 76
Format processing 44
FORMAT Reference Inspection
decision table 83
routine description 91
FORMAT Statement Processor
decision table 50
flowchart 279-284
routine description 58-59
FORMAT Symbols
encoding of 58-59
translation of 59
FORTRAN Compiler Output Generator
decision table 196
routine description 195
FSCAN
(see: Process Label References and Definitions)
FUNC
(see: External Function Generator)
function call 67-68
Function Classifier
flowchart 330
routine description 70
General Register Availability for Integer Divide
decision table 150
flowchart 590
routine description 177
Generate End Loop 53
Get Next Sources Statement
flowchart 223-226
overview 9
routine description 24-26
GIRL 106-107
global register assignment 101
Global Register list 106-107
General Register Selection 134
GNSS
(see: Get Next Source Statement)
GOTO
(see: GO TO PF Entry Processor: GO TO Statement Processor)
GO TO entry in PF
Assigned 643
Computed 643
Unconditional 643
GO TO Entry in PRF
Assigned 634
Computed 634
Unconditional 633
GO TO PF Entry Processor 155
GO TO Statement Processor
decision table 49,142
flowchart 252-255
routine description 56

I/O library subprograms 2
I/O List Element PF Entry Processor
decision table 143
flowchart 509-510
routine description 158
I/O List Entry in PRF 43
I/O List Processor
decision table 143
flowchart 353-356
routine description 76
I/O Statement PF Entry Processor
flowchart 508
routine description 158
I/O statement processor
decision table 50
flowchart 273-278
routine description 58
I/O statement processor routines 45
ICNV
(see: Decimal to Binary Integer Conversion)
IDATA
(see: Initial Value Data Specification Processor)
IDVDE
(see: Integer Divide Generation)
IF
(see: IF statement processor)
IF entry in PF
arithmetic 643
logical 643
IF Entry in PRF
Arithmetic 634
Logical 634

IF statement processor
decision table 49
flowchart 256-257
routine description 56
IMPL
(see: IMPLICIT Statement Processor)
IMPLICIT statement final processing 51
IMPLICIT Statement Processor
decision table 50
flowchart 293-295
routine description 60
IMPLICIT Statements, Pass 2 62
IMPLY
(see: Integer Multiply Generator)
IMPL2
(see: IMPLICIT Statements, Pass 2)
INITIAL entry to Compiler Executive 2
Initial Value Data Specification Processor
flowchart 360-361
routine description 77
Initial Value Processing routines 45
Initial Value Processor
flowchart 362-367
routine description 77
Initialize for Statement Function Definition 53
Input List Entry
in PF 645
in PRF 636
input/output
(see I/O)
INSCON
(see: Inspect a Console Character)
Insert Variable in Symbol Table
decision table 55
flowchart 346
routine description 74
INSOT
(see: Code File Output Subroutine)
Inspect a Console Character
flowchart 242
overview 11
routine description 35
INTBLE format 87
Integer Divide Generator
decision table 146-147
flowchart 532
routine description 164
Integer Multiply Generator
decision table 146
flowchart 529-531
routine description 164
Integer Plus Generator
decision table 145
flowchart 528
routine description 163-164
Intercom Table 650
Internal Symbol Dictionary 194-195,2
Internal Symbol Dictionary Generator
decision table 197
flowchart 619
routine description 206-207
Interphase file controller 12-13,14
INVOKE macro instruction 659-660
IOLST
(see: I/O List Processor)
IPLUS
(see: Interger Plus Generator)
ISP
(see: Statement Label Reference Inspection)
ISD
(see: internal symbol dictionary)
IVAL
(see: Initial Value Processor)
IVST
(see: Insert Variable in Symbol Table)

KEY
(see: Operand Status Routine)
KEY1
(see: Single Operand Locating Routine)

LAB
(see: Label Reference Processor)
Label Common Expression
decision table 105
flowchart 452
routine description 123
Label Definition Entry in PRF 41
Label Entry in PF 642
Label Reference Processor
decision table 83
flowchart 398-400
routine description 90
Label String Processor
flowchart 370
routine description 77
LABEL
(see: Referenced Label PF Entry Processor)
Labeled Branch Generator
decision table 150
flowchart 378
routine description 173
LABL
(see: Statement Label Processor)
LADDR
(see: Add by Load Address)
language processor controller 2
LBL
(see: Labeled Branch Generator)
LBSTR
(see: Label String Processor
LDMOVE
(see: Move a Line to the List Data Set)
LDPNT macro instruction 663
LIBN
(see: Library Function Selector)
Library Function Names 71
Library Function Selector
decision table 54
flowchart 331
routine description 70-71
LIF
(see: Logical IF PF Entry Processor)
LINDUMP
(see: Dump Line Preparation and Output)
linkage conventions 657,8
Load Covering Adcon Routine
decision table 151
routine description
Local Branch Generator
decision table 151
flowchart 577
routine description 172-173

Logical Expression Generator
decision table 148
routine description 167
Logical IF entry
in PF 643
in PRF 634,42
Logical IF PF Entry Processor
decision table 142
flowchart 481
routine description 155
loop
table 137
processing routines 25
removing expressions from 98
variable expressions 100
Loop Test-Expression Generator
decision table 104
flowchart 466-471
routine description 127-128
LPC
(see: language processor controller)
LPC to FORTRAN Continue 22-23
LPC to FORTRAN Early End Call 23-24
LPC to FORTRAN Initial 20,22

macro instructions 662-663,8
Make Initial Assignment to Floating-Point Register
decision table 151
routine description 182
Make Initial Assignment to General Register
descision table 152
flowchart 605
routine description 181-182
Make Synonym Assignment to Floating Register
decision table 144
flowchart 607
routine description 182-183
Make Synonym Assignment to General Register
decision table 144
flowchart 606
routine description 182
Master Input/Output Routine
flowchart 238-239
overview 10
routine description 32-34
mathematical subprograms 2
MAX
(see: Maximum Operator Generator)
Maximum Operator Generator
decision table 148
flowchart 543
routine description 167-168
MEMAC
(see: Memory Access Routine)
Memory Access Routine
decision table 151
flowchart 575-576
routine description 172
Memory Assignments for Variables
flowchart 379-389
Memory Reference Processing 96
MIO
(see: Master Input/Output)
MOD
(see: Processor Terminal Modification)

Move a Line to a List Data Set
flowchart 243
overview 11
routine description 35-36
MRM Table 133
MRMFR Table 134

Name Table 133
NAML
(see: NAMELIST Statement Processor)
NAMELIST Name Processor
(see: FORMAT or NAMELIST Name Processor)
NAMELIST statement processor
decision table 50
flowchart 289-290
routine description 60
NAMELIST Table 44
NDLST
(see: End List PF Entry Processor)
Nonassignment
Character Table 78
Precedence Table 78
Type Statement Identification 79

Object Program Documentation
decision table 197
flowchart 620-622
overview 2
routine description 207-211
Object program module 187,1
Object Program Module Builder
decision table 197
flowchart 612
routine description 198-199
OLIST
(see: I/O List Element PF Entry Processor)
Open Function Control Routine
decision table 148
flowchart 550
routine description 170
Open Function Processing Routine (OPEN1)
decision table 149
flowchart 551-556
routine description 170
Open Function Processing Routine (OPEN2)
decision table 149
flowchart 557-559
routine description 171
Open Function Processing Routine (OPEN3)
decision table 149
flowchart 560-562
routine description 171
Open Function Processing Routine (OPEN4)
decision table 149
flowchart 563-566
routine description 171
Open Function Processing Routine (OPEN5)
decision table 150
flowchart 567-572
routine description 171
Open Function Processing Routine (OPEN6)
decision table 150
flowchart 573-574
routine description 171-172

OPEN1
(see: Open Function Processing Routine)
OPEN2
(see: Open Function Processing Routine)
OPEN3
see: Open Function Processing Routine)
OPEN4
(see: Open Function Processing Routine)
OPEN5
(see: Open Function Processing Routine)
OPEN6
(see: Open Function Processing Routine)
Operand Fetch Complement/Store Routine
decision table 150
flowchart 579
routine description 173
Operand List Expression Formation Routine
decision table 104
flowchart 439
routine description 119-120
Operand Processing Routine
decision table 151
flowchart 595-596
routine description 178-179
Operand Status Routine
decision table 151
flowchart 591
routine description 177
Operator precedence 66
OPM
(see: object program module)
OPND
(see: Operand Processing Routine)
Optimizing subscripts 98
Output List Element Entry
in PF 644
in PRF 635
Output Page Heading
decision table 198
routine description 212

Pass 1 Statement Processors 45
Pass 2 Statement Processors 45
PAUSE Entry
in PF 645
in PRF 636,43
PAUSE Statement PF Entry Processor
(see: STOP and PAUSE Statement PF Entry Processor)
PAUSE, STOP, RETURN statement processor
decision table 52
flowchart 285-287
routine description 59-60
PF
(see: Program File)
Phase Controller
flowchart 218-222
overview 9
routine description 12-13,14,20
Phase Controller PSECT 11
Phase 1 3,39-40
Phase 1 Main loop
decision table 48
flowchart 246-249
routine description 55
Phase 2 4
PHASE2
decision table 83
routine description 82
Phase 3 92,6
Phase 3 Master Control Routine
decision table 102
flowchart 403-406
routine description 109-110
Phase 3 Storage PSECT 106-109
Phase 4 129,7
Phase 4 Master Control
decision table 141
routine description 153,154
PHASE4 (see: Phase 4 Master Control)
Phase 5 186,7
PHASE5
(see: FORTRAN Compiler Output Generator)
PHC
(see: Phase Controller)
PHEAD
(see: Output Page Heading)
PH1M
(see: Phase 1 Main loop)
PMD
(see: program module dictionary)
Polish Expression Generation Routine
decision table 103
flowchart 459-463
routine description 125-126
Preset Data Processor
decision table 198
flowchart 618
routine description 206
Preset Data Tables 639-640
PRF
(see: Program Representation File)
PRF Processing Routine
decision table 103
flowchart 407-418
routine description 110-113
PRINT entry
in PF 644
in PRF 635,43
Process and Generate Begin Loop Elements 53
Process Array demension specification 52
Process END and ERR statement numbers in READ statement 52
Process expression 52
Process FORMAT and NAMELIST name in I/O statement
decision table 52
Process FORMAT statement number in I/O statement
decision table 52
Process Function Argument 53
Process I/O statement list 52
Process initial data specifications 52
Process initial values in DATA or type statement 53
Process Label References and Definitions
decision table 83
flowchart 390-393
routine description 88
Process Label String in Assigned and Computed GO TO Statement 52
Process Statement Number 53

Process Subscripts 53
Process Terminal Modifications
  flowchart 227-228
  overview 9
  routine description 26-28,651-654
Program File 641-645
Program moudle dictionary 187-189,1
Program Representation File 40-43,631-633
PRSEC
  (see: PSECT Builder)
PSECT Builder
  decision table 197
  flowchart 617
  routine description 203-206
PSR
  (see: PAUSE, STOP, RETURN Statement Processor)
PUNCH entry
  in PF 644
  in PRF 635,43
Pushdown Premitive Operand Routine
  decision table 104
  flowchart 435-436
  routine description 118-119

RD
  (see: I/O Statement PF Entry Processor)
RDIV
  (see: Real Divide Generator)
RDM
  (see: Receive Diagnostic Message)
READ entry
  in PF 644
  in PRF 635,42
Read Transfer Processor for I/O Statements
  flowchart 358
  routine description 76
Real Divide Generator
  decision table 147
  flowchart 526-527
  routine description 163
Real Multiply Generator
  decision table 146
  flowchart 524-525
  routine description 162-163
Real Plus Generator
  decision table 145
  flowchart 521-523
  routine description 162
Receive Diagnostic Message
  flowchart 229
  overview 10
  routine description 28
Reference Table 192
Referenced Label PF Entry Processor
  decision table 141
  flowchart 474
  routine description 153-155
register
  conventions 9
  notation 9
  storage 133
Register Storage Clear Routine
  flowchart 610
  routine description 183-184
Relational Expression Generator
  decision table 148
  flowchart 542
  routine description 166-167
Release Temporary Storage
  decision table 152
  flowchart 609
  routine description 183
Relocation Dictionary 192-194
restricted linkage conventions 658
Result-Register Operand Processing Subroutine
  decision table 151
  flowchart 597
  routine description 179-180
RESUME macro instruction 660
RETURN entry
  in PF 643
  in PRF 634,42
RETURN Processor
  decision table 142
  flowchart 486-488
  routine description 156
RETURN statement processor
  (see: PAUSE, STOP, RETURN statement processor)
REWIND Entry in PRF 43
RLD
  (see: Relocation Dictionary)
RLSWS
  (see: Release Temporary Storage)
RLTNL
  (see: Relational Expression Generator)
RMUL
  (see: Real Multiply Generator)
RPLUS
  (see: Real Plus Generator)
RSLT
  (see: Result-Register Operand Processing Subroutine)
RTN1
  (see: Label Reference Processor)
RTRAN
  (see: Read Transfer Processor for I/O Statements)
RTRN
  (see: RETURN Processor)
RWIO
  (see: Input/Output Statement Processor)

SADDR
  (see: Local Branch Generator)
Save area format 657-658
Save Popularity Counts for Register Assignment
  decision table 105
  flowchart 464
  routine description 126
Search and Insert Triads
  decision table 105
  flowchart 442
  routine description 120-121
Search Floating Registers
  decision table 151
  flowchart 594
  routine description 178
Search General Registers
  decision table 152
  flowchart 593
  routine description 178

SELDR
(see: Select Even/Odd General Register Pair)
Select Even/Odd General Register Pair
decision table 151
flowchart 600-601
routine description 181
Select Floating Register
decision table 151
flowchart 602-604
routine description 181
Select One Operand in a Register
decision table 150
flowchart 584-587
routine description 175-176
Select Operand Routine
decision table 150
flowchart 580-581
routine description 173-175
Select Position for Operation
decision table 150
flowchart 582-583
routine description 175
Select Single General Register
decision table 152
flowchart 598-599
routine description 180-181
SELFR
(see: Select Floating Register)
SELGD
(see: General Register Availability for Integer Divide)
SELGM
(see: Determine Availability of Register for Multiplication)
SELOP
(see: Select Operand Routine)
SELSR
(see: Select Single General Register)
service routine interface 11,12,2
service subprograms 2
SFDEF
(see: Statement Function Definition
SFEXP
(see: Statement Function Expansion)
SID
(see: Statement of Identification)
Single Operand Locating Routine
decision table 151
flowchart 592
routine description 177-178
SLONE
(see: Select One Operand in a Register)
SLPOS
(see: Select Position for Operation)
Source conversion routines 45
Source Extraction routines 45
source statement preparation 12,13
Source Statement Identification 49
SPECS
(see: Preset Data Processor)
Statement Function Definition
flowchart 327
routine description 69-70
Statement Function Definition, Pass 2 63
Statement Function Expansion
flowchart 328-329
routine description 70
Statement Function Statement Final Processing 51
Statement ID numbers 79
Statement Label Processor
flowchart 372-373
routine description 78-79
Statement Label Reference Inspection
decision table 83
routine description 94
Statement of Identification
flowchart 371
routine description 78
STCLTB
(see: Storage Class Table)
STFN2
(see: Statement Function Definition, Pass 2)
STOP and PAUSE Statement PF Entry Processor
decision table 144
flowchart 512
routine description 159
STOP Entry
in PF 645
in PRF 43
STOP statement processor
(see: PAUSE, STOP, RETURN statement processor)
STOP
(see: STOP and PAUSE Statement PF Entry Processor)
Storage Assignments for Variables
decision table 83
routine description 81
Storage Class Table 641,44
storage map executive routine 9-10
Storage Reference Processing 135
Storage Specification Tables 636-638,43,44
STORE macro instruction 660
STPNT macro instruction 663
SUBE
(see: Subprogram Entry Statements Processor)
SUBE2
(see: Subprogram Entry Statements, Pass 2)
subprogram calls 1
Subprogram Entry
in PF 642
in PRF 633,41
Subprogram Entry Statement Processor
decision table 51
flowchart 296-300
routine description 61
Subprogram entry statements final processing 51
Subprogram Entry Statements, Pass2
flowchart 303
routine description 62
subroutine call 67-68
SUBS
(see: Subscript Processor)
subscript optimization 98
Subscript Expression Revision Routine
decision table 104
flowchart 455-457
routine description 124
Subscript Expressions 43
Subscript Processor

flowchart 305-310
routine description 63-64
Symbol Table 645
Symbol Table Sort
decision table 197
flowchart 623
routine description 211-212
SYMSRT
(see: Symbol Table Sort)

TEMPRO 54
Term Processor
flowchart 333
routine description 71
TEVCRL 152
TEVC16 152
TEVC8 152
TEVFLL 152
TEVFL4 152
TEVFL5 152
TEVI2 152
TEVI4 152
TEVRDM 152
TEVR4 152
TEVR8 152
TEVVR 152
TRBLD 144
TRBLD
(see: Expression Tree Builder)
Triad File Manipulation Routine
decision table 104
flowchart 440-441
routine description 120
Triad Table 107,109
TRMPRO
(see: Term Processor)
TYPE
(see: Type Statements processor)
Type Statements Processor
flowchart 258-261
routine description 56

Unconditional GO TO entry
in PF 643
in PRF 633

Variable Compute Point and Removal Level Routine
decision table 105
flowchart 437-438
routine description 119
VSCAN
(see: Storage Assignments for Variables)
VSCAN1 84
VSCAN2 84
VSCAN3 85

Weight Subroutine
decision table 144
flowchart 517
routine description 161
WGHT
(see: Weight Subroutine)
work areas 12-13,14,15
WRITE entry
in PF 644
in PRF 635,42