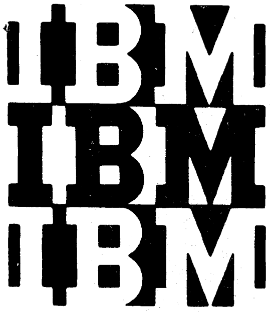
File Number S360-26
Order No. GY33-8000-0



Program Logic

OS ALGOL (F) Compiler Logic

Program Numbers: 360S-AL-531 (Compiler)
360S-LM-532 (Library Routines)

OS Release 21

This manual describes the internal logic of the ALGOL (F) Compiler. It is intended for the use of IBM field engineers, systems analysts and programmers.

The ALGOL (F) Compiler is a processing program of the IBM System/360 Operating System. It translates a source module written in the ALGOL language into an object module that can be processed into an executable load module by the Linkage Editor.

PREFACE

The IBM System/360 Operating System ALGOL Compiler consists of ten phases, or load modules. Chapter 1 of this manual provides an introductory survey of the main functions of the several phases. A more detailed description of the individual phases is provided in the subsequent chapters, as follows:

Directory (IEX00)	Chapter 2
Initialization (IEX10)	Chapter 3
Scan I/II (IEX11)	Chapter 4
Identifier Table Manipulation (IEX20)	Chapter 5
Diagnostic Output (IEX21)	Chapter 9
Scan III (IEX30)	Chapter 6
Diagnostic Output (IEX31)	Chapter 9
Subscript Handling (IEX40)	Chapter 7
Compilation Phase (IEX50)	Chapter 8
Termination Phase (IEX51)	Chapter 8

Two of the phases (Load Modules IEX21 and IEX31) are devoted exclusively to the editing and output of diagnostic messages. Diagnostic output is also provided for in the Termination Phase (Load Module IEX51). The Error Message Editing Routine, which handles the output of diagnostic messages in the three phases mentioned, is described in Chapter 9.

Chapter 10 describes the ALGOL Library, which consists of a set of load modules representing standard I/O procedures, mathematical functions, and the Fixed Storage Area.

Chapter 11 describes the composition of the object module generated by the Compiler, and the organization of the load module at execution time.

Other publications that will be useful to the reader in understanding the Compiler are:

OS ALGOL Language, Order No.GC28-6615

OS ALGOL Programmer's Guide, Order No. GC33-4000

OS FORTRAN IV Library, Order No. Order No. GC28-6596

First Edition (September 1967)

This edition applies to release 21 of the IBM System/360 Operating System, and to all subsequent modifications unless otherwise indicated in new editions or Technical Newsletters. Changes are continually made to the specifications herein; before using this publication in connection with the operation of IBM systems, consult the latest SRL Newsletter, Order No. GN20-0360 for the editions that are applicable and current.

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 is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Nordic Laboratory, Publications Development, Box 962, S-181 09 Lidingö 9, Sweden. Comments become the property of IBM.

©Copyright International Business Machines Corporation 1967

CHAPTER 1: INTRODUCTION. 13

Purpose of the Compiler. 13

The Compiler and System/360 Operating System. 13

Machine System 13

Organization of the Compiler 13
- Directory (IEX00) 13
- Initialization Phase (IEX10). 13
- Scan I/II Phase (IEX11) 15
- Identifier Table Manipulation Phase (IEX20). 15
- Diagnostic Output (IEX21) 15
- Scan III Phase (IEX30). 15
- Diagnostic Output (IEX31) 15
- Subscript Handling Phase (IEX40). . . . 15
- Compilation Phase (IEX50) 16
- Termination (IEX51) 16
- Diagnostic Output (IEX21, IEX31, and IEX51) 16
- ALGOL Library 16

The Object Module. 16

Input/Output Activity. 16

Interphase Communication by Source Text and Table. 17

Use of Main Storage. 18
- Area Occupied by Directory Auxiliary Routines 18
- The Common Work Area. 18
- Area Occupied by Operative Module . . . 19
- Private Area Acquired by Operative Module 19
- Common Area 19

Conventions. 20

CHAPTER 2: DIRECTORY (IEX00) 21

Purpose of the Directory 21

Organization of the Directory. 21
- Control Section IEX00000. 21
  - Initial Entry Routine. 21
  - Final Exit Routine 21
  - Program Interrupt Routine (PIROUT). 21
  - I/O Error Routine (SYNAD). 22
  - Sysprint I/O Error Routine (SYNPR) 22
  - End of Data Routines (EODAD1, EODAD2, EODAD3, AND EODADIN). . . . 22
  - Print Subroutine (PRINT) 23
  - Data Control Blocks. 23
- Control Section IEX00001 (Common Work Area) 23
  - Register Save Area 23
  - DCB Addresses. 24
  - End of Data Exit Addresses 24
  - Compiler Control Field (HCOMPMOD). 24
  - Communication Area 24
  - Area Size Table (INBLKS) 24
  - Headline Storage Area (PAGEHEAD) . . 24
  - Preliminary Error Pool 24
  - Data Control Blocks for SYSIN and SYSUT1. 25
  - Tables 25
  - Other Data 25

CHAPTER 3: INITIALIZATION PHASE (IEX10) 26

Purpose of the Phase 26

Execution of the SPIE Macro. 26

Processing Compiler Options, DDnames, and Heading Information 26
- Compiler Options. 27
- DDnames 28
- Heading Information 28

Selection of Area Size Table (FNDARSIZ). 28

Acquisition of Common Area 28

Opening of Data Sets 28

CHAPTER 4: SCAN I/II PHASE (IEX11) 30

Purpose of the Phase 30

Scan I/II Phase Operations 31
- Opening of Scopes 33
- Processing of Declarations and Specifications 34
- Close of Scopes 34
- End of Phase. 34

Phase Input/Output 35

Identifier Table (ITAB). 35
- Identifier Entries. 36
- Program Block Heading Entries 38
- For Statement Heading and Closing Entries. 39
- Processing of the Identifier Table. . . 40

Scope Identification 41
- Scope Handling Stack. 42

Modification Level 1 Source Text 43

Group Table (GPTAB). 45

Scope Table (SPTAB). 45

Program Block Number Table (PBTAB1). . . 46

Processing of Opening Source Text. . . . 46

Close of Scan I/II Phase 46

Switches 48

Constituent Routines of Scan I/II Phase 52
Phase Initialization. 52
Main Loop (TESTLOOP). 54
Blank (BLANK) 55
Test and Transfer Operator (TRANSOP). 55
RIGHTPAR. 55
POINT 55
Decimal Point (DECPOINT). 56
Assignment (ASSIGN) 56
Statement (STATE) 56
Apostrophe (APOSTROF) 56
Scale Factor (SCALE). 56
Blank after Apostrophe (BLKAPOS). . . 56
Zeta after Apostrophe (ZETAAPO) . . . 57
Invalid Character after Apostrophe (NPAFTAPO) 57
Colon (COLON) 57
Label (LABEL) 57
Letter Delimiter (LETDEL) 57
Semicolon (SEMCO and SEMC60). 57
Error Recording Routines. 57
Change Input Buffer (CIB) 59
Identifier Test (IDCHECK1). 59
Change Output Buffer (COB and COBSPEC) 59
Delimiter (DELIMIT) 60
Delimiter Error Routine (EROUT) . . . 60
Type Specification (TYPESPEC) 62
Comment (COMSPEC) 62
Opening Delimiter (STARTDEL). 62
Begin (BEGIN) 62
String (STRING) 62
Normal Action (NORMAL). 62
Boolean Constant (BOLCON) 62
Goto-If (GIF) 63
Then-Else-Do (TED). 63
First Begin (FIRSTBEG). 63
Program Block (BEG1 Subroutine) . . . 63
End (END) 63
Compound End (COMPDEND) 63
For Statement End (FOREND). 63
Program Block End (PBLCKEND Subroutine). 64
Comment (COM) 64
For statement (FOR) 64
Type Declaration (TYPE) 64
Identifier Error (IER). 64
Code Procedure (CODE) 65
Specification (SPEC). 65
Parameter Specification (SPECENT and IDCHECK) 65
Type Array (TYPEARRY) 65
Array Declaration (ARRAY) 65
Array/Switch List (LIST). 66
Point in List (PONTLST) 66
Right Parenthesis in List (RIGHTPARL). 66
Left Parenthesis in List (LEFTPARL) . 66
Comma in List (COMMALST). 66
Colon in List (COLONLST). 66
Semicolon in List (SEMCLST) 66
Slash in List (SLASHLST). 67
Switch Declaration (SWITCH) 67
Procedure Declaration (PROCEDUR). . . 67
Procedure Identifier (PROCID) 67
Termination (EODADIN) 67
Generate Subroutine 68

CHAPTER 5: IDENTIFIER TABLE MANIPULATION PHASE (IEX20). 69

Purpose of the Phase 69

Identifier Table Manipulation Phase Operations. 69

Phase Input/Output 70

Identifier Table (ITAB). 70

Program Block Table II (PBTAB2). 70

Constituent Routines of Identifier Table Manipulation Phase. 71
Phase Initialization. 71
Identifier Scan (READBLK) 72
Storage Allocation (ALLOSTOR) 72
Write Identifier Table (WRITITAB) . . 73
Print Identifier Table (ITABPRNT) . . 73
Termination (CLOSE) 73

CHAPTER 6: SCAN III PHASE (IEX30). . . . 74

Purpose of the Phase 74

Scan III Phase Operations. 75
Opening and Close of Blocks and Procedures 75
Identifier Handling 75
Number Handling 77
Array Subscript Handling. 77
Handling of Other Operators 77
Phase Termination 77

Phase Input/Output 77

Processing of the Identifier Table . . . 78

Classification of For Statements 79
Processing of For Statements. 80
Detection of Operators in For List. 80
Recognition of Identifiers in For Statements. 80

Optimizable Subscript Expressions. . . . 81

For Statement Table (FSTAB). 81

Left Variable Table (LVTAB). 82

Subscript Table (SUTAB). 82

Critical Identifier Table (CRIDTAB). . . 82

Array Identifier Stack (ARIDSTAB). . . . 84
Modification Level 2 Source Text. . . . 84

Switches 85

Constituent Routines of Scan III Phase . 86
Phase Initialization (INITIATE) . . . 86
General Test (GENTEST). 88
Identifier Test (LETTER). 88
ITAB Search (IDENT) 88
Identifier Classification (FOLI). . . 88
Noncritical Identifier (NOCRI). . . . 89
Procedure/Parameter (PROFU) 89
Switch/Label (SWILA). 89
Critical Identifier (CRITI) 89
Make Cridtab Entry (CRIMA). 90
CRIDTAB Overflow (CRIFLOW). 90
Erase CRIDTAB (DELCRIV) 91
Update CRIDTAB (CRIFODEL) 91
Make LVTAB Entry (LETRAF) 91
Nonzero Digit (DIGIT19) 91
Zero Digit (DIGIT0) 92
Decimal Point (DECPOIN) 92
Scale Factor (SCAFACT). 93
Integer Conversion (INTCON) 93
Real Conversion (REALCON) 94
Integer Handling (INTHAN) 94
Real Handling (REALHAN) 94
Change Constant Pool (CPOLEX) 94
Output TXT Record (TXTTRAF) 95
Generate (GENTXT) 95
Apostrophe (QUOTE). 95
Block Begin (BETA). 95
Procedure Declaration (PIPHI) 95
Read ITAB Record (ITABMOVE) 95
For Statement (FOR) 95
Program Block End (EPSILON) 95
For Statement End (ETA) 95
Do (DO) 96
While (WHILE) 96
Semicolon/Delta (SEMIDELT). 96
Opening Bracket (OPBRACK) 96
Comma (COMMA) 96
Closing Bracket (CLOBRACK). 96
Scan Subscript (SUCRIDEL) 96
Subscript Test (SUSCRITE) 96
Operand Test (OPERAND). 97
Multiplier-Operand (SUBMULT). 97
Make SUTAB Entry (SUTABENT) 98
Input Record End (ZETA) 98
Change Input Buffer (ICHA). 98
Code Procedure (GAMMA). 98
Program End (OMEGA) 98
Other Operators (OTHOP) 98
Letter Delimiter (RHO). 99
Step (STEP) 99
Array (ARRAY) 99
Switch (SWITCH) 99
Divide/Power(DIPOW) 99
Change Output Buffer (OUCHA). 99
Incorrect Operand (INCOROP) 99
Store Error (MOVERRO) 99
Move Operand (MOVE) 99
Check-Write (CHECK) 100
Write SUTAB/LVTAB Record (WRITE). . .100

CHAPTER 7: SUBSCRIPT HANDLING PHASE (IEX40) 101

Purpose of the Phase 101

Subscript Handling Phase Operations. . .101

Phase Input/Output 102

Optimization Table (OPTAB) 103

Subscript, Left Variable and For Statement Tables. 103

Constituent Routines of Subscript Handling Phase. 103
Initialization. 103
Read SUTAB. 105
Scan SUTAB. 105
Sort SUTAB (SORTSU) 106
Read and Sort LVTAB (SORTLE and SORTLE1) 106
Construct OPTAB (OPTAB) 106
Termination (TERMIN). 107
Write OPTAB (OTACHA). 107
Read SUTAB/LVTAB (READ) 107
Sort SUTAB/LVTAB (SORT) 107

CHAPTER 8: COMPILATION PHASE (IEX50) . .108

Purpose of the Phase 108

Compilation Phase Operations 108

Phase Input/Output 110

Operator/Operand Stacks. 110

Control of Object Time Registers 113

Decision Matrices. 116

Compile Time Register Use. 116

Constituent Routines of the Compilation Phase 116
Phase Initialization. 118
Scan to Next Operator (SNOT). 119
Compare (COMP). 119
Blocks and Compound Statements. . . .120
Compiler Program No.0 (CP0). . . .120
Compiler Program No.16 (CP16). . .120
Switches. 120
Compiler Program No.4 (CP4). . . .121
Compiler Program No.85 (CP85). . .122
Compiler Program No.56 (CP56). . .122
Compiler Program No.59 (CP59). . .122
Compiler Program No.41 (CP41). . .122
Compiler Program No.38 (CP38). . .122
Labels. 122
Compiler Program No.1 (CP1). . . .122
Goto Statements 123
Compiler Program No.6 (CP6). . . .123
Compiler Program No.56 (CP56). . .123
Compiler Program No.62 (CP62). . .123
Arrays. 124
Array Declarations 125
Subscripted Variables. 126
Compiler Program No.4 (CP4). . . .127
Compiler Program No.52 (CP52). . .127
Compiler Program No.36 (CP36). . .128
Compiler Program No.51 (CP51). . .128
Compiler Program No.54 (CP54). . .130
Compiler Program No.41 (CP41). . .130
Compiler Program No.38 (CP38). . .130

Procedures 131

Procedure Declaration 131
Procedure Call. 132
Compiler Program No.4 (CP4). . . .133
Operand Recognizer (OPDREC). . . .134
Compiler Program No.16 (CP16). . .134
Compiler Program No.64 (CP64). . .134
Compiler Program No.57 (CP57). . .134

Code Procedures. 135
Compiler Program No.83 (CP83). . .136

Standard Procedures. 136
Compiler Program No.64 (CP64). . .136
Compiler Program No.61 (CP61). . .137

For Statements 138
Counting Loops 138
Elementary Loops 139
Normal Loops 140
Subscript Optimization 143
Compiler Program No.6 (CP6). . . .144
Compiler Program No.40 (CP40). . .144
Compiler Program No.43 (CP43). . .147
Compiler Program No.45 (CP45). . .147
Compiler Program No.47 (CP47). . .147
Compiler Program No.49 (CP49). . .148
Subscript Initialization Routine (DWG3 or USA1). 148
Subscript Incrementation Routine (UVA1). 151
Compiler Program No.81 (CP81). . .151

Assignment Statements. 151
Compiler Program No.12 (CP12). . .151
Compiler Program No.21 (CP21). . .151
Compiler Program No.20 (CP20). . .152

Conditional Statements 153
Compiler Program No.8 (CP8). . . .154
Compiler Program No.78 (CP78). . .154
Compiler Program No.17 (CP17). . .154
Compiler Program No.18 (CP18). . .154

Conditional Expressions. 154
Compiler Program No.64 (CP64). . .156
Compiler Program No.80 (CP80). . .156
Compiler Program No.34 (CP34). . .156
Compiler Program No.65 (CP65). . .156
Compiler Program No.78 (CP78). . .156
Compiler Program No.87 (CP87). . .156
Compiler Program No.79 (CP79). . .157

Boolean Expressions. 157
Compiler Program No.64 (CP64). . .158
Compiler Program No.65 (CP65). . .158
Compiler Program No.67 (CP67). . .158
Compiler Program No.76 (CP76). . .158
Compiler Program No.77 (CP77). . .158

Arithmetic Expressions and Relations . .158
Compiler Program No.64 (CP64). . .158
Compiler Program No.66 (CP66). . .158
Compiler Program No.67 (CP67). . .159
Compiler Program No.63 (CP63). . .159
Compiler Program No.68 (CP68). . .159
Compiler Program No.69 (CP69). . .159
Integer-Integer Routine (DHZB1). .160
Integer Division Routine (ISB1). .160
Integer Multiplication Routine (IPB1). 160
Integer Power Routine (IUB1) . . .160
Real-Real Routine (DHEB2). 160
Real-Integer Power Routine (I1B1). 160
Real Power Routine (HOB1). 161
Compiler Program No.68 (CP68). . .161

Semicolon Handling 161
Compiler Program No.24 (CP24). . .161
Compiler Program No.25 (CP25). . .161
Compiler Program No.23 (CP23). . .161
Compiler Program No.7 (CP7). . . .161

Context Switching. 161
Compiler Program No.19 (CP19). . .161
Compiler Program No.22 (CP22). . .162
Compiler Program No.33 (CP33). . .162
Compiler Program No.70 (CP70). . .162
Compiler Program No.71 (CP71). . .162

Logical Error Recognition. 162
Compiler Program No.26 (CP26). . .162
Compiler Program No.27 (CP27). . .162
Compiler Program No.28 (CP28). . .163
Compiler Program No.29 (CP29). . .163
Compiler Program No.30 (CP30). . .163
Compiler Program No.31 (CP31). . .163
Compiler Program No.72 (CP72). . .163
Compiler Program No.73 (CP73). . .163
Compiler Program No.74 (CP74). . .163
Compiler Program No.75 (CP75). . .163
Compiler Program No.84 (CP84). . .163
Compiler Program No.86 (CP86). . .164

Close of Source Module 164
Compiler Program No.3 (CP3). . . .164

Subroutine Pool. 164
Change Input Buffer (JBUFFER). . .164
Next OPTAB Entry (NXTOPT). 164
Error Recording (SERR) 164
Conversion Integer-Real (TRINRE) .164
Conversion Real-Integer (TRREIN) .164
Generate Object Code (GENERATE). .165
Store Object Time Registers (CLEARRG) 165
Operand Recognizer (OPDREC). . . .165
Update DSA Pointer (MAXCH) 165
Semicolon Handling (SCHDL) 165
ROUTINE1 165
ROUTINE2 166
ROUTINE3 166
ROUTINE4 166
ROUTINE5 166
ROUTINE6 166
ROUTINE7 166
ROUTINE8 166
ROUTINE9 166
ROUTIN10 166
ROUTIN11 166
ROUTIN12 166
ROUTIN13 166
ROUTIN14 167
ROUTIN15 167
Program Block Number Handling (PBNHDL). 167
Parameterless Procedure Statement (PLPRST). 167

Termination Phase (IEX51). 167

Program Block Table IV (PBTAB4) 167
Label Address Table (LAT) . . .167
Data Set Table (DSTAB). 168
Address Table and END Record .169
Statement of Object Time Storage Requirements. 169
Diagnostic Output 169
End of compilation 169

CHAPTER 9: COMPILE TIME ERROR DETECTION AND DIAGNOSTIC OUTPUT 170

Error Detection. 170
Warning Errors (Severity Code W). . .170
Serious Errors (Severity Code S). . .170
Scan I/II Phase (IEX11). . .170
Identifier Table Manipulation Phase (IEX20) 170
Scan III Phase (IEX30) . . .170
Subscript Handling Phase (IEX40) 170
Compilation Phase (IEX50). .170
Termination Phase (IEX51). .171
Terminating Errors (Severity Code T) 171

Diagnostic Output. 171

Error Pool 172

Message Pool 172
Error Message 174
Logic of the Error Message Editing Routine. 174

CHAPTER 10: ALGOL LIBRARY. 176

Fixed Storage Area (IHIFSA). 176
Common Data Area. 176
Fixed Storage Area Routines 176
Initialization (ALGIN) 177
Prologue (PROLOG). 177
Epilogue (EPILOG). 177
Call Actual Parameter, Part 1 (CAP1). 177
Call Actual Parameter, Part 2 (CAP2). 178
Value Call (VALUCALL). 178
Return Routine (RETPROG) 178
Call Switch Element, Part 1 (CSWE1) 178
Call Switch Element, Part 2 (CSWE2) 179
Trace (TRACE). 179
Load Precompiled Procedure (LOADPP). 179
Standard Procedure Declaration (SPDECL). 179
Get Main Storage (GETMSTO) 179
Program Interrupt (PIEROUT). . . .179
FSAERR 179
Termination (ALGTRMN). 179
Integer to Real Conversion (CNVIRD). 179
Real to Integer Conversion (CNVRDI/ENTIER) 179

Input/Output Procedures. 180

PUT/GET (IHIGPR) 180
PUT 180
GET 181
OUTPUT. 181
INPUT 181
OPENGP. 181
CLOSEGP 181
OPENEXIT. 181
CAP1GP. 181
THUNKOUT. 181
THUNKIN 181

INARRAY/INTARRAY(IHIIAR) 181

INBARRAY (IHIIBA). 181

INBOOLEAN (IHIBO). 182

INREAL/ININTEGER (IHIIDE). 182

INSYMBOL (IHIISY). 182

OUTREAL (IHISOR) 182

OUTREAL (IHILOR) 182

OUTARRAY (IHIOAR). 182

OUTBARRAY (IHIOBA) 182

OUTBOOLEAN (IHIOBO). 182

OUTINTEGER (IHIOIN). 182

OUTSTRING (IHIOST) 182

OUTSYMBOL (IHIOSY) 183

OUTTARRAY (IHIOTA) 183

SYSACT (IHISYS). 183
SYSACT. 183
SYSACT1 183
SYSACT2 183
SYSACT3 183
SYSACT4 183
SYSACT5 183
SYSACT6 183
SYSACT7 183
SYSACT8 183
SYSACT9 184
SYSACT10. 184
SYSACT11. 184
SYSACT12. 184
SYSACT13. 184
SYSACT14. 184
SYSACT15. 184

Subroutine Pool (IHIIOR) 184
CLEARNOTTAB 184
CLOSE 184
CLOSEPE 185
CONVERT 185
DCBEXIT 185
ENDOFDATA 185
ENTRYNOTTAB 185
EVDSN 185
NEXTREC 185
OPEN. 185
SYNAD 185

Mathematical Standard Functions. 185

Object Time Error Routine (IHIERR) 186

CHAPTER 11: THE OBJECT MODULE. 187

Object Module. 187

Load Module. 188
  Object Time Tables. 188
    Program Block Table (PBT) 188
    Label Address Table (LAT) 189
    Data Set Table (DSTAB). 190
    Note Table (NOTTAB). 191
    Data Storage Area (DSA) 191
    Storage Mapping Function (SMF) . .193
    Return Address Stack (RAS) 193
  Object Time Register Use. 194

FLOWCHARTS 196

APPENDIX I-A: CHARACTER SET -- FIRST TRANSLATION OF THE SOURCE MODULE IN THE SCAN I/II PHASE 272

APPENDIX I-B: CHARACTER SET -- MODIFICATION LEVEL 1 TEXT 272

APPENDIX I-C: CHARACTER SET -- MODIFICATION LEVEL 2 TEXT 272

APPENDIX I-D: CHARACTER SET -- STACK OPERATORS USED IN THE COMPILATION PHASE 273

APPENDIX II: INTERNAL REPRESENTATION OF OPERANDS 274

APPENDIX III: INTERNAL REPRESENTATION OF STANDARD PROCEDURE DESIGNATORS . . .275

APPENDIX IV: COMPILER CONTROL FIELD (HCOMPMOD). 276

APPENDIX V-A: PROGRAM CONTEXT MATRIX . .278

APPENDIX V-B: STATEMENT CONTEXT MATRIX .279

APPENDIX V-C: EXPRESSION CONTEXT MATRIX. 279

APPENDIX VI: COMPILE TIME ERROR DETECTION 280

APPENDIX VII: OBJECT TIME ERROR DETECTION 283

APPENDIX VIII: COMPILE TIME WORK AREA SIZES, AS A FUNCTION OF THE SIZE OPTION. 285

APPENDIX IX-A: STORAGE MAPS OF THE CONSTITUENT LOAD MODULES OF THE ALGOL COMPILER. 286
    IEX00 - Directory. 286
    IEX10 - Initialization Phase . . .287
    IEX11 - Scan I/II Phase. 288
    IEX20 - Identifier Table Manipulation Phase. 289
    IEX21 - Diagnostic Output. 290
    IEX30 - Scan III Phase 290
    IEX31 - Diagnostic Output. 291
    IEX40 - Subscript Handling Phase .292
    IEX50 - Compilation Phase. 293
    IEX51 - Termination Phase. 294

APPENDIX IX-B: STORAGE MAP OF THE OBJECT MODULE (AT EXECUTION). 295

APPENDIX X: SUMMARY OF COMPILER PROGRAMS. 296

APPENDIX XI: INDEX OF ROUTINES 307

Figure 1. Constituent phases of the ALGOL Compiler. 14
Figure 2. I/O Activity by Data Set and Phase 17
Figure 3. Activity Table showing the processing of source text and tables by phase. 18
Figure 4. Use of main storage by ALGOL Compiler. 19
Figure 5. Option, DDname and Heading fields, and pointers. 27
Figure 6. PARMLIST Table entry for a Compiler option key-word. 27
Figure 7. Scan I/II Phase. Diagram illustrating functions. 32
Figure 8. Scan I/II Phase Input/Output. 35
Figure 9. Identifier Characteristic . . 36
Figure 10. Identifier Table entry for all identifiers except declared array, procedure and switch identifiers and labels. 37
Figure 11. Identifier Table entry as constructed in the Scan I/II Phase for a declared array identifier 37
Figure 12. Identifier Table entry for a declared procedure identifier 38
Figure 13. Identifier Table entry constructed in the Scan I/II Phase for a declared switch identifier. . . . 38
Figure 14. Identifier Table Entry constructed in the Scan I/II Phase for a declared label. 38
Figure 15. Program block heading entry . 39
Figure 16. Program block heading entry, as transmitted to the SYSUT3 data set. 39
Figure 17. For statement heading entry 39
Figure 18. For statement closing entry 39
Figure 19. Diagram illustrating the processing of the Identifier Table. . . 40
Figure 20. Scope Handling Stack operators 42
Figure 21. Group Table entries for a for statement and for a block or procedure 45
Figure 22. One-byte Scope Table entry . 46
Figure 23. One-byte Program Block Number Table entry. 46
Figure 24. Chart showing the logical flow in the search for the opening delimiter 47
Figure 25. Exits from Scan I/II Phase . 49
Figure 26. Private Area acquired by the Scan I/II Phase, showing pointers initialized 52
Figure 27. Source text buffers and pointers. 53
Figure 28. Heading Entry constructed at initialization in Identifier Table for Program Block 0 53
Figure 29. Switches used in Scan I/II Phase 53
Figure 30. Branch Address Table BPRTAB. 54
Figure 31. KEYTAB keys used in TRANSOP routine 55
Figure 32. Delimiter Table (WITAB) . . . 61
Figure 33. Internal Names of boolean constants 'TRUE' and 'FALSE'. 63
Figure 34. Identifier Table Manipulation Phase. Diagram illustrating the functions of the principal constituent routines. 69
Figure 35. Identifier Table Manipulation Phase Input/Output 70
Figure 36. Identifier Table (ITAB) entry, showing the identifier's Data Storage Area displacement address, as inserted by the Identifier Table Manipulation Phase in bytes 9 and 10, for all identifiers except those of declared procedures, switches and labels. 70
Figure 37. Two-byte entry in Program Block Table II (PBTAB2) 71
Figure 38. Private Area acquired by the Identifier Table Manipulation Phase 72
Figure 39. Scan III Phase. 76
Figure 40. Scan III Phase input/output . 78
Figure 41. Function of pointers NOTER and NOTEW in input/output operations on the SYSUT3 data set. 78
Figure 42. Diagram illustrating the handling of Identifier Table (ITAB) records 79
Figure 43. Entry in Left Variable Table 82
Figure 44. Fourteen-byte Subscript Table entry 83
Figure 45. Entry in Critical Identifier Table (CRIDTAB). 83
Figure 46. Entry for an array identifier in the Array Identifier Stack (ARIDSTAB). 84
Figure 47. Private Area acquired by Scan III Phase. 87
Figure 48. Subscript Handling Phase. . .102
Figure 49. Subscript Handling Phase Input/Output. 103
Figure 50. Optimization Table (OPTAB) entry 104
Figure 51. Diagram illustrating use of the private area. 104
Figure 52. Compilation Phase. Diagram illustrating phase operations 109
Figure 53. Compilation Phase Input/Output. 110

Figure 54. Diagram illustrating the function of the Operator/Operand Stacks. 112
Figure 55. Five-byte operand representing an intermediate value or address contained in an object time register or temporarily stored in the register's reserved storage field in a Data Storage Area.. 113
Figure 56. Control Fields governing use of object time general purpose registers 114
Figure 57. Control Fields governing use of floating point registers,. . . .115
Figure 58. Diagram showing the compiler programs 117
Figure 59. Private Area acquired by Control Section IEX40001 for the Compilation Phase (IEX50) 118
Figure 60. Entry in Program Block Table III (PBTAB3). 120
Figure 61. Diagram showing code generated for switch declaration and switch designator 121
Figure 62. Object Time Storage Mapping Function of an array. 125
Figure 63. Code generated for declared type procedure and procedure call . . .132
Figure 64. I/O Table (IOTAB) 138
Figure 65. For statement classification byte in the For Statement Table 139
Figure 66. Logical structure of the code generated for a Counting Loop. . .141
Figure 67. Logical structure of the code generated for an Elementary Loop or Normal Loop. 141
Figure 68. Logical structure of the code generated for a Counting Loop. . .142
Figure 69. Logical structure of the code generated for an Elementary Loop .145
Figure 70. Logical structure of the code generated for an Elementary Loop .146
Figure 71. Entry in Subscript Table-C (SUTABC). 148
Figure 72. Logical structure of the code generated for a Normal Loop. . . .149
Figure 73. Logical structure of code generated for Elementary Loop or Normal Loop 150
Figure 74. Composition and execution sequence of Load Modules IEX21, IEX31, and IEX51, containing the Error Message Editing Routine (IEX60000). 171
Figure 75. Error pattern stored in Error Pool. 172
Figure 76. Message Pool entry. 172
Figure 77. Three-byte Insertion Code in the Message Pool entry (see Figure 76) 172
Figure 78. Format of the printed error message 174
Figure 79. Four-byte parameter list entry for a standard procedure call . .180
Figure 80. Label of a PUT/GET record . .180
Figure 81. Module names of mathematical standard functions contained in the ALGOL Library. 186
Figure 82. Composition of the object module. 187
Figure 83. Sketch showing the organization of the load module 188
Figure 84. Object time Program Block Table 189
Figure 85. Object time Label Address Table (LAT) 189
Figure 86. Content of the data set entries and the PUT/GET Control Field .190
Figure 87. Entry in the object time Note Table (NOTTAB) 191
Figure 88. Content of the Data Storage Area. 192
Figure 89. Content of the 8-byte storage field of a formal parameter called. 193
Figure 90. Entry in the object time Return Address Stack (RAS). 194
Figure 91. Object time register use. . .195

CHARTS

ALGOL Compiler - Overall Flow. 197
Directory (IEX00). 198
Initialization Phase (IEX10) 201
Scan I/II Phase (IEX11). 203
Identifier Table Manipulation Phase
(IEX20) 215
Diagnostic Output (IEX21). 218
Scan III Phase (IEX30) 220
Diagnostic Output (IEX31). 234
Subscript Handling Phase (IEX40) 235
Compilation Phase (IEX50). 238
Termination Phase (IEX51). 262
ALGOL Library. 265

SUMMARY OF AMENDMENTS

FOR GY33-8000-0

OS Release 21

TITLE CHANGES

Maintenance

Names of reference publications have been changed to reflect their current titles.

PURPOSE OF THE COMPILER

The OS/360 ALGOL Compiler translates a source program written in the OS/360 ALGOL Language into an object module which may be linkage edited and executed by an IBM System/360 computer. The final load module consists in part of code generated by the Compiler and, in part, of routines (in load module form) drawn from the ALGOL Library. The Library is a data set containing ALGOL standard I/O procedures and mathematical functions, as well as auxiliary routines required by the object module at execution time. The Library routines are combined with the generated code at linkage edit time, to form an executable load module.

The Compiler prints out a listing of the source module and of the Identifier Table, if the SOURCE option is specified, and prints out diagnostic messages reflecting syntactical errors detected in the source module, as well as other errors occurring during compilation.

THE COMPILER AND SYSTEM/360 OPERATING SYSTEM

The ALGOL Compiler is a processing program of the System/360 Operating System. It is executed under the control of the OS Supervisor, and utilizes the I/O and other services of the OS Control Program.

A compilation is executed as a job step by means of the job control facilities of the Operating System. The use of the Compiler is explained in the OS ALGOL Programmer's Guide.

MACHINE SYSTEM

The minimum machine configuration required for execution of a compilation using the ALGOL Compiler is as follows:

1. An IBM System/360 Model 30, 40, 50, 65, 75, or 91, or an IBM System/370 Model 135 (or higher) with the scientific instruction set and at least 64K bytes of main storage capacity.

2. At least one direct access input/output device; a printer; a console typewriter, and a sequential device (magnetic tape unit or card reader).

ORGANIZATION OF THE COMPILER

Figure 1 indicates the modular structure of the ALGOL Compiler as well as the essential operations performed in each of the constituent phases. The Compiler consists of ten load modules, the first of which, called the Directory, remains in main storage throughout compilation. The other nine modules, representing the working phases of the compiler, are loaded and executed in sequence.

DIRECTORY (IEX00)

The Directory consists of a preassembled Common Work Area, used by all phases, as well as a number of auxiliary routines providing interface with the Operating System. The latter include the initial entry and final exit routines which receive control from, and return control to, the Operating System.

INITIALIZATION PHASE (IEX10)

The Initialization Phase:

1. Sets up a control field in the Common Work Area, reflecting the options specified in the EXEC Statement invoking the Compiler.

2. Determines the sizes of the private work areas required by the individual phases.

3. Acquires main storage for a source text input buffer and for the Error Pool.

4. Opens data sets.

5. Executes a SPIE macro which specifies the address of the program interrupt routine in the Directory.

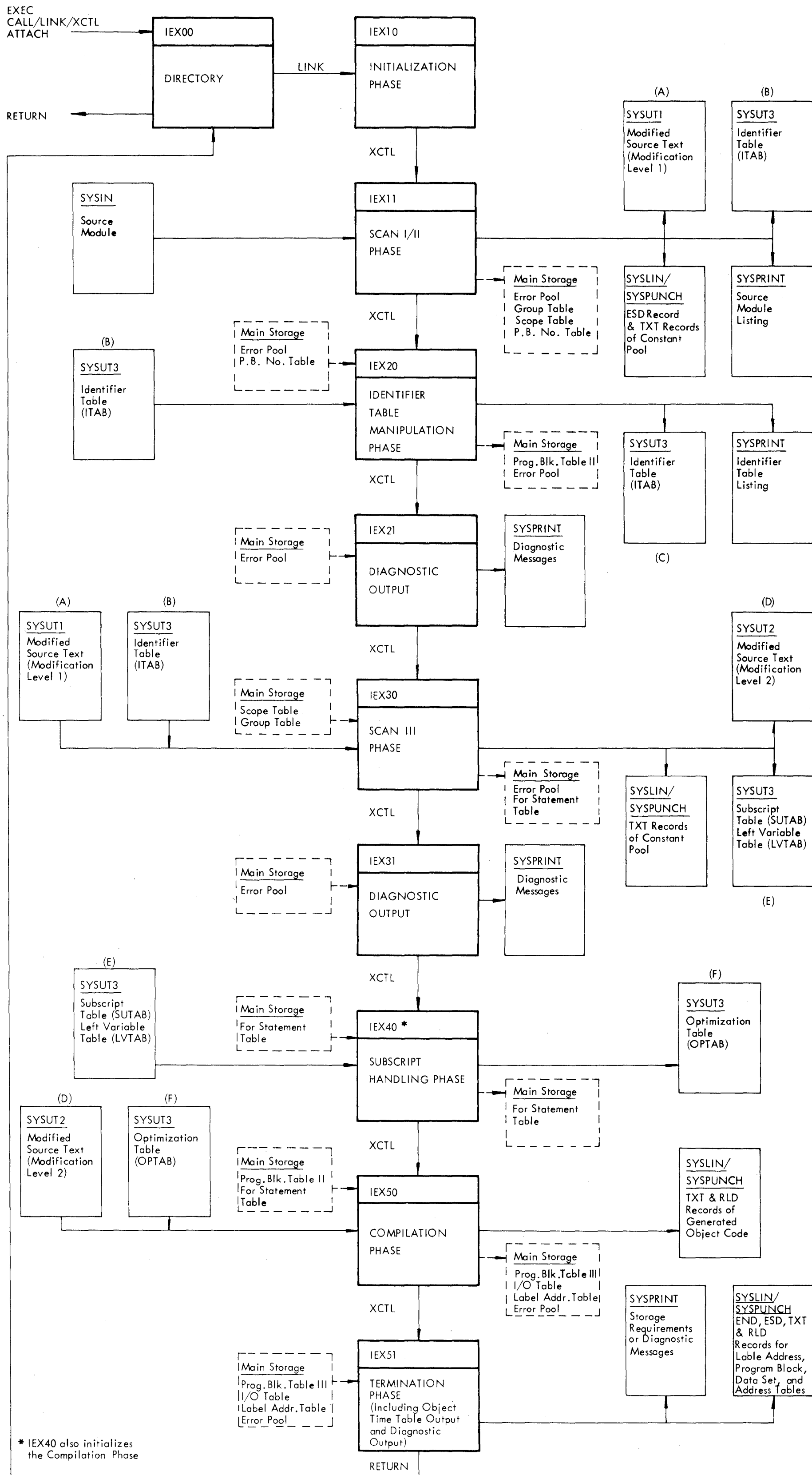


Figure 1. Constituent phases of the ALGOL Compiler

Directory
Contains entry and exit routines which receive control from and return control to the Operating System. Also contains Common Work Area for inter-phase communication, as well as program interrupt, SYNAD and EOD routines. Resident in main storage during compilation.

Initialization
Determines sizes of work areas acquired by the individual phases, according to the SIZE option; opens data sets; sets switches reflecting compilation options; and acquires storage for an input buffer and the Error Pool. Also executes the SPIE macro-instruction.

Scan I/II
Reads the source module and lists all valid identifiers declared or specified in the source module in the Identifier Table, together with descriptive internal names; stores character strings in the Constant Pool, replacing them in the output text by internal names containing the string's relative address; and generates TXT records of the strings stored in the Constant Pool. Replaces all delimiter words and multi-character operators by one-byte symbols in the output text (called Modification Level 1). Detects syntactical errors and records them in the Error Pool. Prints out a listing of the source module if the SOURCE option is specified.

Identifier Table Manipulation
Allocates object time storage addresses to all identifiers (other than declared procedure and switch identifiers and labels) listed in the Identifier Table, noting the relative addresses in the corresponding internal names in the table; records the total storage allocation for identifiers declared or specified in each block or procedure in Program Block Table II; and records multiple declaration errors in the Error Pool. Prints out a listing of the Identifier Table if the SOURCE option is specified.

Diagnostic Output
Edits the contents of the Error Pool and prints out diagnostic messages reflecting the errors detected by the preceding phases.

Scan III
Reads the Modification Level 1 source text output by Scan I/II and generates a new source text (Modification Level 2), in which externally represented operands in statements are replaced by the corresponding internal names in the Identifier Table. Stores constants in the Constant Pool, replacing them by internal names containing the storage address, and generates TXT records of the Constant Pool. Classifies for statements in the For Statement Table and lists linear subscript expressions and left variables in counting loop and elementary loop for statements, in the Subscript Table and Left Variable Table, respectively.

Diagnostic Output
Edits the contents of the Error Pool and prints out diagnostic messages reflecting the errors detected by the Scan III Phase.

Subscript Handling
Constructs the Optimization Table, listing optimizable subscript expressions in for statements, in which no term occurs as a left variable in the for statement. Optimizable subscripts are identified by comparing each term in the expressions listed in the Subscript Table with the entries from the same for statement in the Left Variable Table. Also re-classifies for statements in the For Statement Table.

Compilation
Reads the Modification Level 2 source text and generates an object module. Uses the For Statement Table to determine the logical structure of the generated code for each for statement. Uses the Optimization Table to generate code which pre-calculates a base address and an incremental displacement for optimizable subscripts of arrays occurring in for statements. Generates code to link precompiled standard functions and I/O procedures in the Library to the object module.

Termination
Generates TXT and RLD records of tables used by the object module at execution time, as well as ESD records for standard I/O procedures and mathematical functions, and the END record. Edits and prints out errors recorded in the Error Pool or prints a statement of main storage requirements at object time. Releases main storage and returns control to the exit routine in the Directory.

SCAN I/II PHASE (IEX11)

The Scan I/II Phase reads the source module and constructs the Identifier Table, listing all identifiers declared or specified un the source module. The Identifier Table is used in constructing a five-byte internal name for each and every identifier declared or specified in the source module. In the case of declared labels, procedures, and switches, the internal name (constructed in its entirety in this phase) contains the relative address of an entry in the object time Label Address Table. In the case of all other identifiers, the internal name (constructed partly in this phase and partly in the succeeding phase) contains the relative address of an object time storage field. The internal name ultimately replaces all externally represented operands in the source text (see Scan III Phase below).

The Scan I/II Phase also generates the first of two intermediate transformations of the source text, called Modification Level 1. The principal changes reflected in the first transformation include:

1. An initial translation of all characters to an internal code.
2. The removal of all type declarations and specifications.
3. The replacement of ALGOL delimiter words and multicharacter operators by one-byte symbols.

IDENTIFIER TABLE MANIPULATION PHASE (IEX20)

The Identifier Table Manipulation Phase processes the Identifier Table constructed by the Scan I/II Phase. To each identifier listed in the table, excepting declared procedure and switch identifiers and labels, an object time storage field is assigned, the relative address being inserted in the corresponding entry in the Identifier Table. This address specifies the position of the identifiers storage field, relative to the beginning of a Data Storage Area. The Data Storage Area consists of the total amount of object time storage space allocated to all identifiers declared or specified in the particular block or procedure. The size of the Data Storage Area allocated to each block and procedure is recorded in Program Block Table II and transmitted to the Compilation Phase via the Common Work Area.

DIAGNOSTIC OUTPUT (IEX21)

See "Diagnostic Output" below.

SCAN III PHASE (IEX30)

The Scan III Phase reads the Modification Level 1 text output by the Scan I/II Phase and generates a further transformation of the source text (called Modification Level 2). In this version, the external names of operands in statements are replaced by the internal names constructed for declared or specified identifiers in the Identifier Table. Similarly, all constants are replaced by internal names containing a Constant Pool address. After being stored in the Constant Pool, constants are subsequently transferred to TXT records.

Logical features of all for statements are detected and recorded in the For Statement Table. Among other things, the For Statement Table assigns each for statement to one of three loop classifications (Normal Loops, Counting Loops and Elementary Loops). The loop classification specifies the logical structure of the code generated in the Compilation Phase for each for statement.

Subscript expressions of arrays found in for statements, classified Counting Loops or Elementary Loops, are analyzed and stored in the Subscript Table, provided they satisfy certain criteria with respect to the terms in the expression and their linearity within the for statement. Integer left variables in Counting and Elementary Loops are listed in the Left Variable Table.

DIAGNOSTIC OUTPUT (IEX31)

See "Diagnostic Output" below.

SUBSCRIPT HANDLING PHASE (IEX40)

The Subscript Handling Phase constructs the Optimization Table, listing those subscript expressions of arrays contained in for statements, which can be optimized in the code generated for for statements. Optimization refers to the minimization of computing time involved in addressing the elements of an array.

COMPILATION PHASE (IEX50)

The Compilation Phase reads the Modification Level 2 text output by the Scan III Phase and generates object code to perform the operations designated by statements in the source module.

Operand addresses in the generated code are obtained from the internal names of operands in the Modification Level 2 text. The logical structure of the object code generated for a for statement is governed by the particular for statement's loop classification in the For Statement Table. Where a for statement contains optimizable subscripts, the Optimization Table is used in generating code which minimizes the computing time involved in addressing array elements.

TERMINATION (IEX51)

The Termination Phase constructs the Data Set Table and Program Block Table IV; generates TXT and RLD records for the latter two tables and for the Label Address Table; generates an END record as well as ESD records for all Library routines to be combined with the object module; processes any errors detected in the Compilation Phase; and terminates the Compiler by releasing main storage and returning control to the invoking program via the Final Exit routine in the Directory. The Termination Phase logically constitutes an extension of the Compilation Phase and is described in the same chapter, namely Chapter 8.

DIAGNOSTIC OUTPUT (IEX21, IEX31, AND IEX51)

The compile time Error Message Editing Routine, which forms a control section of each of load modules IEX21, IEX31, and IEX51, prints out diagnostic messages reflecting errors detected by the preceding phase or phases in the source module. Any errors detected are recorded by the particular phase in the Error Pool, in the form of error patterns. At the conclusion of a phase, the Error Message Editing Routine processes the contents of the Error Pool and prints out appropriate diagnostic messages. Errors are classified as warning errors, serious errors, or terminating errors. The recognition by any phase of a terminating error causes control to be transferred directly to a terminating routine in the Termination Phase, after all recorded errors have been printed out by the Error Message Editing Routine in the appropriate diagnostic output module (see Figure 1).

ALGOL LIBRARY

The Library is a partitioned data set (SYS1.ALGLIB) consisting of routines which perform the standard mathematical functions and I/O procedures defined in the ALGOL Language. The appropriate routines, corresponding to the standard functions or I/O procedures called in the source module, are linked to the object module at linkage edit time. ESD records to call standard functions or I/O procedures are generated in the Termination Phase.

The Library also contains the Fixed Storage Area, which consists of a set of auxiliary routines and control fields required for execution of the object module. The auxiliary routines include the Initialization and Termination routines, as well as other routines which acquire or release main storage and administer the calling of procedures. The Library is further described in Chapter 10.

An object time Error Routine is provided, which forms a module of the SYS1.LINKLIB data set. The Error Routine, which is loaded only if an object time error is detected, prints out an appropriate error message and terminates the object program. The error routine is described in Chapter 10.

THE OBJECT MODULE

The structure of the object module is described in Chapter 11.

INPUT/OUTPUT ACTIVITY

The data sets used by the Compiler are indicated in Figure 1. I/O operations in each of the several working phases are discussed in further detail in the relevant chapters. The table in Figure 2 summarizes I/O activity during compilation, in terms of the macro instructions issued.

Data Set Table

Phase	Data Set						
	SYSIN	SYSUT1	SYSUT2	SYSUT3	SYSLIN	SYSPRINT	SYSPUNCH
Access method used:	QSAM	BSAM	BSAM	BSAM	QSAM	QSAM	QSAM
IEX00	CLOSE*	CLOSE*	CLOSE*	CLOSE*	CLOSE*	CLOSE* PUT	CLOSE*
IEX10	OPEN CLOSE*	OPEN CLOSE*	OPEN	OPEN	OPEN (if used)	OPEN PUT	OPEN (if used)
IEX11	GET CLOSE	WRITE CHECK		WRITE CHECK	PUT	**	PUT
IEX20				READ, CHECK WRITE, CHECK NOTE, POINT		**	
IEX21						**	
IEX30		READ CHECK CLOSE	WRITE CHECK	CLOSE (T) READ, CHECK WRITE, CHECK NOTE, POINT	PUT		PUT
IEX31						**	
IEX40			READ CHECK	READ, CHECK POINT WRITE, CHECK			
IEX50			READ, CHECK	READ, CHECK	PUT		PUT
IEX51 IEX51002		CLOSE*	CLOSE	CLOSE	PUT CLOSE	** CLOSE	PUT CLOSE

* Data set closed in event of program interrupt or unrecoverable I/O error.

** In each of the modules indicated, a call is made to the PRINT subroutine in the Directory (IEX00), which executes the PUT macro instruction.

Figure 2. I/O Activity by Data Set and Phase

INTERPHASE COMMUNICATION BY SOURCE TEXT AND TABLE

The source module is subjected to two transformations before object code is generated in the Compilation Phase. These transformed versions of the source text are named Modification Level 1 and Modification Level 2. They are described in Chapters 3 and 5, respectively. Modification Level 1 is generated by the Scan I/II Phase, Modification Level 2 by the Scan III Phase. Modification Level 2 forms the main input to the Compilation Phase, which generates the ultimate object code.

The Tables constructed in the several phases and transmitted to one or more subsequent phases are indicated in Figure 3. A detailed description of the function and contents of each table is given in the chapters indicated:

Name of Table	Described in Chapter
Address Table	8
Data Set Table (DSTAB)	8, 11
For Statement Table (FSTAB)	6
Group Table (GPTAB)	4
Identifier Table (ITAB)	4, 5, 6
I/O Table (IOTAB)	8
Label Address Table (LAT)	8, 11
Left Variable Table (LVTAB)	6
Optimization Table (OPTAB)	7
Program Block Number Table (PBTAB1)	4
Program Block Table II (PBTAB2)	5
Program Block Table III (PBTAB3)	8
Program Block Table IV (PBTAB4)	8, 11
Scope Table (SPTAB)	4
Semicolon Table (SCTAB)	4
Subscript Table (SUTAB)	6

Activity Table

Text / Table (All tables except those marked by asterisks are transmitted between phases via the Common Work Area)	Initialization Phase (IEX10)	Scan I/II Phase (IEX11)	Identifier Table Manipulation Phase (IEX20)	Diagnostic Output (IEX21)	Scan III Phase (IEX30)	Diagnostic Output (IEX31)	Subscript Handling Phase (IEX40) -excluding Compilation Phase Initialization	Compilation Phase (IEX50)	Termination Phase (IEX51)
Source Text *		A			A			B	
Address Table									C,W
Data Set Table (DSTAB)									C,W
For Statement Table (FSTAB)					C		M	T	
Group Table (GPTAB)		C			T				
Identifier Table (ITAB) **		C	M		T				
I/O Table (IOTAB)								C	T
Label Address Table (LAT)								C	W
Left Variable Table (LVTAB) **					C		T		
Optimization Table (OPTAB) **							C	T	
Program Block Number Table (PBTAB1)		C	T						
Program Block Table II (PBTAB2)			C					M,T	
Program Block Table III (PBTAB3)								C	T
Program Block Table IV (PBTAB4)									C,W
Scope Table (SPTAB)		C			T				
Semicolon Table (SCTAB)		C,T							
Subscript Table (SUTAB) **					C		T		

* The source text is transmitted between phases via external storage, unless the text is less than a full buffer in length. In the latter case it is transmitted by way of Source Text Buffer 1 in the Common Area.

** Table transmitted between phases by way of an external storage device (see Figure 1).

Key:
- A - Source Text transformed
- B - Source Text terminated (object code generated)
- C - Table constucted
- M - Table completed or modified
- T - Table utilized and terminated
- W - Table transmitted to object module

Figure 3. Activity Table showing the processing of source text and tables by phase

USE OF MAIN STORAGE

The storage maps in Appendix IX indicate the layout of routines and tables in main storage in each of the phases of the Compiler. In terms of function, the main storage utilized by the Compiler may be divided into five main areas:

1. Area occupied by auxiliary routines of the Directory (Control Section IEX00000)

2. Common Work Area (Control Section IEX00001 of the Directory)

3. Areas occupied by the operative phase (the operative module)

4. Private area acquired by the operative phase

5. Common Area occupied by the Error Pool and Source Buffer 1

These areas are pictured in Figure 4.

AREA OCCUPIED BY DIRECTORY AUXILIARY ROUTINES

The composition of this control section, which contains auxiliary routines interfacing with the Operating System, as well as Data Control Blocks for all data sets except SYSUT1 and SYSIN, remains unchanged during compilation.

THE COMMON WORK AREA

The Common Work Area is an area of approximately 3500 bytes, resident in main storage throughout compilation. Except for the lower 540 bytes, whose assignment is fixed, the composition of the Common Work Area varies between phases and is defined by a Dummy Control Section in each phase. The Common Work Area functions as an interphase communication and control area. It contains a control field, initialized by the Initialization Phase and modified in the subsequent phases; a save area; a general transmission area used for communicating addresses, parameters and counters

used by all phases in common; and a general work area. The general work area, which represents the major part of the Common Work Area, provides space for the construction and/or transmission between successive phases of small-size tables. In the Scan I/II Phase, an 80-byte field of the Common Work Area is used for processing card-image records of the (translated) source module.

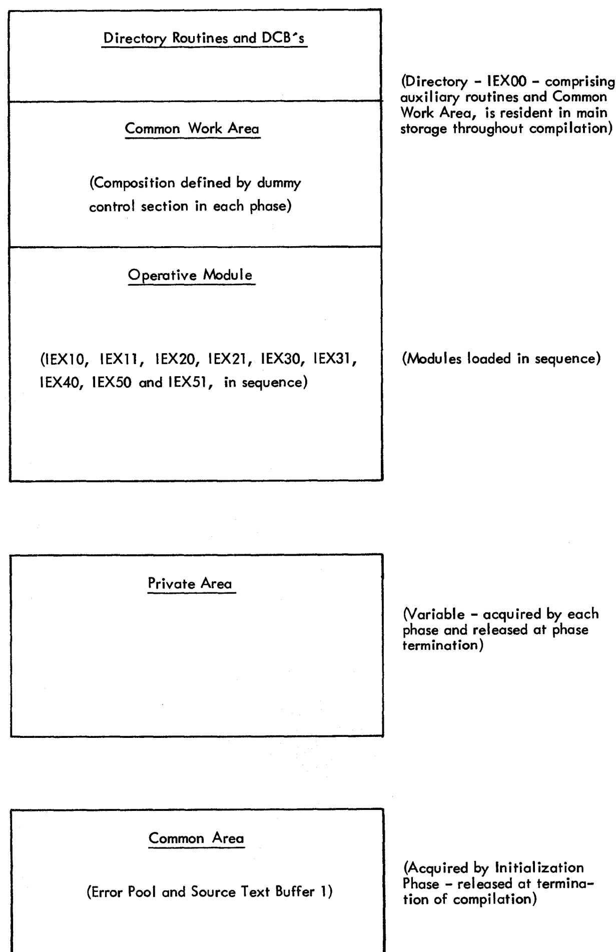


Figure 4. Use of main storage by ALGOL Compiler

AREA OCCUPIED BY OPERATIVE MODULE

The operative module, which varies in size, is loaded adjacent to the Common Work Area.

PRIVATE AREA ACQUIRED BY OPERATIVE MODULE

All of the phases of the Compiler, except load modules IEX21 and IEX31 (diagnostic output modules), acquire a private area for the construction of relatively large size tables which are transferred to external storage devices. The private area is in every case released at phase termination. The private work areas are described in the relevant chapters under the heading "Initialization". In the Scan I/II and Compilation Phases, the private area provides space for one source text buffer, while the Scan III Phase acquires three buffers. (See Common Area.)

COMMON AREA

The Common Area is acquired by the Initialization Phase and is not released until Compiler termination. It provides space for the Error Pool and for Source Buffer No. 1. The Common Area buffer is provided in order to enable the source text to be transmitted between phases via main storage, in the event the text occupies less than a full buffer. If either or both of the intermediate versions of the source text exceeds the buffer length, the text is transferred to external storage. In each of the phases which process the source text, one or more additional buffers are provided for in the private area acquired (and subsequently released) by the phase. In the Scan I/II and Compilation Phases, the private area contains one source buffer, while in the Scan III Phase, the private area contains three buffers.

In the Scan I/II Phase, the Modification Level 1 text is assembled and transmitted to the Scan III Phase in the Common Area buffer, unless the text exceeds the buffer length. In the latter case, the text is transferred to SYSUT1, using the Common Area buffer and the private area buffer as output buffers.

In the Scan III Phase, the Modification Level 1 source text is processed in the Common Area buffer (if the text was transmitted in main storage), or alternatively, in the Common Area buffer and a private area buffer (if the text is input from the SYSUT1 data set). The Modification Level 2 text is assembled in one (or two) buffers in the private area and, if it exceeds the buffer length, it is transferred to SYSUT2. If the text is less than the buffer length, it is moved to the Common Area buffer, before the private area is released, for transmission to the Compilation Phase in main storage.

In the Compilation Phase, the Modification Level 2 source text is processed in the Common Area buffer (if the entire text was transmitted in main storage) or, alternatively, in the Common Area buffer and a

private area buffer (if the text was transmitted on the SYSUT2 data set).

CONVENTIONS

The following conventions are observed in this manual:

1. ALGOL delimiter words in the text of a source module are represented in the manner defined by the IBM System/360 Operating System ALGOL Language, e.g. 'BEGIN' or 'REAL'.

2. With certain exceptions, one-byte characters in the internal code of the Compiler, representing ALGOL delimiter words, as well as other conventional delimiters, are represented as in the following examples: Begin, Goto, Power, Or, Comma, Decimal Point, Array. The same convention applies to operators used by the Compiler internally, e.g. Beta, Proc, Epsilon. Except for the power, assignment and scale factor operators (represented respectively as Power, Assign, and Scale Factor), arithmetic and relational operators are represented by their commonly understood symbols, e.g. +, <, = . Parentheses and brackets are also represented symbolically, as in (,), [,]. The complete range of internal character representation during the various phases of the Compiler is indicated in the code tables in Appendices 1-a through 1-d.

3. Syntactical, logical, or operational errors detected during compilation are identified by the serial number in the corresponding diagnostic message key. Thus, for example, the error whose detection produces a diagnostic message with the message key IEX034I, is identified in this manual as "error No. 34."

PURPOSE OF THE DIRECTORY

The Directory (IEX00) is the first of ten load modules of the ALGOL Compiler. It is the first module to be loaded in main storage, and unlike the other nine modules, which are loaded, executed and then displaced by the succeeding module, the Directory remains in main storage throughout compilation.

The function of the Directory is:

1. To provide the requisite interface between the Compiler, on the one hand, and the invoking program and the Operating System, on the other. This interface is provided by (a) the Initial Entry routine, which receives control from the invoking program and loads the next module (IEX10); and the Final Exit routine, which returns control to the invoking program at the close of the Termination Phase (IEX51);(b) the Program Interrupt, SYNAD and EODAD routines, which receive control from the Operating System in the event of an unexpected interrupt and pass control to an appropriate routine in the operative phase; and (c) data control blocks for data sets used by the Compiler.

2. To provide a PRINT Subroutine, used in common by several phases, which prints out compilation output on the SYSPRINT data set, on call from the operative phase. The printed output includes diagnostic messages indicating syntactical errors detected in the source module, and, depending on the Computer options specified, listings of the source module and the Identifier Table.

3. To provide the Common Work Area, an area of main storage used for the transmission of tables, addresses and other data between phases. Among other things, the Common Work Area contains a common register save area, a Control Field (HCOMPMOD -- see Appendix IV) which governs operations in each phase, and an Area Size Table which specifies the sizes of the private areas acquired by the several phases. The Control Field and Area Size Table, which are initialized or constructed by the Initialization Phase (IEX11), reflect the Compiler options specified by the user.

The Directory is loaded by the invoking program by means of a LOAD, XCTL, LINK, or ATTACH macro instruction, or by an EXEC control card.

ORGANIZATION OF THE DIRECTORY

The Directory consists of two control sections, named IEX00000 and IEX00001, respectively. Control Section IEX00000 contains the Initial Entry, Final Exit, Program Interrupt, SYNAD, EODAD, and Print routines, as well as data control blocks for all except two of the data sets used by the Compiler (the other two are contained in the Common Work Area). Control Section IEX00001 comprises the Common Work Area.

CONTROL SECTION IEX00000

The principal components of Control Section IEX00000 are as follows:

Initial Entry Routine

The Initial Entry routine receives control from the invoking program. The routine saves registers in the Invoker's save area, loads register 13 with the address of a save area in IEX00000, and executes a LINK macro instruction to load and activate the Initialization Phase (IEX10).

Final Exit Routine

The Final Exit routine is entered from the Termination Phase (IEX51). Registers are restored and control returned to the Invoker by a RETURN macro instruction.

Program Interrupt Routine (PIROUT)

PIROUT is activated by the control program in the event of a program interrupt. The address of PIROUT is specified by a SPIE macro instruction in the Initialization Phase (IEX10).

PIROUT records error No.209 in the Error Pool (indicating a program interrupt) and transfers control to a closing routine in the operative phase, the address of which is stored at a location named ERET in the Common Work Area. ERET is updated by the initialization routine (as well as by other routines) in each of the several phases, so as to indicate the correct entry point of the closing routine in the particular phase. A switch (TERR -- see Appendix IV) turned on by PIROUT to indicate a terminating error, causes the terminating routine in the operative phase to transfer control to the Error Message Editing routine in the next diagnostic output module (IEX21, 31, or 51) for print-out of the errors recorded in the Error Pool. The same TERR switch causes the Error Message Editing Routine in the particular diagnostic output module to transfer control to the terminating routine in the Termination Phase (IEX51), rather than to the next successive phase.

Where a program interrupt occurs in the closing routine of the operative phase (in which case the same program interrupt will recur after PIROUT has returned control to the defective closing routine), PIROUT exits directly to the terminating routine in the Termination Phase.

PIROUT is temporarily replaced as the program interrupt exit by the execution of a second, SPIE macro instruction in the initialization routine of the Scan III Phase. The substitute routine provides for special handling of exponent overflow and underflow interrupts, but passes control to PIROUT in all other cases. PIROUT is restored as the program interrupt exit by a final SPIE macro instruction in the closing routine of the Scan III Phase.

I/O Error Routine (SYNAD)

SYNAD is activated by the control program in the event of an unrecoverable I/O error involving the SYSIN, SYSLIN, SYSPUNCH, SYSUT1, SYSUT2, and SYSUT3 data sets. The address of the routine is stored in the relevant DCBs.

The routine closes the affected DCB, records error No. 210 in the Error Pool (using the ddname contained in the DCB), sets the TERR switch on to indicate a terminating error, and passes control to the closing routine in the operative phase, whose entry point is specified in the location ERET. (See also Program Interrupt Routine PIROUT).

Sysprint I/O Error Routine (SYNPR)

The Sysprint I/O Error Routine is activated by the control program in the event of an unrecoverable I/O error involving the SYSPRINT data set. The address of the routine is stored in the relevant DCB.

The routine turns on a switch named PRT (Appendix IV) to indicate that the printer is down, and then enters the SYNAD routine to take the same actions as that taken for all other data sets. The PRT switch (if turned on) causes the Error Message Editing routine to print out a single message (for Error No. 210) on the console typewriter, indicating that the printer is inoperative.

End of Data Routines (EODAD1, EODAD2, EODAD3, AND EODADIN).

The End of Data routines are entered from the control program when a data input operation from the SYSUT1, SYSUT2, SYSUT3, or SYSIN data set is terminated at the end of the data set. The address of the particular End of Data routine is stored in the data set's DCB.

The End of Data routine loads the entry point of the appropriate ECD exit routine in the operative phase, and then passes control to that routine. The entry point of the EOD exit routine is stored by the initialization routine in each phase which processes a data set, at the appropriate one of the locations EODUT1, EODUT2, EODUT3, and EODIN in the Common Work Area. The phases which specify an EOD exit routine for an end of data condition, the data sets involved, and the locations where the entry points are stored, are as follows:

Phase	Data Set	Storage Field for EOD Exit
IEX11	SYSIN	EODIN
IEX20	SYSUT3	EODUT3
IEX30	SYSUT1	ECDUT1
	SYSUT3	ECDUT3
IEX40	SYSUT3	ECDUT3
IEX50	SYSUT2	ECDUT2
	SYSUT3	ECDUT3

Note that End of Data exit routines for SYSUT2 and SYSUT3 are specified both in the the Compilation Phase initialization routine in IEX40, and at the start of IEX50 (see "Phase Initialization" in Chapter 8).

Print Subroutine (PRINT)

The PRINT subroutine prints out text on the SYSPRINT data set on call from the operative phase. Depending on the compiler options specified, the subroutine may be called by the following routines in the modules indicated:

CIB (IEX11) - Source module listing

PRINTITB (IEX20) - Identifier Table listing

COT27 (IEX21, 31, 51) - Diagnostic messages

PRINTT (IEX51) - Object module storage requirements

The text printed out includes front page titles, headlines, as well as variable (compiler-generated) text. A single line of text is printed by each call to PRINT. After a page shift, one or more headlines are printed at the top of the new page before the next line of text is printed. Text other than headlines is assembled by the calling routine in a print buffer previously specified by PRINT (in register 1). Headlines are transmitted by the calling routine in a Common Work Area field named PAGEHEAD (which accommodates up to three lines of text) and are subsequently moved by PRINT to a print buffer for output. The headlines are assembled at PAGEHEAD during initialization of each particular phase.

PRINT maintains both a line and page count, and inserts the control character in the appropriate line of text to effect the required page shift. Before a page is shifted, the next line of text is temporarily moved from the print buffer to a save area, to enable the headline(s), together with the page number, to be printed at the top of a new page. Control characters governing line spacing between headlines are supplied by the calling routine in the headlines.

These characters are used by PRINT to add the correct increment to the line count. The control character to effect a standard single-space line change is inserted by PRINT at the beginning of each new print buffer. Special page shifts, e.g. following the title page, are specified by the calling routine by arbitrarily raising the line count, maintained in the Common Work Area. The calling routine may also suppress one or more headlines by inserting a special character at the beginning of the particular headline.

Data Control Blocks

Control Section IEX00000 contains Data Control Blocks (DCBs) for the following data sets:

SYSPRINT
SYSLIN
SYSPUNCH
SYSUT2
SYSUT3

The DCB addresses are listed in the Common Work Area, following the register save area. The foregoing data sets are required throughout compilation. DCBs for the SYSIN data set, which is not used after the Scan I/II Phase (IEX11), and the SYSUT1 data set, which is not used after the Scan III Phase (IEX30), are stored in the Common Work Area. Data Sets are opened by the Initialization Phase (IEX10), which also modifies the information in the DCBs to reflect special user requirements concerning block sizes and record lengths.

CONTROL SECTION IEX00001 (COMMON WORK AREA)

The Common Work Area is an area of approximately 3500 bytes used by all phases of the Compiler, principally for the construction and/or transmission between phases of small-size tables, essential control information, and address data. Except for a limited number of fields which remain essentially unchanged throughout compilation, the composition of the Common Work Area varies between phases. Its composition is defined by a dummy control section in each phase. The general layout of tables and other data in the Common Work Area in each phase is indicated in the storage maps in Appendix IX-a.

The principal fields which remain fixed in position in the Common Work Area are the following.

Register Save Area

A standard format save area of 72 bytes, addressed throughout compilation by Register 13, is provided for saving registers when control is passed to the control program at any point during execution of the phases IEX10-IEX51.

DCB Addresses

The addresses of the Data Control Blocks of all seven data sets used by the Compiler are recorded in the Common Work Area, immediately below the general save area.

End of Data Exit Addresses

This field contains the entry point(s) of the closing routine(s) to be entered in the operative phase in the event of an End of Data condition on any one of the data sets SYSIN, SYSUT1, SYSUT2, and SYSUT3. The appropriate entry point is fetched from this field by the EODAD routine in the Directory when an EOD condition occurs. The field is updated by each phase at initialization so as to specify the correct closing routine in the phase.

Compiler Control Field (HCOMPMOD)

A three-byte field in the Common Work Area named HCOMPMOD is used as a Compiler Control Field. All except one of the 24 binary positions in this field are used as switches to govern operations in each phase of the Compiler. The significance of each switch is indicated in Appendix IV.

The Control Field, which is initialized by the Initialization Phase (IEX10), indicates, among other things, the Compiler options specified by the user. It also indicates significant error conditions detected by any one phase, which may cause the Compiler to enter Syntax Check Mode, or alternatively, to terminate operations. The compiler options are listed in Chapter 3.

Communication Area

The Communication Area contains addresses, pointers, counters, and other information used by two or more phases in common. The address information may be variable (as in the case of the program interrupt or I/O error closing routine address at ERET, which changes with each phase) or invariable (as in the case of the address of the Common Area Source Text Buffer 1, stored at SRCE1ADD). Counters designate literal number values (e.g. the line count referenced by the PRINT subroutine at LINCNT). Pointers designate address displacement values which may be incremented by several phases in succession (e.g. the pointer PRPT, which is incremented in the Scan I/II and Scan III Phases to indicate the displacement of each point in the object module, beginning with the Constant Pool).

Area Size Table (INBLKS)

The Area Size Table specifies the sizes of work areas or buffers acquired by the individual phases for the construction of tables transferred to auxiliary storage. It also specifies minimum block sizes for certain data sets. The relevant entries in the table are referenced by the initialization routines of the several phases, before the GETMAIN instruction for the particular phase's private area is issued.

The Area Size Table is set up by the Initialization Phase (IEX10), which determines the appropriate size for each work area, according to the SIZE option specified by the user. The table in Appendix VIII shows the increase in work area sizes as the value of the SIZE option increases.

Work areas for small-size tables transmitted between phases via the Common Work Area are defined by a DS statement in the dummy control section defining the Common Work Area in each phase.

Headline Storage Area (PAGEHEAD)

This area is provided for the headlines used in the printed output of the individual phases. The area accommodates up to three 90-character headlines. The appropriate headlines, which are stored in the area at initialization of each phase generating printed output, are fetched by the PRINT subroutine on call from the operative phase.

The principal contents of the variable part of the Common Work Area during the several phases are as follows.

Preliminary Error Pool

A Preliminary Error Pool is provided in the originally assembled Common Work Area, for the recording of any errors which may occur before the main Error Pool is acquired by the Initialization Phase. Any recorded errors are immediately moved to the main Error Pool, after main storage for the latter has been acquired. The Prelimi-

nary Error Pool is deleted after the close of the Initialization Phase.

Data ontrol Blocks for SYSIN and SYSUT1

The DCBs for the SYSIN and SYSUT1 data sets are stored in the variable part of the Common Work Area, since the data sets are not used beyond a certain point, and the area occupied by the DCB's can be released for other uses. The DCB for SYSIN is deleted after the close of the SCAN I/II Phase, while the DCB for SYSUT1 is deleted after the close of the Scan III Phase.

Tables

The following tables are constructed by the several phases in the variable part of the Common Work Area. A majority of these is transmitted to at least one or more subsequent phases via the Common Work Area. A few are used locally only.

IEX11 P.B. No. Table (PBTAB1)
Scope Table (SPTAB)
Group Table (GPTAB)
Semicolon Table (SCTAB) -- local use

IEX20 Program Block Table II (PBTAB2)

IEX30 For Statement Table (FSTAB)

IEX40 Address Table (ATAB) -- local use

IEX50 Program Block Table III (PBTAB3)

IEX51 Program Block Table IV (PBTAB4)

The above list does not include those tables which are transferred to external storage. The processing of all tables, except those used locally, is indicated in detail in Figure 3.

Other Data

The remainder of the variable part of the Common Work Area is used in the various phases for switches, addresses, counters, and pointers of local significance only (i.e. used exclusively by the operative phase). In the storage maps in Appendix IX-a, these areas are identified as "private work areas".

PURPOSE OF THE PHASE

The Initialization Phase:

1. Saves registers used by the Initial Entry Routine in the Directory, and addresses a save area (by loading register 13) for storing registers when lower-level routines, e.g. in the control program, are invoked by any of the subsequent phases. The save area addressed comprises the first 72 bytes of the Common Work Area.

2. Executes the SPIE macro, specifying the PIROUT routine in the Directory as the program interrupt exit.

3. Reads the options specified for the Compiler by the invoking program and turns on a set of switches in the HCOMPMOD Control Field to reflect the options specified.

4. Inserts ddnames (if any are specified by the invoking program) in the corresponding Data Control Blocks.

5. Selects an Area Size Table, according to the machine system capacity indicated by the SIZE option. The Area Size Table specifies the main storage space to be provided in each phase for work areas and buffers, as well as maximum data set block sizes.

6. Acquires main storage for the Common Area, containing the main Error Pool and Source Buffer 1. Any errors detected before the main Error Pool is acquired are recorded in the Preliminary Error Pool in the Common Work Area.

7. Opens all data sets, after specifying the addresses of Open-Exit routines in the Data Control Blocks of the SYSIN, SYSLIN, SYSPUNCH, and SYSPRINT data sets, and after inserting block sizes in the Data Control Blocks of the SYSUT1 and SYSUT2 data sets (the block size is equal to the length of the Source Text Buffer). Block sizes for SYSIN, SYSLIN, SYSPUNCH, and SYSPRINT are inserted by the particular Open-Exit routine, using the block sizes (if any) specified in the DD statements, or the maximum block size specified in the Area Size Table. The block size for SYSUT3 is included in the assembled Data Control Block.

The logic of the Initialization Phase is outlined in Flowcharts 007-010 in the Flowchart section. The following sections describe the principal functions performed.

EXECUTION OF THE SPIE MACRO

At entry to the Initialization Phase, after registers used by the Initial Entry routine have been saved, and after Register 13 has been loaded with the address of the general save area in the Common Work Area, a SPIE macro instruction is executed which specifies the address of the Program Interrupt Exit routine (PIROUT) in the Directory. By virtue of the SPIE macro instruction, the Operating System passes control to PIROUT in the event of a program interrupt. When entered (in the event of a program interrupt), PIROUT passes control to the routine whose address is stored at the location named ERET in the Common Work Area. ERET is updated in each phase so as to indicate the address of the appropriate closing routine in that phase. Immediately after execution of the SPIE macro, the address of the Initialization Phase closing routine GOTOTERM is stored at ERET. GOTOTERM transfers control directly to the Termination Phase (IEX51), after releasing main storage and closing data sets.

GOTOTERM is subsequently replaced as the program interrupt exit by OPEXERR and GOTOEDIT (the latter exits to IEX21 for output of any recorded errors, before transferring control to IEX51).

PROCESSING COMPILER OPTIONS, DDNAMES, AND HEADING INFORMATION

The Compiler may be invoked(a) by means of the job control EXEC statement, i.e. using the facilities of the control program, or (b) by a user-made program. The options open to the user, as well as the concomittant obligations, insofar as the execution of the Compiler is concerned, differ under each of these alternatives.

Where the Compiler is invoked by the EXEC statement, the options specifiable are limited to the compiler control options listed under "Compiler Options" below. Under this alternative, the key-words representing the compiler options specified

are assembled by the control program in an option field addressed by a pointer. At entry to the Compiler, the address of the pointer is contained in Register 1.

Where the Compiler is invoked by a user-made program, the user may specify (a) any of the compiler options, (b) ddnames for data sets, and (c) heading information, consisting of an opening page number for the printed output of the Compiler. Under this alternative, it is the obligation of the user to assemble the key-words representing the compiler options exercised, ddnames (if any) specified, and the heading information, in three separate fields of main storage (hereafter called, respectively, the option field, the ddname field, and the heading field). Each field must be addressed by a pointer in a three-word address list, and the address of the address list must be contained in Register 1 when control is transferred to the Compiler.

Figure 5 pictures the arrangement of the option, ddname and heading fields, and the related pointers. The arrangement is completely analogous under both invocation alternatives, except that in the case of invocation by EXEC statement, the ddname and heading fields are always vacant (the latter fields may also be vacant under the alternative invocation procedure). A vacant field is indicated by the value zero in the corresponding pointers; value zero in Register 1 indicates that all three fields are vacant. A vacant option field indicates that the options exercised are the default options.

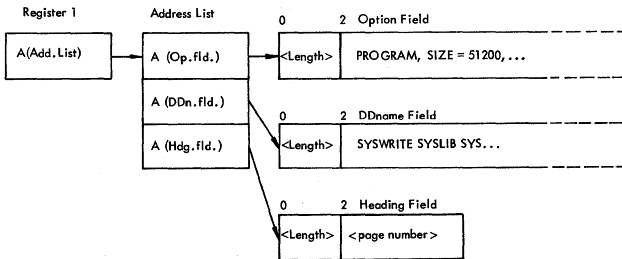


Figure 5. Option, DDname and Heading fields, and pointers

The first bit of each full-word pointer in the address list functions as a flag, to indicate whether or not the field currently being processed is the last to contain significant data. The bit is tested after each field has been processed, to determine whether the next field is to be processed. Thus, for example, if the flag bit is on in the option field pointer, indicating that the ddname and heading fields are vacant, the DDNAMES and HEADINFO routines, which process the ddname and heading fields will be bypassed. The address of the address list is obtained from the Operating System's save area, in which the contents of Register 1 will have been stored after entry to the Compiler.

COMPILER OPTIONS

The processing of Compiler options consists in reading the key-words listed in the option field and in setting appropriate switches in the HCOMPMOD Control Field (Appendix IV) to reflect the particular options specified.

The key-words representing valid options which may be specified for a compilation are as follows (the first key-word corresponds to the default option):

SIZE = [a number ≥45056]
PROGRAM (PG) or PROCEDURE (PC)
SHORT (SP) or LONG (LP)
SOURCE (S) or NOSOURCE (NS)
LOAD (L) or NOLOAD (NL)
NODECK (ND) DECK (D)
EBCDIC (EB) or ISO (I)
TEST (T) or NOTEST (NT)

The letters within parentheses represent the alternative (abbreviated) form in which the option may be specified. The key-words are recorded in the option field in EBCDIC code and are separated by commas. Except in the case of the SIZE option, each option is identified by comparing the key-word with a list of 28 possible key-words in a table named PARMLIST.

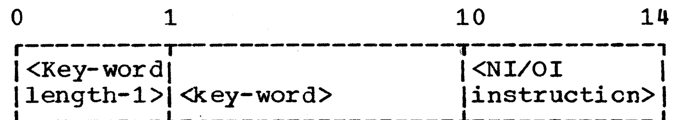


Figure 6. PARMLIST Table entry for a Compiler option key-word

In addition to the option key-word and the key-word length (-1), each entry in the PARMLIST Table contains a logical instruction (NI or OI) which, when executed, turns on a specified switch in the HCOMPMOD Control Field (Appendix IV). As soon as a key-word in the PARMLIST Table is found which matches the key-word in the option field, the instruction in the table is EXECUTEd, turning on the appropriate switch in the Control Field and thus recording the option specified.

The SIZE option is identified by a CLI instruction. After recognition, the size specified is converted to binary and stored

at SIZE, provided it is not less than the minimum capacity required. The size is subsequently referenced in selecting the Area Size Table (see below). If an option is incorrectly specified, error No. 200 is recorded in the Preliminary Error Pool in the Common Work Area and the default option is assumed. If the main storage size specified is less than the minimum, error No. 208 is recorded, and the minimum size of 45,056 is assumed. The contents of the Preliminary Error Pool are subsequently moved to the main Error Pool after the Common Area has been acquired.

DDNAMES

The processing of ddnames consists in transferring the ddnames (if any) from the ddname field to the relevant Data Control Blocks. Unless a ddname field is provided in a user-written program which invokes the Compiler, the ddname field is vacant (Data Control Blocks contain the ddnames required by the Compiler).

The ddnames, each consisting of a maximum of eight EBCDIC characters, will have been entered in the ddname field in prescribed positions, according to the physical device involved. Data Control Block addresses are listed, in corresponding order and position, in the Common Work Area, beginning at LINADD. This enables the ddnames to be transferred to the appropriate DCB in sequence.

HEADING INFORMATION

The heading information which may be specified, consists solely of a starting page number for the printed output of the Compiler. Where no page number is specified, page numbering begins with the number 1.

The page number (if any) in the heading field is moved to a counter named PAGECNT, which is updated and referenced by the PRINT subroutine in the Directory.

SELECTION OF AREA SIZE TABLE (FNDARSIZ)

With the exception of load modules IEX10, IEX21, IEX31, and IEX51, each phase of the Compiler acquires a private area containing one or more work areas or buffers for the construction, processing, or output of working tables. The Initialization Phase (IEX10) acquires a Common Area used by all phases (see below).

To enable the Compiler to adapt itself flexibly to the available storage capacity, the space allotment for certain work areas is scaled to the capacity of the particular machine system as specified in the SIZE option. Twelve capacity levels are established, beginning at 45,056 bytes and graduated upwards at increasing intervals, up to a maximum of 999,999 bytes. At each capacity level, specific area sizes are defined for all work areas and buffers. Capacity levels and area sizes are defined by twelve Area Size Tables, the first of which is named ARTAB.

The FNDARSIZ routine selects the appropriate Area Size Table, according to the machine capacity specified in the SIZE option, and moves the table to the Common Work Area at the field beginning at INBLKS. The table thus selected, which specifies the main storage space to be acquired for all work areas, is referenced by the main working phases at initialization, before the GETMAIN instruction for the phase's private area is executed.

In addition to work area sizes, the Area Size Table also specifies the maximum block sizes for the SYSIN, SYSPRINT, SYSLIN, and SYSPUNCH data sets. The maximum block sizes are referenced by the Open-Exit routines (see below).

Appendix VIII shows the increase in the size of work areas, buffers and maximum block sizes as the SIZE option increases.

ACQUISITION OF COMMON AREA

The Initialization Phase acquires a Common Area containing Source Buffer No.1 and the Error Pool, in which compile time errors detected in the several phases are recorded. The sizes of the buffer and Error Pool are obtained from the Area Size Table. The use of Source Buffer No. 1 is discussed under "Use of Main Storage" in Chapter 1.

After acquisition of the Error Pool, the contents (if any) of the Preliminary Error Pool in the Common Work Area are moved to the newly acquired Error Pool.

OPENING OF DATA SETS

The Initialization Phase opens all data sets used by the Compiler, namely SYSLIN,

SYSPRINT, SYSIN, SYSPUNCH, SYSUT1, SYSUT2, and SYSUT3. The DCBs of SYSIN and SYSUT1 are contained in the Common Work Area (this facilitates the release of main storage for other uses when the data set is no longer needed after the Scan I/II and Scan III Phases, respectively); all other DCBs are contained in Control Section IEX00000 of the Directory. DCB addresses are listed in the Common Work Area, beginning at LINADD.

Immediately before the OPEN macro instruction is executed, the addresses of the Open-Exit routines INEXRT, LINEXRT, PCHEXRT, and PRTEXRT are stored in the SYSIN, SYSLIN, SYSPUNCH, and SYSPRINT DCBs. The Open-Exit routines, which are entered from the Operating System when the OPEN macro instruction is issued, serve to verify that the block size (if any) specified, is a multiple of the record length and does not exceed the maximum specified in the Area Size Table. If the block size is not specified at invocation or if the block size is incorrectly specified, the Open-Exit routine inserts the record length as the block size. If the block size is incorrectly specified, an error is recorded, and in the case of SYSIN, the NOGO switch (Appendix IV) is turned on, causing compilation to be subsequently terminated. In the case of the SYSUT1 and SYSUT2 data sets, the block size (equal to the source buffer length specified in the Area Size Table) is inserted directly, before the OPEN macro instruction is issued. In the case of SYSUT3, the block size is specified in the DCB at assembly time.

When control is recovered from the Operating System OPEN routine, a test is made to determine if the SYSPRINT data set has been opened (in the negative case, Error No. 201 is recorded and the PRTNO and NOGO switches are turned on, causing compilation to be terminated after the error message has been printed out by Load Module IEX21 on the console typewriter). If the data set has been successfully opened, the date is derived and edited from the system clock, and the title "LEVEL 1 JUL 67 OS ALGOL F DATE [date]" is printed on a new page.

Tests are then made to determine if the remaining data sets have been opened. If all data sets have been correctly opened, control is passed to the Scan I/II Phase (IEX11). If any data set has not been opened, an error is recorded, and in the case of SYSIN, SYSUT1, SYSUT2, or SYSUT3, the NOGO and TERR switches are turned on, causing compilation to be terminated after recorded error messages have been printed out by Load Module IEX21.

PURPOSE OF THE PHASE

The purpose of the Scan I/II Phase is to read the source module and perform the following principal tasks.

1. To tabulate and classify all valid identifiers declared or specified in the source module, in the Identifier Table. Declared identifiers include those designated by such declarators as 'REAL', 'INTEGER', 'ARRAY', or 'PROCEDURE', among others, as well as labels. Specified identifiers are formal parameters of procedures, specified in a procedure heading.

   The Identifier Table, which is further processed in the two subsequent phases, facilitates the construction of the internal names of identifiers and the replacement of identifiers in the source text by their internal names. An identifier's internal name consists of a five-byte unit containing a descriptive characteristic, a Program Block Number, and a displacement address.

   The Program Block Number specifies (indirectly) a Data Storage Area, comprising the object time storage area required for all identifiers declared or specified in the particular block or procedure. The displacement address specifies (in the case of a declared label, switch, or procedure identifier) the displacement of an entry in the object time Label Address Table, or (in the case of all other identifiers) the displacement of a storage field in the particular Data Storage Area.

   The entries in the Identifier Table consist of the identifier's external name (represented by a maximum of six characters translated to internal code), followed by the five-byte internal name described above. For declared label, switch, and procedure identifiers, the complete entry, comprising external and internal name, is constructed by the present phase. For all other identifiers, the present phase enters the external name and constructs all except the address part of the internal name. The Data Storage Area displacement address is inserted in the entry by the Identifier Table Manipulation Phase, in which object time storage fields are allocated to all identifiers listed in the table, other than declared label, switch, and procedure identifiers.

   The Identifier Table is terminated in the Scan III Phase, when all externally represented operands in the source text are replaced by their internal names in the table.

2. To assign a serial Program Block Number to every block and procedure in the source text. The same Program Block Number appears in the internal names of all identifiers declared or specified in the particular block or procedure.

   At object time, the Program Block Number references an entry in the Program Block Table, containing, among other things, the size of a Data Storage Area. In the object code generated by the Compilation Phase, an operand is represented by the address of the Data Storage Area (loaded in a base register) and the displacement contained in the operand's internal name.

3. To generate a transformed source text, called Modification Level 1. A second transformation of the source text, called Modification Level 2, is generated by the Scan III Phase. The changes reflected in the first transformation include an initial one-for-one translation of all characters in the source text to the internal code, the replacement of all ALGOL delimiter words by one-byte operators, and the removal of declarations, except procedure, array, and switch declarations, from the source text. These and other changes are described in a later section under the heading "Modification Level 1 Source Text".

4. To store strings enclosed by string quotes, '(' ')', in the Constant Pool, and to replace the string in the transformed source text by an internal name referencing the location where the string was stored. All constants other than strings are stored in the Constant Pool by the Scan III Phase.

5. To recognize syntactical errors in the source module and to store appropriate error patterns in the Error Pool. The contents of the Error Pool are printed

out in the form of diagnostic messages by the Error Message Editing routine in the next module but one (IEX21), after execution of the Identifier Table Manipulation Phase.

6. To print a listing of the source module, if the SOURCE option is specified.

7. To assign a serial Identifier Group Number to every block, procedure, and for statement in the source module. The Identifier Group Number is used in the Scan III Phase to verify the validity of goto statements, and to facilitate the classification of for statements (see Item 8).

8. To construct a Group Table listing all Identifier Group Numbers and identifying each for statement represented in the list. The Group Table is used in the Scan III Phase, in classifying the optimizability of for statements containing goto statements which imply a branch out of the for statement.

9. To construct a Scope Table indicating the Program Block Number of the block or procedure enclosing every for statement. The Scope Table is used in the Scan III Phase to ascertain if all terms of subscript expressions of array identifiers occurring in for statements are valid (i.e. declared) outside the for statement. This is one of several conditions for subscript optimization.

10. To construct the Program Block Number Table, indicating the Program Block Number of the block or procedure immediately enclosing every block and procedure in the source program. The table is constructed for purposes of user-information and is used in the Identifier Manipulation Phase in the print-out of the Identifier Table.

SCAN I/II PHASE OPERATIONS

The two primary functions of the Scan I/II Phase are:

1. To tabulate all identifiers declared or specified in the source module, in the Identifier Table.

2. To generate a transformed source text (Modification Level 1).

In principle, these functions are performed by searching the source text for ALGOL delimiter words (e.g., 'BEGIN' or 'STEP'), as well as other multicharacter operators (e.g., := or .,). If a delimiter constitutes a declarator (e.g., 'INTEGER'), or a specificator, entries are made for the immediately following identifiers in the Identifier Table, after each identifier has been checked for validity. Otherwise, a one-byte symbol representing the delimiter is transferred to the output buffer. Other multicharacter operators are similarly replaced by one-byte symbols. Statements, containing externally represented operands (identifiers) and operators are transferred unchanged, except that any delimiter words and multicharacter operators within the statement are replaced by one-byte symbols. (See "Modification Level 1 Source Text" in this chapter.)

The following provides a general description of the main operations performed in the Scan I/II Phase, illustrated graphically by the diagram in Figure 7. The description is intended to be read in conjunction with the diagram.

At the extreme left of the diagram, it will be seen that the source module (in card or card-image records, EBCDIC or ISO code) is read from the SYSIN data set into an 80-byte field of the Common Work Area by the CIB subroutine. Immediately after read-in, a copy of the record is moved to a print area (or a dummy print area, if the SOURCE option was not specified), and the record in the Work Area is then translated to the internal code (Appendix I-a. Appendix I-b shows the same character set, expanded by the characters which replace delimiter words). The untranslated source text in the print area is used in printing a listing of the source module. It is also used to enable character strings to be stored in the Constant Pool in their original EBCDIC or ISO code. The CIB subroutine, first activated at phase initialization, is subsequently called by any routine which detects the record-end operator Zeta (the operator is inserted by CIB at the end of each translated source record).

In the Work Area, the translated source text is scanned by the TESTLOOP routine, which searches for any of 14 different characters. As soon as any one of these characters is identified, TESTLOOP moves the preceding scanned characters to the Modification Level 1 text in an output buffer, and then activates the appropriate routine. The diagram indicates the routines activated in the case of 12 of the 14 characters (the remaining two are the Blank and the Invalid Character, which are, in effect, ignored).

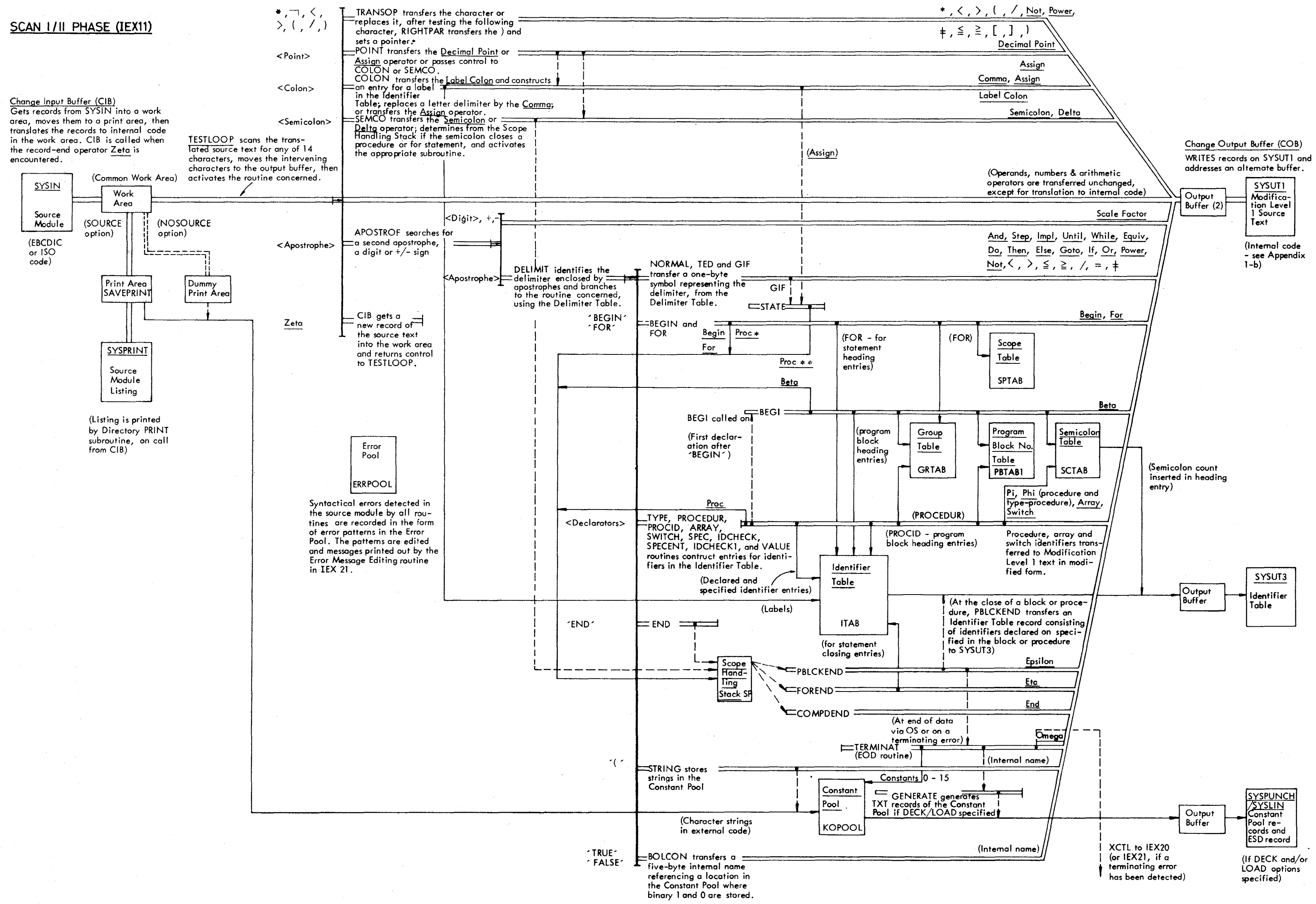


Figure 7. Scan I/II Phase. Diagram illustrating functions of principal constituent routines

For a majority of the characters, the character is simply transferred to the output buffer or replaced by another character, depending on the character which follows. In the case of a colon, the COLON routine may:

1. Transfer a Label Colon and construct an entry in the Identifier Table for the preceding label;

2. Replace a letter string by a Comma; or

3. Transfer the Assign operator.

In the case of a semicolon, the SEMCO routine inspects the Scope Handling Stack to determine if the semicolon closes a procedure or a for statement, and if so, activates the appropriate subroutine (PBLCKEND or FOREND). See "Close of Scopes". If a semicolon terminates a declaration, SEMCO transfers the Delta operator to the Modification Level 1 text; otherwise, the Semicolon operator is transferred.

The record-end operator Zeta causes TESTLOOP to call the CIB subroutine, which reads in a new record and translates it to the internal code.

The apostrophe leads into the Apostrophe routine (APOSTROF). APOSTROF scans the text immediately following the apostrophe, for a digit or +/- sign, a second apostrophe, or one of a set of logical operators. A digit or +/- sign identifies the apostrophe as the Scale Factor. A second apostrophe indicates an ALGOL delimiter word (that is, a string of letters or an operator enclosed by apostrophes). In this case, the Delimiter routine (DELIMIT) is entered. If the scan is terminated by a logical operator (indicating that the closing apostrophe of a delimiter is missing), the Delimiter Error routine (EROUT -- not shown in the diagram) is activated. EROUT differs from DELIMIT, described below, only in point of procedural detail.

DELIMIT compares the characters enclosed by apostrophes with a list of 38 delimiter words in the Delimiter Table (WITAB) and branches to the routine specified in the table for the particular delimiter. A majority of delimiters (21) lead into the NORMAL, TED, or GIF routines, which simply transfer the one-byte symbol in the Delimiter Table to the Modification Level 1 text. Declarators and specificators lead into routines which construct entries in the Identifier Table for the immediately following identifiers.

OPENING OF SCOPES

Whenever the delimiter opening a block, a procedure, a for statement, or a compound statement is encountered, a one-byte operator identifying the particular scope is entered in the Scope Handling Stack. The operators Beta (for a block), Proc (for a procedure), For (for a for statement), and Begin (for a compound statement), are stacked by the BEG1, PROCEDUR, FOR, and BEGIN routines, respectively. Depending on the structure of the body of a procedure, the operator Proc may subsequently be replaced by the operators Proc* or Proc** in the BEGIN, STATE, or FOR routines. See "Scope Handling Stack".

At the beginning of every block and procedure, a program block heading entry, containing a new Program Block Number, is constructed in the Identifier Table. The Program Block Number in the heading entry is copied into the following identifier entries representing identifiers declared or specified in the particular block or procedure. Similarly, at the opening of every for statement, a for statement heading entry is constructed in the Identifier Table. The for statement heading entry is subsequently deleted unless it is followed by one or more identifier entries representing a label or labels declared inside the particular for statement. In the latter case, a for statement closing entry is made at the end of the for statement. Program block heading entries are constructed by BEG1 (for a block) and PROCID (for a procedure); for statement heading and closing entries by FOR and FOREND, respectively.

The BEG1 subroutine, which stacks the operator Beta and constructs the program block heading entry at the opening of a new block, is entered from any routine processing the first declaration following the delimiter 'BEGIN'. Entry to BEG1 is governed by a switch named BEGBIT, which is turned on by the BEGIN routine, entered from DELIMIT on recognition of the delimiter 'BEGIN'. BEGBIT is tested in all declaration-processing routines (immediately after entry from DELIMIT), and if the switch is on, a call is made to BEG1 before the particular declaration is processed.

BEG1 and PROCEDUR also construct entries in the Group Table, Program Block Number Table, and Semicolon Table. FOR makes entries in the Scope Table and Group Table.

PROCESSING OF DECLARATIONS AND SPECIFICATIONS

In the construction of entries in the Identifier Table for declared or specified identifiers, the external name is copied from the translated source text in the Work Area, while the characteristic is inserted by an MVI instruction or, in the case of specified identifiers, copied from the Delimiter Table.

Type declarations ('REAL', 'INTEGER', and 'BOOLEAN') are processed by the TYPE routine.

All type declarations are completely removed from the Modification Level 1 source text, whereas procedure, switch and array declarations are represented in the modified source text by a one-byte declarator, followed by the identifier(s), as well as parameters, components, or dimensions.

Array and switch declarations are processed by the ARRAY, SWITCH, and LIST routines. The main function of the LIST routine, which branches to several subroutines is to count the number of dimensions or components of arrays and switches, and to store this information in the appropriate identifier entries.

Entries for declared procedure identifiers are made by the PROCEDUR, PROCID, and IDCHECK1 routines. The external names of formal parameters in the parameter list following a procedure identifier are copied into the Identifier Table by the IDCHECK1 subroutine on call from PROCID. The characteristics of formal parameters are entered subsequently when the specifications in the procedure heading are processed. The routines which process specifications include, firstly, the TYPE, VALUE, SPEC, ARRAY, SWITCH, and PROCEDUR routines (depending on the particular specificator), and secondly, the SPECENT and IDCHECK routines (SPECENT is a special entry point of IDCHECK).

To distinguish between declarations and specifications, a switch named PROBIT is used. PROBIT is turned on by PROCEDUR, as soon as a procedure declaration is recognized, to signify that a procedure heading has been entered. If a delimiter (say 'REAL') is subsequently encountered, the condition PROBIT=1 signifies that the delimiter is a specificator rather than a declarator and causes the particular routine activated (TYPE in this case) to branch directly to SPECENT.

After copying the appropriate characteristic from the Delimiter Table to a standard storage location, SPECENT (or IDCHECK) compares each identifier following the specificator ('REAL' in this example) with the formal parameters previously copied into the Identifier Table from the parameter list, and when the matching identifier is found, moves the characteristic into the identifier entry.

No part of the procedure heading except the procedure identifier and the parameter list is transferred to the Modification Level 1 text. Type-qualified procedure and array declarations are processed by the TYPE, TYPPROC, or TYPARRAY, and PROCEDUR or ARRAY routines, in that order.

CLOSE OF SCOPES

When the delimiter 'END' is encountered, the END routine inspects the operator at the top of the Scope Handling Stack and calls an appropriate subroutine (PBLCKEND, FOREND, or COMPDEND), according to the stack operator detected.

PBLCKEND is called if 'END' closes a block or a procedure (indicated by the stack operators _Beta_, _Proc_, _Proc*_ or _Proc**_). PBLCKEND transfers the last block of entries in the Identifier Table representing identifiers declared or specified in the closed block or procedure, to the SYSUT3 data set; releases the stack operator; and transfers the closing operator _Epsilon_ to the Modification Level 1 text. FOREND and COMPDEND (which are called if the stack operator is _For_ or _Begin_, respectively), transfer the operators _Eta_ or _End_ to the modified text, and release the stack operator. FOREND may also construct a for statement closing entry in the Identifier Table, or delete the preceding for statement heading entry.

The Scope Handling Stack is also inspected by the SEMCO routine in case a semicolon closes a procedure or a for statement. In the affirmative case, the PBLCKEND or FOREND subroutine is called.

END OF PHASE

The Termination routine (EODADIN), which closes the Scan I/II Phase, is normally entered as an EOD (End of Data) routine from the Operating System, after the PBLCKEND subroutine has detected the final exit from the outermost scope of the source module and has initiated a special scan of the closing text, designed to detect possible logical errors. EODADIN may also be entered when a terminating error has been

detected in the source module, in which case control is passed directly to Diagnostic Output Module IEX21, rather than to the Identifier Table Manipulation Phase (IEX20). The conditions under which EODADIN is entered are described more fully under "Close of Scan I/II Phase".

Flowcharts 011 and 012 in the Flowchart Section indicate the logical arrangement of the principal routines in the Scan I/II Phase. All of the major routines illustrated in the diagram in Figure 7, namely TESTLOOP, APOSTROF, and DELIMIT, can be readily distinguished in the charts. The various levels of routines entered from each of these routines may be seen in both the chart and the illustrative diagram.

The name of this phase, Scan I/II, derives from the fact that the source module is twice scanned in the phase, first by the Change Input Buffer subroutine (CIB), when the source text is translated to the internal code, and second by TESTLOOP, APOSTROF, or some other lower level routine.

PHASE INPUT/OUTPUT

Figure 8 pictures the data input to and output from the Scan I/II Phase. The figure also indicates the tables and other data transmitted to the subsequent phases via main storage.

Input consists of the source module on the SYSIN data set (card reader, disk unit, or magnetic tape unit). Input records, 80 characters in length, are read into the Work Area (WA) by means of a GET macro instruction.

The transformed source text (Modification Level 1) output by the phase is transferred to the SYSUT1 data set by a WRITE macro instruction from two alternating output buffers in unblocked, fixed length records. At phase termination, the data set is closed by a Type T CLOSE (no repositioning to the beginning of the data set). Records are numbered serially from 0. In the event the transformed source text occupies less than one full buffer, it is transmitted to the Scan III Phase via main storage.

The Identifier Table is transferred to the SYSUT3 data set by means of a WRITE macro instruction, in variable-length records of up to 2000 bytes (181 Identifier Table entries of eleven bytes each). Each record comprises the set of identifiers declared or specified in a block or procedure. The record number, represented by the Program Block Number of the block or procedure, and the record length are contained in the first (heading) entry.

An ESD record for the object module and TXT records of the strings stored in the Constant Pool are generated on the SYSLIN and/or SYSPUNCH data sets, provided the options LOAD and/or DECK are specified in the EXEC job control statement. If the source module is a precompiled procedure to be stored on a partitioned data set, the ESD record will contain the procedure name. If the SOURCE option is specified, a listing of the source module is printed out on SYSPRINT.

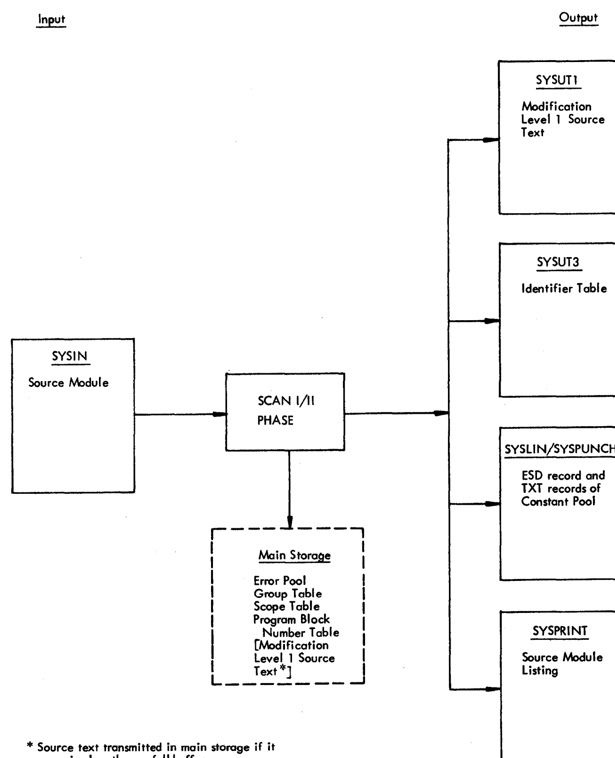


Figure 8. Scan I/II Phase Input/Output

IDENTIFIER TABLE (ITAB)

The Identifier Table (ITAB) is a working record in which an internal form of operand representation, facilitating later compilation operations, is constructed for every valid identifier declared or specified in the source module. This internal representation, referred to as an identifier's internal name, replaces all externally represented operands in the source module. The replacement is made in the Scan III Phase after the construction of the Identifier Table has been completed by the Identifier Table Manipulation Phase.

The entry constructed for an identifier, called an identifier entry, is eleven bytes in length. It contains up to six characters of the identifier's external name, translated to internal code, and a five-byte internal name. For declared procedure and switch identifiers and labels, the complete entry, comprising external and internal name, is constructed in the Scan I/II Phase. For all other identifiers, the external name and all except the address part of the internal name is constructed in the present phase, the address part being inserted in the Identifier Table Manipulation Phase.

Each set of identifier entries representing identifiers declared or specified in a block or procedure, is headed by a program block heading entry. The heading entry contains the Program Block Number assigned to that block or procedure. At the close of a block or procedure, the block of entries relating to that block or procedure is transferred as a record to the SYSUT3 data set.

Within a given block of entries, an entry (or entries) representing a label (or labels) declared inside one or more for statements, is enclosed by one or more for statement heading entries and a for statement closing entry. The Identifier Group Numbers in the for statement heading and closing entries are used, in the Scan III Phase, in detecting illegal branches into for statements. (SWILA routine in IEX30.) The processing of the Identifier Table is described in further detail in a later section.

IDENTIFIER ENTRIES

Figure 10 shows the content of the eleven-byte entry constructed in the Scan I/II Phase for all identifiers except those of declared arrays, procedures, switches, and labels. The identifier's external name, represented by a maximum of six characters in internal code (Appendixes I-a and I-b), is copied from the translated source text in the Work Area, after the full identifier has been checked for validity. If the identifier does not satisfy the specifications of the OS/360 ALGOL Language with respect to validity, no entry is made, and an error is recorded in the Error Pool.

The two-byte characteristic, in the case of declared identifiers, is provided by the program (i.e. by an MVI instruction). In the case of specified identifiers, the characteristic is copied from the Delimiter Table (see DELIMIT routine). If an array or procedure identifier is type-qualified, the characteristic is modified by a logical instruction to show the type.

The hexadecimal value of the characteristic for each type of identifier is shown in the table in Appendix II. The characteristic, which serves to describe the identifier, is inspected in the subsequent phases. Each of the binary positions in the characteristic identifies (when set = 1) a particular characteristic of the identifier. The significance identified with each position is shown in Figure 9. Bits 5 and 6 of the first byte are designated Special Use Bits because they may be manipulated in the Scan III Phase if the identifier is a critical identifier, that is, if the identifier occurs in a for list.

First Byte (Byte 6 in identifier entry)

Bit No:	Description	
0 }	Operand	(See use of bits 0-2 in "Operator/Operand Stacks" - Chapter 8)
1 }		
2	Not used	
3	Not used	
4	No Assignment	
5	Special Use 1	
6	Special Use 2	
7	String	

Second Byte (Byte 7 in identifier entry)

Bit No:	Description	
0	Standard Procedure	} Procedure
1	Code Procedure	}
2	Call by Value	} Simple Variable
3	Call by Name	}
4	Label	
5	Array	
6	Real	} Boolean
7	Integer	}

Figure 9. Identifier Characteristic

The Program Block Number (P.B.No.) is copied from the program block heading entry of the block or procedure in which the identifier is declared or specified.

With the exceptions already noted and described more fully below, the last two bytes of the identifier entry as constructed in the Scan I/II Phase, are filled with zeros. They are reserved for a relative address which is inserted by the Identifier Table Manipulation Phase. The address specifies the identifier's object time storage field within the Data Storage Area provided for the block or procedure in which the identifier was declared or specified.

Figure 11 shows the content of the entry constructed for a declared array identifier. The external name, characteristic, and

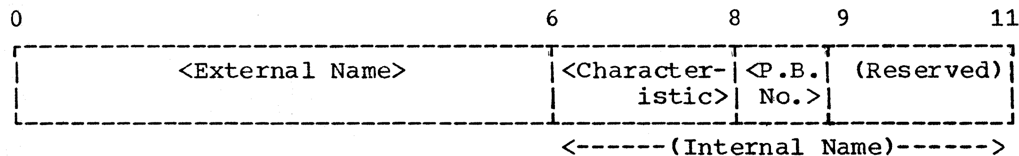
Program Block Number are entered in the manner described above. The number of subscripts (or dimensions) of the array is entered in the first half of byte 9. The last one-and-a-half bytes, filled with zeros in the Scan I/II Phase, are reserved for the relative address of the array's Storage Mapping Function in the particular Data Storage Area. The address is inserted in the Identifier Table Manipulation Phase.

The entry constructed for a declared procedure identifier is shown in Figure 12. The external name and characteristic are entered in the manner described earlier. A new Program Block Number is assigned to the procedure. This same Program Block Number appears in the immediately following program block heading entry, which heads the set of entries representing formal parameters specified in the procedure. The number of parameters of the procedure is entered in the first half of byte 9. The last one-and-a-half bytes of the entry contain the relative address, referred to as the Label Number (LN), of a four-byte entry reserved in the object time Label Address Table (LAT). At object time, the Label Address Table entry contains the absolute address of the object code generated for the procedure.

In the case of a declared type-procedure, the heading entry which follows the procedure identifier entry is followed by a second entry for the procedure identifier. The two identifier entries for a type-procedure are identical, except that in the entry which precedes the heading entry, the first byte of the characteristic is equal to hexadecimal CA, while in the entry which follows the heading entry, the first byte of the characteristic is equal to hexadecimal C2.

Figure 13 shows the entry constructed for a declared switch identifier. The entry is identifical with that for a declared procedure identifier, except that the first half of byte 9 contains the number of components of the switch, minus one.

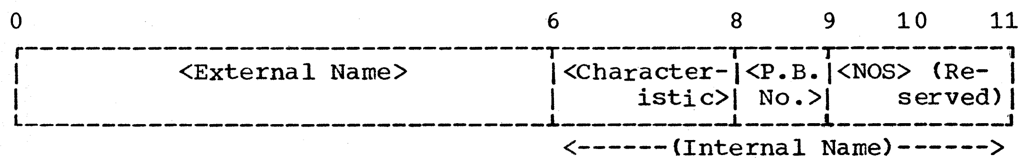
The entry constructed for a declared label is shown in Figure 14. The entry differs from that for a procedure identifier only in that the first half of byte 9 is unused and set to zero.



<P.B.No.> = <Program Block Number>

(Reserved) = The last one-and-one-half bytes are reserved for the relative address of the identifiers's object time storage field -- inserted by the Identifier Table Manipulation Phase

Figure 10. Identifier Table entry for all identifiers except declared array, procedure and switch identifiers and labels



<P.B.No.> = <Program Block Number>
<NOS> = <Number of subscripts, minus one>

(Reserved) = The last one-and-one-half bytes are reserved for the relative address of the array's Storage Mapping Function, inserted by the Identifier Table Manipulation Phase.

Figure 11. Identifier Table entry as constructed in the Scan I/II Phase for a declared array identifier

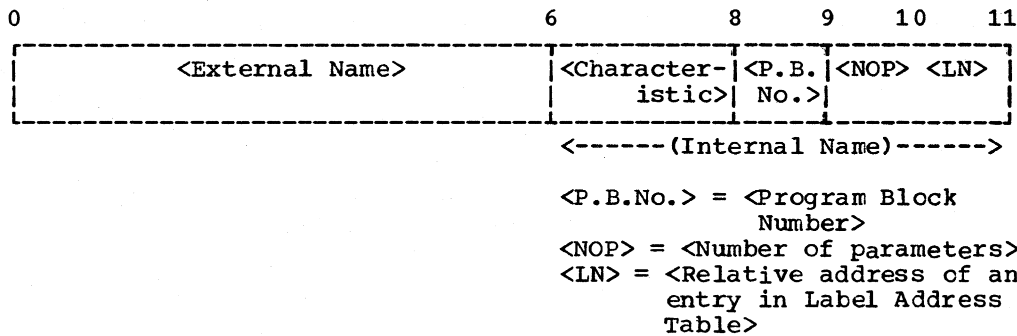


Figure 12. Identifier Table entry for a declared procedure identifier

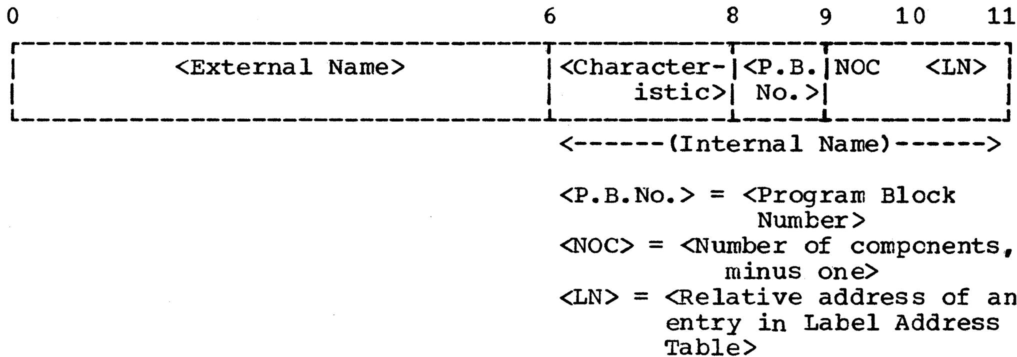


Figure 13. Identifier Table entry constructed in the Scan I/II Phase for a declared switch identifier

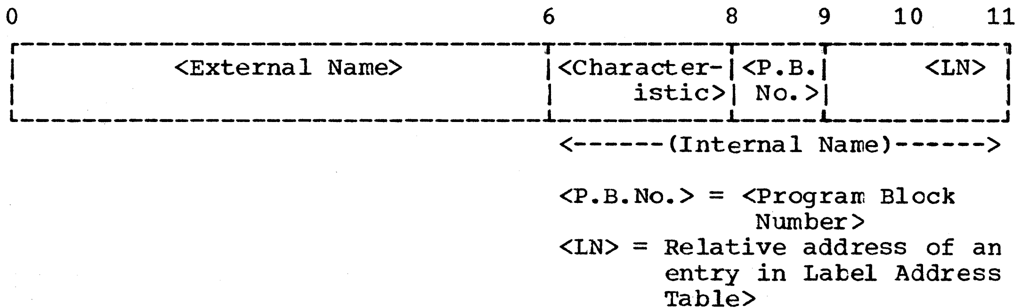


Figure 14. Identifier Table Entry constructed in the Scan I/II Phase for a declared label

PROGRAM BLOCK HEADING ENTRIES

A program block heading entry heads every set of identifier entries representing identifiers declared or specified in a block or procedure.

Figure 15 indicates the content of the eleven-byte program block heading entry. The first eight bytes provide two four-byte save areas, in which the contents of the pointers LIGP and LPBP are stored, before these pointers are set to the address of the heading entry itself. The first bit of byte 8 functions as a switch to indicate if the scope is a type-procedure. In this case, the bit is set = 1; in all other cases, it is set = 0. The Identifier Group Number (I.G.No.) and Program Block Number (P.B.No.) are copied from two counters (IGN and PBN). IGN is incremented for every block, procedure, and for statement, while PBN is incremented for blocks and procedures only.

At the close of a block or procedure, when the set of entries representing identifiers declared or specified in the block or procedure is transferred to a utility data set, the length of the record to be transferred and the semicolon count (copied from the corresponding entry in the Semico-

lon Table) are inserted in the heading entry, as indicated in Figure 16.

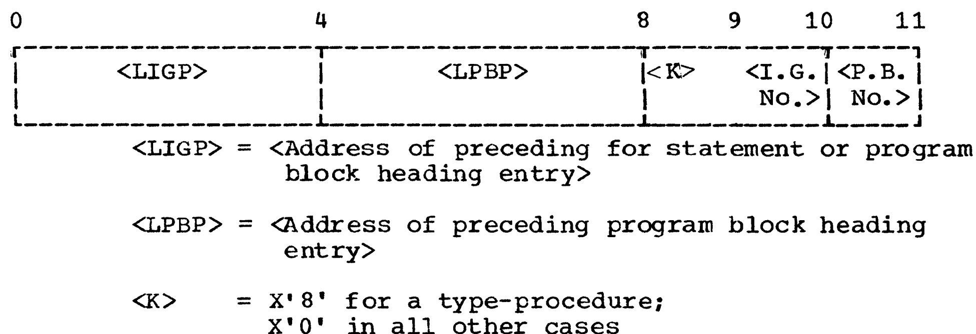


Figure 15. Program block heading entry

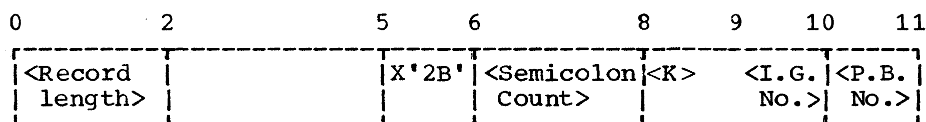


Figure 16. Program block heading entry, as transmitted to the SYSUT3 data set

FOR STATEMENT HEADING AND CLOSING ENTRIES

A for statement heading entry is constructed in the Identifier Table as soon as the delimiter FOR is encountered. If no labels are declared inside the for statement (or a nested for statement), the entry is deleted at the close of the for statement. If, however, any labels are declared inside the for statement, a for statement closing entry is constructed at the close of the for statatement. Where a label is declared inside a series of nested for statements, the entry for the declared label is preceded by a heading entry for each enclosing for statement, and is followed by a single closing entry containing the Identifier Group Number of the embracing block or procedure.

The first four bytes of the for statement heading entry are used as a save area in which the contents of the pointer LIGP are stored before that pointer is reset to the address of the heading entry itself. The Identifier Group Number (I.G.No.) is copied from the counter IGN, which is incremented successively for every block, procedure, and for statement in the source module.

The Identifier Group Number in the for statement closing entry is copied from the heading entry of the reentered scope.

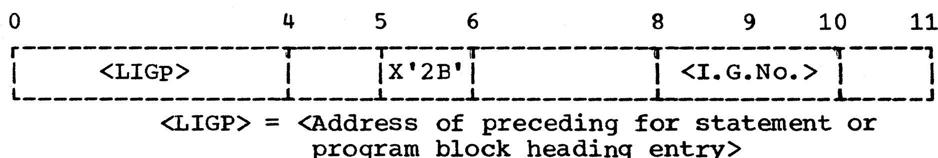


Figure 17. For statement heading entry

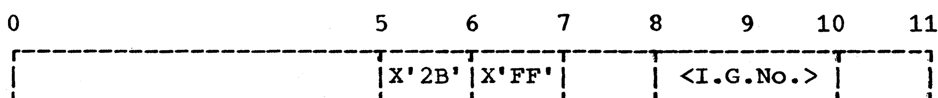


Figure 18. For statement closing entry

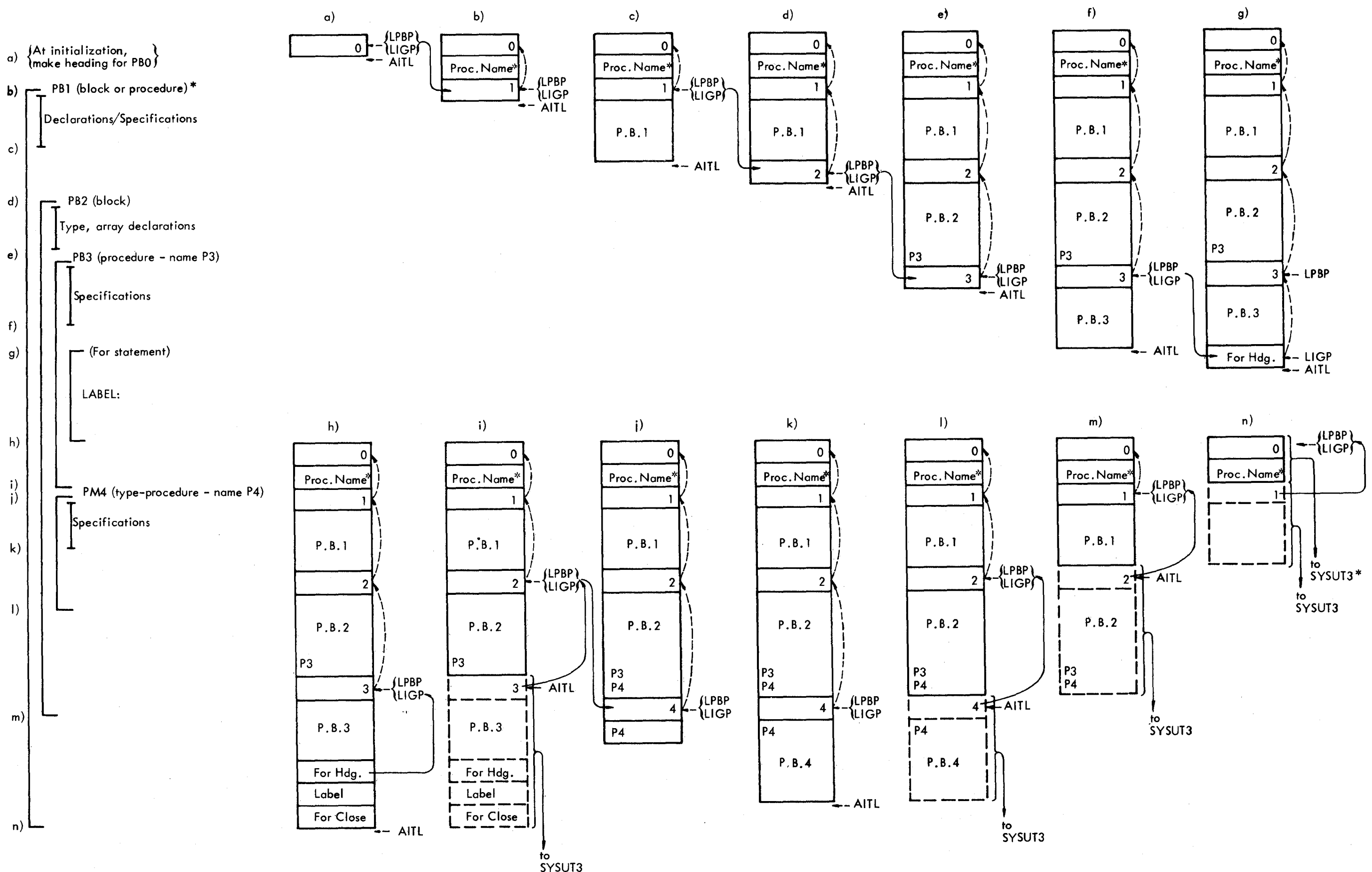
PROCESSING OF THE IDENTIFIER TABLE

The diagram in Figure 19 illustrates the processing of the Identifier Table in the Scan I/II Phase.

At entry to every block or procedure, a program block heading entry, containing a new Program Block Number, is constructed. (In the case of a procedure, the heading entry is preceded by an entry containing the procedure identifier.) Program block heading entries are constructed by the BEG1 subroutine, on call from declaration-processing routines, and by the PROCID routine, entered from PROCEDUR. At entry to a for statement, a for statement heading entry is constructed by the FOR routine. At the close of a for statement, the heading entry may be deleted, or if any labels are declared in the for statement, a for statement closing entry is constructed following the entry for the label. At the close of a block or procedure, the set of entries representing identifiers declared or specified in the block or procedure are transferred to the SYSUT3 data set. The transfer is handled by the PBLCKEND subroutine on call from the END or SEMCO routine.

A pointer named LPBP at all times addresses the heading entry of the current (embracing) block or procedure. LPBP is used

1. In copying the Program Block Number into the following identifier entries,
2. In transferring the Program Block Number of a reentered block to the Modification Level 1 text following the operator <u>Epsilon</u>, which closes a block



* If the source module, as specified by the PROCEDURE option, is a precompiled procedure, the heading entry for Program Block 0 is followed by an entry containing the precompiled procedure name, and at phase termination, Program Block 0, containing the procedure name, is transferred to SYSUT3. If, however, the source module is a program, the heading entry for Program Block 0 is immediately followed by a heading entry for Program Block 1, and Program Block 0 is <u>not</u> transferred to SYSUT3 at phase termination.

Figure 19. Diagram illustrating the processing of the Identifier Table in the Scan I/II Phas

or procedure (see "Scope Identification"), and

3. In specifying the start of a record when the block of identifiers is transferred to the SYSUT3 data set at the close of a block or procedure.

Pointer LIGP addresses the heading entry of the current block or procedure, or the heading entry of the current for statement. Pointer LIGP is used in transferring the Identifier Group Number of the reentered block, procedure, or for statement to the Modification Level 1 text following the operator Epsilon or Eta, which closes a block, a procedure, or a for statement (see "Scope Identification").

Pointer AITL addresses the next free entry in the Identifier Table.

When a program block heading entry is constructed at entry to a new block or procedure, the contents of LPBP and LIGP are saved in the newly constructed heading entry and both pointers are then reset to point to the new heading entry (the addresses saved in the heading entry both point to the preceding program block heading entry). If, subsequently, a for statement is encountered in the block or procedure, pointer LIGP is set to point to the corresponding for statement heading entry after its contents have been stored in the for statement heading entry. In the case of a series of nested for statements, this procedure is repeated for each for statement heading entry. At exit from each enclosing for statement, LIGP is reloaded with the address previously saved in the last constructed heading entry until, at reentry to the current (embracing) block or procedure, LIGP again points to the corresponding program block heading entry.

When the close of the current block or procedure is reached, and the set of identifiers declared or specified in the block or procedure have been transferred to the SYSUT3 data set, the current entry pointer AITL is reset to the beginning of the vacated area (by setting AITL=LPBP), in readiness for a further identifier entry or for a new block or procedure. Pointers LPBP and LIGP are then reloaded with the addresses previously saved in the heading entry of the closed block or procedure, so that they now address the heading entry of the reentered block or procedure.

SCOPE IDENTIFICATION

In the transformed source text (Modification Level 1) generated by the Scan I/II Phase, each scope is identified as to type, by distinguishing one-byte operators which replace the opening and closing delimiters in the source module. The opening and closing operators for each type of scope are as follows:

Scope	Opening Operator	Closing Operator
Block	Beta (X'0D')	Epsilon (X'2A')
Procedure	Pi (X'0E')	Epsilon (X'2A')
Type Procedure	Phi (X'0F')	Epsilon (X'2A')
For Statement	For (X'18')	Eta (X'2B')
Compound Statement	Begin (X'0C')	End (X'2C')

The first delimiter 'BEGIN' which may open the body of a procedure is eliminated in the Modification Level 1 text.

Every block and procedure is assigned a serial Program Block Number. The Program Block Number appears in all entries in the Identifier Table representing identifiers declared or specified in the block or procedure; it also appears in the Modification Level 1 text following the operator which opens a block or procedure.

Every block, procedure, and for statement is assigned a serial Identifier Group Number. The Identifier Group Number appears in the heading entries in the Identifier Table; it also appears in the Modification Level 1 text following the opening operator of a block or for statement. The identifier Group Number is used, in the Scan III Phase, in detecting illegal branches.

The Program Block Number and the Identifier Group Number of the reentered scope also appear in the Modification Level 1 text following the operator which closes blocks, for statements, and procedures. The following list indicates the Program Block Number and/or Identifier Group Number which follow the opening and closing operators.

Scope	Opening	Close
Block	Beta<PBN><IGN>	Epsilon <PBN>*<IGN>*
Procedure	Pi<IGN>	Epsilon <PBN>*<IGN>*
Type-procedure	Phi<IGN>	Epsilon <PBN>*<IGN>*
For Statement	For<IGN>	Eta<IGN>*
Compound Statement	Begin	End

* PBN or IGN of the reentered scope

The Program Block Number (PBN) occupies one byte, the Identifier Group Number (IGN) two bytes.

SCOPE HANDLING STACK

The action required at the close of every scope depends on whether the scope is a block, a procedure, a for statement, or a compound statement. Thus, at the close of a block or a procedure, the set of entries in the Identifier Table representing the identifiers declared or specified in the block or procedure, is transferred to the SYSUT3 data set, and the operator Epsilon is transferred to the Modification Level 1 text. At the close of a for statement, a closing entry may be made in the Identifier Table and the operator Eta transferred. At the close of a compound statement, the operator End is simply transferred.

Owing to the fact that the same delimiter ('BEGIN') opens both a block and compound statement, and owing also to the fact that procedures and for statements may be closed by the delimiters 'END' or a semicolon, depending on their structure, a method of classifying each scope is required, so as to specify both the delimiter to be identified as the closing delimiter and the

Stack Operator		Significance	Stacked by	Released by
Name	Hex.			
Begin	08	Designates a compound statement, closed by 'END'.	BEGIN	COMPEND
Beta	04	Designates a block, closed by 'END'	BEGIN	PBICKEND
Proc*	10	Designates a procedure, closed by 'END' The procedure body consists of an unlabelled block or compound statement.	BEGIN	PBLCKEND
Proc	0C	Designates a procedure, closed by a semicolon or by 'END'. 'END' unconditionally closes the embracing scope. The procedure body consists of a procedure statement, a dummy statement or a delimiter 'CODE'.	PROCEDURE	PBLCKEND
Proc**	14	Designates a procedure, closed by a semicolon or by 'END'. 'END' unconditionally closes the embracing scope. The procedure body consists of a labelled statement or block, or a single assignment, goto, conditional or for statement.	STATE FOR	PBICKEND PBLCKEND
For	18	Designates a for statement, closed by a semicolon or by 'END'. A semicolon may close an embracing procedure or for statement. 'END' unconditionally closes the embracing scope.	FOR	FOREND
Alpha	00	Marks the bottom of the stack. ALPHA is stacked only at phase initialization and released only at phase termination	Initialization	Termination

Figure 20. Scope Handling Stack operators

particular action to be taken at the close. The device used for the classification of scopes is the Scope Handling Stack.

The Scope Handling Stack employs a set of six stack operators, each of which identifies a characteristic scope structure. Whenever a delimiter is detected which marks the opening of a new scope, an appropriate operator is placed in the stack. If, subsequently, some feature is detected in the scope which indicates a change in structure, the operator originally placed in the stack is replaced by another operator which correctly reflects the structure of the scope. When the delimiter specified by the stack operator as the closing delimiter is encountered, the operator is released from the stack. In this way, all embracing scopes at every point in the source module are classified by the operators in the stack, the innermost scope being classified by the last stack entry.

The list in Figure 20 indicates the stack operators, their significance, and the routines which stack and release the operators.

Stack operators are tested in the SEMCO, STATE, BEGIN, CODE, FOR, END, PBLCKEND, ENDMISS, and COMPEND2 routines.

<u>Begin</u> is stacked when the delimiter 'BEGIN' is encountered, provided the stack operator is not <u>Proc</u>. (<u>Proc</u> indicates that the current scope is a procedure and hence that the delimiter 'BEGIN' marks the opening of the procedure body; in this eventuality, <u>Proc</u> is replaced by <u>Proc*</u>). Until released, <u>Begin</u> remains unchanged in the stack unless a following declaration is encountered (see <u>Beta</u>).

<u>Beta</u> replaces <u>Begin</u> if a declaration is encountered immediately after the delimiter 'BEGIN'.

<u>Proc</u> is stacked whenever the delimiter 'PROCEDURE' is encountered. <u>Proc</u> may be changed to <u>Proc*</u> or <u>Proc**</u>, depending on the structure of the procedure body (see <u>Proc*</u> and <u>Proc**</u>).

<u>Proc*</u> replaces <u>Proc</u> if the procedure body consists of an unlabeled block or compound statement (indicated by the delimiter 'BEGIN' following the procedure heading).

<u>Proc**</u> replaces <u>Proc</u> if the procedure body consists of a labeled statement or a single statement other than a block or compound statement (indicated by a label preceding a colon, an assignment operator, or by the delimiters 'GOTO', 'IF', or 'FOR').

The stack operator is tested in the SEMCO and END routines for every semicolon and 'END' encountered. In principle, a semicolon closes a scope if the scope is a for statement or a procedure, the body of which consists of a single statement (other than a compound statement or block) or, alternatively, a labeled statement. No other scope may be closed by a semicolon. This principle is reflected in the logic of the SEMCO routine: the occurrence of a semicolon constitutes the close of a scope only if the stack operator is <u>For</u>, <u>Proc</u>, or <u>Proc**</u> (i.e., only if the scope has the characteristics indicated by these operators). For these three operators, the PBLCKEND or FOREND subroutine is activated. For all other operators, the semicolon does not constitute the closing delimiter, and the <u>Semicolon</u> operator is simply transferred to the output string. A single semicolon may close a series of nested for statements or the procedure embracing a for statement. For this reason, a further test of the stack operator is made after the operator <u>For</u> has been released.

The delimiter 'END' in every case constitutes the close of the current scope. 'END' must close all blocks and compound statements as well as procedures whose procedure body consists of an unlabeled block or compound statement. 'END' also closes for statements and procedures, otherwise closed by a semicolon, where the semicolon is omitted or supplanted by the immediately following 'END' of an enclosing scope. This reasoning is reflected in the logic of the END routine. The occurrence of 'END' marks the close of the current scope for any stack operators except Alpha; if the stack operator is <u>Beta</u>, <u>Proc</u>, <u>Proc*</u>, or <u>Proc**</u>, the PBLCKEND subroutine is activated; if <u>For</u>, the FOREND subroutine; and if <u>Begin</u>, the COMPDEND routine is entered. A single 'END' may close a series of nested for statements, or the procedure embracing a for statement, or a procedure of type <u>Proc</u> or <u>Proc**</u>. For this reason, the stack operator is tested anew, after the operators <u>Proc</u>, <u>Proc**</u>, and <u>For</u> have been released.

MODIFICATION LEVEL 1 SOURCE TEXT

The Scan I/II Phase generates a transformed source text, called Modification Level 1, which is transferred to the SYSUT1 data set and forms the primary input to the Scan III Phase. The principal changes reflected in the Modification Level I source text are as follows:

1. Initially, all characters are translated from the external code (EBCDIC

or ISO) to an internal code (see Appendix I-a). Over and above this initial conversion, the following additional changes are made.

2. ALGOL delimiter words are replaced by one-byte symbols or eliminated, as follows:

   a. The delimiter words 'GOTO', 'IF', 'THEN', 'ELSE', 'FOR', 'DO', 'STEP', 'UNTIL', 'WHILE', 'EQUIV', 'IMPL', 'OR', 'AND', 'NOT', 'GREATER', 'NOTGREATER', 'LESS', 'NOTLESS', 'EQUAL', 'NOTEQUAL', and 'POWER' are replaced by unique one-byte symbols. See Appendix I-b.

   b. The delimiter 'BEGIN' is variously represented by two symbols Begin or Beta, or eliminated, depending on the scope opened by the delimiter. The delimiter 'END' is variously represented by the symbols End, Eta, or Epsilon, depending on the scope closed by the delimiter. See "Scope Identification" in this chapter.

   c. The declarators 'ARRAY' and 'SWITCH' are replaced by unique one-byte symbols, but the declarator 'PROCEDURE' is variously represented by the symbols Pi or Phi, according to whether the declarator is preceded by a type qualifier. The type declarators 'REAL', 'INTEGER', and 'BOOLEAN' are eliminated, as are the specificators 'STRING', 'LABEL', and 'VALUE', and the delimiters 'COMMENT' and 'CODE'. See item 4 below.

3. All other delimiters (including operators) are represented in the Modification Level 2 text by one-byte operators, or eliminated, as follows:

   a. Dual-character arithmetic, relational, and logical operators are replaced by one-byte symbols. No change, beyond the initial translation to internal code, is made in single-character operators.

   b. The separators comma, colon, decimal point, and := are uniquely represented by one-byte symbols. The semicolon is represented by the operator Semicolon, unless the semicolon follows a declaration (in which case it is replaced by the operator Delta) or a comment (in which case it is eliminated). Semicolon or Delta is followed by a two-byte Semicolon Count. If a semicolon closes a declared procedure, the operator Delta is preceded by the symbol Epsilon; if a semicolon closes a for statement, the operator Semicolon is preceded by the symbol Eta.

   A colon following a declared label is represented by the operator Label Colon. A lone period is eliminated, unless it is preceded or followed by a digit or +/- sign, in which case it is represented resented as a Decimal Point. Apostrophes are eliminated except when followed by a digit or +/- sign (in which case the apostrophe is replaced by the Scale Factor operator). The operator Apostrophe appears in the Modification Level 1 text solely in front of internal names representing character strings and logical values. The parentheses (and) are transferred, except when they occur in parameter delimiters (which are replaced by the Comma). The brackets (/ and /) are replaced by the symbols [and], respectively. The string quote signs '(' and ')' enclosing character strings, are eliminated.

4. Declarations are removed or transferred in modified form, as indicated below. All valid declared or specified identifiers are entered in the Identifier Table.

   a. Type declarations are eliminated in their entirety from the Modification Level 1 text.

   b. Array and switch declarations are transferred in modified form. The declarators are represented by one-byte operators, the declared identifiers by a maximum of six characters in internal code, together with switch components or array dimensions.

   c. Procedure declarations are transferred in modified form. The declarator is represented by the symbol Pi (if the procedure is not type-qualified) or Phi (if the procedure is type-qualified). Procedure identifiers and the formal parameters in parameter lists, each represented by a maximum of six characters, are transferred, but the value and specification parts of procedure headings are eliminated. The delimiter 'CODE', representing the body of a code procedure, is replaced by the symbol Gamma, followed by six charac-

ters of the code procedure identifier.

d. Semicolons following all declarations, whether the declaration is represented in the Modification Level 1 text or not, are represented by the operator Delta.

5. The principal remaining changes represented in the Modification Level 1 text are as follows:

   a. Character strings are replaced by five-byte internal names referencing the location where each string was stored in the Constant Pool. The internal name is preceded by the Apostrophe operator.

   b. The logical values 'TRUE' and 'FALSE' are replaced by internal names referencing the values one and zero, respectively, in the Constant Pool. The internal name is preceded by the Apostrophe.

   c. Number constants are transferred unchanged, except for the translation to internal code mentioned in item 1, and the replacement of the point and apostrophe (representing respectively, the decimal point and the scale factor) by the Decimal Point and Scale Factor operators.

   d. Valid labels are transferred, but the colon following a declared label is replaced by the Label Colon.

   e. Parameter delimiters of the form)LETTERS:(are replaced by the Comma. If a parameter delimiter extends across two output buffers, the symbol Rho is inserted at the beginning of the second buffer, indicating to the Scan III Phase that the letters at the end of the preceding record are to be replaced by the Comma.

   f. The record-end operator Zeta is inserted at the end of every output record, except the last, in which the character Omega marks the end of the Modification Level 1 text.

All operands (identifiers) contained in statements in the source module are transferred unchanged to the Modification Level 1 text, except for the initial translation to the internal code mentioned in item 1 above.

The table in Appendix I-b indicates the complete scope of coded characters appearing in the Modification Level 1 source text.

GROUP TABLE (GPTAB)

The Group Table is constructed in the Scan I/II Phase and transmitted to the Scan III Phase via the Common Work Area. A three-byte entry is constructed for every block, procedure and for statement, indicating the Identifier Group Number (I.G.No.) of the enclosing block, procedure, or for statement, and, in the event the scope for which the entry is constructed is a for statement, indicating its serial For Statement Number incremented by one (F.S.No.+1). The Group Table is used in the Scan III Phase, in finding the Identifier Group Number of the enclosing scope, and in classifying the optimizability of for statements containing goto statements involving a branch out of the for statement. Entries are referenced by Identifier Group Number.

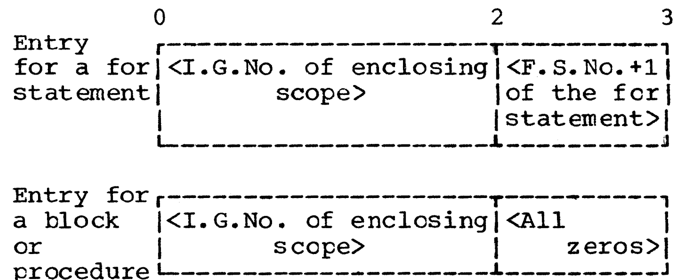


Figure 21. Group Table entries for a for statement and for a block or procedure

SCOPE TABLE (SPTAB)

The Scope Table is constructed in the Scan I/II Phase and transmitted to the Scan III Phase in main storage. A one-byte entry is constructed for every for statement, indicating the Program Block Number (P.B.No.) of the enclosing block or procedure. The Scope Table is used in the Scan III Phase in determining whether all terms of array subscript expressions occurring in for statements are declared outside the for statement (i.e. not in a block enclosed by the for statement).

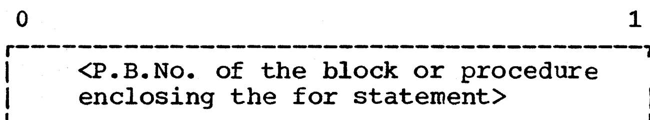


Figure 22. One-byte Scope Table entry

PROGRAM BLOCK NUMBER TABLE (PBTAB1)

The Program Block Number Table is constructed in the Scan I/II Phase and transmitted to the Identifier Table Manipulation Phase in main storage. A one-byte entry is constructed for every block and procedure, indicating the Program Block Number of the enclosing block or procedure. The Program Block Number Table is used in connection with the print-out of the Identifier Table listing in the next phase, in which the Program Block Number of the block or procedure embracing each block and procedure is shown.

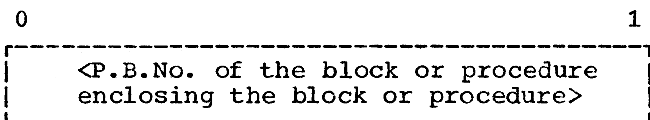


Figure 23. One-byte Program Block Number Table entry

PROCESSING OF OPENING SOURCE TEXT

The source module as specified in the EXEC statement may be a program or a precompiled procedure. If the source module is a program, the operative (programming) text in the source module must be opened by the delimiter 'BEGIN'. If the source module is a precompiled procedure, the operative text in the source module must be opened by the delimiter 'PROCEDURE' or by one of the delimiter sequences 'REAL' 'PROCEDURE', 'INTEGER' 'PROCEDURE', or 'BOOLEAN' 'PROCEDURE'.

Since the opening delimiter may be preceded by comment, provision is made in the Compiler to assure that, at the start, all text is disregarded until the correct delimiter or delimiter sequence is found. To facilitate the search for the correct opening delimiter, a number of special-purpose routines, as well as a switch named STARTBIT, are used.

STARTBIT = 0 (off) signifies that the opening delimiter has not been found and specifies, in general, that scanning for the appropriate character sequence is to continue. STARTBIT = 1 (on) signifies that the opening delimiter has been found.

The chart in Figure 24 shows the logical flow through the routines which process the opening delimiter, and the function of the STARTBIT. In the TESTLOOP routine the condition STARTBIT off has the effect of limiting the character search to an apostrophe (the first of two apostrophes enclosing a delimiter word). When an apostrophe is found, control is passed to APOSTROF, which searches for the second apostrophe and then branches to DELIMIT. In DELIMIT, the condition STARTBIT off causes a branch to a special-purpose routine, called STARTDEL, whose function is to activate FIRSTBEG, PROCEDUR, or TYPE, according to whether the delimiter is 'BEGIN', 'PROCEDURE' or 'REAL', 'INTEGER', or 'BOOLEAN', respectively, and to return control to TESTLOOP in all other cases. If the source module is a program and the delimiter is 'BEGIN', STARTBIT is turned on by FIRSTBEG, thus signifying that the correct opening delimiter has been found. If the source module is a precompiled procedure and the delimiter is 'PROCEDURE' or '<type>' 'PROCEDURE', STARTBIT is turned on by PROCEDUR. In all other cases, an error is stored in the Error Pool, and control returned to the TESTLOOP, which continues to scan for an apostrophe.

CLOSE OF SCAN I/II PHASE

The EODADIN routine, which closes the Scan I/II Phase and which transfers control to the succeeding phase, may be entered under four main conditions:

1. At the logical close of the source module, when the logical terminal delimiter ('END' in most cases) has closed the outermost scope of the source module
2. When an unexpected End of Data condition occurs
3. When a terminating syntactical error is detected in the source module
4. When a program interrupt or unrecoverable I/O error occurs.

Charts A, B, C and D in Figure 25 show the flow of control through the various routines before EODADIN is finally entered, under the four conditions mentioned.

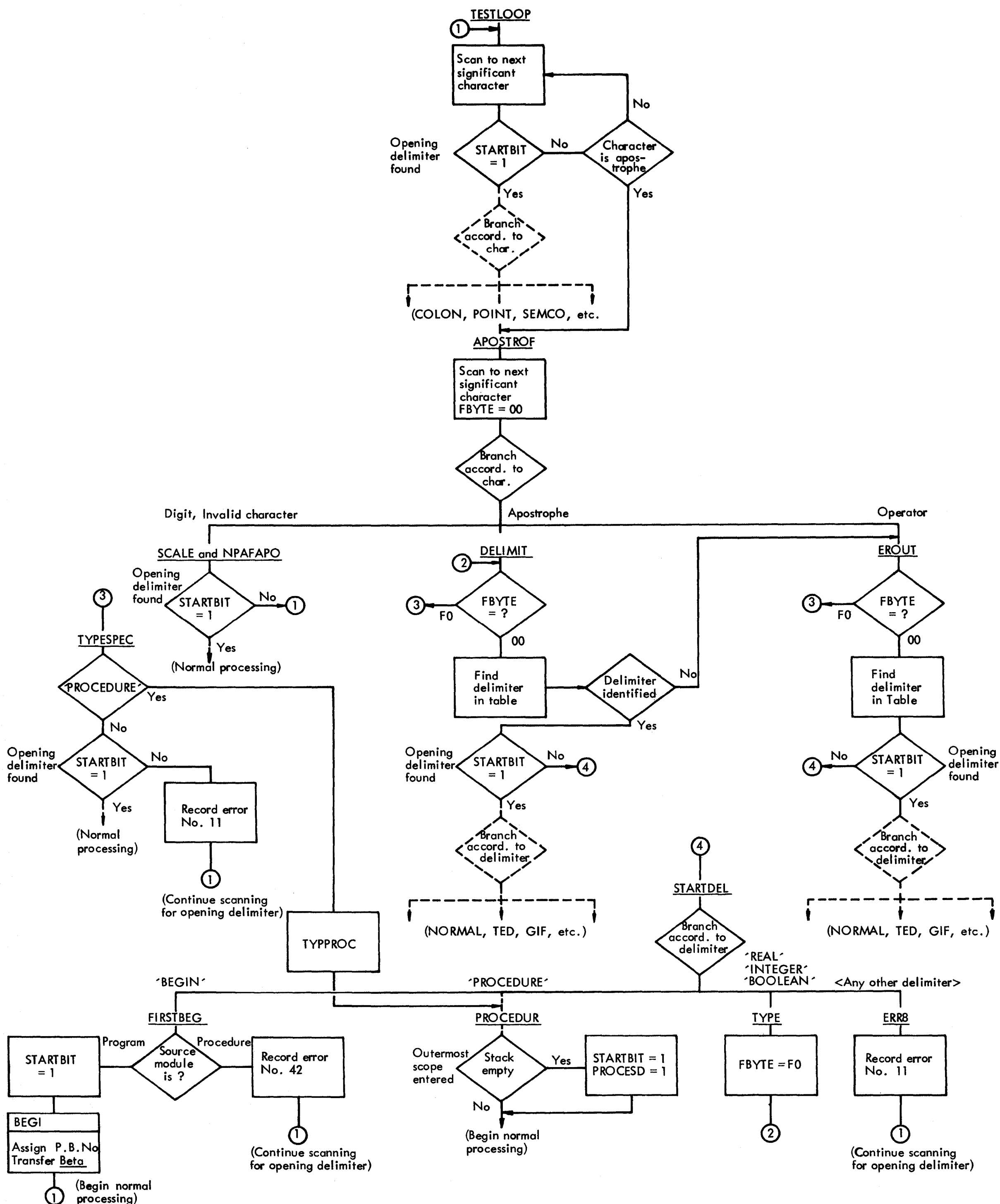


Figure 24. Chart showing the logical flow in the search for the opening delimiter and showing the function of the STARTBIT

In Chart A, the logical close of the source module is detected by the PBLCKEND subroutine (activated by the delimiter 'END' or a semicolon closing a block or procedure), when the stack operator Alpha shows that the bottom of the Scope Handling Stack has been reached, and that, accordingly, the outermost scope of the source module has been closed. Under these conditions, PBLCKEND specifies EODADIN as the EOD routine; turns a switch named ENDBIT on, and transfers control to COMMEND, whose function is to bypass any comment and to find the semicolon (or the first apostrophe

of 'END' or 'ELSE') which terminates such comment. If the terminating delimiter is a semicolon, the condition ENDBIT=1 causes control to be passed to READROUT, which scans the remaining source text for any characters other than a blank or the record-end operator <u>Zeta</u>. If any significant character is detected, an error is recorded in the Error Pool by ERR9 before control is passed to EODADIN; otherwise, READROUT continues reading the remaining source record (if any) until the Operating System transfers control to EODADIN at End of Data. The closing comment (if any) may be terminated (incorrectly) by 'END' or 'ELSE'. When either of these delimiters has been identified by the DELIMIT and COMSPEC routines, control is passed to END or TED, in which the condition ENDBIT=1 causes a branch to be taken to ERR9 before EODADIN is entered.

In Chart B, the ENDMISS routine (which is specified at phase initialization as the EOD routine) is entered when the Operating System has identified an unexpected End of Data condition. ENDMISS activates the PBLCKEND, FOREND, and/or TERMBGN routines until all remaining entries of the Identifier Table have been transferred to external storage and all stack operators have been released. Thereafter, an error message is stored in the Error Pool, and control is passed to EODADIN.

In Chart C, the ERR4 subroutine is called when a terminating error has been detected. ERR4 records the error in the Error Pool and then transfers control, through COMPFIN (which turns the TERR switch on -- see Appendix IV), to EODADIN. The condition TERR on causes EODADIN to transfer control to Diagnostic Output Module IEX 21, rather than to IEX20 (Identifier Table Manipulation Phase).

In Chart D, EODADIN is entered directly from the Directory routine PIROUT in the event of a program interrupt or an unrecoverable I/O error. PIROUT, whose address is specified in the SPIE macro instruction executed in the Initialization Phase, passes control to the routine whose address is stored at ERET. ERET is updated at initialization of the Scan I/II Phase, to specify the entry point of EODADIN.

SWITCHES

The Scan I/II Phase employs some 20 local switches, over and above the common switches in the HCOMPMOD Control Field (Appendix IV). The significance of the various switches, all of which are located in the Common Work Area, is indicated in the list which follows. The first switch in the list (FBYTE) comprises a full byte, which may have three hexadecimal values. The remaining 19 switches are represented by the binary positions of three contiguous bytes named BITS1, BITS2, and BITS3 (Figure 19), and may be either on (=1) or off (=0).

FBYTE = X'00' (set =X'00' by APOSTROF) signifies that no particular set of delimiters is being sought, and specifies to DELIMIT that a normal branch is to be taken, according to the delimiter identified.

= X'F0' (set =X'F0' by TYPE and IDCHECK) signifies that one of the delimiters 'REAL', 'INTEGER', or 'BOOLEAN' is followed by a second apostrophe, indicating another delimiter (which may logically only be 'PROCEDURE' or 'ARRAY') and specifies to DELIMIT that TYPESPEC is to be entered.

= X'FF' (set =X'FF' by COM) signifies that an end comment is followed by an apostrophe, indicating a delimiter word (which may logically only be 'END' or 'ELSE'), and specifies to DELIMIT that COMSPEC is to be entered.

BEGBIT:

<u>on</u> (turned on by the BEGIN routine) identifies the fact that the delimiter 'BEGIN' has been encountered and specifies to all routines which process declarations, namely TYPE, PROCEDUR, SWITCH, and ARRAY, that the BEG1 subroutine is to be called before any declarations are processed. Among other things, BEG1 assigns a new Program Block Number and transmits the one-byte operator <u>Beta</u> to the output buffer.

<u>off</u> (turned off by the BEG1 subroutine) specifies that the BEG1 has been called and that the subroutine is not to be reactived until the delimiter 'BEGIN' opening a new block has been identified.

Chart A: End of Scan I/II Phase - Normal Close of Source Module's Outermost Scope

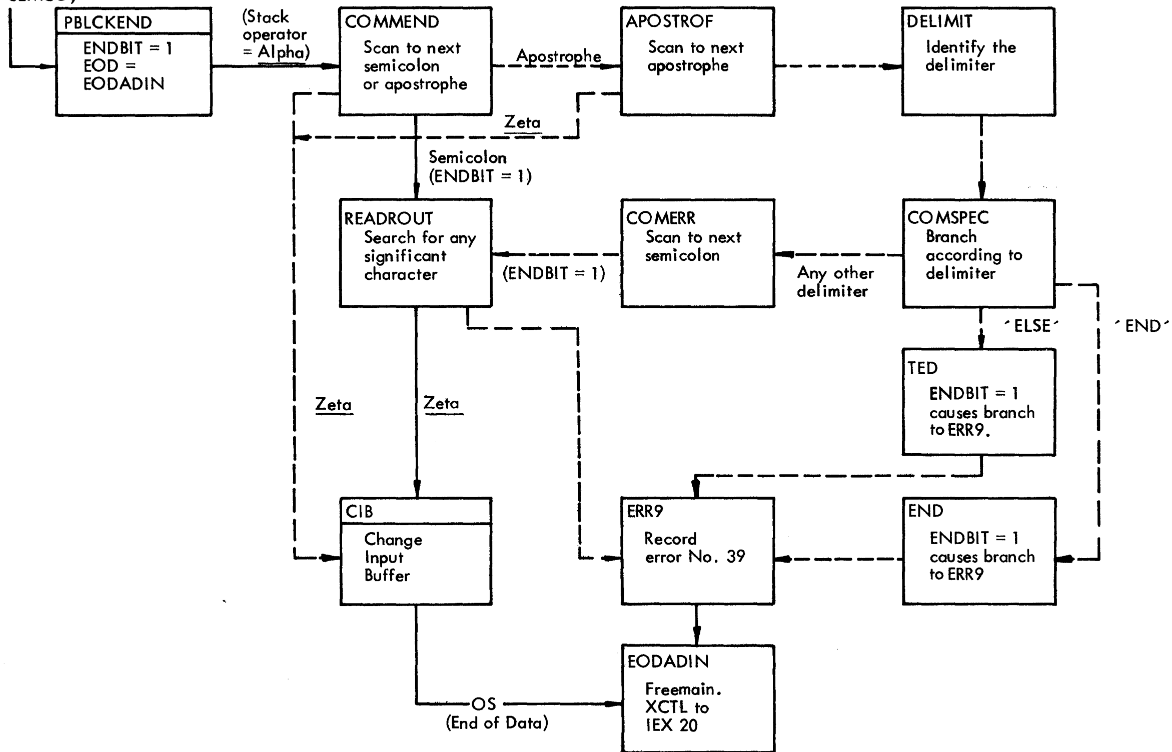


Chart B: End of Scan I/II Phase - Unexpected End of Data

(OS at End of Data)

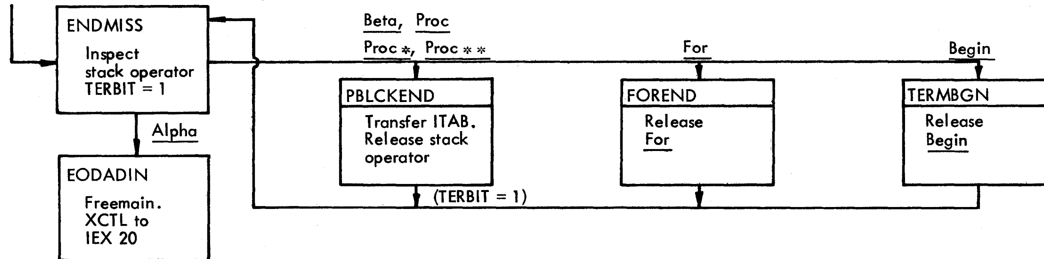


Chart C: End of Scan I/II Phase - Terminating Error

(Any routine on detection
of a terminating error)

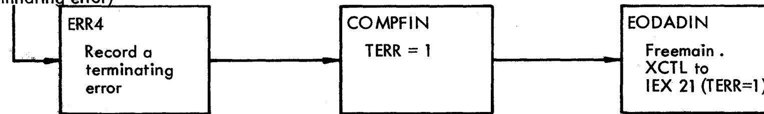


Chart D: End of Scan I/II Phase - Program Interrupt or Unrecoverable I/O Error

(from PIROUT in Directory)

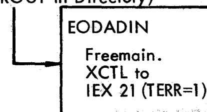


Figure 25. Exits from Scan I/II Phase

PROBIT:

on (turned on by the PROCEDUR routine) signifies that a procedure heading has been entered and specifies to declaration-processing routines that, until turned off, all declarative delimiters such as 'INTEGER', are to be processed as type specifiers of formal parameters, as opposed to type declarators.

off (turned off by the STATE, BEGIN, FOR, and CODE routines, when a statement, or the delimiter 'BEGIN', indicating the end of a procedure heading, is identified) specifies to declaration-processing routines that, until turned on, all declarative delimiters are to be processed as type declarators, not as specifiers.

DELTABIT:

on (turned on by declaration-processing routines) identifies the fact that a declaration has been detected and specifies to the SEMCO routine that the semicolon immediately following is to be replaced in the output string by the one-byte operator Delta.

off: (turned off by the SEMCO routine) signifies that, unless subsequently turned on, the next semicolon is to be represented by the Semicolon operator.

IDBIT:

on (turned on by the PROCID routine) signifies that the next identifier is the procedure identifier and specifies that a program block heading entry is to be made in the Identifier Table to mark the beginning of a new identifier group.

off (turned off by the PROCID routine) signifies that the procedure heading entry has been made in the Identifier Table and specifies that the formal parameter part of the procedure heading is being processed.

ARBIT:

on (turned on by the ARRAY routine) signifies that an array declaration has been identified and specifies to all routines that the character being processed forms part of an array list.

off (turned off upon identification of a semicolon terminating an array declaration and upon entry to the SWITCH routine) signifies to all routines that, unless turned on, the character being processed forms part of a switch list.

LISTBIT:

on (turned on by the ARRAY routine on recognition of a comma following an array identifier) signifies that the next identifier is a continuation of a list of declared array identifiers with the same dimension list, and specifies that a Comma is to be transferred to the output buffer to separate the last identifier from the next.

off (turned off in the ARRAY routine) has no significance.

TERBIT:

on (turned on by the ENDMISS routine after control has been passed to it by the Operating System at End of Data) specifies to the PBICKEND subroutine that control is to be returned to ENDMISS.

off (turned off at phase initialization) has no significance.

ENDBIT:

on (turned on by the PBICKEND subroutine when a test shows that the Scope Handling Stack is empty, indicated by stack operator Alpha) signals the final exit from the outermost scope of the source module, and specifies that any remaining text in the source module is to be disregarded, and that if any text, other than comment, terminated by a semicolon colon, is found, error No. 43 is to be recorded.

off (turned off at phase initialization) signifies that the delimiter which logically closes the source module has not yet been reached.

COBIT:

on (turned on in the COM routine) specifies to the COM routine that the source string up to the next semicolon is to be deleted. The characters deleted may be a segment of the form: 'COMMENT'<comment>;.

off (turned off by the COMMEND routine) specifies that the source text up to the next semicolon or the delimiter 'ELSE' or 'END', is to be deleted. This deletes:

1. Any comment enclosed as follows: 'END'<comment>'ELSE'/'END'/; or

2. An erroneous statement or declaration (or portion thereof).

STARTBIT:

on (turned on by the FIRSTBEG and PROCEDUR routines) signifies that the opening delimiter of the source module has been found.

off (turned off at phase termination) signifies that the opening delimiter of the source module has not been found. See "Processing of Opening Source Text".

VALBIT:

on (turned on by the VALUE routine) signifies to the SPEC routine that a value specification is being processed.

off (turned off in the IDCHECK routine) signifies that, unless turned on, a type specification (not a value specification) is being processed.

PROBIT:

on (turned on by EODADIN in the event the source module is a precompiled procedure) specifies to PBLCKEND that control is to be returned to EODADIN, after Program Block 0 in the Identifier Table has been transferred to the SYSUT3 data set.

off (turned off at phase termination) has no significance.

FRSTPUT:

on (turned on by the GENERATE routine) signifies that the first PUT instruction has been issued, and that the address of an output buffer is available.

off (turned off by the GENERATE routine) signifies that the first PUT instruction has not been issued.

ENDELSEBIT:

on (turned on by the END routine when, after the delimiter 'END' has closed a block or compound statement, a test shows that the stack operator is For or Proc**) signifies that the embracing scope is a for statement or a procedure which may be closed by a semicolon, and specifies to the COM routine that, if a semicolon is found to terminate the comment following 'END', a branch is to be taken to the COMPEND2 routine. The latter activates the FOREND or PBLCKEND subroutine, depending on whether the stack operator is For or Proc**.

off (turned off by the END, COMPEND2, and TED routines) has no significance.

E11BIT:

on (turned on by ERR8) signifies that error No. 11 has been recorded, and that the error should not be recorded again.

off (turned off at phase initialization) signifies that error No. 11 has not previously been recorded.

FMBIT:

on (turned on by PROCID) signifies that the formal parameter list of a declared procedure is being processed and that the end of the list has not been reached; and specifies to IER that control is to be returned to PROCID after a defective parameter has been processed.

off (turned off by PROCID when the semicolon following a formal parameter list is found) has no particular significance.

NOFREE:

on (turned on in CLOSE2) signifies that main storage for the private area has not been acquired, and specifies to EODADIN that a FREEMAIN macro instruction is not required.

off (turned off at initialization) has no particular significance.

FRSITB:

on (turned on in PBLCKEND after the first Identifier Table record has been output on SYSUT3) signifies that a CHECK macro instruction should be issued before each subsequent output operation.

off (turned off at initialization) signifies that no previous output has taken place on SYSUT3 and that, accordingly, a CHECK macro instruction is not required before the next output operation.

PROCESD:

on (turned on in the PROCEDUR routine)

signifies that the source module is a precompiled procedure and specifies to the PROCID routine that an ESD record is to be made for the procedure name.

off (turned off by the PROCID routine) signifies that the source module is a program, or that the ESD record for a precompiled procedure has been generated.

CONSTITUENT ROUTINES OF SCAN I/II PHASE

The principal constituent routines of the Scan I/II Phase are described below. The page on which each routine is described and the flowchart in the Flowchart Section in which the general logic of the routine is set forth may be found with the aid of the Index in Appendix XI.

The position of the major routines in the overall logical organization of the phase may be determined by reference to Flowcharts 011 and 012 in the Flowchart Section.

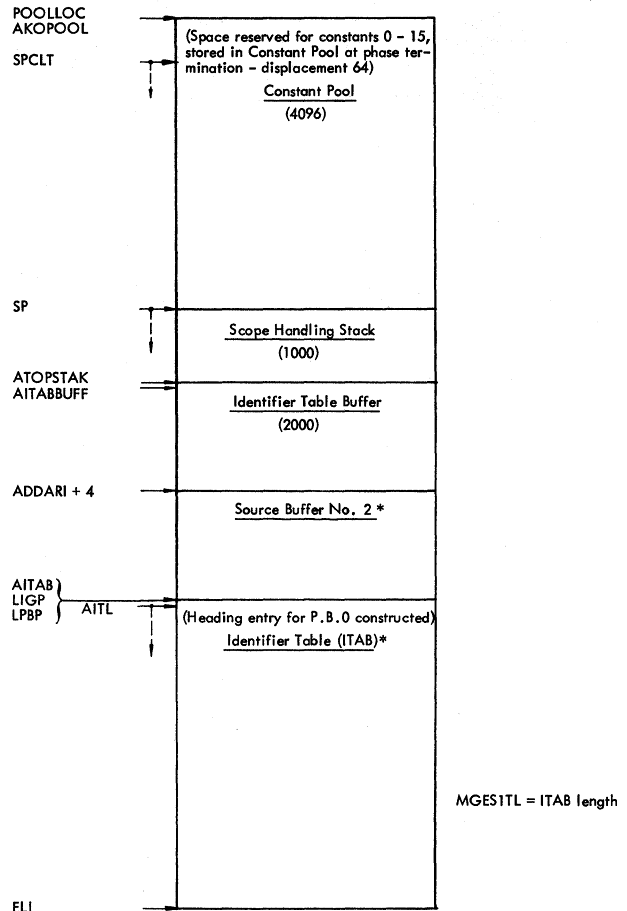


Figure 26. Private Area acquired by the Scan I/II Phase, showing pointers initialized

PHASE INITIALIZATION

The Initialization routine gets main storage for the private work area shown in Figure 26; initializes pointers; specifies EOD and program interrupt-I/O error routines; assembles headlines for the source module listing; and activates the Change Input Buffer subroutine (CIB). The routine exits to TESTLOOP.

The entry point of the routine activated in the event of a program interrupt or an I/O error (both of which terminate compilation) is stored in ERET, the location referenced by the Program Interrupt routine (PIROUT) and the I/O Error routines (SYNAD and SYNPR) in the Directory. The entry point CLOSE2, specified at entry is changed, after the GETMAIN instruction has been issued, to EODADIN. Both CLOSE2 and EODADIN close data sets and transfer control to Diagnostic Output Module IEX21. EODADIN in addition releases main storage.

The entry, ENDMISS, in the event of an End of Data (EOD) condition on the SYSIN data set is stored at EODIN, the location referenced by the End of Data Exit routine in the Directory.

The GETMAIN instruction for the private work area is executed after the total area required has been computed. The area sizes needed for the Identifier Table and Source Buffer No. 2, which depend on the capacity of the system used, are obtained from the Area Size Table entries named ITAB1OS and SRCE1S, respectively. The areas allocated to the Constant Pool, Stack, and ITAB buffer are fixed at 4096, 1000 and 2000 bytes, respectively, for all systems. The various pointers initialized are shown in Figure 26. A fuller explanation of the pointers LPBP and LIGP is given under the heading "Processing of the Identifier Table".

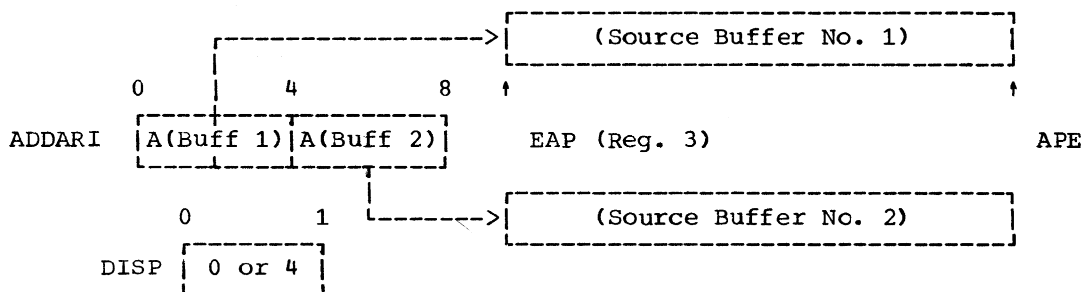


Figure 27. Source text buffers and pointers

Source Buffer No. 2 is the second of two buffers used for output of the modified source text generated by the phase. Buffer No. 1 is set up in the Common Area by the Initialization Phase, its beginning address being stored at SRCE1ADD and its end address at SRCE1END. The present initialization routine stores the addresses of both buffers in an eight-byte field named ADDARI, and then initializes the pointers EAP (Register 3) and APE for Source Buffer No. 1 (see Figure 27). EAP and APE are updated whenever buffers are exchanged by the Change Output Buffer subroutine (COB). The particular address loaded in EAP from ADDARI is determined with the aid of a control byte named DISP (reset from 0 to 4 and vice versa just before EAP is updated), which specifies the displacement (0 or 4) from ADDARI.

A heading entry (Figure 28) for Program Block 0 (an arbitrarily defined block enclosing the source module) is constructed in the Identifier Table. The current entry position AITL is set to point to the next free entry in the Identifier Table.

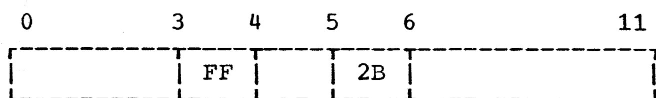


Figure 28. Heading Entry constructed at initialization in Identifier Table for Program Block 0

The following dispositions are made in the Common Work Area, in which the addresses of the various tables and other fields are defined by a dummy control section in IEX11. The address of an 88-byte dummy print area named SAVEPRNT is stored at APRINTAR. If the SOURCE option is not specified, the Change Input Buffer subroutine (CIB) moves each source record to SAVEPRNT, in order that strings may be stored in the Constant Pool in external code. If, however, SOURCE is specified, indicating that a listing of the source module is to be printed, source records are moved instead to a print buffer specified by the PRINT routine in the Directory. In this case, the address in APRINTAR will be replaced by the address of the print buffer.

In preparation for the print-out of a source module listing, the headlines ("SOURCE PROGRAM" for the first line and "SC SOURCE STATEMENT" for the second line) are moved to a field named PAGEHEAD in the Common Work Area from the locations HDING1 and HDING2. The headlines are printed out by the Directory PRINT subroutine, on call from CIB, if the SOURCE option is specified.

The

Bit	BITS1	BITS2	BITS3
0	BEGBIT	ENBIT	E11BIT
1	PROBIT	COBIT	FMBIT
2	DELTABIT	STARTBIT	NOFREE
3	IDBIT	VALBIT	FRSITB
4	ARBIT	PBOBIT	PROCESD
5	LISTBIT	Not used	Not used
6	Not used	FRSTPUT	Not used
7	TERBIT	ENDELSEBIT	Not used

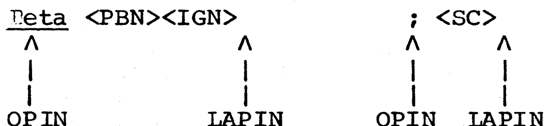
Figure 29. Switches used in Scan I/II Phase

See "Switches" in this chapter.

The Program Block Counter (PBC), Identifier Group Counter (IGC), Semicolon Counter (SC), For Statement Counter (FSN), and Output Record Counter (ONC) are initialized at 0, and the first entries (0) for Program Block 0 are made in the Program Block Number Table (PBTAB1), Group Table (GPTAB), and Scope Table (SPTAB). The control

switches used in the phase, which are contained in three bytes named BITS1, BITS2, and BITS3, are zero-set. The switches in each byte are shown in Figure 29. Their function and significance is explained elsewhere in this chapter under "Switches".

OPIN and LAPIN are the names of two special-purpose output buffer pointers. OPIN is always adjusted to point to the character that may precede a label or begin a parameter delimiter. These include Begin, Beta, Do, Else, Delta, Semicolon, and). At OPIN + 4 is noted the number of the output record (ONC), in which the character pointed at by OPIN is to be found. LAPIN points to the first byte following that pointed at by OPIN, where the letter string or label may begin. OPIN and LAPIN may be separated by two or more characters. OPIN and LAPIN are used when declared labels are entered in the Identifier Table or when a letter string is replaced by a Comma.



Before exit to the TESTLOOP routine, the Change Input Buffer subroutine (CIB) is called. CIB activates the PRINT subroutine in the Directory (if SOURCE is specified), which prints out the headlines assembled at PAGEHEAD and returns with the address of the print buffer. CIB then gets the first record in the Work Area (WA), moves it to the print buffer (or a dummy print area), translates the record to internal code, and returns control to Initialization, after having loaded the address of WA in REGI (Register 1). In the TESTLOOP routine, which is now entered, as well as in all routines which scan or inspect characters in the translated source text, REGI functions as the Work Area pointer.

MAIN LOOP (TESTLOOP)

TESTLOOP scans the translated source text in the Work Area, by means of a Translate and Test instruction, for any one of 14 characters assigned a nonzero function byte in Translation Table TESTTABL; moves the scanned text to the output buffer; and branches to the routine whose address is specified in an entry of Branch Address Table BPRTAB. The displacement of the entry in BPRTAB is given by the value of the character's function byte.

Hexadecimal Displacement	Content of Entry
00-03	All zeros
04	Address of TRANSOP
08	Address of TRANSOP
0C	Address of TRANSOP
10	Address of TRANSOP
14	Address of TRANSOP
18	Address of TRANSOP
1C	Address of CCLCN
20	Address of SEMCO
24	Address of RIGHTPAR
28	Address of BLANK
2C	Address of ERR1
30	Address of POINT
34	Address of APOSTROF
38	Address of CIB
3C	Address of ASSIGN
40	Address of DECPOINT
44	Address of ERR5
48	Address of BLKAPOS
4C	Address of NPAFTAPO
50	Address of SCALE
54	Address of COLONLIST
58	Address of SEMCLST
5C	Address of DELIMIT
60	Address of ZETAAPO
64	Address of EROUT
68	Address of LEFTPARL
6C	Address of RIGHTPARL
70	Address of PZETA
74	Address of ASSIGN
78	Address of DECPOINT
7C	Address of ERR5A
80	Address of COMMALST
84	Address of POINLST
88	Address of SLASHLST
8C	Address of QUOTE
90	Address of SEMC60

Figure 30. Branch Address Table BPRTAB

The function byte assigned by TESTTABL to each character, and the routines entered from TESTLOOP, are as follows:

Character	Function Byte	Routine Entered
Blank	28	BLANK
*	04	TRANSOP
/	08	"
(	0C	"
>	10	"
<	14	"
Not	18	"
)	24	RIGHTPAR
Point	30	POINT
Apostrophe	34	APOSTROF
Colon	1C	COLON
Semicolon	90	SEMC60
Invalid Character	2C	ERR1
Zeta	38	CIB
<All other characters>	00	(No branch, scanning continues)

The branching action just described is dependent on the condition that the correct delimiter word opening the source module has been found (STARBIT=1). See "Processing of Opening Source Text".

The transfer of scanned source text from the Work Area to the output buffer is handled by MSBLOOP (Move Scanned Bytes Loop).

Branch Address Table BPRTAB is referenced by most routines which determine a branch on the basis of a Translate and Test instruction.

BLANK (BLANK)

BLANK steps the Work Area pointer REGI to the next nonblank character and returns control to the calling routine (TESTLOOP or LIST). A scan, using Translation Table BTABLE, is initiated if a string of blanks is indicated by a second blank following the first. BTABLE assigns function byte FF to all characters except a blank, which is assigned a zero function byte.

TEST AND TRANSFER OPERATOR (TRANSOP)

TRANSOP determines if any of the characters *, /, (, <, >, or Not is associated with an immediately following character, and if so, transfers a one-byte operator representing the two characters in combination. Otherwise, the character is transferred unchanged.

The determination is made by comparing the succeeding character with a key, contained in a table named KEYTAB (Figure 31). The key used is specified by the function byte assigned the particular character in the TESTLOOP or LIST routines, from which TRANSOP is entered.

Current Operator (CO) represents the character in the source text which activates TRANSOP. The Expected Operator (EO) is the character with which the succeeding character is compared. The Resultant Operator (RO) represents the logical result of CO in combination with EO. RO is transferred to the output buffer, if the succeeding character agrees with EO. CO is transferred if the succeeding character is any character other than EO (excepting blank, which is disregarded, and the record-end operator <u>Zeta</u>, which causes the CIB subroutine to be called).

	Resultant Operator (RO)	Current Operator (CO)	Expected Operator (EO)
Asterisk	<u>Power</u>	*	*
Slash	]	/	)
Left Parenthesis	[	(	/
Less than	≤	<	=
Greater than	≥	>	=
Not	≠	¬	=

Figure 31. KEYTAB keys used in TRANSOP routine

RIGHTPAR

RIGHTPAR transers the) operator to the output buffer, and sets the pointers CPIN and LAPIN (see "Phase Initialization").

POINT

POINT inspects the character which follows a point, using a Translate and Test instruction, and branches to one of six routines according to the value of the function byte assigned the character in Translation Table PTTABLE. The address of the routine entered is obtained from an entry in Branch Address Table (BPRTAB), whose displacement equals the value of the assigned function byte.

The function byte assigned by PTTABLE to each character, and the routines entered from POINT, are as follows:

<u>Character</u>	Function <u>Byte</u>	Routine <u>Entered</u>
<Digit 0-9>	40	DECPOINT
Equal Sign	3C	ASSIGN
Point	1C	COLON
Comma	20	SEMCO
<Letter or Delimiter>	44	ERR5
Zeta	38	CIB
<All other characters>	00	(No branch, scanning continues)

DECIMAL POINT (DECPOINT)

DECPOINT transfers the Decimal Point operator.

ASSIGNMENT (ASSIGN)

ASSIGN transfers the Assign operator and passes control to STATE.

STATEMENT (STATE)

STATE is entered when a statement, identified by an assignment operator or a label, or by the delimiters 'GOTO', 'FOR', or 'IF', has been recognized. It serves to determine if the statement constitutes the body of a procedure, and if so, to stack the operator Proc in place of Proc** (see "Scope Handling Stack").

APOSTROPHE (APOSTROF)

APOSTROF has the main function of determining if an apostrophe opens a delimiter or if it represents a scale factor. APOSTROF inspects the characters following the apostrophe by means of a Translate and Test instruction, and branches to one of six routines, determined by the function byte assigned the particular character in Translation Table ATABLE. The branch is made by reference to Branch Address Table (BPRTAB).

The function byte assigned to each character in ATABLE, and the routines entered from APOSTROF, are as follows:

Character	Function Byte	Routine Entered
<Digit 0-9 or +/- sign>	50	SCALE
Blank	48	BLKAPOS
Zeta	60	ZETAAPO
Invalid Character	4C	NPAFTAPO
Apostrophe	5C	DELIMIT
Not,Or,And,Comma,) or Point	64	EROUT
<Any letter>, / or (	00	(No branch, scanning continues)

As indicated above, letters are assigned zero function bytes and are accordingly by-passed in the scan. Thus, for example, the letters separating the apostrophes in the delimiter word 'BEGIN' would be bypassed. The closing apostrophe, however, would terminate the scanning operation, and a branch would be taken to the Delimiter routine (DELIMIT).

The particular action taken by the SCALE, NDAFAPO, DELIMIT, and EROUT routines is governed by a control byte called FBYTE, which may have one of three hexadecimal values: 00, F0, or FF. FBYTE is set to X'00' in APOSTROF; to X'F0' in TYPE and SPEC; and to X'FF' in COM. The function of FBYTE is to specify whether or not a specific choice of delimiter words is being sought in the source text. Thus, for example, when the TYPE routine determines that an apostrophe immediately follows one of the delimiters 'REAL', 'INTEGER', or 'BOOLEAN', indicating a second delimiter in a sequence '<type>' 'PROCEDURE' or '<type>' 'ARRAY', it sets FBYTE to X'F0' in order to specify that the following delimiter must be 'PROCEDURE' or 'ARRAY', and that an error is to be recorded if any other delimiter is found. TYPE then passes control to the APOSTROF routine (by way of the ENTRAPR entry point), which then enters DELIMIT or EROUT. FBYTE='F0' causes DELIMIT or EROUT to pass control to TYPESPEC, which then determines if either of the acceptable delimiters follows. When the COM routine encounters an apostrophe at the end of a sequence of comment following 'END', indicating that a delimiter follows, it sets FBYTE to X'FF' to specify that any delimiter except "END" or "ELSE" is to be disregarded, and passes control to the APOSTROF routine by way of the ENTRAPR entry point. In the DELIMIT and EROUT routines which are subsequently entered, FBYTE=X'FF' causes a branch to COMSPEC, which then determines if the following delimiter is 'END' or 'ELSE'. FBYTE=X'00' signifies generally that no specific choice of delimiter words is being sought. FBYTE is set to X'00' by APOSTROF on entry from TESTLOOP. When a specific choice of delimiters is sought, as in the conditions described above, the APOSTROF routine is entered (from TYPE,SPEC, and COM) by way of ENTRAPR, in which case FBYTE will have been previously set to X'F0' or X'FF'.

SCALE FACTOR (SCALE)

SCALE transfers the Scale Factor operator.

BLANK AFTER APOSTROPHE (BLKAPOS)

The source text is scanned, with the aid of Translation Table BTABLE, to the next

non-blank character, after which the first apostrophe and any following characters are right-shifted by the number of blanks scanned.

ZETA AFTER APOSTROPHE (ZETAAPO)

The first apostrophe and any following characters are moved to a field named WABEFOR in front of the Work Area, after which a new source record is read in by the CIB subroutine.

INVALID CHARACTER AFTER APOSTROPHE (NPAFTAPO)

An error (No.1) is recorded, after which BLKAPOS is activated, which moves the apostrophe and following characters right, deleting the invalid character.

COLON (COLON)

The COLON routine leads into ASSIGN,LETDEL, or LABEL according to whether the colon in the source text precedes an equal sign, a left parenthesis, or any other character, identifying respectively, the assignment operator :=, a parameter delimiter of the form)LETTERS:(, or a label, e.g., LABEL: xyz.

The character string representing a label or a letter string is processed in the output buffer (where it will have been transferred by the TESTLOOP routine before control was passed to COLON on detection of a colon), rather than in the Work Area. The start of the character string is found with the aid of the pointers OPIN and LAPIN (see "Phase Initialization"). OPIN always points to the last transferred operator which may precede a label or letter string. LAPIN points to the first byte where a label or letter string may begin (usually the next byte to the right of that pointed at by OPIN). Individual characters of a label or parameter delimiter in the output buffer are addressed with the aid of pointer PIN.

LABEL (LABEL)

Provided the label does not begin in the output buffer preceding the last one (in which case the label exceeds the output buffer length and compilation is terminated), the label is checked for validity. If all characters of the label are valid, up to six characters of the label are copied into an entry for the label in the Identifier Table. If any one character is invalid, Error No.8 or 7 is recorded in the Error Pool, and the Identifier Table entry is erased, by a branch to ITABCLEAR.

LETTER DELIMITER (LETDEL)

Beginning with the character pointed at by pointer LAPIN, the character string in the output buffer is checked. If all of the characters are letters, in which case the character string qualifies as a letter delimiter, and if the letter delimiter begins in the current output buffer, a <u>Comma</u> is transferred to the output buffer, replacing the right parenthesis, and the letter delimiter is deleted. If the letter delimiter began in the preceding output buffer, the operator <u>Rho</u> is transferred to the first byte of the current output buffer, and that part of the letter delimiter in the current buffer is deleted. <u>Rho</u> serves in the Scan III Phase to signal that the immediately preceding characters up to, and including, the last right parenthesis are to be deleted and replaced by a <u>Comma</u>.

SEMICOLON (SEMCO AND SEMC60)

SEMCO: Depending on whether the semicolon in the source text terminates a statement or a declaration (indicated by DELTABIT=1), a <u>Semicolon</u> or <u>Delta</u> symbol is transferred, followed by the current Semicolon Count. If the semicolon terminates a statement (DELTABIT=0), the operator in the Scope Handling Stack is tested to determine if the semicolon closes a single-statement procedure of a for statement. If so, the PBLCKEND or FOREND subroutine is called.

SEMC60, entered on detection of the single-character semicolon, turns on the SET60 switch in the HCOMPMOD Control Field (Appendix IV), before entering SEMCO.

ERROR RECORDING ROUTINES

The error recording routines, named below, administer the storage of error patterns in the Error Pool. They are called by the various routines of the Scan I/II Phase on detection of syntactical errors in the source text. A majority are

subroutines which return control to the calling routine after the error pattern has been stored. Certain of the routines exit to the TESTLOOP routine, while those routines which administer the storage of terminating error patterns pass control (via COMPFIN) to EODADIN, which transfers control to diagnostic output module IEX21. The content of the error pattern is described in Chapter 9.

The typical call to an error recording subroutine has the form:

```
BAL REGB, <Error Routine Name>
DC X'041C'
```

On entry to the called subroutine, REGB contains the address of the immediately following parameter list, the first byte of which specifies the error pattern length while the second byte specifies the error number.

The typical return to the calling routine from the called subroutine is of the form:

```
BC 15, 2(0, REGB)
```

This specifies a return to the instruction following the parameter list in the calling routine.

The error recording routines may be divided into service routines and call routines. Service routines are those which actually store message patterns in the Error Pool, or which handle the necessary processing preliminary to the storage of error patterns. Call routines are those which receive calls for the recording of an error and which, in turn, issue calls to the appropriate service routines. Call routines may also move source text into an error pattern at an address specified by a service routine.

The service routines are the following:

ERROR1: Stores the first four bytes of every Error Pool entry, containing the entry length, error number, and semicolon count. The entry length and error number are fetched from the parameter list specified by the calling routine. ERROR1 also updates the Error Pool pointer (NEXTERR) in readiness for the next entry, making allowance for any source text to be subsequently inserted. The address of the current Error Pool entry is transmitted in REGY. In the event of an Error Pool overflow, a branch is made to ERRO.

ERROR1 is activated every time an error pattern is stored in the Error Pool, except in the case of Error No. 0.

ERROR2: Calculates the length of an identifier or delimiter addressed by a pointer named IN and stores the length in the parameter list of the calling routine.

ERR2D: Activates ERROR2 (which computes an identifier's or delimiter's length and stores the length in a parameter list), and ERROR1 (which stores the length, error number and semicolon count in an error pattern); moves the identifier or delimiter addressed by IN to the Error Pool entry addressed by REGY; and returns control to the calling routine.

ERR2: Sets the pointer IN to the last entry for an identifier in the Identifier Table, then branches to ERR2D (which stores the error pattern with the aid of ERROR2 and ERROR1).

ERR2B: Sets the pointer IN to an entry in the Identifier Table for a procedure identifier, then branches to ERR2D (which stores the error pattern with the aid of ERROR2 and ERROR1).

ERR2E: Sets the pointer IN to a location called IDBUCKET (see Type Specification routine IDCHECK) containing a procedure parameter, then branches to ERR2D (which stores the error pattern with the aid of ERROR2 and ERROR1).

ERR2C: Moves six characters of an erroneous delimiter to a location named BUCKET, sets pointer IN to that location, and branches to ERR2D (which stores the error pattern with the aid of ERROR2 and ERROR1).

ERR7: Activates ERROR1 (which stores the length error number and semicolon count in an error pattern). ERR7 is called where the error pattern contains no source text.

ERRO: Records a terminating error indicating an overflow of the Error Pool, and transfers control to the terminating routine EODADIN (via COMPFIN, which turns the TERR switch on).

The call routines are described below. Certain of the routines handle specific errors and, in calling the service routine, specify a parameter list for the particular error. Other routines handle more than one error, the parameter list being specified by the calling routine in which the error is detected.

ERR1: Calls ERR7, specifying a parameter list for Error No.1.

ERR3: Calls ERROR1, specifying a parameter list for Error No.3, then moves the characters previously stored at BUCKET by the Colon routine into the error pattern set up by ERROR1.

ERR4: Calls ERROR1 and passes control (via COMPFIN) to the EODADIN routine. ERR4 is called by numerous routines on detection of any terminating error. (See also "Close of Scan I/II Phase").

ERR5A: Calls ERROR1, specifying a parameter list for Error No. 35.

ERR5: Calls ERROR1, specifying a parameter list for Error No.2. Exits to TESTLOOP or LIST.

ERR6: Depending on two switches (which may cause a branch to other routines), calls ERROR1 and moves six characters of a delimiter from the Work Area into the error pattern set up by ERROR1.

ERR8: Depending on a switch, calls ERROR1, specifying a parameter list for Error No.11. Exits to TESTLOOP or LIST.

ERR18: Calls ERROR1, specifying a parameter list for Error No. 18.

ERR9: Calls ERR7, then transfers control to EODADIN (see also "Close of Scan I/II Phase").

ERROR10: Calls ERR2B, specifying a parameter list for Error No. 10 (which indicates that certain parameters of a procedure have not been specified). On return, ERROR10 inserts an all-purpose internal name in the Identifier Table entries representing the unspecified parameters.

ERR21: Stores the length of a declarative delimiter in a parameter list for Error No.21; calls ERROR1, specifying the parameter list; and then moves the delimiter from the Delimiter Table into the error pattern set up by ERROR1.

CHANGE INPUT BUFFER (CIB)

CIB gets an 80-character record of the source text from the SYSIN data set into the Work Area (WA); copies the record into a print buffer or a dummy print area; and translates the record in the Work Area to the internal code (Appendix I-a). If the SOURCE option is specified, a branch is made to the PRINT subroutine in the Directory, which prints out the record previously moved to a print buffer, and transmits the address of a new print buffer, to which the newly obtained source record will be moved. CIB is called by all routines which scan the source text, on recognition of the recorded operator Zeta. The latter is inserted by CIB at the end of each translated record in the Work Area.

In the event the source module is in ISO code, each record is first translated to EBCDIC code by searching for the characters (,), =, + and the apostrophe (the only characters whose representation differs between the EBCDIC and ISO codes) and replacing these characters by their EBCDIC combinations. This conversion simplifies the subsequent translation to internal code and facilitates printing the source text on the printer, in which the code implemented is EBCDIC.

The translation to internal code is made with the aid of translation table TRLTABLE, and produces the character set shown in Appendix I-a.

IDENTIFIER TEST (IDCHECK1)

IDCHECK1 is entered from the PROCID, ARRYID, and SWITCH routines, after a test has determined that the first character of a procedure, array, or switch identifier is a letter. IDCHECK1 transfers the letter to an entry in the Identifier Table and to the output buffer; inspects the following characters of the identifier, similarly transferring the next five characters, provided they are letters or digits; and returns control on detection of any character other than a letter or digit.

CHANGE OUTPUT BUFFER (COB AND COBSPEC)

See also "Phase Initialization".

COB determines if the last byte but one in the current output buffer has been filled (by comparing pointer EAP (register 3) with buffer-end pointer APE), and if so, transfers the buffer-end indicator Zeta to the last byte pointed to by EAP; writes out the current buffer (whose address is stored at WADDARI); and resets pointers EAP and APE to an alternate buffer, addressed by ADDARI + DISP storing the address of the new buffer at WADDARI. If the current buffer has not been filled, COB returns control to the calling routine. COB is

called in advance of every transfer of one or more characters to the output buffer.

COBSPEC, a special entry point of COB, includes a test as to whether a variable number of unfilled bytes (two or more) remain in the current buffer. The test consists in comparing REGO (instead of EAP) with APE, where REGO, preset by the calling routine, indicates the current address value of EAP, incremented by the required number of bytes. COBSPEC is called when a unit of data may not be split between records (e.g. the three-byte unit transferred by SEMCO, containing the Semicolon (or Delta) and the semicolon count).

DELIMITER (DELIMIT)

DELIMIT is entered from APOSTROF when the second of two apostrophes enclosing a delimiter word has been identified. DELIMIT compares the characters enclosed by apostrophes with a set of delimiter words in the Delimiter Table (W1TAB -- Figure 32), and when the corresponding word has been located, branches to the routine whose address is specified in an entry of the Branch Address Table (DELPRGTB). The displacement of the entry in the Branch Address Table is indicated opposite the delimiter in the Delimiter Table.

The delimiter in the source text is compared with the group of words in the Delimiter Table having the same number of characters. The length of the delimiter in the source text is contained in REGL. The particular word group in the Delimiter Table, with which the comparison is to be made, is found with the aid of a look-up table (L1TAB) consisting of ten four-byte entries each containing the address of the particular word group. Thus, the address of a given word group comprising words of the same length (REGL) as the source delimiter, is contained in the entry specified by L1TAB + 4*C(REGL). Within a given word group, the entries for all words are uniform in length, being equal to the number of characters in the word, plus three (a two-byte characteristic or operator and a one-byte displacement - the displacement of the corresponding entry in the Branch Address Table, DELPRGTAB). The number of entries in the word group is indicated in the byte preceding the word group (loaded in REGY).

If the apostrophes enclose no characters, Error No. 12 is recorded. If the apostrophes enclose more than ten characters, or if the comparison described above produces no corresponding Delimiter Table entry, control is passed to the Delimiter Error routine (EROUT).

After a delimiter has been correctly identified, a test is made of the STARTBIT to determine if the correct delimiter opening the source module has been found (indicated by STARTBIT=1). If not, control is passed to STARTDEL (see "Processing of Opening Source Text"). Otherwise, the routine corresponding to the delimiter identified is entered.

Before the comparison described above is initiated, a test is made of the switch named FBYTE. FBYTE=X'F0' signifies that one of the delimiters 'PROCEDURE' or 'ARRAY' is being sought; while FBYTE=X'FF' signifies that one of the delimiters 'ELSE' or 'END' is being sought. In either of these cases, control is passed to TYPESPEC or COMSPEC (See also APOSTROF). Otherwise (FBYTE=X'00'), a normal comparison is initiated.

DELIMITER ERROR ROUTINE (EROUT)

EROUT is entered from APOSTROF, when the closing apostrophe of a delimiter word is missing, and from DELIMIT, when a misspelling is detected in a delimiter word. EROUT compares the characters following the opening apostrophe with each of the words in the Delimiter Table (W1TAB), moving downward through the table, and if a matching word is found, branches to the routine specified. (See DELIMIT routine). If no matching delimiter is found, Error No.14 is recorded, the apostrophe is disregarded, and control is returned to TESTLOOP. The comparison proceeds by comparing (1) the first character of the defective delimiter with each of the entries in the first word group of the Delimiter Table, (2) the first two characters with the entries in the second word group, (3) the first three characters with the entries of the third word group, and so on, until a matching delimiter word is found, or until the last word group has been compared.

The comparison is conditional on the switch FBYTE=X'00'. If FBYTE=X'F0' or X'FF', control is passed to TYPESPEC or COMERR directly (see below).

After identification of a delimiter, the same test of the STARTBIT is made as that described under DELIMIT.

1	2	3	4	5	6	7	8
			DELIMITER TABLE (WITAB) — Hexadecimal Representation				
Word Group No.	Delimiter Word	One-byte Operator Notation used in this Manual (see column 6)	No. of Entries in Word Group (First byte in Word Group)	Delimiter Word	One-byte Operator (second byte = X'00') or Two-byte Characteristic for Specificators or Null Operator (both bytes = X'00')	Displacement of Full-Word Entry in Branch Address Table DELPRGTAB, containing Entry Point of Routine Entered	Name of Routine Entered
1	'/'	/	02	03	04 00	00	NORMAL
	'('			06	00 00	04	STRING
2	'DO'	Do	03	43 4E	1C 00	08	TED
	'IF'	If		48 45	1D 00	0C	GIF
	'OR'	Or		4E 51	22 00	00	NORMAL
3	'END'	2)	04	44 4D 43	00 00	10	END
	'FOR'	3)		45 4E 51	00 00	14	FOR
	'AND'	And		40 4D 43	23 00	00	NORMAL
	'NOT'	Not		4D 4E 53	20 00	00	NORMAL
4	'REAL'		09	51 44 40 4B	C2 12	18	TYPE
	'STEP'	Step		52 53 44 4F	19 00	00	NORMAL
	'THEN'	Then		53 47 44 4D	1E 00	08	TED
	'ELSE'	Else		44 4B 52 44	1F 00	08	TED
	'GOTO'	Goto		46 4E 53 4E	17 00	0C	GIF
	'TRUE'			53 51 54 44	07 00 4)	1C	BOLCON
	LESS'	<		4B 44 52 52	11 00	00	NORMAL
	'CODE'	3)		42 4E 43 44	00 00	20	CODE
	'IMPL'	Impl		48 4C 4F 4B	21 00	00	NORMAL
5	'BEGIN'	2)	0A	41 44 46 48 4D	00 00	24	BEGIN
	'UNTIL'	Until		54 4D 53 48 4B	1A 00	00	NORMAL
	'ARRAY'	3)		40 51 51 40 58	CA 16	28	ARRAY
	'VALUE'			55 40 4B 54 44	00 00	2C	VALUE
	'LABEL'			4B 40 41 44 4B	CA 18	30	SPEC
	'WHILE'	While		56 47 48 4B 44	1B 00	00	NORMAL
	'FALSE'			45 40 4B 52 44	00 00 4)	1C	BOLCON
	'POWER'	Power		4F 4E 56 44 51	05 00	00	NORMAL
	'EQUAL'	=		44 50 54 40 4B	10 00	00	NORMAL
	'EQUIV'	Equiv		44 50 54 48 55	24 00	00	NORMAL
6	'SWITCH'	3)	02	52 56 48 53 42 47	CA 1C	34	SWITCH
	'STRING'			52 53 51 48 4D 46	CB 10	30	SPEC
7	'INTEGER'		05	48 4D 53 44 46 44 51	C2 11	18	TYPE
	'BOOLEAN'			41 4E 4E 4B 44 40 4D	C2 13	18	TYPE
	'COMMENT'			42 4E 4C 4C 44 4D 53	00 00	38	COM
	'NOTLESS'	≧		4D 4E 53 4B 44 52 52	15 00	00	NORMAL
	'GREATER'	>		46 51 44 40 53 44 51	12 00	00	NORMAL
8	'NOTEQUAL'	≠	01	4D 4E 53 44 50 54 40 4B	13 00	00	NORMAL
9	'PROCEDURE'	3)	01	4F 51 4E 42 44 43 54 51 44	CA D0	3C	PROCEDUR
10	'NOTGREATER'	≦	01	4D 4E 53 46 51 44 40 53 44 51	14 00	00	NORMAL

Notes:

1. For the specificators 'REAL', 'ARRAY', 'LABEL', 'SWITCH', 'STRING', 'INTEGER', 'BOOLEAN' and 'PROCEDURE', the two-byte characteristic is copied into the Identifier Table entries of specified formal parameters. In the case of all other delimiter words (except for 'TRUE' and 'FALSE' and all delimiters for which both bytes in the column = X'00'), the first byte is transferred to the Modification Level 1 text as a one-byte operator representing the delimiter. The notation in column 3 indicates the name by which the operator is identified in the text.
2. Delimiter variously represented in the Modification Level 1 and 2 versions of source text by two or more one-byte operators, supplied by program. See "Scope Identification", "Modification Level 1 Source Text" and Appendix I-b.
3. Delimiter represented in the Modification Level 1 text by one-byte operator supplied by program. See Appendix I-b and "Modification Level 1 Source Text".
4. First byte specifies the displacement of the constant 0 (False) or 1 (True) in Constant Pool No. 0.

Figure 32. Delimiter Table (WITAB)

TYPE SPECIFICATION (TYPESPEC)

TYPESPEC is entered from DELIMIT and EROUT by virtue of the switch FBYTE=X'F0'. FBYTE is set to X'F0' by the TYPE routine when a test shows that a type declarator ('REAL', 'INTEGER', or 'BOOLEAN') is immediately followed by another apostrophe, indicating a further delimiter. (Unless the latter delimiter is 'PROCEDURE' or 'ARRAY', the source text is in error). TYPE passes control to ENTRAPR (an entry point of APOSTROF), which scans to the next apostrophe and branches to DELIMIT or EROUT, which branch in turn to TYPESPEC on finding FBYTE=X'F0'. TYPESPEC inspects the delimiter and passes control to TYPPROC or TYPEARRY, if the delimiter is 'PROCEDURE' or 'ARRAY', respectively. If any other delimiter is identified, control is passed to the Identifier Error routine IERSPEC. The latter serves to bypass the defective declaration and to record an error.

COMMENT (COMSPEC)

COMSPEC is entered from DELIMIT by virtue of the switch FBYTE=X'FF'. FBYTE is set to X'FF' by the COM routine when an apostrophe is found in a sequence of comment following 'END', indicating that the comment is terminated by a delimiter word. The latter should be 'END' or 'ELSE'. COM passes control to ENTRAPR (an entry point of APOSTROF), which scans to the next apostrophe and branches to DELIMIT, which branches in turn to COMSPEC on finding FBYTE=X'FF'. COMSPEC inspects the delimiter and passes control to END or TED, if the delimiter is 'END' or 'ELSE'. If any other delimiter is identified, control is passed to COMCED2 (an entry point of the COM routine). The latter continues to scan to the next semicolon or apostrophe, disregarding the delimiter.

OPENING DELIMITER (STARTDEL)

See "Processing of Source Module Opening Text".

BEGIN (BEGIN)

BEGIN is entered from DELIMIT and EROUT on recognition of the delimiter 'BEGIN'.

BEGIN inspects the Scope Handling Stack and, unless the stack operator is Proc, transfers the operator Begin to the Modification Level 1 text, stacks Begin, and turns the BEGBIT switch on.

If the stack operator is Proc, indicating that the delimiter 'BEGIN' opens the body of a procedure closed by 'END', the stack operator Proc is replaced by Proc* and the PROBIT switch turned off.

STRING (STRING)

STRING is entered from DELIMIT and EROUT on recognition of the first of two string quote signs '('...')' enclosing a character string. STRING stores the enclosed character string in the Constant Pool and transfers a five-byte internal name, referencing the location where the string is stored, to the Modification Level 1 text. The internal name (see Appendix II) is preceded by the Apostrophe operator.

The string is stored in the Constant Pool in the external code (EBCDIC or ISO) of the source module -- it is copied from the print area (or dummy print area) to which each source module record is moved by CIB, before the record is translated to the internal code. See Figure 7.

NORMAL ACTION (NORMAL)

NORMAL is entered from DELIMIT or EROUT when any one of the following delimiters is identified: '/', 'OR', 'AND', 'NOT', 'STEP', 'LESS', 'UNTIL', 'NOTLESS', 'EQUAL', 'EQUIV', 'IMPL', 'WHILE', 'GREATER', 'NOTEQUAL', and 'NOTGREATER'. NORMAL transfers the corresponding one-byte operator in the Delimiter Table (Figure 32) to the output buffer and returns control to TESTLOOP or LIST.

BOOLEAN CONSTANT (BOLCON)

BOLCON is entered from DELIMIT or EROUT when the boolean constant 'TRUE' or 'FALSE' is encountered. A five-byte internal name (Figure 33) is transferred to the output buffer, indicating the character of the boolean constant, and referencing a location in the Constant Pool where the constant 0 (False) or 1 (True) is stored. The internal name is preceded by the Apostrophe (X'2E'), which signals the Scan III Phase that an internal name follows.

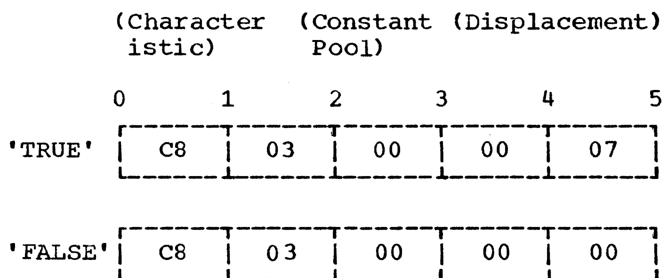


Figure 33. Internal Names of boolean constants 'TRUE' and 'FALSE'

GOTO-IF (GIF)

GIF is entered from DELIMIT or EROUT on recognition of the delimiters 'GOTO' and 'IF'. The one-byte operator given in the Delimiter Table (Figure 32) is transferred to the output buffer and control passed to STATE.

THEN-ELSE-DO (TED)

TED is entered from DELIMIT or EROUT on recognition of the delimiters 'THEN', 'ELSE', or 'DO'. A one-byte symbol representing the delimiter in the Delimiter Table (Figure 32) is transferred to the output buffer. Before control is returned to TESTLOOP or LIST, pointers OPIN and LAPIN are set to point, respectively, to the delimiter symbol transferred and to the next byte in the output buffer (see "Phase Initialization").

FIRST BEGIN (FIRSTBEG)

See "Processing of Source Module Opening Text".

PROGRAM BLOCK (BEG1 SUBROUTINE)

BEG1 is activated as soon as a new block has been identified. It is entered from all declaration-handling routines (e.g. TYPE, PROCEDUR, ARRAY) processing the first declaration following the delimiter 'BEGIN' (indicated by the switch BEGBIT=1). BEG1 constructs a program block heading entry in the Identifier Table, containing a new Program Block Number; resets the pointers LPBP and LIGP to the new heading entry; and replaces the operator Begin by Beta in the output text and in the Scope Handling Stack. It also constructs entries in the Group Table, Program Block Number Table and Semicolon Table.

END (END)

END is entered from DELIMIT or EROUT when the delimiter 'END' is recognized. Its function is to inspect the operator in the Scope Handling Stack and to activate the appropriate closing subroutine, according to the stack operator detected:

Stack Operator	Subroutine Activated
Beta, Proc Proc*, Proc**	PBLCKEND
Begin	COMPDEND
For	FOREND
Alpha	ERR8 (see "Close of Scan I/II Phase")

The subroutines are described below.

COMPOUND END (COMPDEND)

COMPDEND releases the stack operator Begin and transfers the operator End to the output text, marking the close of a compound statement. See END.

FOR STATEMENT END (FOREND)

FOREND, which is entered from END and SEMCO on detection of the delimiter 'END' or a semicolon closing a for statement, constructs a for statement closing entry in the Identifier Table, if the closed for statement contained a declared label. If the closed for statement contained no declared labels, the for statement heading entry is deleted. FOREND also transfers the operator Eta, followed by the Identifier Group Number, to the output text, and releases the stack operator For. Pointer LIGP is reset to point to the heading entry of the reentered for statement, if any, or to the heading entry of the enclosing block or procedure.

PROGRAM BLOCK END (PBLCKEND SUBROUTINE)

PBLCKEND, which is called by END and SEMCO on detection of the delimiter 'END' or a semicolon closing a block or procedure, transfers the set of entries in the Identifier Table representing identifiers declared or specified in the block or procedure to the SYSUT3 data set. The program block heading entry which heads this set of identifiers is indicated by the pointer LPBP. Before the transfer is executed, pointers LPBP and LIGP are reset to the address of the heading entry corresponding to the enclosing block or procedure (see "Processing of Identifier Table"). PBLCKEND also transfers the operator _Epsilon_ to the output text, followed by the Program Block and Identifier Group Number of the enclosing block, procedure or for statement, and releases the stack operator.

COMMENT (COM)

COM has two main functions: to bypass comments, and to delete (or bypass) erroneous declarations. The routine scans the source text (using translation table COMTABLE) for a semicolon, an apostrophe, a blank, or _Zeta_. The function bytes assigned these characters specify displacements to subprograms of the COM routine.

There are three entry points: COM, COMMEND, and COMERR.

COM is entered from DELIMIT and EROUT when the delimiter 'COMMENT' has been encountered. Scanning continues until a semicolon is found. This deletes (or bypasses) all source text beginning with 'COMMENT' and extending up to and including the semicolon.

COMMEND is entered from END when a comment of the following form is to be eliminated: 'END'<comment> 'END'/;/ 'ELSE'. Scanning terminates when a semicolon or an apostrophe is found, deleting the preceding comment. In case a semicolon is found, SEMCO is entered. If an apostrophe is found, the switch FBYTE is set to X'FF' and control passed to ENTRAPR (an entry point of APOSTROF). APOSTROF scans to the next apostrophe, branches to DELIMIT (or EROUT), which branches to COMSPEC on finding FBYTE=FF. COMSPEC inspects the delimiter and branches to END or TED if the delimiter is 'END' or 'ELSE', respectively. In all other cases, COMERR is entered.

COMERR is entered from several declaration-processing routines when an erroneous declaration is identified. Scanning continues until a semicolon is found. This deletes all source text beginning with the declarator and extending up to (but not including) the next semicolon. When the semicolon is found, SEMCO is entered.

FOR STATEMENT (FOR)

FOR is entered from DELIMIT and EROUT on recognition of the delimiter 'FOR'. A for statement heading entry is constructed in the Identifier Table and entries are made in the Scope and Group Tables. If the stack operator is _Proc_, it is replaced by _Proc**_. The operator _For_ is stacked and transferred to the output text, followed by a new Identifier Group Number.

TYPE DECLARATION (TYPE)

TYPE is entered from DELIMIT and EROUT on recognition of any of the declarators 'REAL', 'INTEGER', or 'BOOLEAN'. The routine makes an entry in the Identifier Table for each of the identifiers following the declarator, provided the identifier is valid. If any invalid character is found in the identifier, control is passed to the Identifier Error routine (IER), which deletes the entry made in the Identifier Table and records Error No. 5 or 16. If the declarator is immediately followed by another apostrophe, indicating a further delimiter, the switch FBYTE is set = X'F0' and ENTRAPR (an entry point of APOSTROF) is entered.

At entry, the switches PROBIT and BEGBIT are tested, in that order. If PROBIT=1, indicating that the delimiter specifies a formal parameter in a procedure heading, control is passed to the SPECENT routine. If BEGBIT=1, indicating that the declarator represents the first declaration following 'BEGIN' and that, accordingly, a new block has been entered, a call is made to the BEG1 subroutine (which assigns a new Program Block Number) before entries for the declared identifier(s) are made in the Identifier Table.

IDENTIFIER ERROR (IER)

IER is entered from declaration-processing detection of a defect in a declared identifier. The routine deletes all or part of an entry for the identifier in the Identifier Table and records Error No. 5 or 16, depending on the entry point (IER or

IERSPEC). It also skips over the source text up the next comma, semicolon, or right parenthesis, in the case of a formal parameter list.

CODE PROCEDURE (CODE)

CODE is entered from DELIMIT or EROUT on recognition of the delimiter 'CODE', representing the body of a code procedure. The routine verifies that 'CODE' follows a procedure heading; modifies the characteristic in the entry previously made (by the PROCEDUR and PROCID routines) for the procedure identifier in the Identifier Table, so as to designate a code procedure, and transfers up to six characters of the procedure identifier, followed by two Blanks, to the output text, preceded by the operator Gamma. After finding the semicolon which should follow 'CODE', the PBLCKEND subroutine is called and control then passed to SEMCO.

At entry, the switches PROBIT and BEGBIT are tested, in that order. If PROBIT=1 indicating that the delimiter specifies a formal parameter in a procedure heading, control is passed to the SPECENT routine. If BEGBIT=1, indicating that the declarator represents the first delimiter following 'BEGIN', and that, accordingly, a new block has been entered, a call is made to the BEG1 subroutine (which assigns a new Program Block Number) before processing continues.

SPECIFICATION (SPEC)

SPEC is entered from DELIMIT or EROUT on recognition of the specificators 'LABEL' and 'STRING'. Its function is to verify that the specificators occur in a procedure heading. If they do, control is passed to the Type Specification routines (SPECENT and IDCHECK). If not, Error No.25 is recorded, and the declaration is skipped by branching to COMERR.

VALUE (VALUE)

VALUE is entered from DELIMIT or EROUT on recognition of the delimiter 'VALUE'

After testing the switch PROBIT to insure that the delimiter occurs in a procedure heading (signified by PROBIT=1), the switch VALBIT is turned on and control is passed to IDCHECK. The latter locates the Identifier Table entry corresponding to each formal parameter which follows'VALUE', and, by virtue of VALBIT=1, sets the value bit in the identifier characteristic (Figure 9) so as to designate a value-called parameter.

PARAMETER SPECIFICATION (SPECENT and IDCHECK)

SPECENT is entered from TYPE, ARRAY, SWITCH and PROCEDUR, when a specificator is encountered in a procedure heading (indicated by PROBIT=1). SPECENT moves the corresponding two-byte characteristic contained in the Delimiter Table (Figure 32) to a field named KB and then enters IDCHECK.

IDCHECK is entered from SPECENT and from VALUE. IDCHECK's function is to locate the appropriate entry (entries) in the Identifier Table and a) to insert the characteristic and Program Block Number, or b) to set the value bit in the characteristic. (Before the value or specification parts of a procedure heading are processed, the external names of all formal parameters are copied into a sequence of Identifier Table entries, from the parameter list which follows the procedure identifier. The first of these entries is addressed by the pointer PRIMPAR). The characteristic is inserted by ORing the relevant bytes of the Identifier Table entry with the contents of the location KB.

TYPE ARRAY (TYPEARRY)

TYPEARRY is entered from TYPESPEC when a delimiter sequence of the type '<type>' 'ARRAY' has been identified. If PROBIT=1 (indicating the delimiter sequence occurs in a procedure heading) control is passed to IDCHECK which proceeds to complete the Identifier Table entry for a type-array parameter of a procedure. If BEGBIT=1 (indicating the delimiter sequence represents the first declaration following 'BEGIN', and that accordingly 'BEGIN' opens a block), the BEG1 subroutine is called. Thereafter, control is passed to the ARRAY routine (by way of ARRYDME1), which constructs an entry for a type-array identifier in the Identifier Table.

ARRAY DECLARATION (ARRAY)

ARRAY is entered from DELIMIT and EROUT on recognition of the delimiter 'ARRAY'. The routine constructs an entry in the Identifier Table for each array identifier following the declarator (by call to the IDCHECK1 subroutine); and transfers the operator Array to the output text, followed by up to six characters of each identifier. On recognition of the left bracket, (/, marking the beginning of the dimension list, the operator [is transferred to the output text and control passed to the LIST routine. The LIST routine analyzes the dimension list, records a count of the number of dimensions in the corresponding Identifier Table entries, transfers the dimension list to the output text, and returns control to ARRAY if the dimension list is followed by a further identifier.

At entry, the switches PROBIT and BEGBIT are tested, in that order. If PROBIT=1, indicating that the delimiter specifies a formal parameter in a procedure heading, control is passed to the SPECENT routine. If BEGBIT=1, indicating that the declarator represents the first declaration following 'BEGIN' and that a new block has been entered, a call is made to the BEG1 subroutine (which assigns a new Program Block Number), before entries for the declared array(s) are made in the Identifier Table.

ARRAY/SWITCH LIST (LIST)

The LIST routine is entered from the ARRAY and SWITCH routines upon recognition of a dimension list in an array declaration or a component list in a switch declaration. LIST scans the source text (beginning with the first character following the left bracket in an array declaration or the first character following the assignment operator in a switch declaration) for any one of 15 characters assigned a non-zero function byte in Translation Table (ARTABLE); moves the scanned text to the output buffer; and branches to the routine whose address is specified in a full-word entry of Branch Address Table (BPRTAB) given by the value of the character's function byte.

The function bytes assigned by ARTABLE to the character set and the routines entered from LIST are as follows:

Character	Function Byte	Routine Entered
Apostrophe	34	APOSTROF
*	04	TRANSOP
>	14	"
<	10	"
Not	18	"
<u>Zeta</u>	38	CIB
Blank	28	BLANK
Invalid Character	2C	ERR1
Comma	80	COMMALST
/	88	SLASHLST
)	6C	RIGHTPARL
(	68	LEFTPARL
Point	84	PONTLST
Colon	54	COLONLST
Semicolon	58	SEMCLST
<All other characters>	00	(No branch, scanning continues)

The latter seven routines recognize separators in a dimension on component list; transfer representative operators to the output buffer; count the number of dimensions or components in a dimension or component list; and record the dimension count or component count in the corresponding Identifier Table entries made by the ARRAY or SWITCH routines for the array or switch identifiers. The actual dimensions in a dimension list or the components in a component list are transferred to the output text by the LIST routine before branching to the routine concerned. The switch ARBIT=1 specifies an array dimension list, while ARBIT=0 specifies a switch component list.

POINT IN LIST (PONTLST)

PONTLST inspects the character following the point and passes control to COLONLST or SEMCLST or transfers a <u>Decimal Point</u>.

RIGHT PARENTHESIS IN LIST (RIGHTPARL)

RIGHTPARL transfers a right parenthesis and decrements the bracket count.

LEFT PARENTHESIS IN LIST (LEFTPARL)

LEFTPARL transfers a left parenthesis or a left bracket, [, representing (/, and increments the bracket count.

COMMA IN LIST (COMMALST)

COMMALST increments the dimension count and transfers the <u>Comma</u> operator.

COLON IN LIST (COLONLST)

Transfers a colon, provided it occurs in an array dimension list. If it occurs in a switch component list, the colon is disregarded and Error No.3 is recorded.

SEMICOLON IN LIST (SEMCLST)

SEMCLST stores the component count in the Identifier Table entry specified by a pointer named DIM, and transfers control to SEMCO, after specifying the return address of TESTLOOP. If the semicolon occurs in an array dimension list, Error No.32 is

recorded, and the identifier entry is deleted.

SLASH IN LIST (SLASHLST)

SLASHLST inspects the character following the slash and transfers the slash or a right bracket,]; enters the dimension count for a declared array in the Identifier Table entry indicated by pointer DIM; and transfers control to ARRAY, SEMCO, or COMERR, according to whether the character following is a comma, a semicolon, or any other character, excepting Zeta or a blank.

SWITCH DECLARATION (SWITCH)

SWITCH is entered from DELIMIT and EROUT on recognition of the declarator 'SWITCH'. SWITCH constructs an entry in the Identifier Table for the identifier following the declarator, and transfers up to six characters of the identifier to the output text, preceded by the operator Switch. On detection of the assignment operator marking the beginning of the component list, the Assign operator is transferred and control passed to the LIST routine. LIST transfers the component list to the output text, counts the number of components in the list, and enters the component count in the Identifier Table entry for the switch identifier.

At entry to the routine, the switches PROBIT and BEGBIT are tested, in that order. If PROBIT=1 (indicating that the delimiter specifies a formal parameter in a procedure heading), control is passed to the SPECENT program. If BEGBIT=1 (indicating that the declarator represents the first declaration following 'BEGIN' and that, accordingly, a new block has been entered), the subroutine BEG1 is called before entries for the declared switches are made in the Identifier Table.

PROCEDURE DECLARATION (PROCEDUR)

PROCEDUR is entered from the DELIMIT and EROUT routines on recognition of the delimiter 'PROCEDURE'. PROCEDUR makes entries in the Group, Semicolon, and Program Block Tables; transfers the operator Pi to the output text and the Stack; inserts the characteristic for a declared procedure identifier into the next entry of the Identifier Table; and passes control to PROCID, which constructs an entry in the Identifier Table, containing a new Program Block Number for the procedure identifier, followed by a program block heading entry, and copies the external names of the formal parameters in the parameter list into the following entries.

Initially, the PROBIT and BEGBIT switches are tested, in that order. If PROBIT=1 (indicating that the delimiter 'PROCEDURE' specifies a formal parameter in a procedure heading), control is passed directly to SPECENT. If BEGBIT=1 (indicating that the delimiter represents the first declaration following 'BEGIN' and that, accordingly, 'BEGIN' opens a new block), a call is made to the BEG1 subroutine.

PROCEDURE IDENTIFIER (PROCID)

PROCID is entered from PROCEDUR when a procedure declaration has been encountered. PROCID first constructs an entry in the Identifier Table for the procedure identifier. The external name (up to six characters) is copied into the entry and transferred to the output text by call to IDCHECK1. The characteristic for the procedure identifier will have been stored in the entry by PROCEDUR. When a left parenthesis (opening a parameter list) or a semicolon (following the identifier of a parameterless procedure) is encountered, a program block heading entry is constructed. If the procedure is a type-procedure, a second identifier entry for the procedure identifier is made immediately after the heading entry. The external names of the formal parameters, represented by a maximum of six characters, are now copied into the following entries of the Identifier Table and the output text. The two-byte characteristics of these parameters are inserted immediately after, by the SPECENT routine when the specifications in the procedure heading are processed. Control is passed to SEMCO as soon as a semicolon following the closing right parenthesis of the parameter is encountered.

TERMINATION (EODADIN)

EODADIN is entered from:

1. PBLCKEND (via COMMEND and READRCUT) when the stack operator Alpha (marking the bottom of the Scope Handling Stack) indicates that the outermost scope of the source module has been closed;

2. ENDMISS when an unexpected End of Data condition occurs;

3. PIROUT (in the Directory) when a program interrupt or unrecoverable I/O error occurs; and

4. ERR4 when a terminating error is detected in the source module.

See also "Close of Scan I/II Phase".

EODADIN transfers the closing operator <u>Omega</u> to the Modification Level 1 text; writes out the last record of the modified source text (by calling COB), except when the entire text occupies less than a full buffer (in which case it is transmitted to the Scan III Phase in the Common Area buffer); generates TXT records of the character strings in the Constant Pool (by calling the GENERATE - GENTXT5 subroutine) on the SYSPUNCH and/or SYSLIN data sets, provided the DECK and/or LOAD options have been specified; closes the SYSIN, SYSUT1, and SYSUT3 data sets; releases main storage; and transfers control to the Identifier Table Manipulation Phase (IEX20), or, if a terminating error has occurred, to Diagnostic Output Module IEX21.

If the source module is a precompiled procedure, Program Block No. 0 in the Identifier Table, containing an entry for the procedure name, is transferred to the SYSUT3 data set (by call to PBLCKEND) and an ESD record for the procedure name is generated (by calling GENERATE-GENESD). The precompiled procedure name will have been stored in external code at the location named ESDPARAM by the PROCID routine.

GENERATE SUBROUTINE

See Chapter 8.

PURPOSE OF THE PHASE

The main purpose of the Identifier Table Manipulation Phase is to complete the construction of the internal names of all identifiers listed by the Scan I/II Phase in the Identifier Table. Except in the case of entries for declared procedure and switch identifiers and labels, the last two bytes of the internal name provide space for the relative address of the identifier's object time storage field (Figure 36). The Identifier Table Manipulation Phase assigns an object time storage field to each identifier, and stores the corresponding relative address in the space provided in the identifier's internal name.

The processing of the Identifier Table, which forms the main input to the Identifier Table Manipulation Phase, may be divided into the following functions.

1. To search each group of identifiers in the Identifier Table for repeated declarations of the same identifier, and to record appropriate error patterns in the Error Pool.

2. To allocate object time storage fields to the identifiers listed in the Identifier Table, and to record the relative address of each identifier's assigned storage field in the identifier's internal name. The relative address represents a displacement from the beginning of a Data Storage Area, comprising the total number of bytes assigned to identifiers declared or specified in the particular block or procedure.

3. To construct Program Block Table II (PBTAB2), indicating the size of the Data Storage Area required at object time for every block and procedure in the source module. Program Block Table II is transmitted in main storage to the Compilation Phase, in which the space requirements recorded in the table are augmented by additional space allocations for the storage of intermediate results.

4. To transmit the completed Identifier Table (via SYSUT3) to the Scan III Phase according to ascending Program Block Number sequence.

5. To generate a printed listing of the contents of the Identifier Table, if the SOURCE option has been specified.

IDENTIFIER TABLE MANIPULATION PHASE OPERATIONS

The diagram in Figure 34 illustrates the principal operations performed in the Identifier Table Manipulation Phase. The bracketed numbers in the following text refer to the numbered positions in the diagram.

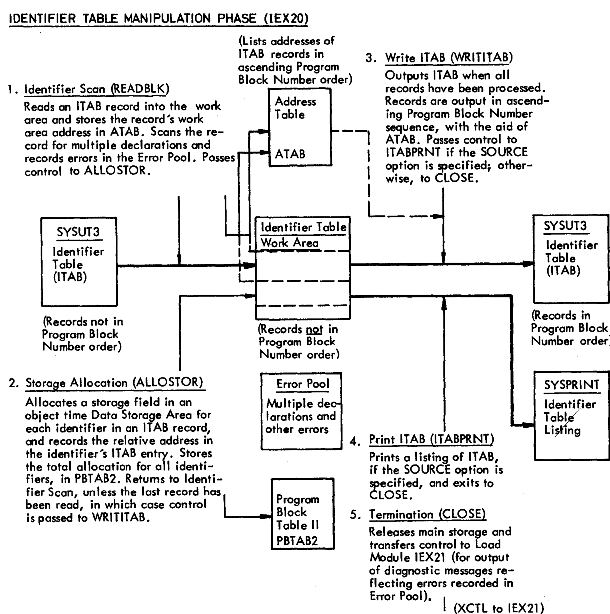


Figure 34. Identifier Table Manipulation Phase. Diagram illustrating the functions of the principal constituent routines.

Identifier Table records (1) are read into a work area and processed, one at a time, in the order in which they were stored on the SYSUT3 data set by the Scan I/II Phase, that is, according to the sequence in which the blocks and procedures were closed in the source module. To enable the records to be output in ascending Program Block Number sequence, the address of each record is stored in the Address Table (ATAB), in an entry determined by the record's Program Block Number. Initially, each record is scanned by the Identifier Scan routine to determine if multiple declarations were made for the same identifier.

After the record has been scanned and appropriate errors recorded in the Error Pool, the Storage Allocation routine (2) allocates an object time storage field to each identifier, and records the address of the allocated bytes (relative to the beginning of the Data Storage Area comprising the total allocation for the block or procedure) in the corresponding identifier entry.

When all records of the Identifier Table have been read in and processed in this manner, the Identifier Table is (3) retransferred to the SYSUT3 data set, records being output in ascending Program Block Number sequence. If the SOURCE option was specified, (4) a listing of the Identifier Table is printed; otherwise, (5) the termination routine (CLOSE) is entered. CLOSE transfers control to the next phase.

PHASE INPUT/OUTPUT

Figure 35 pictures the data input to and output from the Identifier Table Manipulation Phase. The figure also shows the tables transmitted to and from the phase in main storage.

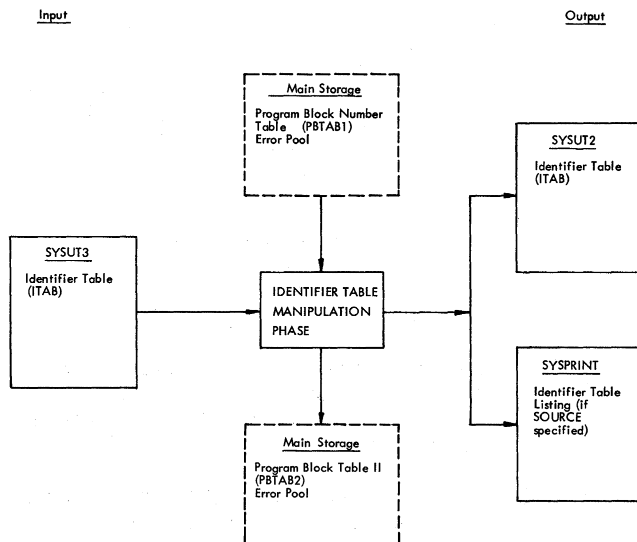


Figure 35. Identifier Table Manipulation Phase Input/Output

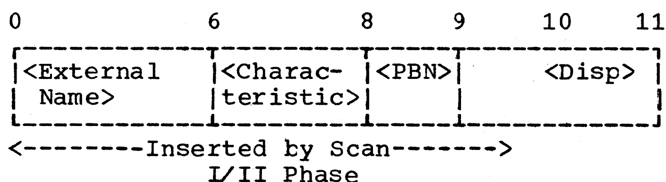
Identifier Table records (variable length) are read from the SYSUT3 data set by a READ macro.

Output of the completed Identifier Table to SYSUT3 is initiated after all records of the Scan I/II version have been read into main storage and processed, and after the data set has been closed by a Type T CLOSE.

The Identifier Table listing is compiled by the ITABPRNT routine and printed on SYSPRINT, line by line, by call to PRINT in the Directory. Phase Input/Output

IDENTIFIER TABLE (ITAB)

Figure 36 shows the space provided (last one-and-one-half bytes) in the typical identifier entry for the relative address of an identifier's storage field in the particular block's or procedure's object time Data Storage Area. The figure is not representative of identifier entries for declared labels and declared procedure and switch identifiers, in which the last 1 1/2 bytes contain a displacement address in the object time Label Address Table, inserted by the Scan I/II Phase.



<Disp> = <Displacement in the block's or procedure's Data Storage Area>

Figure 36. Identifier Table (ITAB) entry, showing the identifier's Data Storage Area displacement address, as inserted by the Identifier Table Manipulation Phase in bytes 9 and 10, for all identifiers except those of declared procedures, switches and labels.

The Identifier Table is described more fully in Chapter 4.

PROGRAM BLOCK TABLE II (PBTAB2)

The Program Block Table II (PBTAB2) indicates the total number of object time storage bytes allocated in the Identifier Table Manipulation Phase to each block and procedure in the source module. PBTAB2 is transmitted in main storage to the Compilation Phase, where it is transferred to Program Block Table III.

```
0                                        2
┌─────────────────────────────────────────┐
|<Total bytes allocated to the block or   |
|             procedure>                  |
└─────────────────────────────────────────┘
```

Figure 37. Two-byte entry in Program Block Table II (PBTAB2)

PBTAB2 is constructed by the ALLOSTOR routine. The total storage allocation for a particular block or procedure is stored in the entry corresponding to the particular Program Block Number.

CONSTITUENT ROUTINES OF IDENTIFIER TABLE MANIPULATION PHASE

The principal constituent routines of the Identifier Table Manipulation Phase are described below. The index in Appendix XI indicates the page on which each routine is described and the flowchart in the Flowchart Section in which the general logic of the routine is set forth.

PHASE INITIALIZATION

The Initialization routine gets main storage for the private work area shown in Figure 38; initializes pointers and switches; specifies EOD and program interrupt-I/O error routines; calls the PRINT subroutine in the Directory, after assembling headlines for the Identifier Table listing, provided the SOURCE option is specified, and exits to READBLK (which reads in Identifier Table records from the SYSUT3 data set).

The program interrupt-I/O error exit, CLOSE2, is stored at ERET, a location in the Common Work Area referenced by the PIROUT routine in the Directory. The exit is changed, after the GETMAIN instruction, to CLOSE. CLOSE releases main storage and transfers control to Diagnostic Module IEX21, while CLOSE2 simply transfers control to IEX21.

The GETMAIN instruction for the private work area is issued after the total area required for the Identifier Table (ITAB) and the Address Table (ATAB) has been computed. The area allotted to the Identifier Table is fetched from the ITAB20S entry in the Area Size Table in the Common Work Area. The area provided for the Address Table is fixed at 1024.

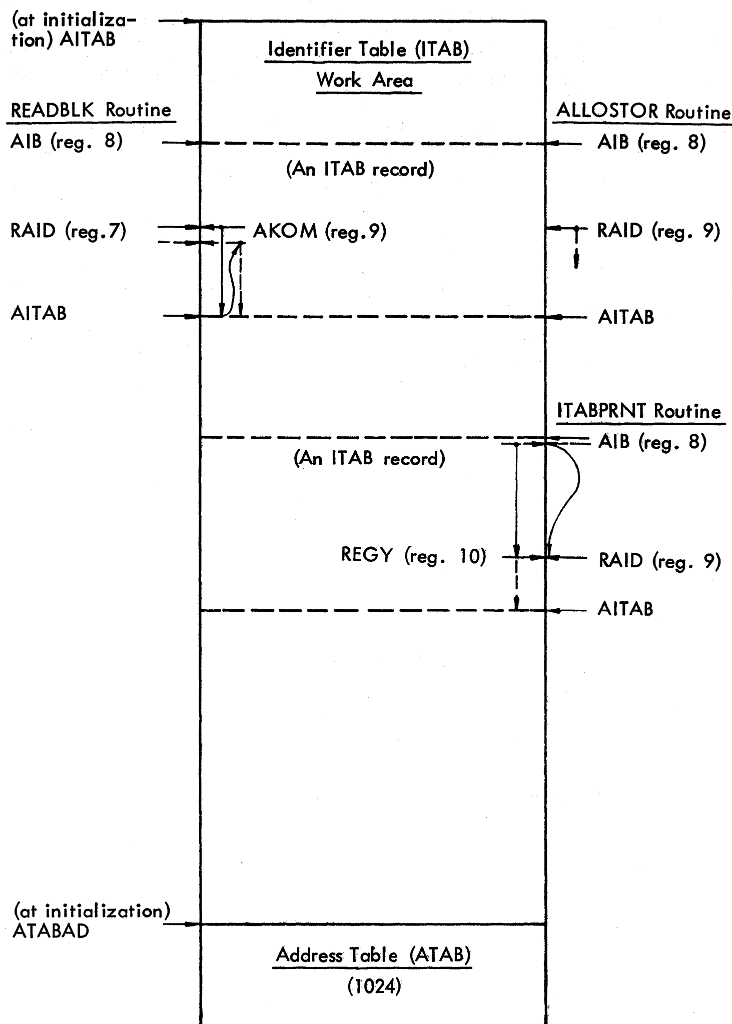
SAVEPB, SAVE, and BITS1 are three Common Work Area locations defined by a dummy control section in Load Module IEX20 (ITAB Manipulation). SAVEPB is the name of the Program Block Counter which is incremented by 1 in the Lastrec routine for every Identifier Table record processed. SAVEPB is compared with PBN (the program block count transmitted by the Scan I/II Phase in the Common Work Area), and if the count is identical (indicating that all Identifier Table records have been read in from SYSUT3 and processed), control is passed to WRITITAB, which outputs the table on SYSUT3.

SAVE is a location used (by the ITABPRNT routine) in converting numerical data, in connection with the print-out of the Identifier Table.

BITS1 contains a switch, named PROCBIT. PROCBIT=1 (turned on in the ITABPRNT routine on recognition of a procedure identifier) signifies that the Identifier Table entry being processed is that of a procedure identifier, and that the parameter count in the internal name is the actual count and should not be increased by 1 when the entry is printed out. PROCBIT=0 signifies that the Identifier Table entry being processed is that of an array or switch identifier, and that the dimension count or component count in the internal name represents the actual count, less one, and should be increased by 1 when the entry is printed out.

If the SOURCE option is specified, the headlines "IDENTIFIER TABLE" for the first line, "PBN SC PBN NAME TYPE DM DSP NAME TYPE DM DSP NAME TYPE DM DSP" for the second, and "SURR PR LN PR LN PR LN" for the third line, are moved to a field named PAGEHEAD in the Common Work Area. A call is then made (via PRINTITB) to the PRINT subroutine in the Directory, which prints out the headings on a new page. Resetting to a new page is governed by presetting the line count to 128 (in LINCNT) before calling PRINT.

If short precision has been specified (determined by testing the ING switch in the HCOMPMOD Control Field), the value 4 is stored in the half-word named C, displacing the defined constant 8, and specifying to the ALLOSTOR routine that arithmetic identifiers are to be allocated four bytes each. If short precision has not been specified, C remains unchanged at 8, and real (or floating point) identifiers will accordingly be assigned eight bytes each.



Notes:

1. In the READBLK routine AITAB initially addresses the location in the Identifier Table Work Area to which the next record is read from SYSUT3. After read-in of the record is complete, AIB is set equal to AITAB, and AITAB is then incremented by the length of the record (in the heading entry), so that AIB and AITAB now point to the beginning and end of the record. RAID addresses successive identifier entries in the record, moving progressively through the record, while AKOM addresses each of the entries following RAID, with which the identifier addressed by RAID is compared.

2. In the ALLOSTOR routine, AIB and AITAB point to the beginning and end of the record. RAID addresses successive identifier entries in the record.

3. In the ITABPRNT routine, AIB and AITAB point to the beginning and end of the record currently being processed. For each record processed, AIB and AITAB are set by loading AIB with the address contained in the Address Table entry corresponding to the next sequential Program Block Number, and then setting AITAB = AIB + (the length of the ITAB record addressed by AIB, the length being contained in the heading entry). Identifiers are printed out in alphabetical order, three on each line of printed text. To find the next identifier in alphabetical order, RAID and REGY are initializedat the first identifier entry in the record. REGY then addresses the following identifiers in turn, each identifier being compared with the identifier addressed by RAID. When REGY addresses an identifier of higher alphabetical order than RAID, RAID is reset to REGY. This procedure is repeated until the end of the record is reached, so that RAID now addresses the next identifier in alphabetical order. After the identifier has been processed, it is deleted, by shifting the identifier entries at the bottom of the record upward by the entry length.

* Area size specified by Area Size Table in Common Work Area. See Appendix VIII for the variation in area sizes as a function of the SIZE option.

Figure 38. Private Area acquired by the Identifier Table Manipulation Phase

IDENTIFIER SCAN (READBLK)

The Identifier Scan routine reads an Identifier Table record from the SYSUT3 data set into the work area provided, and searches the record for duplicate identifiers. The search is made by comparing each identifier in turn with all of the following identifiers. If a duplicate identifier is detected, Error No.45 is recorded, by calling E43. For statement heading and closing entries (identified by X'2B' in byte 6) are disregarded. As soon as all entries in the record have been scanned, the ALLOSTOR routine is entered.

Records are read into the work area location specified in AITAB. The address in AITAB is stored in the Address Table (ATAB) entry corresponding to the Program Block Number of the record. The length of this entry is then added to the address in AITAB, so that AITAB will point to the location where the next record will be read. See Note 1 in Figure 38.

STORAGE ALLOCATION (ALLOSTOR)

ALLOSTOR allocates a specific number of object time storage bytes to each identifier represented in an Identifier Table record (excepting declared procedure, switch, and label identifiers), and records the relative address of the bytes allocated in the corresponding identifier entry. This relative address specifies the displacement in the Data Storage Area to be set up for the particular block or prce-

dure at object time. When all the identifiers in a record have been processed, the total number of bytes allocated is recorded in an entry of Program Block Table II (PBTAB2), corresponding to the Program Block Number of the block or procedure. Control is returned to the READBLK routine (which reads in the next Identifier Table record) if a further record remains to be processed. This is indicated if the Program Block Counter SAVEPB, stepped up for every record processed, is less than the program block count recorded in PBN by the Scan I/II Phase. When SAVEPB=PBN, control is passed to the WRITITAB routine.

The allocation for each type of identifier is as follows:

Identifier	Bytes	Precision
Real variables	4	Short
	8	Long
Integer variables	4	
Boolean variables	1	
Arrays	4(NOS+6)	Short
	4(NOS+6)+X	Long
	[X=4 if NOS is odd, X=0 if NOS is even]	
Formal parameters	8	
Declared labels, procedure and switch Identifiers	None	

NOS denotes the number of dimensions, diminished by one, recorded in the Identifier Table entry for an array identifier. The area allocated for arrays provides space for a Storage Mapping Function (see Figure 62). A 24-byte field is reserved at the beginning of the Data Storage Area for every block and procedure (except in the case of <type>-procedures, for which the area reserved is 32 bytes).

Object time storage space is allocated with the aid of a set of displacement pointers named DP (Double Word Pointer). WP (Word Pointer), HP (Half Word Pointer), and BP (Byte Pointer). These pointers are zero-set at the beginning of every Identifier Table record. DP reflects the total displacement at any point in terms of double- words. It is incremented at double and full-word boundaries by 4 or 8 bytes, depending on the precision specified. Where the allocation to be made for an identifier is less than a double word, pointer BP, HP, or WP may be set equal to DP and incremented by one, two, or four bytes, so as to minimize the number of unused bytes. See Note 2 in Figure 38.

WRITE IDENTIFIER TABLE (WRITITAB)

WRITITAB is entered from ALLOSTOR when all Identifier Table records have been read into main storage and processed. After repositioning SYSUT3 by a Type T CLOSE, WRITITAB transfers the Identifier Table records in the work area to the SYSUT3 data set, in ascending Program Block Number sequence. The address of the record corresponding to the next sequential Program Block Number (in REGZ) is determined by reference to the Address Table (ATAB) entry for that Program Block Number. Control is passed to the ITABPRNT routine when the Program Block Number in REGZ equals the program block count stored in PBN by the Scan I/II Phase.

PRINT IDENTIFIER TABLE (ITABPRNT)

ITABPRNT generates a listing of the contents of the Identifier Table, containing the external name of each identifier and indicating (by means of a system of coded symbols) the characteristics of the identifier. Output of the listing, whose format is described in the OS ALGOL Programmer's Guide, is dependent on the SOURCE option being specified.

The identifier groups are listed in ascending Program Block Number sequence, and within each group the identifiers are listed in alphabetical order. See note in Figure 38.

TERMINATION (CLOSE)

CLOSE releases the main storage area occupied by the Identifier Table and the Address Table, and transfers control to Diagnostic Output Module IEX21 (see Chapter 9).

PURPOSE OF THE PHASE

The purpose of the Scan III Phase is to read the Modification Level 1 source text output by the Scan I/II Phase and to perform the following principal tasks:

1. To replace the external names of all identifiers in the modified source text by their corresponding internal names in the Identifier Table (see Chapter 4).

2. To store constants in the source text in the Constant Pool, and to replace each constant by a five-byte internal name, referencing the location where the constant is stored.

   A constant is stored in the Constant Pool in fixed or floating point representation, depending on whether the constant is an integer number or a real number. TXT records of the constants stored in the Constant Pool are generated on the SYSLIN and/or SYSPUNCH data sets, according to the Compiler options specified (see Item 9).

3. To construct the For Statement Table (FSTAB), indicating the critical features of every for statement in the source text. The For Statement Table, which is transmitted to the two subsequent phases via main storage, serves to determine the structure of the loop generated in the object code for each for statement. Among other things, the For Statement Table assigns each for statement to one of three loop classifications (Normal Loop, Elementary Loop, or Counting Loop) and indicates the character of the for list (e.g. if the for list contains a step or while element). It also indicates if subscript optimization is to be performed for optimizable array subscripts in a for statement.

4. To construct the Subscript Table (SUTAB) listing, under each for statement, all subscript expressions of a defined character occurring in the iterated part of the for statement. The expression must be of the type ±F*V±A, where V is the controlled variable, and the factor F and addend A must be integer variables or constants.

   The Subscript Table is transmitted (on the SYSUT3 data set) to the Subscript Handling Phase, in which optimizable subscript expressions are identified and copied into the Optimization Table (OPTAB) for transmission to the Compilation Phase. To be optimizable, no assignment may be made in the for statement to the factor F or the addend A in the subscript expression. The test for optimizability is performed in the Subscript Handling Phase by comparing the factor and addend with the variables listed in the Left Variable Table (see next item).

5. To construct the Left Variable Table (LVTAB), listing, under each for statement, the integer left variables in the iterated part of the for statement. The Left Variable Table is transmitted (on the SYSUT3 data set) to the Subscript Handling Phase. It is used in identifying subscript expressions listed in the Subscript Table which are not optimizable (see preceding item).

6. To generate a transformed source text (called Modification Level 2). The principal change made in this version of the source text consists in the replacement of externally represented identifiers and constants by five-byte internal names (see items 1 and 2). Other changes are set forth under "Modification Level 2 Source Text".

7. To replace the external names of standard mathematical functions and input/output procedures by five-byte internal designators. The internal designators are stored in the Identifier Table work area by the Initialization routine, before the first record of the Identifier Table is read into main storage from the SYSUT3 data set.

8. To recognize syntactical errors in the source text and to store appropriate error patterns in the Error Pool. The contents of the Error Pool are printed out in the form of diagnostic messages by the Error Message Editing Routine in the immediately following Diagnostic Output Module (IEX31).

9. To generate TXT records of the Constant Pool on the SYSLIN and SYSPUNCH data sets, if the LOAD and/or DECK options have been specified.

SCAN III PHASE OPERATIONS

The primary functions of the Scan III Phase are:

1. To replace externally represented operands in the Modification Level 1 text by their corresponding internal names in the Identifier Table;

2. To store constants found in the Modification Level 1 text in the Constant Pool and to replace the constants by internal names; and

3. To detect critical logical features of all for statements and record these in the For Statement Table. A closely related function is to list integer left variables and linear subscripts of arrays in for statements, in the Left Variable and Subscript Tables.

The diagram in Figure 39 illustrates the main operations performed in the Scan III Phase (the overall logic of the phase is indicated in Flowcharts 044 and 045 in the Flowchart Section). The following description provides a brief comment on the diagram.

The modified source text is scanned, in the first instance, by the GENTEST routine, which branches to approximately 30 other routines, according to the character identified in the source text. Control is in every case returned to GENTEST after the required processing has been completed.

Modification Level 1 text records are read from the SYSUT1 data set by the ICHA subroutine, which is called by all routines on detection of the record-end operator Zeta. The Modification Level 2 text records are output on SYSUT2 by the OUCHA subroutine on call from all routines which transfer operators and internal names to the modified text.

OPENING AND CLOSE OF BLOCKS AND PROCEDURES

At the opening of a block or procedure (indicated by the operators Beta, Pi or Phi), the next sequential Identifier Table record is read into the work area provided. Records are read from the SYSUT3 data set by the ITABMOVE subroutine, on call from BETA or PIPHI. When the end of the current (embracing) block or procedure is reached (indicated by the operator Epsilon), the corresponding Identifier Table record in the work area is deleted by EPSILON. This procedure insures that the Identifier Table work area at all times contains those identifiers which have been validly declared or specified in the current block or procedure, as well as in all enclosing blocks or procedures. The handling of the Identifier Table is described more specifically in a later section.

IDENTIFIER HANDLING

A letter indicates the beginning of an externally represented identifier. The LETTER routine scans the following characters, and when the end of the identifier has been found, branches to IDENT. IDENT initiates a comparison (between the identifier in the source text and the external names contained in the entries in the Identifier Table work area), designed to locate an entry for the same identifier declared or specified in the current (or an enclosing) block or procedure. If no matching identifier is found in the Identifier Table, the identifier in the source text is undefined: an error is recorded in the Error Pool, the Compiler enters Syntax Check Mode (Chapter 9), and, after an all-purpose internal name has been transferred to the Modification Level 2 text, control is returned to GENTEST. If, however, a matching identifier is found in the Identifier Table (indicating that the identifier was duly declared or specified) control is passed to FOLI, which branches to one of four routines (NOCRI, PROFU, SWILA, and CRITI), according to the character of the identifier, indicated by the characteristic in the Identifier Table entry.

The main function of the NOCRI, PROFU, SWILA, and CRITI routines is to determine if the identifier in the source text is contained in an embracing for statement (that is, in the for list or in the iterated part of an embracing for statement); and if so, to make entries in the Left Variable and/or Critical Identifier Tables; and to classify the embracing for statement(s) in the For Statement Table, according to whether the presence of the particular type of identifier in the for statement affects the logical structure of the code to be generated for the for statement(s) in the Compilation Phase. The processing of for statements is discussed in more specific detail in a later section. The SWILA routine, entered if the identifier is a label or a switch, serves to verify the validity of a branch.

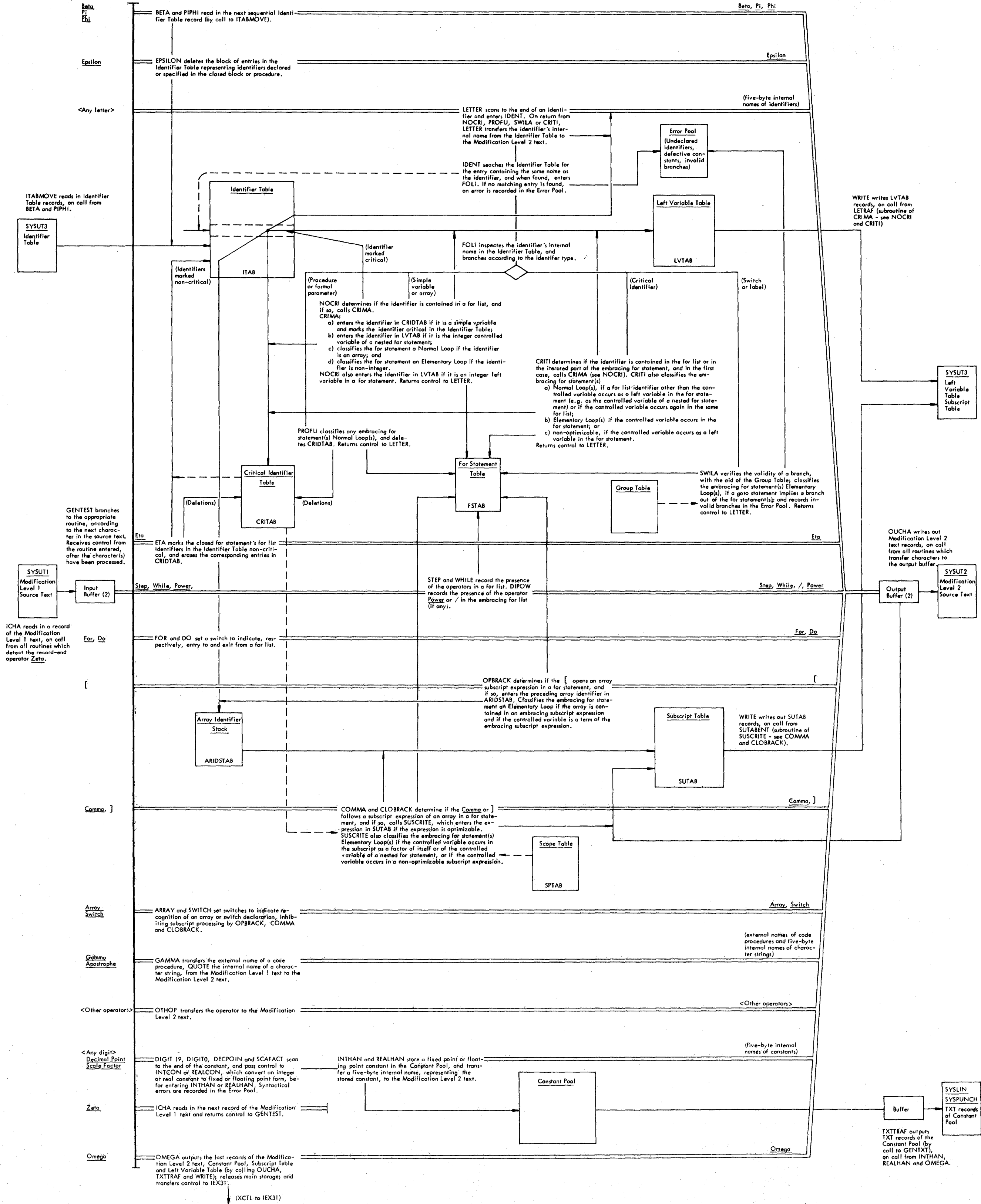


Figure 39. Scan III Phase.
Diagram illustrating the functions of the principal constituent routines

NOCRI, PROFU, SWILA, and CRITI all return control to LETTER (directly if the identifier is not contained in a for statement). LETTER thereafter transfers the identifier's five-byte internal name to the Modification Level 2 text, replacing the external name in the Modification Level 1 text, and returns control to GENTEST. The internal name is obtained from the Identifier Table entry previously located by IDENT. An overall survey of the identifier-handling routines can be found in the Flowchart Section with the aid of the Index of Routines in Appendix XI.

NUMBER HANDLING

Constants in the Modification Level 1 text are handled, in the first instance, by the DIGIT19, DIGIT0, DECPOIN, and SCAFACT routines. The function of these routines, in the case of real constants (e.g. 457.725 or 0.0095'86), is to represent the constant as the product of a mantissa (with the decimal point immediately to the left of the first significant digit) and a power of ten. Thus the constants in the illustrations above would be represented as 0.457725×10^3 and 0.95×10^{84}. When this transformation is complete, control is passed to the REALCON routine, the mantissa being transmitted in a storage location and the exponent in a register. REALCON converts the constant, represented by the mantissa and exponent, to floating point representation in a register. Thereafter, control is passed to the REALHAN routine, which stores the constant in the Constant Pool and transfers a five-byte internal name, referencing the constant's storage location, to the Modification Level 2 text.

Integer constants are handled by the DIGIT19 and/or DIGIT0 routines. When the last digit in the constant has been located, control is passed to the INTCON routine. INTCON converts the constant to fixed point notation in a register, and exits to INTHAN, which stores the constant in the Constant Pool and transfers a five-byte internal name to the Modification Level 2 text.

ARRAY SUBSCRIPT HANDLING

Subscript expressions, identified by the operators [, Comma and], are handled by the OPBRACK, COMMA, and CLOBRACK routines. If a subscript expression relates to an array in a for statement, an analysis of the subscript expression is initiated to determine if the subscript expression is optimizable, that is, if the expression is a linear expression satisfying certain constraints (defined in a later section). If the subscript expression is optimizable, the terms of the expression, together with their signs and a serial number identifying the for statement, are entered in the Subscript Table.

HANDLING OF OTHER OPERATORS

For a majority of the operators in the Modification Level 1 text, the processing is limited to the transfer of the operator to the Modification Level 2 text (by OTHOP). In the case of the operators For and Do, a switch is turned on to indicate, respectively, entry to and exit from a for list, while the appearance of the operators Step, While, Power, or / in a for list is recorded in the appropriate entry of the For Statement Table. The Apostrophe operator indicates that the internal name of a character string or a logical value follows, and in this case the internal name alone is simply transferred to the Modification Level 2 text.

PHASE TERMINATION

The Scan III Phase is terminated on recognition of the closing operator Omega. The OMEGA routine writes out the last records of the Subscript and Left Variable Tables, releases main storage, and transfers control (XCTL) to Diagnostic Output Module IEX31 (Chapter 9).

PHASE INPUT/OUTPUT

Figure 40 pictures the data input to and output from the Scan III Phase. The figure also indicates the tables transmitted via main storage.

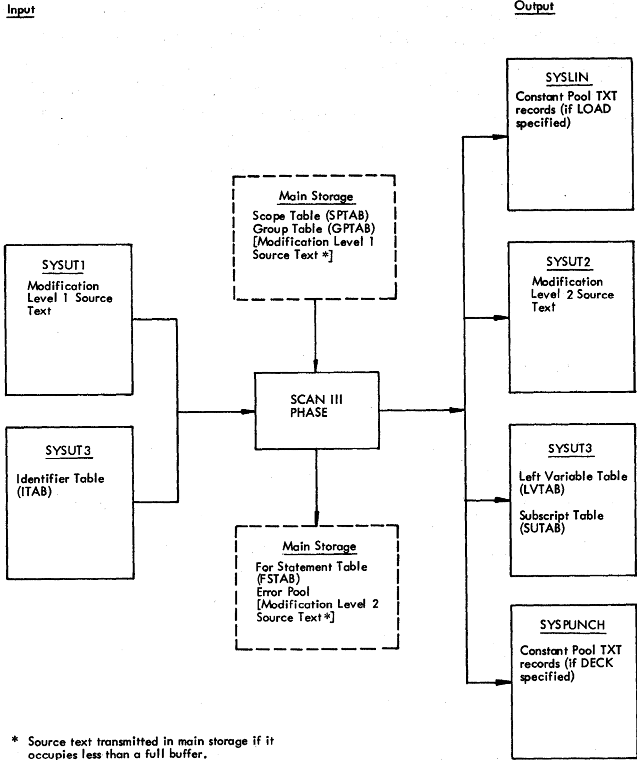


Figure 40. Scan III Phase input/output

The Modification Level 1 source text is input from the SYSUT1 data set, unless the text occupies less than a full buffer. In the latter case the modified source text will have been transmitted from the Scan I/II Phase in main storage. Similarly, the Modification Level 2 source text is output on SYSUT2 or transmitted via main storage, depending on whether the text occupies more than or less than a full buffer.

Input of the Identifier Table (ITAB) proceeds in parallel with output of the Subscript Table (SUTAB) and the Left Variable Table (LVTAB) on the same data set (SYSUT3). ITAB input is handled by the ITABMOVE subroutine, while SUTAB and LVTAB output is handled by the WRITE subroutine. SUTAB and LVTAB records (fixed length = buffer size) are output in random order, accordingly as the respective buffer is filled, starting at the SYSUT3 data set address immediately following the last ITAB record output by the Scan I/II Phase. The data set address is saved at initialization and transmitted to the Subscript Handling Phase in readiness for input of the first SUTAB/LVTAB record. To enable the records to be differentiated in the Subscript Handling phase, each output record contains a leading four-byte key (SUTB in SUTAB records, LVTB in LVTAB records). Before every input and output operation on SYSUT3, a test is made in both the ITABMOVE and WRITE subroutines, to determine if the operation to be performed differs from the last operation (i.e. input of ITAB or output of SUTAB/LVTAB). If the operation to be performed is the same as the last performed operation, input/output is initiated directly from or to the current data set position. If, however, the operation to be performed differs from the last performed operation, the data set position of the last transferred record is saved (with the aid of a NOTE macro) in one of the pointers NOTER or NOTEW (see Figure 41); the data set is then repositioned (by a POINT macro) to the address previously saved in NOTER/NOTEW; and input/output is started at the data set address to which SYSUT3 is positioned.

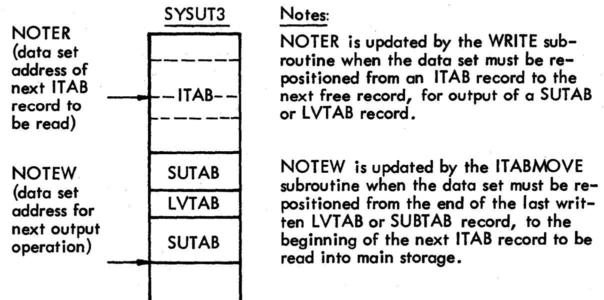


Figure 41. Function of pointers NOTER and NOTEW in input/output operations on the SYSUT3 data set

PROCESSING OF THE IDENTIFIER TABLE

A description of the entries in the Identifier Table (ITAB) is given in Chapter 4. See also Appendix II.

In the Scan III Phase, externally represented operands in the source text are replaced by their corresponding internal names constructed in the Identifier Table.

The processing of the Identifier Table is approximately as follows. A new ITAB record is read into a work area from the SYSUT3 data set, as soon as a new block or procedure is encountered in the Modification Level 1 source text. When the end of a block or procedure is reached, the corresponding record in main storage is erased. In this way, the work area at all times contains those identifiers which have been duly declared (whether in the current scope or in an enclosing scope) and which may validly occur as operands at any given point in the source module. Any operand in the modified source text not represented by an entry in the work area represents an undeclared identifier.

When an operand is recognized in the source text, it is compared with each of the ITAB entries in the work area, beginning with the last, until an entry is found which contains the same external name as the operand in the source text. The internal name in the ITAB entry is then transferred to the Modification Level 2 text in the output buffer, replacing the externally represented operand in the Modification Level 1 text. If no ITAB entry is found to match the operand in the source text, an all-purpose internal name is transferred to the output text, Error No. 81 is recorded, and the Compiler enters Syntax Check Mode (Chapter 9).

An ITAB record is read into the work area on recognition of any of the operators Beta, Pi or Phi, opening a block or a procedure. Input is handled by the ITABMOVE subroutine on call from the BETA and PIPHI routines. (Records are arranged on the data set in ascending Program Block Number sequence -- the same sequence as blocks and procedures are opened (and numbered) in the source module). The corresponding record is erased on detection of the closing operator Epsilon. The record is erased (by the EPSILON routine) by resetting a pointer (ZCURITEN) back to the end of the preceding record. Figure 42 illustrates the handling of the Identifier Table in relation to the block structure of a hypothetical source module.

CLASSIFICATION OF FOR STATEMENTS

Every for statement in the source module is assigned to one of three main loop classifications: Normal Loops, Elementary Loops, and Counting Loops. The loop classification, which is recorded in the For Statement Table, specifies the logical structure of the code generated in the Compilation Phase for each for statement. The structure of the object code generated for each of the various loop types is illustrated in Chapter 8 (Figures 66-72).

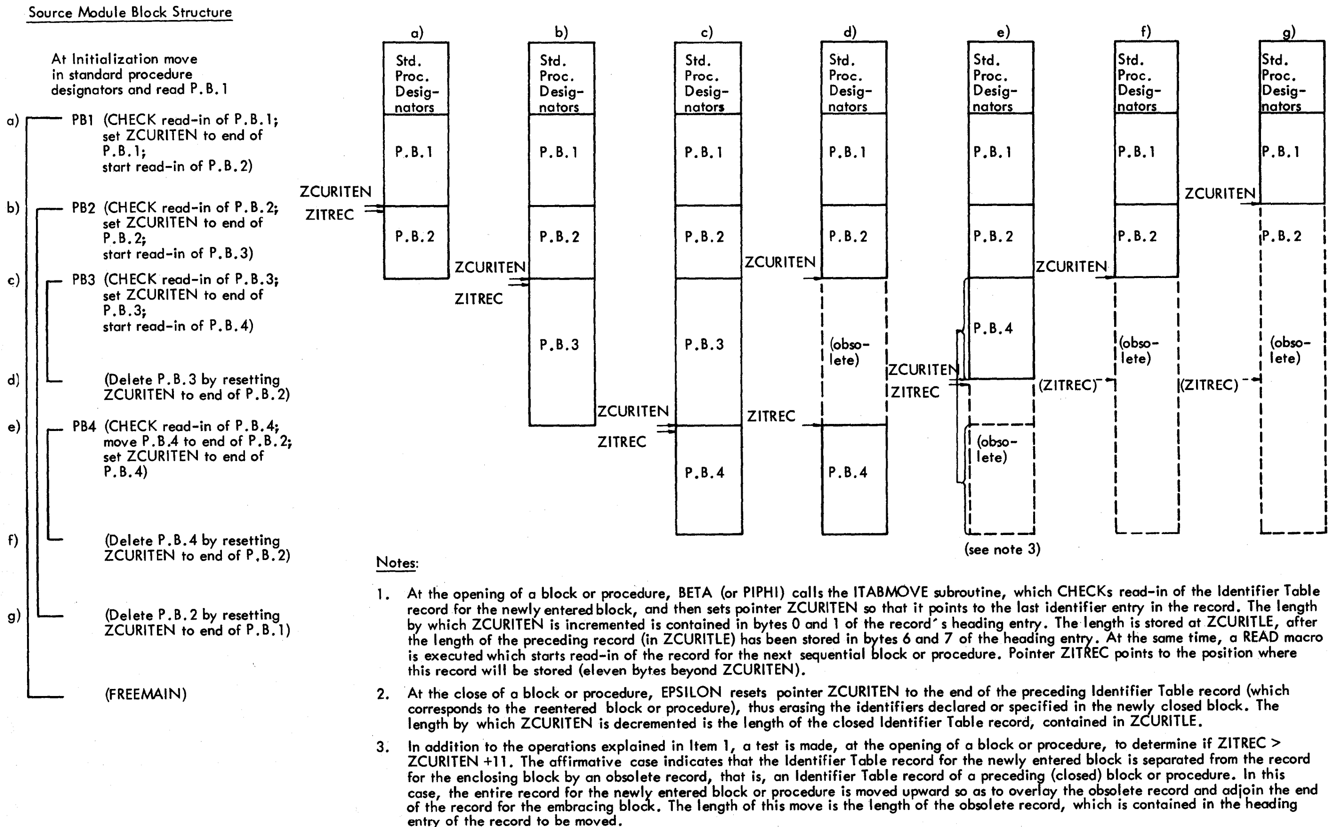


Figure 42. Diagram illustrating the handling of Identifier Table (ITAB) records in the ITAB Work Area

Normal Loop

A for statement is classified a Normal Loop if any one of the following conditions is detected:

1. The for statement contains a procedure identifier (other than a standard mathematical function or a standard output procedure).

2. An assignment is made in the for statement to any identifier in the for list ('FOR'.....'DO'), excepting the controlled variable.

3. An array element occurs in the for list.

4. The entries in the Critical Identifier Table (CRIDTAB), representing the identifiers in the for list, are deleted by reason of CRIDTAB overflow.

Elementary Loop

A for statement is classified an Elementary Loop if any one of the following conditions is detected:

1. A real operand occurs in the for list.

2. Any one of the operators While or Power occurs in the for list.

3. The controlled variable occurs in the for statement (other than in optimizable subscript expressions).

4. A goto statement implying a branch out of the for statement, is contained in the iterated statement.

Counting Loop

A for statement is classified a Counting Loop if it does not qualify as a Normal Loop or as an Elementary Loop.

PROCESSING OF FOR STATEMENTS

The processing of for statements in the Scan III Phase consists in the detection of the logical features listed above under the three loop classifications, and in registering these features in the for statement's classification byte in the For Statement Table.

Processing operations may be divided into (a) the detection of operators in for lists; (b) the recognition of identifiers in for statements (that is, in both the for list and in the iterated part of a for statement).

Detection of Operators in For List

The presence of any of the operators Step, While, or Power in a for list is detected by the routines STEP, WHILE, and DIPOW. Their presence is recorded by bit settings in the for statement's classification byte in the For Statement Table.

Recognition of Identifiers in For Statements

The recognition of the classes of identifiers in a for statement and their position in the for statement as the controlled variable and/or a left variable in the iterated statement, is handled by the routines entered from FOLI. See "Identifier Handling" and Flowchart 056.

FOLI's function is to determine the class of an identifier and to branch to a corresponding routine. The identifier class is determined by inspection of the Special Use Bits of the characteristic (Figure 9) in the Identifier Table entry corresponding to the externally represented identifier in the Modification Level 1 text. The routines entered from FOLI, according to the class of identifier recognized, are as follows:

NOCRI - a type-declared simple variable or an array

SWILA - a label or switch

PROFU - a procedure or formal parameter

CRITI - a critical identifier

An identifier is termed a critical identifier if it occurs in the for list of an embracing for statement, provided the identifier is a declared real, integer, or boolean simple variable. As soon as an identifier of this kind is encountered in a for list, the Special Use Bits of the characteristic in the corresponding Identifier Table entry, originally equal to binary 00, are set to binary 11, thus marking the identifier critical and facilitating the subsequent recognition of critical identifiers in the iterated part of the for statement. A critical identifier's Special Use Bits are reset to binary 00 only at exit from the for statement.

NOCRI determines, by reference to a switch byte set to various values on recognition of the operators <u>For</u> and <u>Io</u>, if an identifier occurs in a for list. If it does, and if the identifier is a type declared simple variable, NOCRI makes an entry for the identifier in the Critical Identifier Table (CRIDTAB), and then marks the identifier "critical" in the Identifier Table, in the manner explained above. These operations are performed by the CRIMA subroutine, which also classifies the enclosing for statement, according to the character of the particular identifier. NOCRI also makes an entry for the identifier in the Left Variable Table by call to LETRAF, if the identifier occurs as an integer left variable in the for statement.

SWILA, which is entered in the case of a switch or label, classifies the enclosing for statement(s), if any, if a jump is detected out of the for statement(s).

PROFU, which is entered in the case of a procedure or formal parameter, classifies the embracing for statement(s), if any, as Normal Loops.

CRITI, which is entered in the case of a critical identifier, classifies the enclosing for statement(s) according to whether an assignment is made to identifiers in the for list, among other things.

A for statement may be reclassified if the controlled variable occurs inside a nonlinear subscript expression. This condition is detected by the subroutines (called by OPBRACK, COMMA, and CLOBRACK) which process array subscript expressions in for statements.

OPTIMIZABLE SUBSCRIPT EXPRESSIONS

A subscript expression of any array may be described as a formula which specifies a displacement (usually in terms of one or more variables). If a subscripted variable occurs in a for statement, and if the controlled variable occurs as a variable in one or more subscript expressions, each subscript expression will specify a different displacement for every value assigned to the controlled variable (and hence for every cycle of the for loop).

The optimization of a subscript expression in a for statement consists in the generation of object code which precalculates (a) the initial value of the subscript, and (b) a constant increment to be added in each cycle of the for loop. Arrays and subscript optimization are discussed in detail in Chapter 8.

Optimization is possible if the following conditions are satisfied:

1. The subscript expression is optimizable, i.e., of the form ±F*V±A, where the factor F is an integer variable or constant, V is the controlled variable in the for statement, and the addend A is an integer variable or constant. Optimizable subscript expressions are entered in the Subscript Table.

2. The for statement is a Counting Loop or an Elementary Loop in which no assignment is made in the iterated statement to the controlled variable.

From the foregoing, it is apparent that subscript optimization is performed only in the case of Counting Loops and Elementary Loops.

A for statement may or may not contain an array identifier. If an array does not occur in a for statement, subscript optimization does not come into question. If, however, one or more arrays occur in a for statement, subscript optimization may be possible, depending on the for statement's loop classification and on whether the subscript expression is of the type specified above.

Each for statement's classification byte in the For Statement Table, specifies if optimization is to be performed for those subscript expressions in the for statement which are optimizable.

Subscript expressions are processed by the OPBRACK, COMMA, and CLOBRACK routines.

FOR STATEMENT TABLE (FSTAB)

As transmitted (via main storage) to the Subscript Handling Phase (and thence to the Compilation Phase), the For Statement Table (FSTAB) contains a classification byte for each for statement in the source module. The classification byte indicates:

1. The for statement's loop classification.

2. The presence of the operators <u>Step</u> and <u>While</u> in the for list.

3. Whether or not optimization is to be performed for optimizable subscript expressions in the for statement.

The foregoing information is indicated by bit settings in each half of the classification byte, as follows:

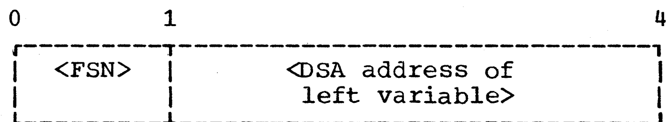
First Half (bits 0-3) :	X'F'	For statement is a Normal Loop
	X'8' X'2'	For statement is an Elementary Loop
	X'0'	For statement is a Counting Loop
	X'4'	Subscript optimization is not to be performed
Second Half (bits 4-7) :	X'8'	For list contains a step element
	X'4'	For list contains a while element (in this case, the for statement is classified an Elementary Loop).

A maximum of 255 bytes is provided for the For Statement Table in the Common Work Area.

The classification byte for a given for statement may be modified or referenced by several routines in the Scan III Phase, including WHILE, STEP, and DIPOW, as well as the routines entered from FOLI.

LEFT VARIABLE TABLE (LVTAB)

As transmitted to the Subscript Handling Phase, the Left Variable Table (LVTAB) contains an entry for every integer left variable occurring in the iterated part of for statements in which subscript optimization is possible. In the case of a series of nested for statements, the entries made for each for statement include all integer left variables in the enclosed for statement(s), including the controlled variable(s).



<FSN> = <Serial For Statement Number>

<DSA address> = <Last three bytes of internal name, containing Program Block Number and displacement in object time Data Storage Area>

Figure 43. Entry in Left Variable Table (LVTAB)

LVTAB is used in the Subscript Handling Phase to identify those subscript expressions listed in the Subscript Table which are optimizable.

Entries are made in LVTAB by the LETRAF subroutine. LVTAB is output to the SYSUT3 data set by the WRITE subroutine, on call from NOCRI or CRIMA.

SUBSCRIPT TABLE (SUTAB)

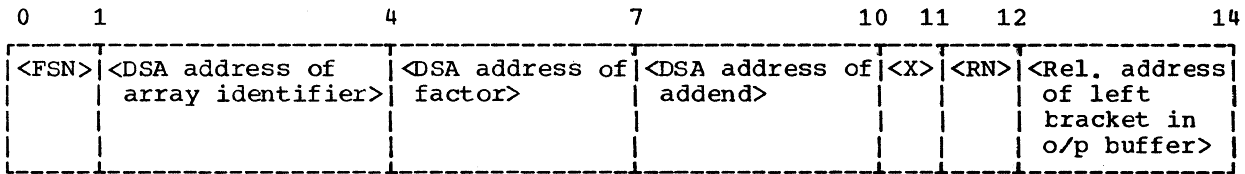
As transmitted to the Subscript Handling Phase, the Subscript Table (SUTAB) contains an entry for every optimizable subscript expression found in the iterated part of a for statement.

Entries are made in SUTAB by the SUTABENT subroutine on call from SUSCRITE (which is called in turn by the COMMA and CLOBRACK routines).

SUTAB is output on the SYSUT3 data set (in parallel with output of LVTAB and input of ITAB) by the WRITE subroutine.

CRITICAL IDENTIFIER TABLE (CRIDTAB)

The Critical Identifier Table provides a temporary record of the critical identifiers in the embracing for statement(s), that is, the nonarray identifiers found in the for list(s) of the embracing for statement(s). It is used primarily in determining if an identifier in the iterated part of a for statement also occurs as the controlled variable in the for list. It also provides a means of identifying the for statement, in whose for list a critical identifier occurs. The latter function assumes importance in the case of a series of nested for statements, where an assignment is made to a critical identifier which occurs in the for list(s) of one or more enclosing for statements. This condition affects the logic of the enclosing for statement(s), and must be reflected in the For Statement Table.



0	1	4	7	10	11	12	14
<FSN>	<DSA address of array identifier>	<DSA address of factor>	<DSA address of addend>	<X>	<RN>	<Rel. address of left bracket in o/p buffer>	

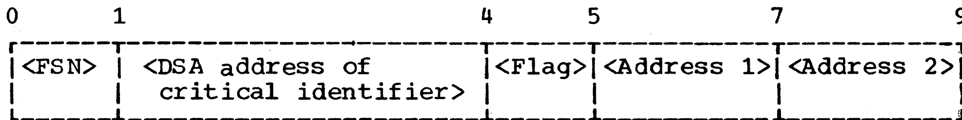
<FSN> = <For Statement Number>

<DSA address> = <Last three bytes of identifier's internal name containing Program Block Number and displacement>

<X>Bit 0 = <Sign of factor: 0 = +, 1 = ->
1 = <Sign of addend: 0 = +, 1 = ->
2 =
3 = (used only in Subscript Handling Phase)
4-7 = <Positional number of subscript>

<RN> = <Output record number in which the left bracket following the array identifier was put out>

Figure 44. Fourteen-byte Subscript Table entry for an optimizable array subscript expression in a for statement



0	1	4	5	7	9
<FSN>	<DSA address of critical identifier>	<Flag>	<Address 1>	<Address 2>	

<FSN> = <For Statement Number>
<DSA address> = <Last three bytes of critical identifier's internal name>
<Flag>: Bit 0 on = Identifier is controlled variable
Bit 1 on = CRIDTAB contains a preceding entry for same identifier
Bit 1 off = This entry is the first or only entry for the identifier
Bit 2 on = CRIDTAB contains a succeeding entry for the same identifier
Bit 2 off = This entry is the last (or only) entry for the identifier

<Address 1> =
First or only entry:
<Relative address of critical identifier's ITAB entry>
Second or subsequent entry:
<Relative address of preceding CRIDTAB entry for same identifier>

<Address 2> =
Any entry except the last:
<Relative address of succeeding CRIDTAB entry for same identifier>
Last or only entry:
(Not used)

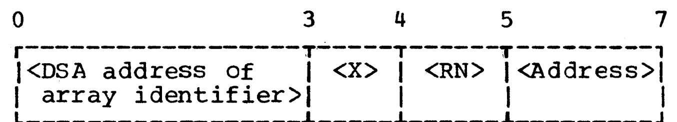
Figure 45. Entry in Critical Identifier Table (CRIDTAB)

An entry is made in CRIDTAB by the CRIMA subroutine as soon as it is determined that an identifier is contained in a for list. At exit from a for statement, all entries for identifiers in the for list are deleted. If the same identifier occurs in the for lists of a series of nested for statements, each entry for that identifier is flagged to show that there is a preceding and/or succeeding entry for the same identifier. If a for statement is classified a Normal Loop, all CRIDTAB entries for identifiers in the for list are deleted (by the DELCRIV subroutine). In the event of CRIDTAB overflow, the entries for the outermost for statement are deleted (by the CRIFLOW subroutine).

As indicated above, CRIDTAB lists the identifiers in the for list(s) of the embracing for statement(s), each entry indicating, first, if the identifier is the controlled variable, and second, if the identifier occurs in any other embracing for statement(s). As soon as it is detected (by NOCRI) that an identifier occurs in a for list, the Special Use Bits (see Figure 9) in the corresponding Identifier Table (ITAB) entry for the identifier are set to binary 11, to indicate that the identifier is a critical identifier, and an entry is made for the identifier in CRIDTAB. The Special Use Bits remain set to binary 11 until exit from the for statement, or until the for statement is classified a Normal Loop, at which time they are reset to their original value by the CRIFODEL routine. When an operand is encountered in the iterated statement (or in the same for list) whose corresponding ITAB entry shows that the identifier is a critical identifier, control is passed by FOLI to the CRITI routine. CRITI locates the corresponding entry in CRIDTAB, and then proceeds to modify the classification byte (in FSTAB) of the for statement(s) corresponding to each entry for the identifier, according to the particular circumstances surrounding the identifier in the iterated statement and in the for list. These may show, for example, that the identifier occurs as the controlled variable or as some other variable in the for list; that the identifier appears to the left of, or to the right of, an assignment operator in the iterated statement; or that the identifier appears only in a subscript expression. Depending on the circumstances identified, the corresponding for statement's classification byte may be modified to change the loop classification, or to specify that subscript optimization is or is not possible.

ARRAY IDENTIFIER STACK (ARIDSTAB)

An entry is made in the Array Identifier Stack for an array identifier in a for statement when the opening bracket following the identifier is encountered. The entry is deleted when the bracket which closes the array list is found. In a series of nested arrays (as, for example: (ARRAY1[K, ARRAY2[L, ARRAY3[M,N]]]), an entry is made for each array, as soon as the opening bracket for the particular array is recognized. The stack entries are released as the relevant closing bracket is identified, the last entry for the innermost nested array being released first, the entry for the embracing array being released second, and so on.



<DSA Address> = <Last three bytes of array identifier's internal name>

<X> = <Positional number of subscript component in which the array occurs> (set to X'00' if the array does not occur in an embracing array list)

<RN> = <Output record number>

<Address> = <Relative address of opening bracket in the output record>

Figure 46. Entry for an array identifier in the Array Identifier Stack (ARIDSTAB)

The Array Identifier Stack provides temporary storage for information concerning an array in a for statement. The recorded information may subsequently be transferred to one or more entries in the Subscript Table, depending on whether the subscript expression(s) in the array list are optimizable. A subscript expression containing an array is not optimizable, but the subscript expressions of the nested array may be optimizable.

MODIFICATION LEVEL 2 SOURCE TEXT

The Scan III Phase generates a second transformation of the source text, called Modification Level 2 (the first transformation being that of Modification Level 1, produced by the Scan I/II Phase). The Modification Level 2 text, which forms the

primary input to the Compilation Phase, is transferred to the SYSUT2 data set, unless it occupies less than a full buffer, in which case it is transmitted via Source Text Buffer 1. The principal changes reflected in the Modification Level 2 source text, as compared to Modification Level 1, are as follows:

1. All externally represented operand identifiers are replaced by five-byte internal names (Appendix II). The internal names are obtained from the Identifier Table. Undeclared identifiers are replaced by an all-purpose internal name.

2. Constants are replaced by five-byte internal names specifying the type of constant (real or integer) and the field in the Constant Pool where the constant is stored, and the Constant Pool Number. Defective constants are replaced by an all-purpose internal name (Appendix II).

3. The two-byte Identifier Group Number following the operators which mark the opening and closing of blocks, procedures, and for statements in the Modification Level 1 text (see "Scope Identification" in Chapter 4) is eliminated in the Modification Level 2 text.

4. The Apostrophe preceding the internal names of character strings and boolean constants is removed, but the internal names are transferred to the Modification Level 2 text unchanged.

5. The operator Rho (inserted by the Scan I/II Phase at the beginning of a record when a parameter delimiter extends across a buffer boundary) is removed, together with the preceding letter string. The right parenthesis at the beginning of the letter string is replaced by a Comma.

SWITCHES

The following switches are used in the routines of the Scan III Phase.

ZFORTEST

X'00' (set to X'00' on detection of operator Do) indicates that the source text currently being processed is not part of a for list.

X'C0' (set to X'C0' on recognition of operator For) signifies that a for statement has been entered.

X'80' (set to X'80' by CRIMA after the controlled variable in a for statement has been recognized) signifies that a for list following the controlled variable is being processed.

IOBYTE (Bits 0 thru 3 are named as follows. They are tested and turned on or off in the ITABMOVE, WRITE, and CHECK subroutines).

READM=1 signifies that the last SYSUT3 operation was a READ.

WRITEM=1 signifies that the last SYSUT3 operation was a WRITE.

READC=1 signifies that the last SYSUT3 operation was a CHECK following a READ.

WRITEC=1 signifies that the last SYSUT3 operation was a CHECK following a WRITE.

SCATEST (Bits 0 thru 4 are named as follows. They are tested and turned on or off in the DIGIT19, DIGIT0, DECPCIN, SCAFACT, and REALCON routines).

SFSIGN=1 signifies that the Scale Factor is followed by a +/- sign.

SFL0=1 signifies that a scale factor exponent contains one or more leading zeros.

SF19=1 signifies that a significant digit has been encountered in a scale factor exponent.

SF=1 signifies that a scale factor has been encountered as part of a real number.

PRECERR=1 signifies that the precision of a real constant exceeds the machine capacity.

STATUS (Bits 0 and 4 are named as follows. They are tested in the OPBRACK, COMMA, and CLOBRACK routines.)

SARRAY=1 (turned on by ARRAY on detection of the operator Array) signifies that an array declaration is being processed.

SSWITCH=1 (turned on by SWITCH on detection of the operator Switch) signifies that a switch declaration is being processed. SARRAY and SSWITCH are both turned off by SEMIDELT at the close of the declaration.

ZCLOBRA

X'00' (set to X'00' by COMMA and

OPBRACK on detection of the operators [or Comma in a subscript expression) signifies that SUSCRITE is to be called.

X'FF' (set to X'FF' by CLOBRACK on detection of an array element in a subscript expression) signifies that the rest of the expression cannot be optimized and specifies that SUSCRITE is not to be called.

ZLVOV

X'00' The maximum capacity of SUTAB or LVTAB has not yet been reached.

X'FF' (set to X'FF' in LETRAF and SUTABENT) signifies that the maximum capacity of LVTAB or SUTAB has been reached, and specifies to SUSCRITE and LETRAF that no more entries are to be made in these tables.

CONSTITUENT ROUTINES OF SCAN III PHASE

The principal constituent routines of the Scan III Phase are described below. The Index of Routines in Appendix XI provides a guide to the flowchart in the Flowchart Section and to the text in which each routine is outlined.

PHASE INITIALIZATION (INITIATE)

The Initialization routine acquires main storage for the private work area shown in Figure 47; initializes pointers; issues a SPIE macro to take care of exponent overflow and underflow interrupts; stores a set of 28 standard procedure designators in the Identifier Table (ITAB) work area; reads in the first Modification Level 1 text record from the SYSUT1 data set; and exits to the GENTEST routine.

TERM1 is the address of the routine entered in the event of a program interrupt or input/output error. It is stored at ERET, the location referenced by the Program Interrupt routine (PIROUT) and the I/O Error routines (SYNAD and SYNPR) in the Directory. TERM1 is changed to TERM2 after the GETMAIN macro has been issued.

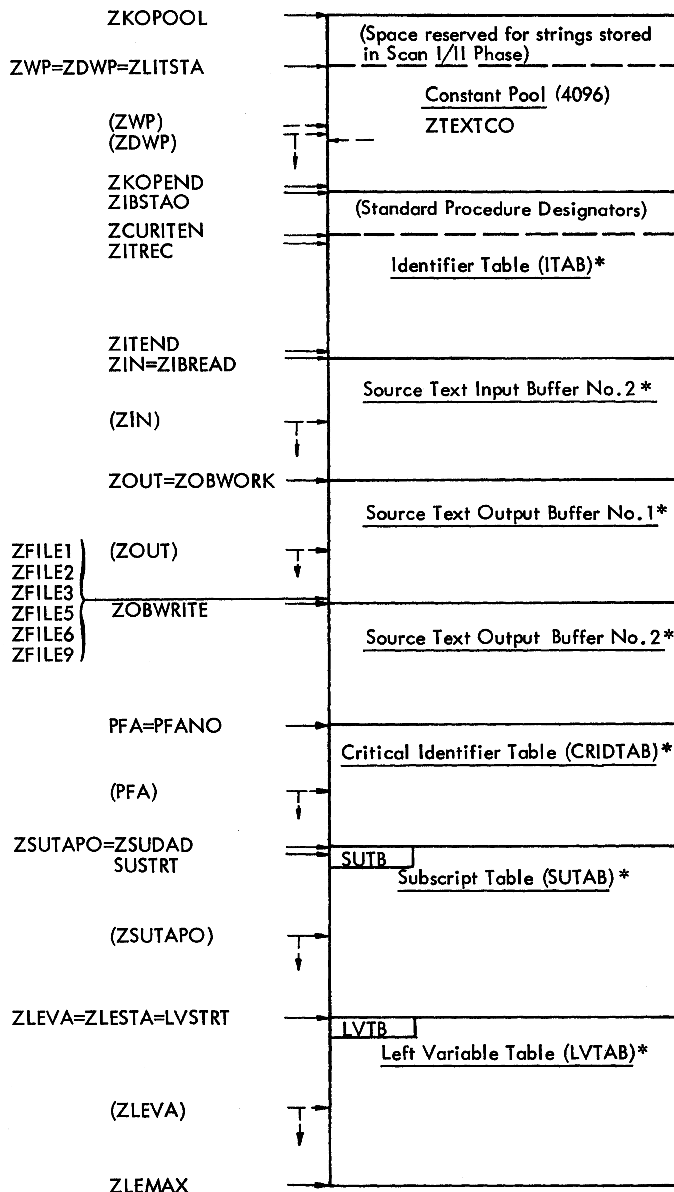
A SPIE macro is then issued to provide for special handling of interrupts due to exponent overflow or underflow. By execution of this macro, the Directory routine PIROUT (the routine specified in the SPIE macro executed in the Initialization Phase) is replaced as the program interrupt exit, by a routine named INTERUPT. INTERUPT determines the type of interrupt involved, and if it is any interrupt other than an exponent overflow, passes control to PIROUT, which then passes control to the entry address TERM2 stored at ERET (see preceding paragraph). If, however, the interrupt is due to exponent overflow, INTERUPT records Error No. 82, disregards the constant in the Modification Level 1 source text, transfers an all-purpose internal name to the Modification Level 2 text, and passes control to the GENTEST routine.

The GETMAIN instruction is issued after the area sizes for all work areas have been totalled. The area sizes are obtained from the Area Size Table in the Common Work Area.

After initializing a pointer to Source Text Input Buffer No. 1 in the Common Area, a call is made to the Change Input Buffer subroutine (ICHA), provided the ONEREC switch in the HCOMPMOD Control Field shows that the source text has not been transmitted from the Scan I/II Phase in main storage. ICHA reads in the first Modification Level 1 text record.

A set of 28 eleven-byte entries containing the external and internal names of all ALGOL standard I/O procedures and mathematical functions, is moved into the Identifier Table area from a table named FIXITAB. Appendix III lists the internal names of standard I/O procedures and mathematical functions.

After fetching (by means of a NOTE macro) the data set address of the last Identifier Table (ITAB) record on SYSUT3, and saving the address in NOTEW and SULTSTRT, the data set is repositioned to the first ITAB record, and a call is made to the ITABMOVE subroutine, which reads in the first record. The data set address in NOTEW is referenced by the WRITE subroutine in the present phase; SULTSTRT is referenced in the Subscript Handling Phase, when SUTAB and LVTAB are read into main storage.



* Area size specified by Area Size Table in Common Work Area. See Appendix VIII for the variation in area sizes as a function of the SIZE option.

Notes:

1. Source Text Input Buffer No. 1 is located in the Common Area acquired by the Initialization Phase. Its address is obtained from the Common Work Area location SRCE1ADD and stored at ZIBRUN. In the ICHA subroutine, input buffers are exchanged by exchanging the contents of ZIBRUN and ZIBREAD and setting ZINR=ZIBRUN, where the latter point to the next record to be processed, while ZIBREAD points to the alternate buffer into which read-in of the following record has been started. The end of a record is identified by the operator Zeta.
2. In the OUCHA subroutine, source text output buffers are exchanged by exchanging the contents of ZOBWORK and ZOBWRITE and setting ZOUT= ZOBWORK, where the latter point to the vacant buffer to be filled next, while ZOBWRITE points to the alternate buffer from which output of a record has been started. Pointers ZFILE1, ZFILE2, ..., ZFILE9 point to the end of the current buffer, less 1, 2,....., 9 bytes.
3. In the Constant Pool, ZLITSTA points to the next free entry, allowing for strings stored by the Scan I/II Phase. The displacement by ZLITSTA from the start of the pool is obtained from the displacement pointer PRPOINT, transmitted from Scan I/II in the Common Work Area.

   ZWP and ZDWP point to the next free entry at a word or double word boundary.

   ZTEXTCO is set equal to ZWP + 56 after a TXT record has been output. A TXT record is output as soon as ZWP>ZTEXTCO.
4. In the Identifier Table, ZCURITEN points to the end of the set of ITAB records representing identifiers declared or specified in the embracing blocks and procedures. ZITREC points to the next entry.
5. A four-byte key is stored at the start of SUTAB and LVTAB records. The key permits the Subscript Handling Phase to identify each record read from SYSUT3.

Figure 47. Private Area acquired by Scan III Phase

GENERAL TEST (GENTEST)

GENTEST scans the Modification Level 1 text in the current input buffer by means of a Translate and Test instruction, and branches to one of 26 routines, according to the function byte assigned the particular character in Translation Table GENER. The function bytes assigned by GENER to the character set and the routines entered from GENTEST are as follows:

Character	Function Byte	Routine Entered
<Any letter>	04	LETTER
<Any digit 1 - 9>	08	DIGIT19
<Digit 0>	0C	DIGIT0
Decimal Point	10	DECPOIN
Scale Factor	14	SCAFACT
Apostrophe	18	QUOTE
Beta	1C	BETA
Pi, Phi	20	PIPHI
For	24	FOR
Epsilon	28	EPSILON
Eta	2C	ETA
Do	30	DO
While	34	WHILE
Semicolon, Delta	38	SEMIDELT
[	3C	OPBRACK
Comma	40	COMMA
]	44	CLOBRACK
Zeta	48	ZETA
Gamma	4C	GAMMA
Omega	50	OMEGA
+, -, *, ÷ , (,), <, >, ≤, ≥, =, ≠, Assign, Not, Impl, Or, And, Equiv, Label Colon, Begin, Goto, Until, If, Then, Else, End, Power	54	OTHOP
Rho	58	RHO
Step	5C	STEP
Array	60	ARRAY
Switch	64	SWITCH
Power,/	68	DIPOW

IDENTIFIER TEST (LETTER)

See "Identifier Handling" in this chapter.

LETTER, which is entered from GENTEST on recognition of a letter, scans the source text to the next nonletter, nondigit character, and branches to one of four routines, according to the function byte assigned by Translation Table IDENTI. The function bytes assigned to the character set and the routines entered from LETTER are as follows:

Character	Function Byte	Routine Entered
<Any letter or digit>	00	(No branch scanning continues)
Zeta	04	ZETALET
Rho	08	RHO
Decimal Point	0c	ERROR1
Scale Factor, Apostrophe <All other operators>	10	IDENT

Entry to IDENT signifies that an identifier in a valid context has been encountered. IDENT searches the Identifier Table for the corresponding entry, and when the entry is found, passes control to FOLI (see below). Control is subsequently returned to LETTER1, which transfers the internal name of the identifier to the Modification Level 2 text, replacing the external name in the Modification Level 1 text.

ZETALET exchanges source text buffers, by call to the ICHA subroutine, and returns control to LETTER.

ERROR1 branches to INCOROP, which records Error No. 80, transfers an all-purpose internal name to the output text and switches the Compiler to Syntax Check Mode before returning control to GENTEST.

ITAB SEARCH (IDENT)

IDENT moves up to six characters of the identifier in the source text to a field named ZIDEX, and then compares the identifier with the external names listed in the Identifier Table. When a matching entry is found, control is passed to the Identifier Classification Routine (FOLI). If an identical external name is not found, the identifier in the source text is undeclared: Error No. 81 is recorded, Syntax Check Mode is entered, and an all-purpose internal name at ZALLPU is addressed. Control is then returned to LETTER (via LETTER1) which transfers the internal name to the output buffer, and returns control to GENTEST.

IDENTIFIER CLASSIFICATION (FOLI)

FOLI inspects the internal name located by IDENT in the Identifier Table, corresponding to the identifier in the source text, and branches to one of four routines, according to the class of identifier designated by the Special Use Bits in the characteristic (Figure 9):

Special Use Bits	Identifier Class	Program Entered
00	Declared simple variable or array identifier	NOCRI
01	Procedure identifier or formal parameter	PROFU
10	Label or switch identifier	SWILA
11	Critical Identifier (the identifier occurs in the for list of the embracing for statement)	CRITI

Only the first three classes are represented in the Identifier Table as constructed in the Scan I/II and Identifier Table Manipulation Phases. A declared real, integer, or boolean simple variable is classed a critical identifier (the fourth class) as soon as it is detected in a for list. In practical terms, this means that the Special Use Bits of the corresponding entry in the Identifier Table are set to binary 11 and that an entry for the identifier is made in the Critical Identifier Table. At exit from the for statement, all identifiers in the for list are restored to the noncritical class by resetting the Special Use Bits of the corresponding Identifier Table entries to 00, and the entries in the Critical Identifier Table for those identifiers are deleted.

NONCRITICAL IDENTIFIER (NOCRI)

NOCRI determines whether the identifier encountered in the source text occurs in a for list, in the iterated part of a for statement, or outside a for statement. If the identifier occurs in a for list (indicated by ZFORTEST≠ X'00'), a call is made to the CRIMA subroutine, which sets the Special Use Bits of the corresponding Identifier Table entry to 11, classifying it a critical identifier, and makes an entry for the identifier in the Critical Identifier Table. If the identifier occurs in the iterated part of a for statement and if it occurs as an integer left variable, a call is made to the LETRAF subroutine, which makes an entry in the Left Variable Table. If the identifier occurs outside a for statement, control is returned to the LETTER routine, which transfers the internal name in the corresponding Identifier Table entry to the output text, replacing the external name in the Modification Level 1 text.

PROCEDURE/PARAMETER (PROFU)

PROFU is entered when the FOLI routine has detected a procedure identifier or a formal parameter. PROFU erases all entries (if any) in the Array Identifier Stack in the event the procedure identifier or formal parameter occurs in an array list; and, if the procedure or formal parameter occurs in a for statement (indicated by entries in the Critical Identifier Table), calls the DELCRIV subroutine, which classifies the embracing for statement(s) Normal Loops and erases CRIDTAB. Control is returned to the LETTER1 routine, which transfers the internal name of the identifier to the output text.

SWITCH/LABEL (SWILA)

SWILA is entered when the FOLI routine has identified a switch or label identifier. The function of SWILA is to determine whether the label or switch implies a branch out of the current scope. If so, it is determined (by reference to the Group Table) if the branch is into a scope enclosing the current scope, or into a scope enclosed by the current scope. In the first case, the jump is valid, but if the current scope is a for statement, its optimization is affected, and the corresponding for statement's classification byte is modified.

In the second case, the branch is invalid, as it is into a for statement, and an error is recorded. In every case, control is returned to the LETTER1 routine.

CRITICAL IDENTIFIER (CRITI)

CRITI is entered when the FOLI routine has identified an operand that is a critical identifier, i.e. the source identifier occurs in the for list of an embracing for statement. (An entry for the identifier will have been previously made in the Critical Identifier Table). The action taken depends on whether the operand occurs in a for list or not (indicated by the switch ZFORTEST).

Identifier in For List: A branch is made to the CRIMA subroutine which constructs an entry for the identifier in the Critical Identier Table.

CRITI then searches the Critical Identifier Table for the previous entry for the

identifier. When the earlier entry has been found, the two entries are chained together, the flag byte in each entry being set so as to indicate respectively that the entry is preceded or followed by another entry for the same operand, and the relative address of the preceding or following entry being stored in byte 5 and 6 or byte 7 and 8 (see Figure 45).

Tests are now made of the appropriate bit in the flag-byte of the two entries to determine whether or not the identifier constitutes the controlled variable in the respective for statements. The action taken for the various alternatives is as follows:

If the identifier occurs as the controlled variable in the current for statement as well as an enclosing for statement, the classification byte in the For Statement Table for the enclosing for statement is modified to show that subscript optimization is not possible. If the identifier in the current for statement is the controlled variable, but in the enclosing for statement is not the controlled variable, the enclosing for statement is classified a Normal Loop.

If the identifier occurs twice in the current for list (once as the controlled variable), the current for statement is classified a Normal Loop.

A test is now made to determine if the preceding entry in the Critical Identifier Table is chained to another preceding entry. If it is, the for statement referenced by that entry is classified in the For Statement Table in the manner described above, according to the position of the identifier in the current for statement and in the enclosing for statement.

Identifier Not in For List: If the Array Identifier Stack contains any entries, indicating that the identifier occurs in a subscript expression, control is returned immediately to GENTEST. If, however, the Array Identifier Stack is empty (indicated by ZARSPO = ZARNO), the processing continues as follows:

A search is made in the Critical Identifier Table for the entry corresponding to the identifier in the source text (the search is made by comparing the contents of bytes 1, 2, and 3 in the Critical Identifier Table entries with the contents of bytes 8, 9, and 10 of the Identifier Table entry previously located by IDENT).

If the identifier in the source text is a left variable (followed by :=), and if the entry in the Critical Identifier Table indicates that the identifier occurs in the for list as the controlled variable, the classification byte of the corresponding for statement is modified to show that subscript optimization is not possible. If the identifier is a left variable and occurs in the for list as an operand other than the controlled variable, the for statement is classified a Normal Loop.

If the identifier in the source text is not a left variable but occurs in the for list as the controlled variable, the current for statement is classified an Elementary Loop. No entry is made in the For Statement Table if the identifier occurs in the for list as an operand other than the controlled variable.

After the action described above, control is returned to the LETTER routine, which transfers the identifier's internal name in the Identifier Table to the Modification Level 2 text.

MAKE CRIDTAB ENTRY (CRIMA)

CRIMA is entered when the NOCRI or CRITI routine has determined that an identifier occurs in a for list. Provided the identifier is not an array, CRIMA makes an entry in the Critical Identifier Table, indicating if the identifier is the controlled variable or not; classifies the for statement a Counting or Elementary Loop, depending on whether the identifier is an integer or not; and changes the Special Use Bits of the corresponding Identifier Table entry to mark the identifier a critical identifier. If the for statement is enclosed by another for statement, and the identifier is a controlled integer variable, a call is made to the LETRAF subroutine, which makes an entry for the identifier in the Left Variable Table.

If the identifier is an array, the for statement is classified a Normal Loop and a call is made to the CRIFODEL subroutine, which erases all entries in the Critical Identifier Table for identifiers in the for list of the for statement.

CRIDTAB OVERFLOW (CRIFLOW)

CRIFLOW is called by the CRIMA subroutine in the event of overflow of the Critical Identifier Table (CRIDTAB). CRIFLOW deletes all CRIDTAB entries for the outermost enclosing for statement and classifies that for statement a Normal Loop.

ERASE CRIDTAB (DELCRIV)

DELCRIV is called by the PROFU routine, when it is determined that a procedure or a formal parameter occurs in a for statement. DELCRIV erases all entries in the Critical Identifier Table, representing identifiers in the for list(s) of the embracing for statement(s); resets the Special Use Bits of the corresponding Identifier Table entries (to indicate they are no longer critical); and reclassifies the embracing for statement(s) Normal Loops.

UPDATE CRIDTAB (CRIFODEL)

CRIFODEL is called by the ETA routine and by the CRIMA subroutine when an array has been encountered in a for statement. CRIFODEL deletes the entries in the Criti- Identifier Table for identifiers in the for list.

MAKE LVTAB ENTRY (LETRAF)

LETRAF is called by NOCRI when a left variable is encountered in the iterated part of a for statement, and by CRIMA when a controlled variable is identified and it is determined that the current for statement is enclosed by another for statement. LETRAF makes one entry in the Left Variable Table (LVTAB) for the left variable, for every for statement which embraces the left variable. If the LVTAB work area is filled, the WRITE subroutine is called.

NONZERO DIGIT (DIGIT19)

See "Number Handling" in this chapter.

DIGIT19 is entered from GENTEST on detection of any nonzero digit in the Modification Level 1 text, and from DIGIT0 on recognition of a nonzero digit following a zero. DIGIT19 scans the source text to the next nondigit character, using Translation Table DIG19, and branches to one of six routines according to the assigned function byte. The function bytes assigned to the character set and the routines entered from DIGIT19 are as follows:

Character	Function Byte	Routine Entered
Decimal Point	04	DECPTM
Scale Factor	08	SCAFACTM
<Any Letter>	0C	QTORLT
Apostrophe		
Zeta	10	ZETAM
Rho	14	RHO
<Any other character>	18	OTHER
<Any digit>	00	(No branch-scanning continues)

Before scanning is initiated, registers are set to specify the address and length of a 19-byte field named NUMBER to which the digits of a constant are subsequently moved.

DECPTM is entered when the Decimal Point in a real constant (e.g. 640.325) is encountered. DECPTM moves the significant integer digits (640 in the example) to the 19-byte field named NUMBER, computes the number of digits moved in REXCORR (register 7), and branches to DECPOIN (see below). The digit count in REXCORR is treated as the exponent of a power-of-ten correction factor which must be applied to the constant, regarded as a mantissa, with the Decimal Point shifted to the left of the high order digit. Thus, the constant 640.325 is treated as the product .640325 x 10^3.

SCAFACTM is entered when the Scale Factor in a constant (e.g. 28'45) is encountered. SCAFACTM moves the significant digits preceding the Scale Factor to the 19-byte field named NUMBER, computes the number of digits moved (in REXCORR), and branches to SCAFACT (see below). The digit count in REXCORR is treated as the exponent of a power-of-ten correction factor which must be applied to the mantissa in which the implied decimal point has been shifted to the left of the high-order digit. The digit count is subsequently added (by SCAFACT) to the exponent following the Scale Factor.

QTORLT is entered if a constant contains an invalid character or if a constant occurs in an invalid context. QTORLT exits to INCOROP, which records Error No. 80, transfers an all-purpose internal name, and returns to GENTEST after switching to Syntax Check Mode. ZETAM exchanges input buffers (after moving the preceding digits to NUMBER and computing the digit count in REXCORR) and returns to DIGIT19, which then scans the remainder of the constant in the new buffer.

RHO returns control directly to GENTEST, disregarding the preceding digit(s). The operator Rho signifies, in this particular

context, that the preceding digit(s) form(s) part of an invalid identifier that is to be disregarded.

ZERO DIGIT (DIGIT0)

DIGIT0 is entered from GENTEST when a lone zero or a nonsignificant zero at the beginning of a constant is encountered in the Modification Level 1 text. DIGIT0 scans the source text to the next nonzero character, using Translation Table DIG0 and branches to one of seven routines, according to the assigned function byte. The function bytes assigned to the character set and the routines entered from DIGIT0 are as follows:

Character	Function Byte	Routine Entered
<Nonzero digit>	04	DIG191
<Any Letter> Apostrophe	08	QTORLT
Decimal Point	0C	DECPOIN1
Scale Factor	10	SCA0
Zeta	14	ZETA0
Rho	18	RHO
<Any other character>	1C	OTHOP0
<Zero Digit>	00	(No branch-scanning continues)

Before scanning is initiated, registers are set to specify the address and length of a 19-byte field named NUMBER to which the nonzero digits following the zero(s) are subsequently moved.

DIG191 (an entry point of the DIGIT19 routine) is entered when a nonzero digit is encountered in a constant (e.g.064) beginning with zero.

DECPOIN1 (an entry point of the DECPOIN routine) is entered when a Decimal Point is encountered in a real constant (e.g. 0.325).

SCA0 is entered when the Scale Factor is encountered immediately following a zero (e.g. 0'45). SCA0 loads the value zero (the equivalent value of a constant of this type) and exits to the SCAFACT routine.

OTHOP0 is entered when the integer zero is identified. OTHOP0 transfers a five-byte internal name, referencing a location in Constant Pool No. 0 where the constant zero is stored, to the Modification Level 2 text and returns control to GENTEST.

QTORLT and RHO perform the same functions as those described under the DIGIT19 routine. ZETA0 exchanges input buffers and returns control to DIGIT0.

DECIMAL POINT (DECPOIN)

DECPOIN is entered

1. From GENTEST on recognition of the Decimal Point at the begining of a real constant (e.g. .325).

2. From DIGIT0 on recognition of the Decimal Point following a zero (e.g. 0.325).

3. From DIGIT19 on recognition of the Decimal Point in a real constant (e.g. 640.325 or 1.325'45). In this case, the integer digits will have been moved, before entry to DECPOIN, to a 19-byte field named NUMBER, and register REXCORR will contain the exponent of a power-of-ten correction factor to be applied to the constant, regarded as a mantissa with the decimal point shifted to the left of the high order digit.

DECPOIN scans the source text to the next character other than a 1-9 digit, using Translation Table DECPO, and branches to one of five routines, according to the assigned function byte. The function bytes assigned to the character set and the routines entered are as follows:

Character	Function Byte	Routine Entered
<Digit Zero>	04	DECPO
<Any letter>, Decimal Point or Apostrophe	08	QTORLTP
Scale Factor	0C	DECPSCA
Zeta	10	DECPZETA
<Any other character>	14	DECPOT
<Any 1-9 Digit>	00	(No branch-scanning continues)

Before scanning is initiated, registers are set to specify the address and length of a 19-byte field named NUMBER, to which the digits following the Decimal Point are subsequently moved.

DECPO is entered when any zero following the Decimal Point is encountered. Provided the zero is not preceded by a significant digit (as in 0.0325), DECPO decrements REXCORR (register 7) for each zero following the Decimal Point, and returns to DECPOIN. If the zero is preceded by a significant digit (as in 6.0325 or 0.3025), REXCORR is not decremented. The resulting count, if any, in REXCORR is treated as the (negative) exponent of a power-of-ten correction factor to be applied to the constant, regarded as a mantissa with the decimal point shifted immediately to the

left of the first nonzero digit. Thus, for example, the constant 0.0325 is regarded as the product 0.325 * 10^{-1}

DECPSCA is entered when the Scale Factor is encountered in a real constant (e.g. 1.325'+45 or 0.0125'-45). DECPSCA moves the decimal digits preceding the Scale Factor to the 19-byte field NUMBER, adjoining the integer digits (if any) previously moved by DIGIT19, and passes control to SCAFACT. DECPOT is entered when the end of a decimal constant (e.g. 1.25) is identified. Provided the constant is not zero, DECPOT passes control to the REALCON routine. If the constant is equivalent to zero, register XFLOAT is loaded with the value zero, and control is passed to REALHAN. QTORLTP and DECPZETA perform essentially the same functions as QTORLT and ZETAM in the DIGIT19 routine.

SCALE FACTOR (SCAFACT)

SCAFACT is entered:

1. from GENTEST on recognition of the Scale Factor in a constant having no mantissa (e.g. '45);
2. from DIGIT0 on recognition of the Scale Factor following a zero mantissa (e.g., 0'45);
3. from DIGIT19 on recognition of the Scale Factor following an integer mantissa (e.g. 28'45); and
4. from DECPOIN on recognition of the Scale Factor following a decimal mantissa (e.g. 1.325'+45).

In both cases (3) and (4), the significant digits of the mantissa will have been moved, before entry to SCAFACT, to a 19-byte field named NUMBER, and register REXCORR will contain the exponent of a power-of-ten correction factor to be applied to the mantissa, in which the decimal point (expicit or implied) has been shifted to the left of the high-order nonzero digit.

SCAFACT scans the source text, using a Translate and Test instruction, and branches to a routine determined by the assigned function byte. The function bytes assigned to the character set and the routines entered are as follows:

Character	Function Byte	Routine Entered
<Any nonzero digit>	04	SCA19
<Zero>	08	SCAZERO
+ or -	0C	SCASIGN
<Any Letter>, Decimal Point, Scale Factor or Apostrophe	10	SCAQL
Zeta	14	SCAZETA
<Any other operator>	18	SCAOT

SCA19, SCAZERO, and SCASIGN each set a switch (named SF19, SFL0, and SFSIGN, respectively) to indicate the detection in the exponent following the Scale Factor of (a) a nonzero digit, (b) a leading zero, or (c) a leading +/- sign. These switches are inspected in SCASIGN to determine if a +/- sign marks the beginning or end of the exponent, and in SCAOT to determine if the exponent is syntactically correct. If a +/- sign precedes the exponent (indicated if none of the switches have been turned on), SCASIGN stores the sign for use when the exponent is converted to binary (in SCAOT). SCAOT in entered when the end of a floating point constant (e.g. 28'45 or 1.325'+45) has been identified. SCAOT moves the digits of the exponent following the Scale Factor to a nine-byte field named SCAWORK; converts the exponent(together with the previously saved exponent sign) to binary form, and adds the resultant to the exponent of the power-of-ten correction factor (if any) in REXCORR. This action has the effect of transforming a floating point constant to a standard format, consisting of a fractional mantissa, with the decimal point shifted to the left of the high-order nonzero digit, and an integer exponent, where the mantissa is stored at the 19-byte field NUMBER and the exponent is contained in REXCORR.

SCAQL and SCAZETA perform essentially the same functions as QTORLT and ZETAM in the DIGIT19 routine.

INTEGER CONVERSION (INTCON)

INTCON converts an integer constant to binary form (in register RBIN) and passes control to INTHAN, which stores the constant in the Constant Pool and transfers a five-byte internal name representing the constant, to the Modification Level 2 text. If a constant exceeds ten digits, Error No. 83 is recorded, the constant is moved to the 19-byte field NUMBER, and control is passed to REALCON.

INTCON is entered from DIGIT19, the limits of the constant in the source text being specified by two pointers, its length being contained in register REXCORR.

REAL CONVERSION (REALCON)

REALCON converts a real constant to floating point form (in floating point register XFLOAT) and passes control to REALHAN, which stores the constant in the Constant Pool and transfers a five-byte internal name respresenting the constant to the Modification Level 2 text.

REALCON is entered

1. From DECPOIN when the end of a decimal constant (e.g. 32.125) has been identified;
2. From SCAFACT when the end of a floating point constant (e.g. 1.25'47) has been identified; and
3. From INTCON when an integer constant exceeds ten decimal digits or the maximum range of a fixed point constant.

At entry to REALCON, the constant to be converted will have been transformed to the standard format of a decimal matissa, with the implied decimal point to the left of the high order digit, and an integer exponent (the constant being equal to: Mantissa *10**Exponent). The mantissa is stored in decimal form in a 19-byte field named NUMBER, while the exponent, in binary form, is contained in REXCORR (register 7).

The conversion to floating point form proceeds as follows:

1. The mantissa is tranformed to an integer mantissa, the implied decimal point being shifted to the right of the lowest order digit, by subtracting the number of digits in the mantissa from the exponent in REXCORR.
2. The mantissa is converted to binary and stored in the second four bytes of an eight-byte field named ZFLOFIEL containing the power-of-sixteen characteristic of 78 (X'4E') in the high-order byte. The contents of ZFLOFIEL are then loaded (normalized) in floating point register XFLOAT. The characteristic of 78 (equivalent in excess-64 notation to a power-of-sixteen exponent of 14) is the exponent required to compensate for an implied 14-place leftward shift of the radix point, i.e., from a position to the right of the low-order mantissa digit (see item 1). The characteristic is reduced, when the mantissa is normalized in XFLOAT, by the number of hexadecimal places the high order digit is shifted left.
3. The mantissa in XFLOAT is multiplied by the hexadecimal equivalent of the power-of-ten exponent in REXCORR. The hexadecimal equivalent is obtained from one of two power-of-ten tables named ZEXTABP and ZEXTABN, the former for positive powers of ten, the latter for negative powers of ten. Each table contains fifteen entries, the first seven for exponents in the range ±1 to ±7, the last eight for the exponents ±8, ±16, ±24, and so on up to ±64. Multiplication of the mantissa is carried out in one or more steps, depending on whether the particular power-of-ten is exactly represented in the table. Thus, for example, if the power-of-ten is 12 (decimal) the mantissa is multiplied first by the equivalent of 10^4 and secondly by the equivalent of 10^8.

INTEGER HANDLING (INTHAN)

INTHAN stores a fixed point constant (contained in register RBIN) in the next free entry of the Constant Pool, and transfers a five-byte internal name to the output text, referencing the location where the constant is stored. If the constant is in the range 0 to 15 inclusive, the internal name references the relevant constant previously stored in the Constant Pool in the Scan I/II Phase. INTHAN is entered from INTCON and from REALHAN if short precision is specified for real constants.

REAL HANDLING (REALHAN)

REALHAN stores a floating point constant (transmitted in floating point register XFLOAT) in the next free entry of the Constant Pool and transfers a five-byte internal name, referencing the location where the constant is stored, to the Modification Level 2 text. If short precision is specified, the constant in XFLOAT is transferred to fixed point register RBIN and the INTHAN routine is entered. After the constant has been stored in the Constant Pool, control is returned to GENTEST.

REALHAN is entered from REALCON.

CHANGE CONSTANT POOL (CPOLEX)

CPOLEX is called by INTHAN and REALHAN in the event the Constant Pool is filled. CPOLEX generates TXT records for the last

constants in the Pool, assigns a new Constant Pool number, and resets pointers to the beginning of the Pool area.

OUTPUT TXT RECORD (TXTTRAF)

TXTTRAF is called by INTHAN and REALHAN when the end of a 56-byte record in the Constant Pool has been reached. TXTTRAF updates Constant Pool pointers and calls the GENTXT subroutine, if the LOAD and/or DECK options are specified. GENTXT generates TXT records of the Constant Pool and outputs the records on SYSLIN and/or SYSPUNCH.

GENERATE (GENTXT)

See Chapter 8.

APOSTROPHE (QUOTE)

QUOTE transfers the five-byte internal name which follows the Apostrophe, to the output buffer. The internal name references a location in the Constant Pool where a string or logical value was stored in the Scan I/II Phase.

BLOCK BEGIN (BETA)

The operator Beta in the source text opens a new block.

BETA calls the ITABMOVE subroutine, which reads in the next Identifier Table record; and transfers Beta and the following Program Block Number to the output text.

PROCEDURE DECLARATION (PIPHI)

The operator Pi opens a declared procedure, while Phi opens a declared <type> -procedure.

PIPHI calls the ITABMOVE subroutine, which reads in the next Identifier Table record, and transfers Pi (or Phi) to the output text.

READ ITAB RECORD (ITABMOVE)

ITABMOVE reads the next Identifier Table record from the SYSUT3 data set into the work area (see Figure 42) and resets a pointer, ZCURITEN, to the end of the previously input record reprsenting identifiers declared or specified in the newly entered block or procedure. It is called by BETA and PIPHI on recognition of the operators Beta, Pi, or Phi, opening a new block or procedure. See "Processing of the Identifier Table".

Identifier Table (ITAB) records are input from SYSUT3 in parallel with the output of Subscript Table (SUTAB) and Left Variable Table (LVTAB) records on the same data set. For this reason, the data set must be repositioned to the appropriate ITAB record, before reading can begin, in the event the immediately preceding operation involved output of a SUTAB or LVTAB record. See "Phase Input/Output".

FOR STATEMENT (FOR)

FOR sets the switch ZFORTEST=X'C0', to indicate that the next operand is the controlled variable, and transfers the operator For.

PROGRAM BLOCK END (EPSILON)

The operator Epsilon marks the close of a block or procedure.

EPSILON resets a pointer in the Identifier Table work area so as to delete the block of identifier entries representing identifiers declared or specified in the block or procedure closed by Epsilon. See "Processing of the Identifier Table" in this chapter. Epsilon and the following Program Block Number of the re-entered block are transferred to the output text.

FOR STATEMENT END (ETA)

The operator Eta in the input text marks the close of a for statement.

ETA sets the switch ZFORTEST=X'00', to indicate the exit from a for statement, and calls the CRIFODEL subroutine, which deletes all entries in the critical Identifier Table for the indentifiers in the closed for statement's for list. Eta is then transferred.

DO (DO)

Do sets the switch ZFORTEST = X'00', to indicate exit from a for list, and transfers the operator Do.

WHILE (WHILE)

WHILE modifies the current for statement's classification byte in the For Statement Table (FSTAB) to indicate the presence of a while element in the list, and transfers the operator While to the output text.

SEMICOLON/DELTA (SEMIDELT)

SEMIDELT transfers the Semicolon or Delta operator, together with the following semicolon count, to the output text, after recording the semicolon count.

OPENING BRACKET (OPBRACK)

The opening bracket, [, marks the beginning of an array list, if the SARRAY switch is on.

If the array occurs in a for statement (indicated by entries in CRIDTAB), OPBRACK makes an entry for the array in the Array Identifier Stack (ARIDSTAB), provided the array does not occur in an embracing array list, and transfers the opening bracket to the output text. (The ARIDSTAB entry is not deleted until the closing bracket is encountered).

If the array occurs in an enclosing array list (indicated by the presence of an entry in ARIDSTAB), a call is made to SUCRIDEL, which scans the subscript expression in which the array occurs, to determine if it contains a controlled variable, and reclassifies the corresponding for statement(s) to Elementary Loops.

COMMA (COMMA)

The Comma marks the end of a subscript expression in an array list.

If the array occurs in a for statement, and if the ZCLOBRA switch indicates that optimization of the subscript expression is possible, COMMA calls the SUSCRITE subroutine, which makes an entry in the Subscript Table (SUTAB) for the subscript expression preceding the Comma, provided the subscript expression is optimizable (see "Optimizable Subscript Expression"). If subscript optimization is not possible, a call is made to SUCRIDEL, which scans the subscript expression to determine if it contains a controlled variable, and, if so, reclassifies the corresponding for statement(s) to Elementary Loops.

CLOSING BRACKET (CLOBRACK)

The closing bracket,], marks the end of an array list, if the SARRAY switch is on.

If the array occurs in a for list (indicated by an entry in ARIDSTAB), and if subscript optimization is possible, a call is made to SUSCRITE, which makes an entry in the Subscript Table, provided the subscript expression is optimizable. If subscript optimization is not possible, a call is made to SUCRIDEL, which scans the subscript expression to determine if it contains a controlled variable, and if so, reclassifies the corresponding for statement(s) to Elementary Loops.

If the Array Identifier Stack contains more than one entry, indicating that the current array occurs in an enclosing array list, a switch is set to indicate that optimization of the embracing subscript expression is not possible.

SCAN SUBSCRIPT (SUCRIDEL)

SUCRIDEL is called by OPBRACK, COMMA, and CLOBRACK when it is determined that an array occurs in a subscript expression of another array. It is also called by SUSCRITE when an unoptimizable subscript is found. Its function is to scan the subscript expression to determine if the controlled variable of an embracing for statement occurs in the expression, and if so, to classify the relevant for statement an Elementary Loop.

SUBSCRIPT TEST (SUSCRITE)

SUSCRITE is called by COMMA and CLOBRACK if a Comma or closing bracket,], terminates a subscript expression of an array in a for statement (indicated by one or more

entries in the Array Identifier Stack). SUSCRITE determines if the subscript expression is optimizable, and if so, makes an entry for the expression in the Subscript Table - SUTAB (see Figure 44). To be optimizable, the expression must be of the type ±F*V±A, where the Factor F is an integer variable or constant, V is the controlled variable, and the addend A is an integer variable or constant. Either F or A may be a zero constant. If F and/or A are variables, they must be declared outside the for statement in which the subscript expression occurs.

A subscript expression which satisfies the requirements of optimizability may have several forms. Thus, the factor, controlled variable, and addend may appear in positions which differ from the standard form given above (e.g., ±V*F±A or ±A±F*V). Alternatively the factor or addend may be equal to zero or one, as in the following cases:

```
±F*V    (addend = 0)
±V      (factor = 1, addend = 0)
±A      (factor = 0)
```

The subscript expression is processed in the output buffer, where the internal names of the operands in the expression will have been transferred, together with any operators, before entry to COMMA or CLOBRACK. An entry will also have been made (by OPBRACK) in the Array Identifier Stack for the array identifier.

Each operand in the subscript expression is inspected by the OPERAND subroutine, which determines if the operand is an integer variable or constant, and if the operand is a controlled variable. If it is, the address of the corresponding entry in the Critical Identifier Table is transmitted in a register (OPPTR). If the operand is not a controlled variable, OPPTR contains the value 0. If an operand is not an integer variable or constant, no entry is made in SUTAB for the subscript expression.

When the factor (zero, one, or the operand factor in the expression) or the addend (zero, or the operand addend in the expression) have been identified, each is moved to a field named, respectively, FACTOR and ADDEND. These fields will contain the one-byte sign of the factor or addend followed by the five-byte internal name of the factor or addend found in the output buffer. In case the factor or addend is absent, the five-byte internal name of the constant 0 or 1 is inserted.

The factor and addend are transferred (from FACTOR and ADDEND) to an entry in the Subscript Table by the SUTABENT subroutine. Before SUTABENT is called, a test is made of the for statement's classification byte to determine if subscript optimization is possible. A test is also made to determine if the array identifier, factor, and addend are declared outside the for statement (this is verified if the Program Block Number of the array adentifier, factor, and addend is equal to or less than the Program Block Number in the Scope Table entry - Figure 22 - corresponding to the for statement). If all tests are positive, the entry for the expression is made in the Subscript Table.

If a subscript expression occurs inside a series of nested for statements, an entry is made for each embracing for statement, for which the optimizability and scope tests, described above, are satisfied. This applies only when the factor = 0.

OPERAND TEST (OPERAND)

OPERAND is called by SUSCRITE and SUBMULT. Its function is to determine if an operand in a subscript expression is an integer variable or constant, and if the operand is a controlled variable. If the operand is a controlled variable, the address of the corresponding entry in the Critical Identifier Table is transmitted in register OPPTR. If the operand is not a controlled variable, OPPTR contains the value zero.

In the event an operand is not an integer operand or constant, OPERAND exits (via SUSCRITE) to the SUCRIDEL subroutine (which scans the subscript expression for a controlled variable and classifies the corresponding for statement(s) Elementary Loops). Thereafter, SUSCRITE returns control to the calling routine (COMMA or CLOBRACK).

MULTIPLIER-OPERAND (SUBMULT)

SUBMULT is called by SUSCRITE when an integer operand is followed by a multiplication sign (*). SUBMULT ascertains (with the aid of OPERAND) if the operand following the multiplication sign is an integer operand, and determines if either (or both) of the operands on both sides of the multiplication sign is a controlled variable. If either (or both) of the operands is a controlled variable, SUBMULT stores the operand representing the factor of the controlled variable for the innermost for statement in the location named FACTOR, and transmits the address of the

Critical Identifier Table entry for the controlled variable in register CVR. If neither of the operands is a controlled variable, or if both operands represent the same controlled variable, SUBMULT exits (via SUSCRITE) to SUCRIDEL, and SUSCRITE thereafter returns control to COMMA or CLOBRACK.

MAKE SUTAB ENTRY (SUTABENT)

SUTABENT is called by SUSCRITE when an entry for an optimizable subscript expression is to be made in the Subscript Table (Figure 44). SUTABENT constructs the entry as follows:

For Statement Number: from Critical Identifier Table entry addressed by CVR

Array identifier address: from last Array Identifier Stack entry

Factor address: from the field named FACTOR

Addend address: from the field named ADDEND

Subscript Positional Number: from ZPOSIX

Sign of factor and addend: from first byte of FACTOR and ADDEND

Output record number and relative address of opening bracket: from last Array Identifier Stack entry

SUTABENT also outputs a SUTAB record when the work area has been filled, by calling WRITE.

INPUT RECORD END (ZETA)

The operator Zeta marks the end of the current Modification Level 1 source text record.

ZETA calls the ICHA subroutine, which exchanges input buffers.

CHANGE INPUT BUFFER (ICHA)

ICHA is called on recognition of the record-end character Zeta. ICHA reads in a Modification Level 1 text record from the SYSUT1 data set to overlay the already processed record in the current buffer and resets pointers to a previous input record in a second buffer.

CODE PROCEDURE (GAMMA)

The Gamma operator precedes a code procedure identifier in the Modification Level 1 source text.

GAMMA transfers an Apostrophe replacing Gamma (X'3C'), together with the following eight-byte external name. The external name is the name by which the precompiled procedure will be called in the object code generated by the Compilation Phase.

PROGRAM END (OMEGA)

The operator Omega marks the end of the Modification Level 1 source text.

OMEGA closes the SYSUT1 data set, from which the source text was input; writes out the last Modification Level 2 text record on SYSUT2 (by calling OUCHA); and closes SYSUT2 temporarily. If the entire Modification Level 2 text occupies less than a full buffer, it is transmitted to the Compilation Phase via this buffer. The last segment of the Constant Pool is transferred to the SYSLIN and/or SYSPUNCH data sets (by calling TXTRAF), if the LOAD and/or DECK options are specified (indicated by switches in the HCOMPMOD Control Field). The last (partial) records of the Subscript Table (SUTAB) and Left Variable Table (LVTAB) are output on SYSUT3 (by calling WRITE), after the respective lengths of these tables have been stored in ZSUTEN and ZLEVEN for use by the Subscript Handling Phase. The main storage occupied by all tables in the private work area is released; and, after issuing a SPIE macro to restore the Directory routine PIROUT as the program interrupt routine, control is transferred to Diagnostic Output Module IEX31.

OTHER OPERATORS (OTHOP)

OTHOP transfers the operator (see GENTEST) to the Modification Level 2 text and returns to GENTEST.

LETTER DELIMITER (RHO)

The operator Rho signifies that the characters at the end of the preceding text record formed (a) part of a parameter delimiter or (b) part of an invalid identifier (see "Modification Level 2 Source Text" in this chapter). In the first case, RHO transfers a Comma to the Modification Level 2 text to replace the parameter delimiter. In the second case, the invalid identifier is disregarded.

STEP (STEP)

STEP modifies the current for statement's classification byte in the For Statement Table (FSTAB) to indicate the presence of the delimiter 'STEP', and transfers the Step operator to the output text.

ARRAY (ARRAY)

ARRAY turns on the SARRAY bit in the STATUS byte, to indicate that an array declaration has been encountered, and transfers the Array operator to the output text.

SWITCH (SWITCH)

SWITCH turns on the SSWITCH bit in the STATUS byte to indicate that a switch declaration has been encountered, and transfers the Switch operator to the output text.

DIVIDE/POWER (DIPOW)

DIPOW, which is entered from GENTEST on recognition of the operators Power and /, determines if the operator occurs in a for list, and if so, marks the corresponding for statement's classification byte to show the for statement is not a Counting Loop. The operator is then transferred to the Modification Level 2 text.

CHANGE OUTPUT BUFFER (OUCHA)

OUCHA resets pointers to a new Modification Level 2 text output buffer and transfers the text in the current buffer to the SYSUT2 data set. OUCHA is called by all routines when the end of the buffer has been reached. If the buffer ends in the middle of a subscript expression, the subscript expression, together with the preceding bracket or comma, is transferred to the new buffer, unless the subscript is unoptimizable. In the latter case, the SUCRIDEL subroutine is called. Before the WRITE instruction is issued, the record-end operator Zeta is transferred to the next byte in the output buffer, and pointers are reset to the new buffer.

INCORRECT OPERAND (INCOROP)

INCOROP is entered from the LETTER, DIGIT19, DIGIT0, DECPOIN, SCAFACT, and QUOTE routines when an illegal character is encountered in an operand. INCOROP searches for the end of the operand (indicated by an operator), moves the remainder of the operand (by calling MOVE) to a field named ZIDEX (the first part of the identifier will have been transferred before entry to INCOROP), calls the MOVERRO subroutine, records an error in the Error Pool, containing up to twelve characters of the erroneous operand, and transfers an all-purpose internal name to the Modification Level 2 text. Before returning control to GENTEXT, the Compiler is switched to Syntax Check Mode.

STORE ERROR (MOVERRO)

MOVERRO stores error patterns in the Error Pool. It is called by INCOROP and by other routines on detection of syntactical errors in the Modification Level 1 text. The content of the error pattern is indicated in Figure 75.

MOVE OPERAND (MOVE)

MOVE transfers all or part of an externally represented operand to a field specified by the calling routine. It is called by all routines which process identifiers and numbers.

CHECK-WRITE (CHECK)

CHECK executes a macro instruction to check the last WRITE operation on the SYSUT3 data set and sets a switch (IOBYTE) to indicate a write-check.

WRITE SUTAB/LVTAB RECORD (WRITE)

WRITE outputs Subscript Table and Left Variable Table records cn the SYSUT3 data set. It is called by OMEGA, LETRAF, and SUTABENT. See "Phase Input/Output".

PURPOSE OF THE PHASE

The main purpose of the Subscript Handling Phase is to identify the array subscript expressions listed in the Subscript Table, for which subscript optimization can be exercised in the for statements in which the expressions occur, and to transfer these subscript expressions to the Optimization Table. The subscript expressions listed in the Subscript Table are optimizable, to the extent that the expressions satisfy certain constraints with respect to the variables and operators in the expression, and the linearity of the expression in the particular for statement (these constraints are explained in Chapter 6). However, subscript optimization can only be exercised in the relevant for statement(s) provided a further condition is satisfied. This condition is that no assignment may be made in the iterated part of the for statement to any variable in the subscript expression.

In the Subscript Handling Phase, each entry in the Subscript Table is compared with the integer left variables of the corresponding for statement listed in the Left Variable Table. (To facilitate the processing, the entries in both tables are sorted according to ascending For Statement Number, before the comparison is initiated.) If an entry is found in the Left Variable Table for the same for statement which matches either the factor or the addend in the subscript expression, the subscript expression contains variables which occur as left variables, and the expression is accordingly not optimizable. If no matching entry is found in the Left Variable Table, the subscript expression is optimizable. In this case, the entry in the Subscript Table is transferred to the Optimization Table.

Under certain conditions a Counting Loop may qualify as a conditional Elementary Loop. The Subscript Handling phase serves to recognize this condition, and to change the classification of a for statement accordingly in the For Statement Table. The specific condition for this change in classification is that the addend A in a linear subscript expression of the type ±F*V±A occurs as left variable in the for statement; and that the factor F is a nonzero constant or integer-declared variable, or that the factor occurs as a left variable.

In the Scan III Phase, certain for statements are classified as non-optimizable after entries have been made in the Subscript Table. These entries are deleted in the Subscript Handling Phase before construction of the Optimization Table. They are identified by reference to the For Statement Table.

When a linear subscript expression occurs inside two or more nested for statements, the Subscript Table contains an entry for each for statement. To prevent the possibility of multiple entries for a subscript expression being transferred to the Optimization Table, all multiple entries in the Subscript Table are identified and appropriately coded, before the Optimization Table is constructed. Multiple entries are coded at the same time as the search for subscript expressions in nonoptimizable for statements is performed.

SUBSCRIPT HANDLING PHASE OPERATIONS

Figure 48 illustrates the sequence of operations performed in the Subscript Handling Phase. In the comment which follows, the numbers in parentheses refer to the numbered positions in the diagram.

After initialization of the phase, (1) the entire Subscript Table is read into main storage from the SYSUT3 data set. Subscript Table records are intermixed with Left Variable Table records on the data set, and the entries may not be in ascending For Statement Number order. The handling of Subscript Table and Left Variable Table input by the READ subroutine is discussed under "Phase Input/Output".

When input of the Subscript Table is complete, (2) a search of the table is initiated to identify and flag for deletion, any entries for subscripts contained in for statements classified non-optimizable in the For Statement Table. At the same time, multiple entries for subscripts contained in nested for statements are identified and coded.

On completion of the foregoing search, (3) the entries in the Subscript Table are sorted according to ascending For Statement Number. The sorting operation involves the transfer of the table to a new work area. After the SYSUT3 data set has been repositioned, the Left Variable Table (4) is read

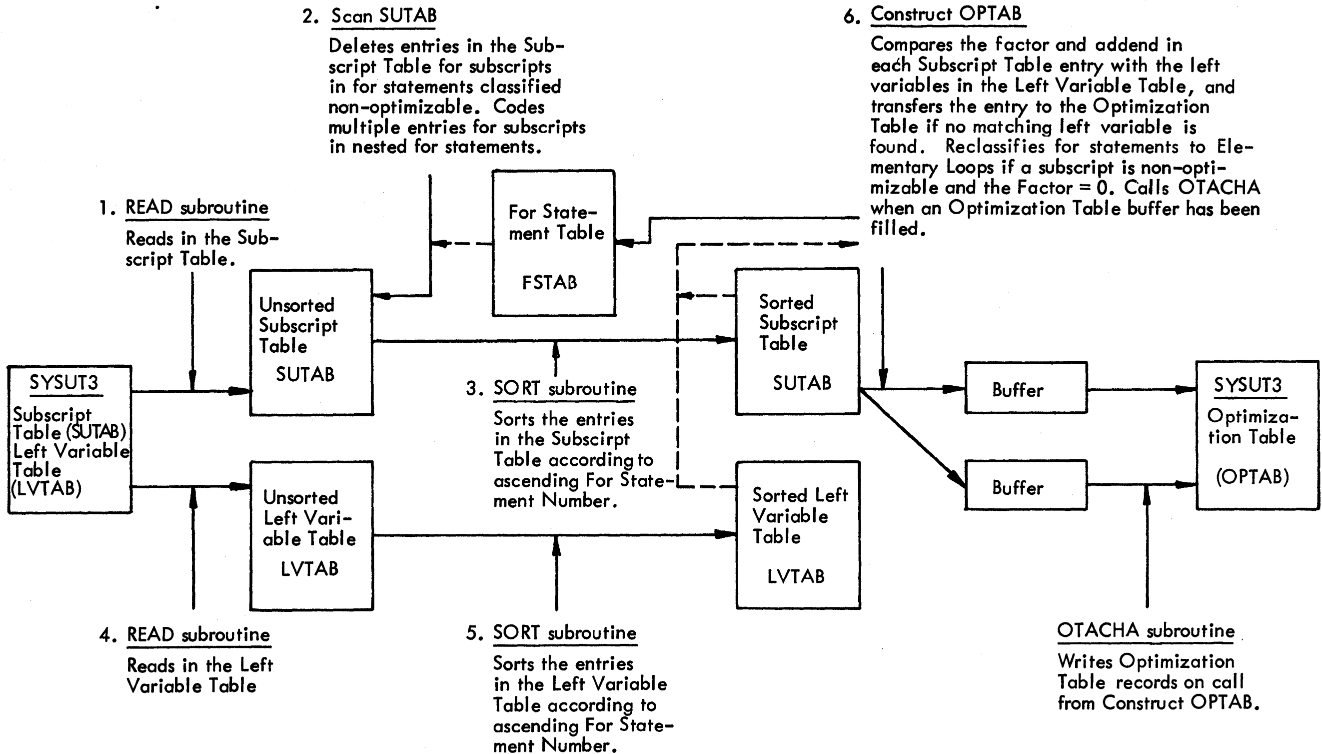


Figure 48. Subscript Handling Phase.
Diagram illustrating functions of principal constituent routines

into main storage and (5) the entries in the table are sorted according to ascending For Statement Number.

The Optimization Table (6) is constructed by transferring those entries in the Subscript Table, representing linear subscript expressions in for statements, which are found to be optimizable in the particular for statement. A subscript expression is optimizable if no assignment is made in the for statement to either the addend or the factor in the expression. This condition is verified by comparing both addend and factor with all left variables in the particular for statement, listed in the Left Variable Table. If the condition for optimizability is satisfied, the entry in the Subscript Table is transferred to the Optimization Table.

If, however, a subscript expression is not optimizable in the for statement, the Subscript Table entry is disregarded. In addition, the particular for statement is classified an Elementary Loop, if an assignment is made to the factor, or if an assignment is made to the addend, and if the factor is equal to zero.

The phase is terminated when all entries in the Subscript Table have been processed. The termination routine releases the private area used by the phase and branches to Control Section IEX40001, which initializes the Compilation Phase (Chapter 8).

PHASE INPUT/OUTPUT

The Subscript Table and Left Variable Table are read into main storage from the SYSUT3 data set independently, in two separate operations. (On the data set, SUTAB and LVTAB records are stored in random order, beginning at a point following the obsolete Identifier Table input to the Scan III Phase). Before each read operation, the data set is positioned (by a POINT macro instruction) at the correct starting address, stored by the Scan III Phase at the location named SULTSTRT.

In each read operation, the records (both SUTAB and LVTAB records) are read into the work area provided for the particular table involved. The records are identified by a key in the first four bytes (SUTB and LVTB, respectively). After a record has been read in, the last four bytes of the record are saved. The following record is then read in, the key of the record overlaying the last (saved) bytes of the previous record. The key of the newly

read-in record is now inspected. If the key shows that the record does not belong to the desired table, it is overlayed by the succeeding record. If, however, the record belongs to the desired table, the previously saved bytes of the preceding record are reinserted, overlaying the key of the last read-in record. The last four bytes of the latter record are now saved and a further record is read in.

Prior to output of the Optimization Table on the same SYSUT3 data set, the data set is repositioned to the start of the data set by a type T CLOSE. At termination of the phase, the data set is again repositioned in readiness for input to the Compilation Phase.

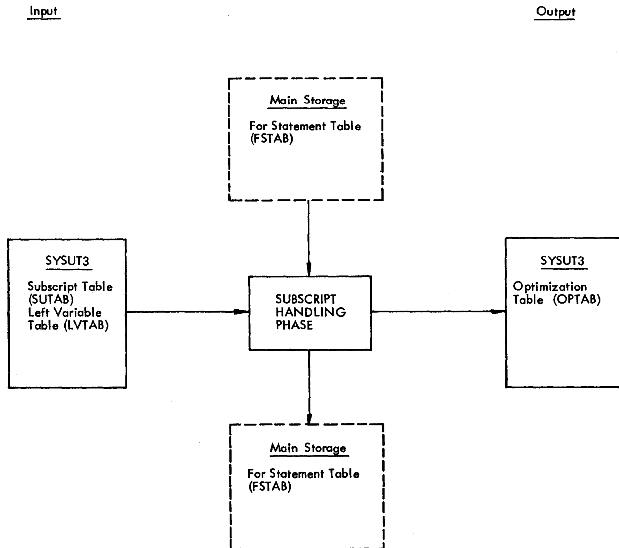


Figure 49. Subscript Handling Phase Input/Output

OPTIMIZATION TABLE (OPTAB)

The Optimization Table (OPTAB) is constructed by the Subscript Handling Phase and transmitted to the Compilation Phase on the SYSUT3 data set. The Optimization Table lists the optimizable subscript expressions found in for statements classified Counting Loops or Elementary Loops. The entries in the Optimization Table are copied from the Subscript Table (Figure 44), provided neither the factor nor the addend in the subscript expression occurs as a left variable in the particular for statement. The entry in the Optimization Table is virtually identical to the corresponding entry in the Subscript Table, except that the Chain Bits in Byte 10 of the entry, set to binary 00 in the Scan III Phase, may be equal to binary 00 or 10 in the Optimization Table.

SUBSCRIPT, LEFT VARIABLE AND FOR STATEMENT TABLES

The Subscript Table (SUTAB), Left Variable Table (LVTAB), and For Statement Table (FSTAB) are described in Chapter 6.

CONSTITUENT ROUTINES OF SUBSCRIPT HANDLING PHASE

The principal constituent routines of the Subscript Handling Phase are described below. The index in Appendix XI provides a cross-reference between the descriptive text and the relevant flowchart in the Flowchart Section.

The position of the routines in the overall logical organization of the phase may be seen in Chart 072.

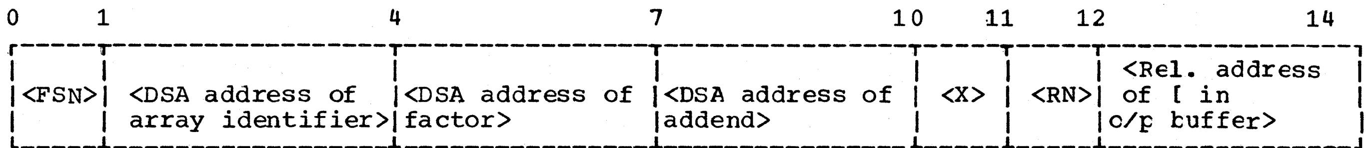
INITIALIZATION

The initialization routine acquires main storage for the private area pictured in Figure 51; sets a pointer to the beginning of the private area; and specifies program interrupt-I/O error exit routines. The Read SUTAB routine is then executed.

The size of the private area is computed as follows: 2 x (Subscript Table) + 2 x (Optimization Table buffer) + 8. The area sizes are obtained from the Area Size Table in the Common Work Area.

The private area provides two storage areas for the Subscript Table, one for input of the unsorted table and one for the sorted table. After the Subscript Table has been sorted, the input area is used, first, for input and sorting of the Left Variable Table, and second, for Optimization Table construction. Figure 51 illustrates the use of the private area in the various stages of the Subscript Handling Phase.

The first program interrupt exit TERMIN2, is stored in the location named ERET, the address referenced by the PIROUT routine in the Directory. TERMIN2 is replaced by TERM1, after the GETMAIN instruction is executed.



0	1	4	7	10	11	12	14
<FSN>	<DSA address of array identifier>	<DSA address of factor>	<DSA address of addend>	<X>	<RN>	<Rel. address of [in o/p buffer>	

<FSN> = <For Statement Number of outermost embracing for statement in which the subscript expression is optimizable>

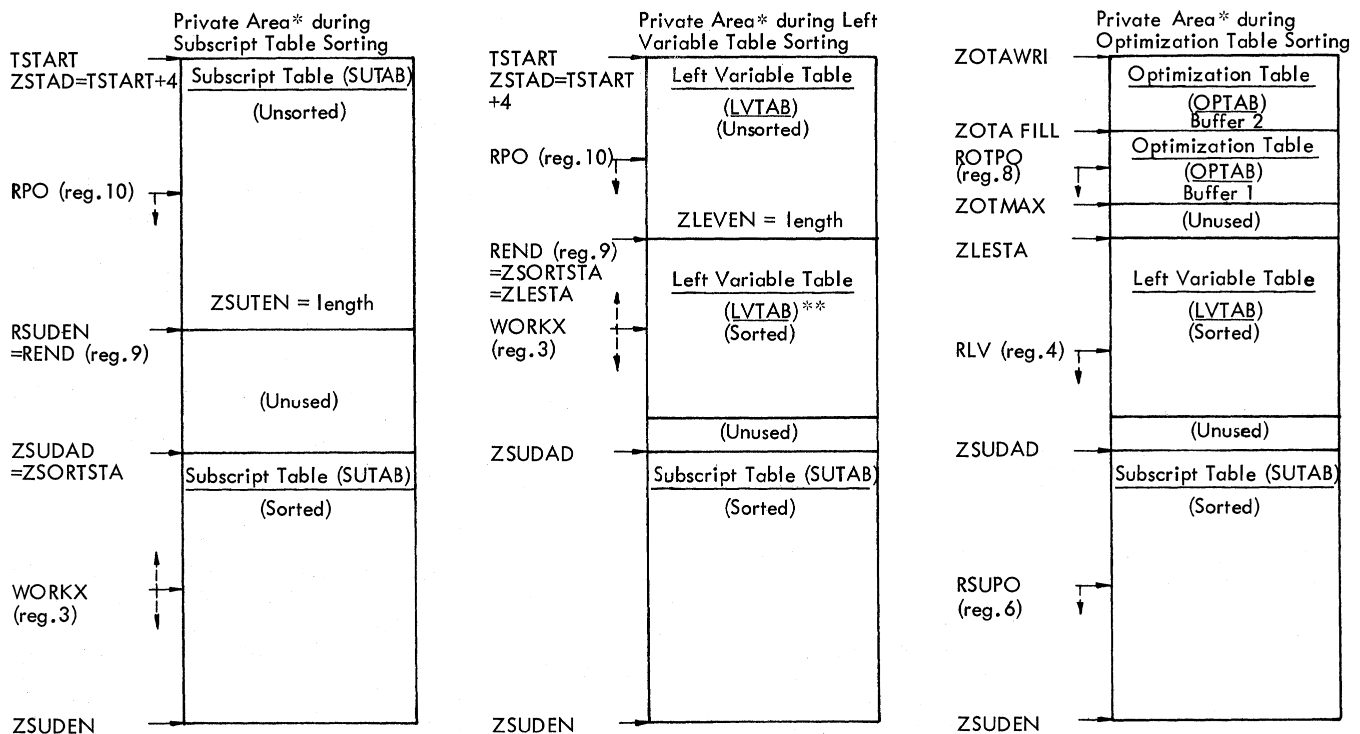
<DSA address> = <Last three bytes of identifier's internal name, containing Program Block Number and displacement>

<X> Bit 0 = <Sign of factor: 1=+, 0=->
Bit 1 = <Sign of addend: 1=+, 0=->
Bit 2-3 = (Chain Bits:* Binary 10 or 00 -- not used in Compilation Phase)
Bits 4-7 = <Positional number of subscript>

<RN> = <Source text record number in which subscript expression occurs>

*The use of the Chain Bits in the Subscript Handling Phase is discussed under the "Scan SUTAB" and "Construct OPTAB" routines.

Figure 50. Optimization Table (OPTAB) entry



* The size of the private area is computed as 2* SUTAB + 2* OPTAB Buffer + 8, the work area sizes being specified by the Area Size Table in the Common Work Area. See Appendix VIII for the Variation in area sizes as a function of the SIZE option.

** If the length of the Left Variable Table is less than twice the Optimization Table buffer length, the sorted Left Variable Table work area begins at the end of Optimization Table Buffer 1 (ZOTMAX).

Figure 51. Diagram illustrating use of the private area acquired by the Subscript Handling Phase

READ SUTAB

Read SUTAB loads registers to specify the number of records, record key (SUTB), and record length of Subscript Table records, and calls the READ subroutine, which reads the Subscript Table into the unsorted table work area. Scan SUTAB is then entered.

SCAN SUTAB

Scan SUTAB deletes all entries in the Subscript Table (Figure 44) for subscript expressions in for statements which are classified as non-optimizable. The routine also identifies multiple entries for the same subscript expression contained in nested for statements, and adjusts the Chain Bits in such entries to indicate if the entry is followed by one or more entries for the same subscript expression.

For every entry in the Subscript Table, a test is made of the classification byte in the For Statement Table, corresponding to the For Statement Number in the entry, to determine if subscript optimization is possible. If optimization is not possible, the first byte of the entry is set to X'FF', marking it for deletion, and the test proceeds with the next entry. A count is maintained of the deleted bytes for subsequent use in setting up the area for the sorted Subscript Table.

Each entry not flagged for deletion is compared with the succeeding entry, to determine if both entries refer to the same subscript expression in two nested for statements. In the affirmative case, the Chain Bits in byte 10 of each entry (originally equal to binary 00) are set, according to the key below, to indicate whether the entry relates to the outermost for statement in which subscript optimization may be possible, or to a nested for statement. The second entry is then compared with the next entry, and if this comparision shows that the same subscript expression occurs in a further nested for statement, the Chain Bits in the second and third entries are set accordingly. This comparison is repeated until the sequence of entries for the same subscript expression in a series of nested for statements has been coded. The various Chain Bit settings are as follows:

Multiple entries for subscript expressions enclosed by two or more for statements:	Chain Bit Settings (Binary)	Significance
First entry	10	Entry relates to outermost embracing for statement in which subscript optimization may be possible. One or more subsequent entries exist.
Second or intermediate entries	01	Entry relates to a nested for statement in which subscript optimization may be possible. One or more subsequent entries exist.
Last entry	11	Entry relates to innermost nested for statement in which subscript optimization may be possible. No subsequent entry exists.
Single entries	00	Entry relates to a single for statement in which subscript optimization may be possible. No subsequent entry exists.

This system of Chain Bit settings serves to insure that only one Subscript Table entry is transferred to the Optimization Table, namely the entry relating to the outermost embracing for statement in which the subscript expression is optimizable. The Chain Bits are inspected, and may be manipulated, in the Construct OPTAB routine (entered after the entries have been sorted) to determine if the entry for the subscript expression relating to an embracing for statement was transferred to the Optimization Table.

SORT SUTAB (SORTSU)

SORTSU sets pointers to specify the addresses of the unsorted and sorted Subscript Table work areas, as well as the entry length, and calls the SORT subroutine, which sorts the Subscript Table entries by ascending For Statement Number, in the sorted work area. Pointers are then set for the Optimization Table buffers (Figure 51) and the SORTLE routine is entered.

READ AND SORT LVTAB (SORTLE AND SORTLE1)

SORTLE loads registers to specify the number of records, the record key (LVTB), and the record length of Left Variable Table records, and calls the READ subroutine, which reads the Left Variable Table into the unsorted table work area. Pointers are then set to specify the addresses of the unsorted and sorted Left Variable Table work areas, as well as the entry length, and a call is made to the SORT subroutine, which sorts the Left Variable Table entries by ascending For Statement Number, in the sorted work area.

When sorting of the Left Variable Table is complete, the SYSUT3 data set is closed in readiness for output of the Optimization Table.

CONSTRUCT OPTAB (OPTAB)

OPTAB compares the addend and factor in each "active" entry of the Subscript Table (Figure 44) with the left variables listed in the Left Variable Table (Figure 43) for the corresponding for statement, and if no left variable is found which matches either addend or factor, transfers the Subscript Table entry to the Optimization Table (Figure 50). An entry in the Subscript Table is said to be "active" if no preceding entry for the same subscript expression relating to an enclosing for statement was transferred to the Optimization Table. An "active" entry is indicated if the entry's Chain Bits in Byte 10 are equal to binary 10 or 00. The entries transferred to the Optimization Table represent the array subscript expressions for which subscript optimization is exercised in the object code generated by the Compilation Phase for the relevant for statements.

If a left variable is found which matches either the addend or the factor, the Subscript Table entry is not transferred (the subscript expression being non-optimizable in the particular for statement), and the for statement in which the expression occurs is classified an Elementary Loop, provided

1. The addend occurs as a left variable in the for statement and the factor is a zero constant; or

2. The factor occurs as a left variable in the for statement.

After the last entry in the Subscript Table has been processed, control is passed to TERMIN.

Where an optimizable subscript expression is enclosed by two or more nested for statements classified Counting or Elementary Loops, the Subscript Table contains an entry for each for statement which embraces the subscript expression. With the aid of the Chain Bits in Byte 10 of the entry, the entries in the case of such a chain of nested for statements are coded, before entry to OPTAB (see Scan SUTAB routine), so as to indicate whether the entry relates to the outermost for statement or a nested for statement, and hence, whether a subsequent entry exists. be possible in an embracing for statement, optimization may be possible in one or more nested for statements. When it is determined in OPTAB that a subscript expression is not optimizable in a particular for statement, a test is made to determine if the Subscript Table contains a further entry for the same subscript expression in a nested for statement. This is determined by inspection of the entry's Chain Bits. A subsequent entry is indicated if the current entry's Chain Bits are equal to binary 10 or 01. No subsequent entry is indicated if the Chain Bit settings are equal to binary 00 or 11.

In the event the current Subscript Table entry's Chain Bits indicate that a subsequent entry exists, a search is made for the next entry representing the same subscript expression. The Chain Bit settings in this subsequent entry must be equal to binary 01 or 11, depending on whether the entry relates to an intermediate nested for statement, or the innermost nested for statement (see Scan SUTAB routine). When the entry has been located, its Chain Bits are inverted (to binary 10 or 00). This action serves to identify that the entry is now "active", i.e., that it relates

1. To the outermost embracing for statement in which the subscript expression may be optimizable, or

2. To the last (innermost) for statement

in which the subscript expression may be optimizable.

TERMINATION (TERMIN)

TERMIN writes out the last Optimization Table record (by call to OTACHA), closes the SYSUT3 data set, releases the main storage acquired for the private area, and passes control to the Initialization routine of the Compilation Phase (Control Section IEX40001), unless a terminating error has occurred (indicated by the switch TERR = 1). In the latter case, control is transferred (by XCTL) to the Compiler termination routine in Load Module IEX51.

WRITE OPTAB (OTACHA)

OTACHA provides the address of an alternate output buffer, and writes out the Optimization Table record in the current buffer on the SYSUT3 data set. OTACHA is called by the Construct OPTAB routine.

READ SUTAB/LVTAB (READ)

READ is called by the Read SUTAB and Read and Sort LVTAB routines. READ reads the Subscript Table and Left Variable Table from the SYSUT3 data set. See "Phase Input/Output".

SORT SUTAB/LVTAB (SORT)

The SORT subroutine sorts the entries in the Subscript Table and the Left Variable Table according to ascending For Statement Number. SORT is called by SORTSU and SORTLE1.

The parameters required by the SORT subroutine in sorting the two tables are specified as follows:

Start of Unsorted Table	ZSTAD
End of Unsorted Table	REND (Reg.9)
Start of Sorted Table	ZSORTSTA
Entry Length (SUTAB-14; LVTAB-4)	RENTRY (Reg.11)

See also Figure 51.

Sorting consists in moving the entries in the unsorted table, one by one, to a sorted area, so that the entries are arranged in groups with a common For Statement Number. The sorting process consists of the following steps:

1. Counting the number of entries in each group of entries having a common For Statement Number. Counting is carried out by inspecting the For Statement Number in each entry, moving sequentially through the unsorted table, and incrementing the count in a corresponding half-word of the Entry Count Table (ZCOSTA).

2. Constructing the Address Table (ZADSTA), containing the displacements in the sorted table where the first entry in each group with a common For Statement Number will be moved. In the case of the first group (For Statement Number 0), the displacement of the first entry in the sorted table is equal to zero. In the case of all subsequent groups, the displacement is computed by multiplying the entry length by the number of entries in the preceding group.

3. Moving the entries from the unsorted table to the sorted table. The destination of each entry in the sorted table is specified by the start address of the sorted table, plus the displacement contained in the corresponding entry of the Address Table. After each move, the displacement in the Address Table is incremented by the entry length, so that the resultant displacement specifies the relative address where the next entry having the same For Statement Number will be moved.

PURPOSE OF THE PHASE

The purpose of the Compilation Phase is to read the Modification Level 2 text produced by the Scan III Phase and to generate an object module which will perform the operations indicated in the source module.

Compilation is performed by approximately 60 individual compiler programs. The compiler programs are activated by action of two central routines (named SNOT - Scan to Next Operator - and COMP-Compare), which scan the Modification Level 2 text and which branch to the appropriate compiler program, according to the sequence of operators found in the source text.

For the most part the object code is determined by the individual operators (or sequences of operators) in the source text. In the case of for statements, however, the overall structure of the code is governed by the particular for statement's loop classification in the For Statement Table, constructed in the Scan III Phase (Chapter 6). The same For Statement Table also specifies, among other things, if subscript optimization is to be performed for optimizable subscript expressions contained in for statements. Optimizable subscripts in each for statement are listed in the Optimization Table.

Operand addresses in the individual instructions are obtained from the five-byte internal names representing operands in the source text.

Provided the requisite options have been specified, TXT records of the object code are generated on an external data set by a subroutine (named GENERATE) on call from the compiler programs.

For all ALGOL-defined I/O procedures or standard mathematical functions invoked in the source module, ESD records are generated to call the appropriate routines from the ALGOL Library (Chapter 10). Library routines are combined with the object module by the Linkage Editor to form an executable load module.

COMPILATION PHASE OPERATIONS

The Compilation Phase is initialized mainly by a routine which forms a control section (IEX40001) of Load Module IEX40. The initialization routine acquires main storage for a private area and initializes tables and constants in the Common Work Area and in the phase's private area. It also initiates input of the first records of the Modification Level 2 text and the Optimization Table from the SYSUT2 and SYSUT3 data sets. After initialization has been completed, control is transferred (by XCTL to Load Module IEX50 (Compilation Phase proper).

The following description provides a brief survey of the basic operating framework of the Compilation Phase, with special reference to the illustrative diagram in Figure 52. The logic of the object code generated is discussed in the later sections describing the compiler programs under appropriate headings. Figure 53 provides a guide to the various compiler programs.

Within the Compilation Phase, the SNOT routine (Scan to Next Operator) receives control after Load Module IEX50 has been loaded, and proceeds to scan the Modification Level 2 source text in the current input buffer. Source text records are read from the SYSUT2 data set by the JBUFFER subroutine, on call from SNOT (and certain compiler programs) on detection of the record-end operator Zeta. The source text consists mainly of a sequence of one-byte operators and five-byte operands. Appendix I-c indicates the internal representation of all operators in the source text. Appendix II indicates the contents of the five-byte internal names of all operands declared or specified in the source module.

SNOT scans the source text to the next operator, stores the operand (if any) which precedes the operator in the Operand Stack, and then enters the COMP routine (Compare). COMP's function is to activate the appropriate compiler program, according to the pair of operators in the source text and in the Operator Stack.

COMP determines the compiler program to be entered, with the aid of one of three decision matrices and a compiler program Address Table. There are three decision matrices: a Program Context Matrix, a Statement Context Matrix and an Expression

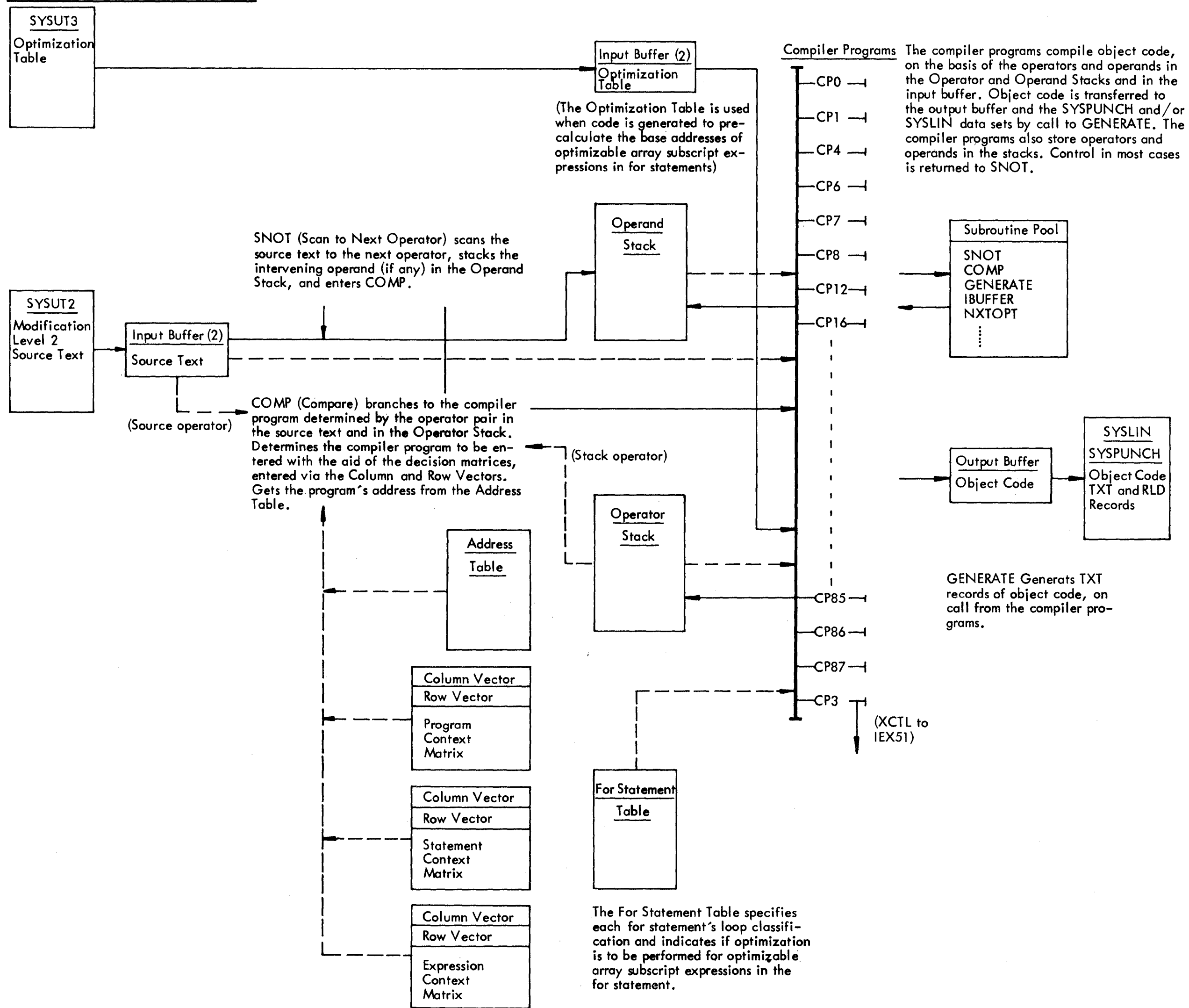


Figure 52. Compilation Phase. Diagram illustrating phase operations

Context Matrix (Appendices V-a to V-c). The particular matrix referenced by COMP at any particular point is determined by the action of a compiler program at an earlier point in time.

The decision matrices specify a particular compiler program number for every pair of operators in the source text and in the Operator Stack. A decision matrix is entered with the aid of a Row Vector and a Column Vector. The Row Vector specifies a displacement for the stack operator while the Column Vector specifies a displacement for the source operator. The sum of these displacements gives the displacement of an element in the decision matrix containing the number of the appropriate compiler program. The address of the compiler program is obtained from an entry in the Address Table which corresponds to the program number in the decision matrix.

Depending on the particular compiler program entered and on the source text, the compiler program may:

1. Compile code in accordance with the stack operator, using the address data in the stack operand; release the stack operator and operand; and return control to SNOT (or COMP); or

2. Store the source operator in the Operator Stack, and return to SNOT; or

3. Release the stack operator and return control to COMP or SNOT; or

4. Compile code in accordance with the source operator, release one or more operands, and return to SNOT.

The foregoing list indicates only a few of the compilation actions taken by the compiler programs, and represents only a small sample of the range of compilation alternatives.

In practical terms, object code is generated by means of a call to an appropriate entry point of the GENERATE subroutine. The call specifies the address and length of a sequence of one or more instructions appropriately edited by the compiler program. Depending on the Compiler options specified, GENERATE produces TXT records of the object code on the SYSLIN and/or SYSPUNCH data sets.

Compilation is terminated in the Termination Phase (IEX51), which receives control from Load Module IEX50, when the CPEND routine (entered from Compiler Program No. 3 on detection of the program-end operator Omega) issues the XCTL macro instruction.

The Termination Phase generates ESD, TXT, and RLD records for the object time Program Block Table, Label Address Table, Data Set Table, and Address Table, or, if any errors were detected during the Compilation Phase, prints out diagnostic messages for the errors recorded in the Error Pool. Control is returned to the final exit routine in the Directory after main storage has been released and data sets closed.

PHASE INPUT/OUTPUT

Input/output operations in the three modules (IEX40,IEX50 and IEX51) of the Compilation Phase are represented in Figure 53. The same figure also indicates the tables transmitted between these modules via main storage.

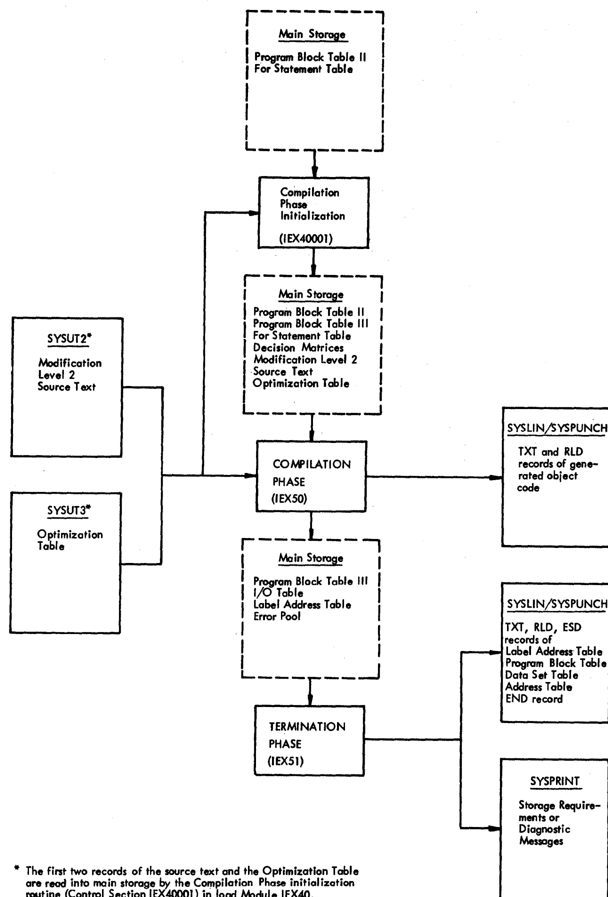


Figure 53. Compilation Phase Input/Output

The first two records of the Modification Level 2 source text, and the Optimization Table are read in by the initialization routine in Load Module IEX40. All subsequent input is handled by the JBUFFER and NXTOPT subroutines.

ESD, TXT, and RLD records are output on the SYSLIN and/or SYSPUNCH data sets by the GENERATE subroutine.

OPERATOR/OPERAND STACKS

The Operator and Operand Stacks occupy opposite ends of a combined Operator/Operand Stack area, acquired at initialization of the Compilation Phase (see Figure 59). When operators (one byte in length) are entered in the Stack, a pointer (OPTK) is incremented. When operands (five bytes in length) are entered in the stack, a pointer (OPDK) is decremented. Operators are released by decrementing OPTK, while operands are released by incrementing OPDK.

The function of the Operator and Operand Stacks is to provide temporary storage for operators and operands in the source text, as well as for other special-purpose operators and operands originating in the compiler programs). In principle, a sequence of operators and operands is stacked until such time as an operator in the source text is encountered which marks the end of a defined logical entity or relationship and which signifies that object code for one or more of the preceding operators may be generated. The stacking of operators and operands is equivalent to deferring the generation of object code until such time as the logical meaning of an operator sequence has been clarified.

Operators are stacked by a majority of the compiler programs. Operands originating in the source text are stacked by the SNOT routine, other special-purpose operands by the compiler programs.

Appendix I-d indicates the internal representation of the various stack operators. The operators are defined in the Explanation accompanying Appendix I-d.

Appendix II indicates the content of the five-byte operands representing identifiers and constants in the source module, while the notes accompanying Appendix X list the special-purpose operands stacked by the compiler programs.

In the object module generated by the Compiler, the object time operands (represented by addresses) include:

1. Operands specifically declared or specified in the source module (e.g. variables, constants or labels); and

2. Intermediate values or addresses. Intermediate values (or addresses) represent the intermediate results obtained at object time from operations (specified in the source module) on operands. Intermediate values or addresses may be contained in registers, or in Data Storage Area locations.

The above breakdown excludes a third category, not considered here, comprising operands involved in purely administrative functions.

With certain exceptions, every operand being processed by a compiler program, whether the operand be a variable, a constant, an address, or an intermediate value or address, is represented by a corresponding internal name in the Operand Stack, pointing to a corresponding storage location in an object time Data Storage Area or an entry in the Label Address Table. As soon as code is generated to load the value or address of an object time operand in a register, the corresponding operand in the Operand Stack is modified to show the value (address) in a register. A new storage field (or save area) is reserved for the register in the current Data Storage Area when the register is first assigned (see "Control of Object Time Registers"). If the contents of the register are subsequently stored at object time in the reserved storage field, the operand is appropriately modified (see below). In the same way, as soon as code is generated to move a value from one Data Storage Area field to another field reserved for an intermediate result, the operand is modified correspondingly. When an operand has been processed in the object code, or when a value or address in a register has been finally stored or disposed of (and the register relinquished), the corresponding entry in the Operand Stack is released.

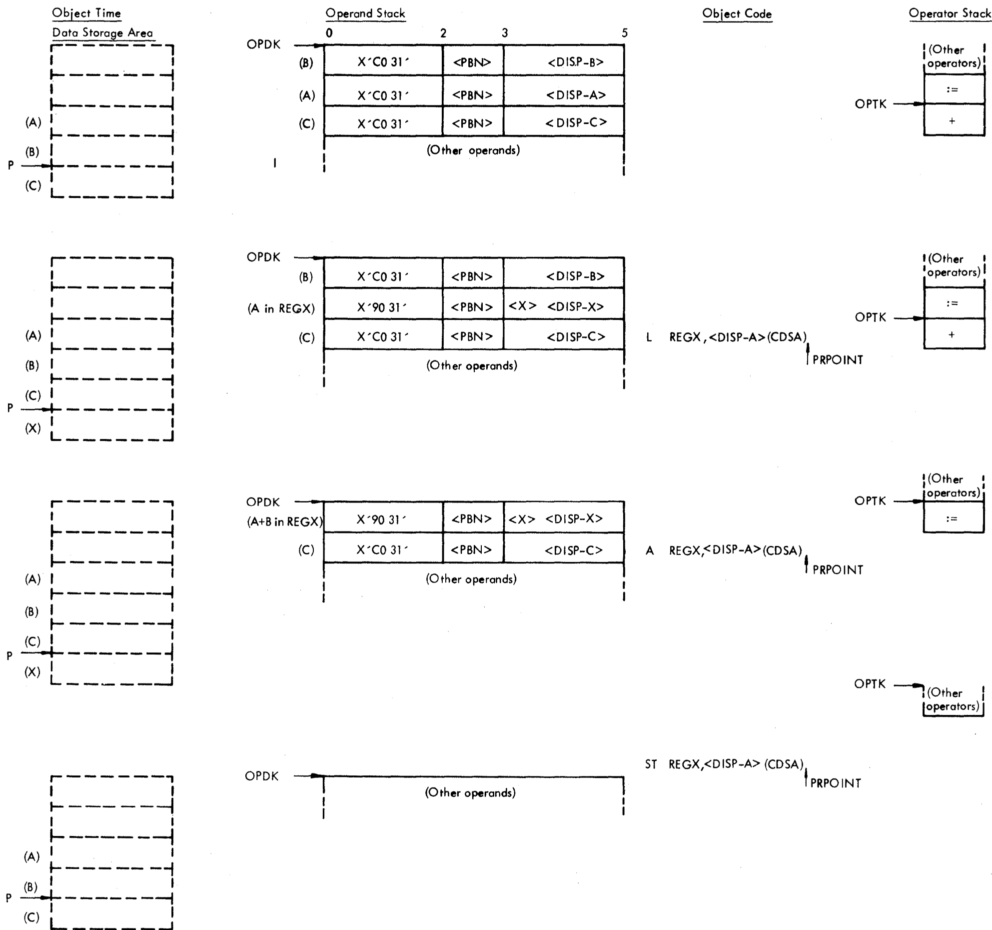
In analogous manner, an operator is released from the Operator Stack when code has been generated in accordance with the operator.

The handling of stack operators and operands is illustrated by the example in Figure 54, which shows the code generated when the semicolon marking the close of a simple arithmetic statement is encountered. The operators and operands in the statement will have been entered in their respective stacks as shown, before the semicolon is encountered. The illustration shows the adjustments to both stacks, following each step in the generation of code. The example assumes that all operands in the statement are variables and that they are all declared in the embracing block.

Identifiers A, B and C declared integer simple variables in the source module.

(In the Modification Level 2 text, the identifiers would be represented by the 5-byte internal names shown in the Operand Stack below, and the operators by one-byte operators, as in Appendix I-d.)

Source Text: . . . ; C := A + B ; . . .



Notes:

1. Pointers OPDK (reg. 9) and OPTK (reg. 10) point to the last entry in the Operand and Operator Stacks.

2. Displacement pointer P (reg. 7) indicates the displacement of the last reserved storage field in the object time Data Storage Area (DSA) of the current block or procedure. Whenever a change in scope occurs, the PBNHDL subroutine stores the contents of P, representing the DSA displacement for the last block or procedure, in a corresponding entry of PBTAB2, and loads P with the DSA displacement for the newly entered (or reentered) block or procedure, contained in the corresponding entry of PBTAB2.

   P is incremented whenever additional storage bytes are reserved in a Data Storage area, e.g. for an intermediate value or address in a register. P may subsequently be decremented, as in the illustration, if code is not subsequently generated to store an intermediate value (address) in the reserved storage field. If, however, code is actually generated to store the intermediate value (address), the displacement in P is stored (by the MAXCH subroutine) in a corresponding entry of PBTAB3, thus recording the minimum length required (up to that particular point) for the particular Data Storage Area at object time.

3. Displacement pointer PRPOINT (reg. 6) indicates the displacement in bytes of the next object code instruction to be generated, counted from the start of Constant Pool No. 0. PRPOINT is updated by the GENERATE subroutine after every segment of code is generated. In the compiler programs, the length in PRPOINT is used as a relative address in specifying branch addresses (as well as addresses of declared labels) in the object module. In such cases, the displacement in PRPOINT is stored in the appropriate entries of the Label Address Table. At linkage edit time, the relative addresses in the Label Address Table are converted to absolute addresses.

4. The abbreviations in the Operand Stack entries have the following significance.

   PBN - Program Block Number of the block or procedure in which the identifier is declared or specified.

   DISP-A
   DISP-B - Displacement of the identifier's assigned storage field in the object time Data Storage Area.
   DISP-C

   DISP-X - Displacement of Register X's reserved storage field in the object time Data Storage Area.

   X - Number of the object time register containing the intermediate value. The register may be any one of the registers available for general computational use (see "Control of Object Time Registers").

   Concerning the characteristic in the first two bytes of the stack operand, see the accompanying text. Compare also with Appendix II.

5. The notation in the illustrated object code has the following significance.

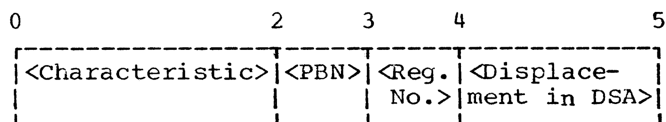
   REGX - The name of the object time register whose number (X) is contained in the stack operand. The register may be any one of the registers available for general computational use (see "Control of Object Time Registers").

   DISP-A
   DISP-B - The displacement of the identifier's assigned storage field in the object time Data Storage Area. The displacement is obtained from the relevant stack operand.
   DISP-C

   CDSA - The name of base register 10, containing the address of the current Data Storage Area, i.e. the object time Data Storage Area of the current block or procedure. CDSA appears in all of the illustrated instructions because, in the assumptions of this simple case, all of the identifiers are declared in the immediately embracing block. If any one of the identifiers had been declared outside the embracing block, the base register in the instruction involved would have been GDSA (reg. 9) -- Global Data Storage Area. In the object module, GDSA is always used to address the relevant Data Storage Area, where an operand is not declared or specified in the current block or procedure; CDSA at all times addresses the current Data Storage Area.

Figure 54. Diagram illustrating the function of the Operator/Operand Stacks

Figure 55 indicates the contents of the operand representing an intermediate value or address contained in a register or contained in a Data Storage Area field, i.e., the Data Storage Area of the block or procedure having the Program Block Number <PBN>.



<Characteristic> - (See text)

<PBN> = <Program Block Number of the embracing block or procedure>

<Reg. No.> = <Number of the register containing the intermediate value/address> or <zeros> if operand in Data Storage Area

<Displacement in DSA> = <Displacement of the register's storage field in the Data Storage Area of the block or procedure indicated by PBN>

Figure 55. Five-byte operand representing an intermediate value or address contained in an object time register or temporarily stored in the register's reserved storage field in a Data Storage Area.

Except for certain special-purpose operands, the location and type of every operand at object time is indicated by specific binary settings in the characteristic of the relevant stack operand (namely, bits 0, 1, and 2 of Byte 0). The significance of the various settings is as follows:

Characteristic Byte 0	Significance
(Bits 0, 1, 2)	Operand represents:
110	a variable or constant with an assigned storage field in a Data Storage Area or a Constant Pool, or the address of a declared procedure, switch or label with an assigned entry in the Label Address Table
100	a value contained in the register indicated in the operand
010	an intermediate value contained in a Data Storage Area field
101	an address contained in the register indicated in the operand
011	an address contained in a Data Storage Area field

CONTROL OF OBJECT TIME REGISTERS

Of the 16 general purpose registers available for use by the object module, seven registers are restricted to specific addressing functions. The remaining eight general purpose registers, as well as all four floating-point registers, are available for general computational or addressing purposes. The division of register assignments is as follows:

Available for general computational use
General Purpose Registers 0-7 (GPR0,...,GPR7)
Floating Point Registers 0,2,4, and 6 (FPR0,...,FPR4)

Available for general addressing use
General Purpose Register 8 (ADR)

Restricted to Specific Addressing Functions
General Purpose Registers 9-15. The use of each of these registers is indicated in Chapter 11, under "Object Time Register Use".

To provide for the logical and efficient assignment of the registers available for general computational and addressing use, two systems of register control are employed, one for general purpose registers, the other for floating-point registers. These control systems, which are administered by a set of subroutines indicated below, have two purposes:

1. To ensure that, whenever a competing demand arises for the use of a given register, any intermediate value contained in that register is duly saved; and
2. To minimize the number of Store Register instructions in the object module (in other words, to maximize the register holding time).

Figure 56 illustrates the control fields used in the control of object time general purpose registers.

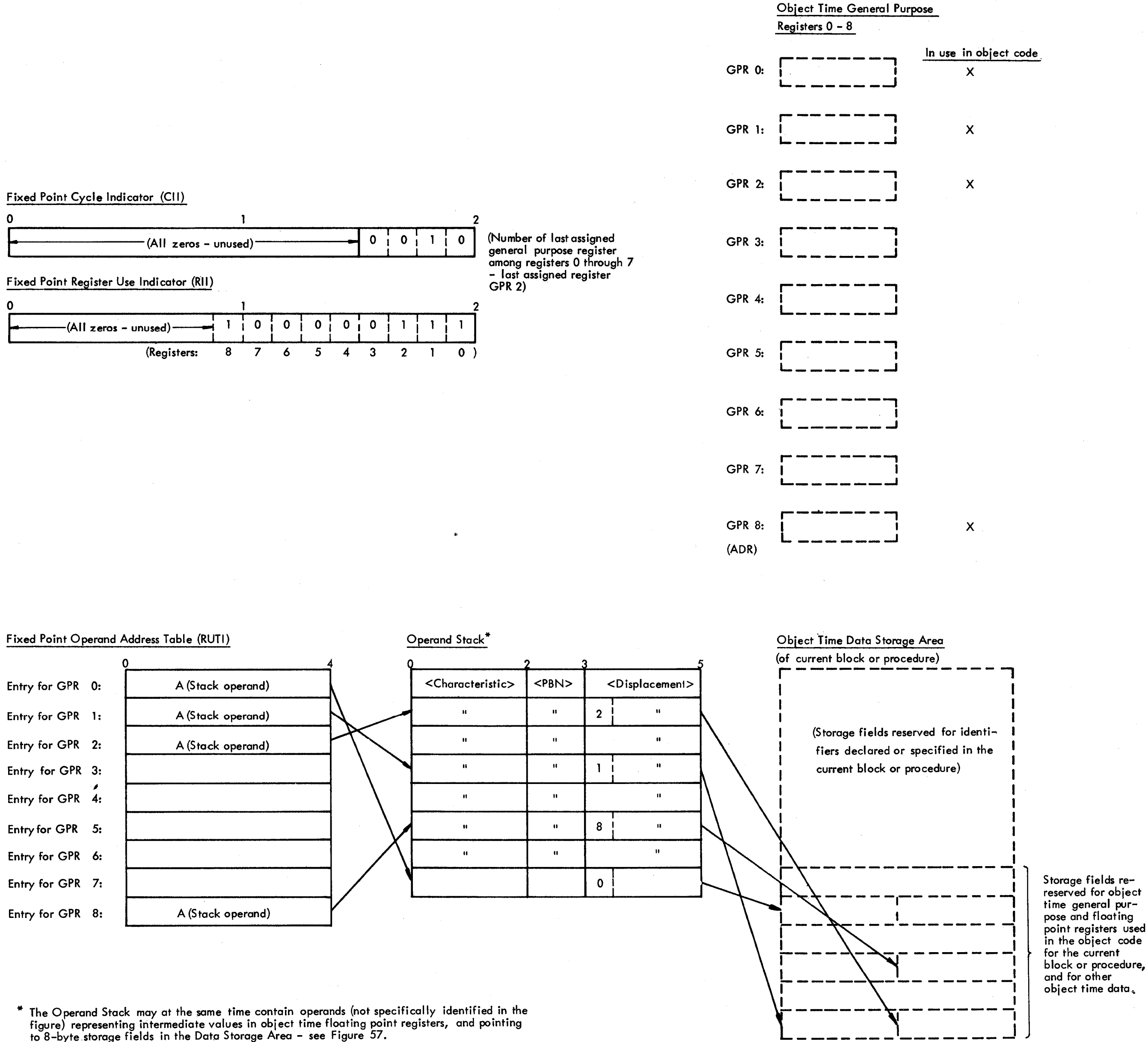


Figure 56. Control Fields governing use of object time general purpose registers, showing relationships to Operand Stack, Data Storage Area and registers

The two-byte Fixed Point Cycle Indicator (CII) indicates the number of the last assigned register among general purpose registers 0 through 7. When a register is required for computational purposes, CII is increased by one (or reset to 0 if CII=7). The resulting number is the number of the register to be used. This rotational assignment of registers ensures a maximum register holding time. (Where an even or odd pair of registers is required for integer division or multiplication, CII may be increased by two).

The two-byte Fixed Point Register Use Indicator (RII) indicates the registers currently in use among general purpose registers 0 through 8. A register in use is indicated if the corresponding bit is turned on. If the register is currently in use, a Store Register instruction must be generated before the register can be used in the subsequent object code (see below).

The 36-byte Fixed Point Operand Address Table (RUTI) provides a full word for each of registers 0 through 8. Whenever code is generated to load a value (or address) in a register, a four-byte storage field, or save area, is reserved for the register in the current Data Storage Area. The relative address of the save area is recorded in the relevant Operand Stack entry, while the address of the Operand Stack entry is stored in the register's full-word entry in the Operand Address Table (RUTI). If, when a given register is to be used, the Register Use Indicator (RII) shows that the register is currently in use, code is first generated to store the contents of the register in the storage field specified by the relevant Operand Stack entry. Thereafter, a new save area is reserved for the register in the current Data Storage Area, its displacement being recorded in the current Operand Stack entry, and the address of the Operand Stack entry being stored in the appropriate entry of the Operand Address Table (RUTI).

When a register is released (i.e., as soon as it is no longer needed), the register's save area in the Current Data Storage Area is relinquished, and the corresponding bit in the Register Use Indicator (RII) is turned off. The Cycle Indicator (CII) is reduced by one (or reset to 7 if CII=0), and the Operand Address Table (RUTI) is unaffected.

Object time register control is handled primarily by ROUTINE1, ROUTINE2,..., through ROUTIN15.

Figure 57 illustrates the control fields used in the control of object time floating-point registers. The function of these fields is entirely anologous to that of the control fields which govern the assignment of general purpose registers (Figure 56).

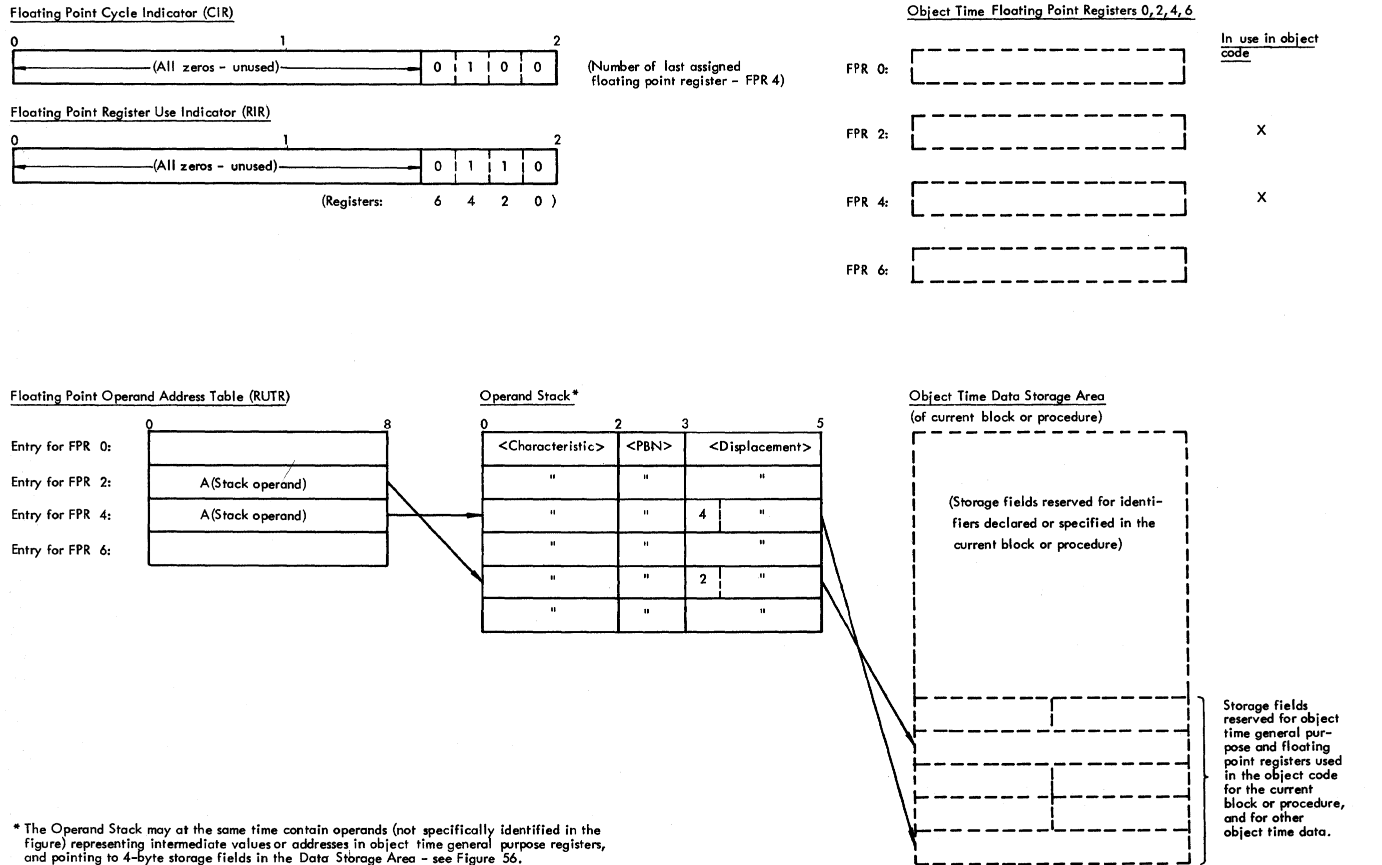


Figure 57. Control Fields governing use of floating point registers, showing relationships to Operand Stack, Data Storage Area and registers

DECISION MATRICES

The basic control framework of the Compilation Phase is expressed by a set of three decision matrices named the Program Context Matrix, the Statement Context Matrix and the Expression Context Matrix (Appendices V-a, V-b, and V-c). Each matrix specifies a particular compiler program to be activated (by the COMP routine) for every possible pair of operators in the source text and in the Operator Stack.

The operative decision matrix at any particular point in time depends, in general, on the logical context of the source module currently being processed. Matrix changes are effected by the compiler programs according to changes in logical context, as indicated by the operators in the source module. A switch from one matrix to another is effected by incrementing or decrementing the matrix base register CCT (register 11). A prospective change to a specific decision matrix may be specified by the storage of an appropriate operator (PRC, STC, or EXC) in the Operator Stack, whose subsequent detection will cause the relevant compiler program to switch to the appropriate matrix.

The operative decision matrix is referenced (by the COMP routine) with the aid of a Column Vector and a Row Vector. The Column Vector consists of a series of 2-byte elements, each containing a displacement value associated with any given operator in the source text. The Row Vector consists of a series of 2-byte elements, each containing a displacement value associated with any given operator in the Operator Stack. The sum of the displacements for the source operator and the stack operator gives the displacement of an appropriate half-word in the operative decision matrix addressed by pointer CCT. The half-word thus specified multiplied by four contains the relative address of an entry in the Address Table, containing the absolute address of the relevant compiler program.

COMPILE TIME REGISTER USE

The general purpose registers are used in the Compilation Phase as follows:

Register Number	Register Name	Use
0	(Variable)	Variable use
1	(Variable)	Variable use
2	(Variable)	Variable use
3	(Variable)	Variable use
4	(Variable)	Variable use
5	SBR	Base register of the Subroutine Pool
6	PRPOINT	Displacement pointer to the next instruction in the generated code
7	P	Displacement pointer to the last reserved storage field in the object time Data Storage Area of the current block or procedure
8	SOURCE	Pointer to the current operator in the Modification Level 2 source text
9	OPDK	Pointer to the latest entry in the Operand Stack
10	OPTK	Pointer to the latest entry in the Operator Stack
11	CCT	Base register of the operative decision matrix
12	BASE	Base register of the operative compiler program
13	WAREG	Base register of the Common Work Area
14	(Variable)	Variable use
15	(Variable)	Variable use

The use of registers 6 (PRPOINT), 7 (P), 9 (OPDK), and 10 (OPTK) is illustrated in Figure 54.

CONSTITUENT ROUTINES OF THE COMPILATION PHASE

The principal constituent routines of the Compilation Phase are described below. The relevant text and flowchart in which each routine is discussed or outlined can be found with the aid of the index in Appendix XI. The diagram in Figure 58 provides a guide to the various routines from the standpoint of the logical elements of the source module processed by each routine.

All primary control routines, i.e., all routines entered directly from COMP, are referred to as "compiler programs". Subsidiary routines are referred to as "routines" or "subroutines".

COMPILATION PHASE
(IEX50)

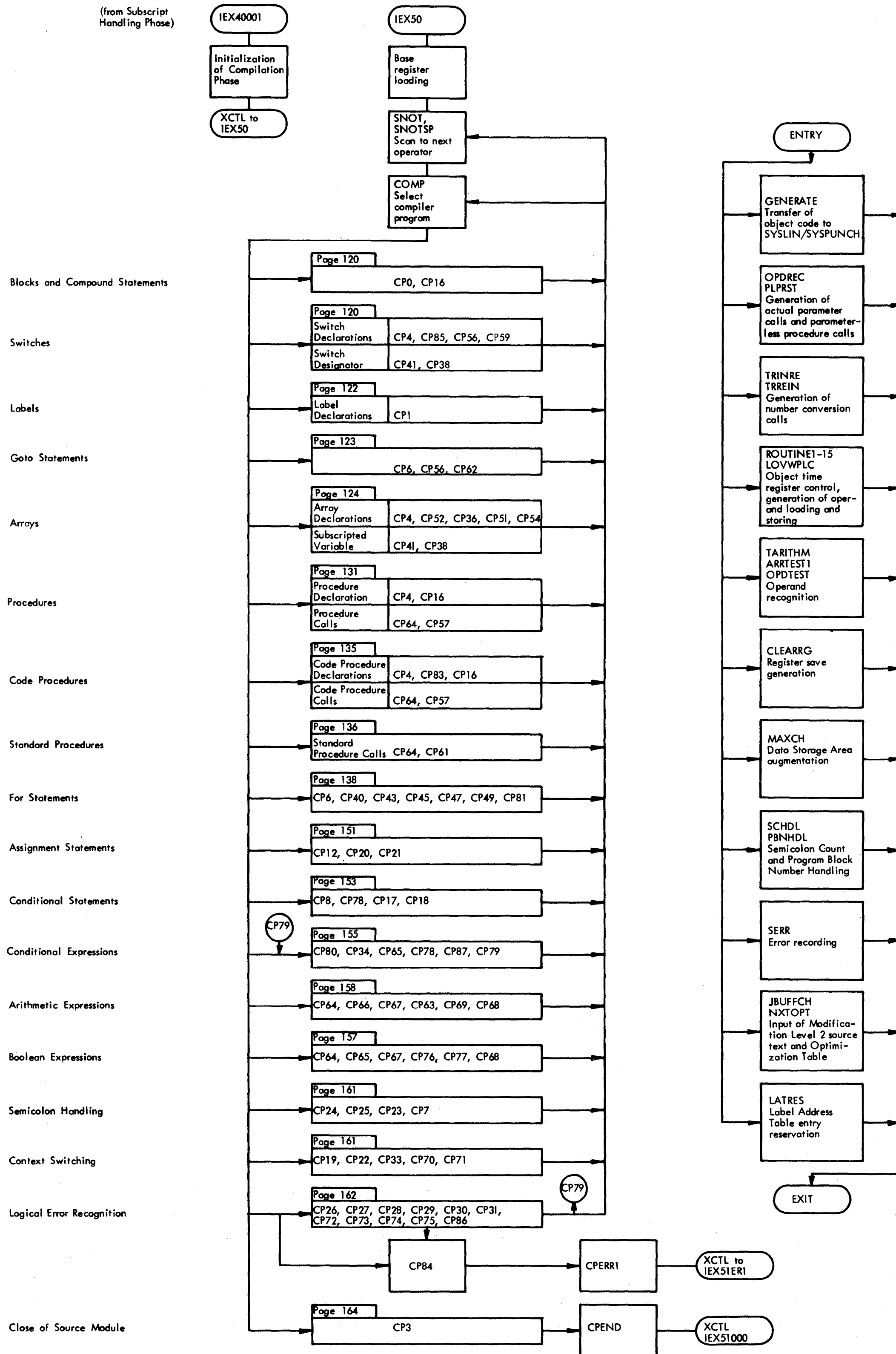


Figure 58. Diagram showing the compiler programs in relation to the logical elements of the source module

A summary of the compiler programs, showing, among other things, the stack operators and operands at entry and exit, errors detected, and subroutines called, is provided in the table in Appendix X.

PHASE INITIALIZATION

Compilation Phase initialization is divided between Control Section IEX40001 in Load Module IEX40, and a short initializating routine at the start of Load Module IEX50. Control Section IEX40001 acquires main storage for the private area pictured in Figure 59; constructs Program Block Table III in the Common Work Area; reads in the first two records of the Modification Level 2 source text and the Optimization Table; and transfers control (XCTL) to Load Module IEX50.

The initializing routine at the start of Load Module IEX50 addresses the Program Context Matrix in the Common Work Area; specifies new program interrupt, I/O error and End of Data exits; and enters the Scan to Next Operator routine.

At entry to IEX40001, the address of the terminating routine INERR1 is stored at ERET, the location referenced by the Program Interrupt routine (PIROUT) and the I/O Error Routines (SYNAD and SYNPR) in the Directory. INERR1 is replaced by INERR2 after the private area has been acquired.

The GETMAIN instruction for the private area pictured in Figure 59 is issued after the area sizes of all work areas have been totalled. The area sizes are obtained from the Area Size Table (see Chapter 3) in the Common Work Area, except in the case of the Label Address Label, for which the area is fixed at 4096 bytes.

Source Text Buffer 2 is acquired only if the Modification Level 2 source text is to be read into main storage from the SYSUT2 data set (i.e., the buffer is not acquired if the entire source text was transmitted by the Scan III Phase in Source Text Buffer 1 in the Common Area, indicated by the SPIC switch in the HCOMPMOD Control Field). Except for the first two records, which are read in by Control Section IEX40001, Modification Level 2 text records are read from the SYSUT2 data set by the JBUFFER subroutine on call from SNOT (Scan to Next Operator), and Compiler Programs 4, 51, and 66.

Similarly, buffers for the Optimization Table are acquired only if the table was constructed by the Subscript Handling Phase, indicated by the OPT switch in the HCOMPMOD Control Field. Except for the first two records, which are read in by Control Section IEX40001, Optimization Table records are read from the SYSUT3 data set by the NXTOPT subroutine, on call from Compiler Programs 40, 47, and 49.

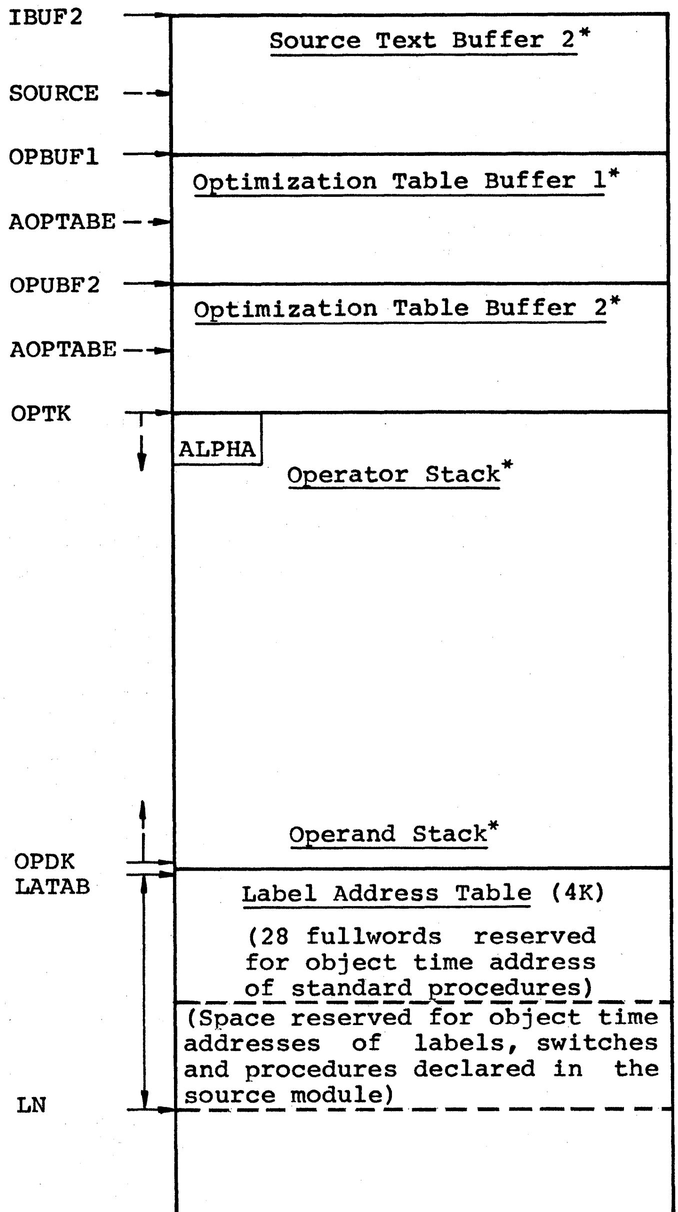


Figure 59. Private Area acquired by Control Section IEX40001 for the Compilation Phase (IEX50)

The Operator and Operand Stacks occupy opposite ends of a combined Operator/Operand Stack area. The handling of the Operator and Operand Stacks is discussed elsewhere in this chapter.

A total of 4096 bytes are allocated for the Label Address Table. This includes space for entries assigned in the Scan I/II Phase to labels, switches, and procedures declared in the source module. The remainder of the Label Address Table is used in the Compilation Phase for branch addresses produced by the Compiler in the object code. The displacement of the last assigned entry is indicated by displacement pointer LN (Label Number), transmitted from the Scan I/II Phase via the Common Work Area.

The first 28 full-words in the Label Address Table are reserved for the object time addresses of standard procedures which may be called in the source module. At initialization, the first bit in each full-word is set = 1. If, subsequently, a call for the standard procedure is detected, the bit in the corresponding entry is reset to 0 (by Compiler Program No. 64). In the Termination Phase, ESD records are generated for all standard procedures which were called, indicated by the first bit in the entry being equal to 0.

Program Block Table III is constructed in the Common Work Area by transferring the contents of each two-byte entry in Program Block Table II (see Chapter 5) to the first two bytes of the corresponding four-byte entry in Program Block Table III. The last two bytes in the new entry are zero-set.

Program Block Table III is located in the Common Work Area. In the same way, space is provided for various other tables, including:

- Data Set Table (DSTAB)
- I/O Table (IOTAB)
- Subscript Table (SUTABC)

The tables are described elsewhere in this chapter.

As soon as the first two records of the Modification Level 2 source text and the Optimization Table have been READ (and CHECKed) from the SYSUT2 and SYSUT3 data sets, control is passed (by XCTL) to Load Module IEX50 (Compilation Phase proper). Before the respective READ instructions are issued, the address of a point to be entered in the event of an End of Data conditon, is stored at EODUT2 and EODUT3 in the Common Work Area, the locations referenced by the End of Data routines EODAD2 and EODAD3 in the Directory. If the entire source text was transmitted by the Scan III Phase in the Common Area buffer, or if no Optimization Table was constructed (indicated, respectively, by the SPIC and OPT switches in the HCOMPMOD Control Field), the corresponding READ and CHECK macro instructions are bypassed.

On entry to Load Module IEX50, the address of the terminating routine CPERR1 is stored at ERET, the location referenced by the Program Interrupt routine PIROUT and the I/O Error routines SYNAD and SYNPR in the Directory. After loading register 11 (CCT) with the address of the Column Vector of the Program Context Matrix (see Appendix IV-a), the End of Data exits JB3 and NX4 are specified for the SYSUT2 and SYSUT3 data sets, and the Scan to Next Operator routine (SNOT) is entered.

SCAN TO NEXT OPERATOR (SNOT)

SNOT scans the Modification Level 2 source text to the immediately following operator, and branches to COMP. The new operator is addressed by pointer SOURCE (register 8). If the operator was preceded by an operand, represented by a five-byte internal name, the operand is stored in the Operand Stack and a switch named OPDT is turned on. OPDT is inspected by the various compiler programs to determine whether specific processing treatment of a preceding operand is required, or to identify a syntactical error condition.

In a majority of the compiler programs, the exit (or return address) is that of SNOT. In other compiler programs the return address is SNOTSP (or SPEC), a special entry point of COMP. In CP3, entered on recognition of the program-end operator <u>Omega</u>, the exit is to CPEND, which transfers control to Load Module IEX51.

SNOTSP is specified as the return address where the operator may not logically be followed by an operand. If an operand follows, SNOTSP records Error No. 191, and continues to search for the next operator.

In both SNOT and SNOTSP, a call is made to the JBUFFER subroutine (which addresses the alternate buffer containing a new source text record), if the record-end operator <u>Zeta</u> is detected.

COMPARE (COMP)

COMP branches to the compiler program specified in the current decision matrix for the particular pair of operators in the source text and in the Operator Stack, addressed respectively by pointers SOURCE (register 8) and OPTK (register 10).

COMP is entered from SNOT, SNOTSP and from certain compiler programs.

BLOCKS AND COMPOUND STATEMENTS

Compiler Program No.0 (CP0)

Source Operator: Beta, Begin (both stacked)
Stack Operator: Alpha, Beta, Pi, Phi, Then-s, Else-s, Do or Semicolon

Beta opens a block, Begin a compound statement.

For Beta, a call is made to CLEARREG (which produces object code to save registers), and code is then generated to activate the Fixed Storage Area routine PROLOGP (see Library), which acquires a Data Storage Area for the new block at object time. Before exit to SNOT, a call is made to PBNHDL, which loads register P with the length (size) reserved for the block's object time Data Storage Area, contained in Program Block Table III.

For Begin, the operator is stacked and control returned to SNOT.

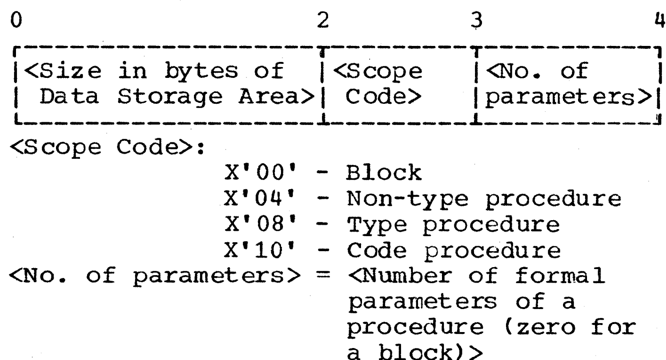


Figure 60. Entry in Program Block Table III (PBTAB3)

Compiler Program No.16 (CP16)

	CASE A	CASE B	CASE C
Source Operator:	Epsilon	End	Epsilon
Stack Operator:	Beta	Begin	Pi, Phi
	(all released)		

CASE A: Epsilon closes a block. Code is generated to call the Fixed Storage Area routine EPILOGB, which releases the Data Storage Area of the block exited and activates the Data Storage Area of the enclosing block. The PBNHDL subroutine is also called.

CASE B: End closes a compound statement. Begin is released.

CASE C: See "Procedures" below.

All cases: If the source operator was preceded by an operand (in which case the operand logically represents a parameterless procedure statement), a call is made to the subroutine PLPRST, which generates code to call the procedure.

SWITCHES

The object code generated in the case of a switch consists of several separate (but interrelated) segments. The first segment comprises the code generated for the switch declaration. The subsequent segments comprise the code generated for each switch designator.

Figure 61 illustrates the code generated for a declared switch having three simple label components, and for a single goto statement containing a switch designator. The figure also indicates the compiler programs which generate the various segments of the object code, and the Fixed Storage Area routines invoked in the object code. The latter are described more fully in Chapter 10.

The code for a switch declaration consists essentially of

1. An opening instruction to branch around the declaration,
2. A series of code sequences, one for each component in the switch list, and each ending with a branch to the Fixed Storage Area routine CSWE2, and
3. A list of address constants, each pointing to one of the preceding component code sequences.

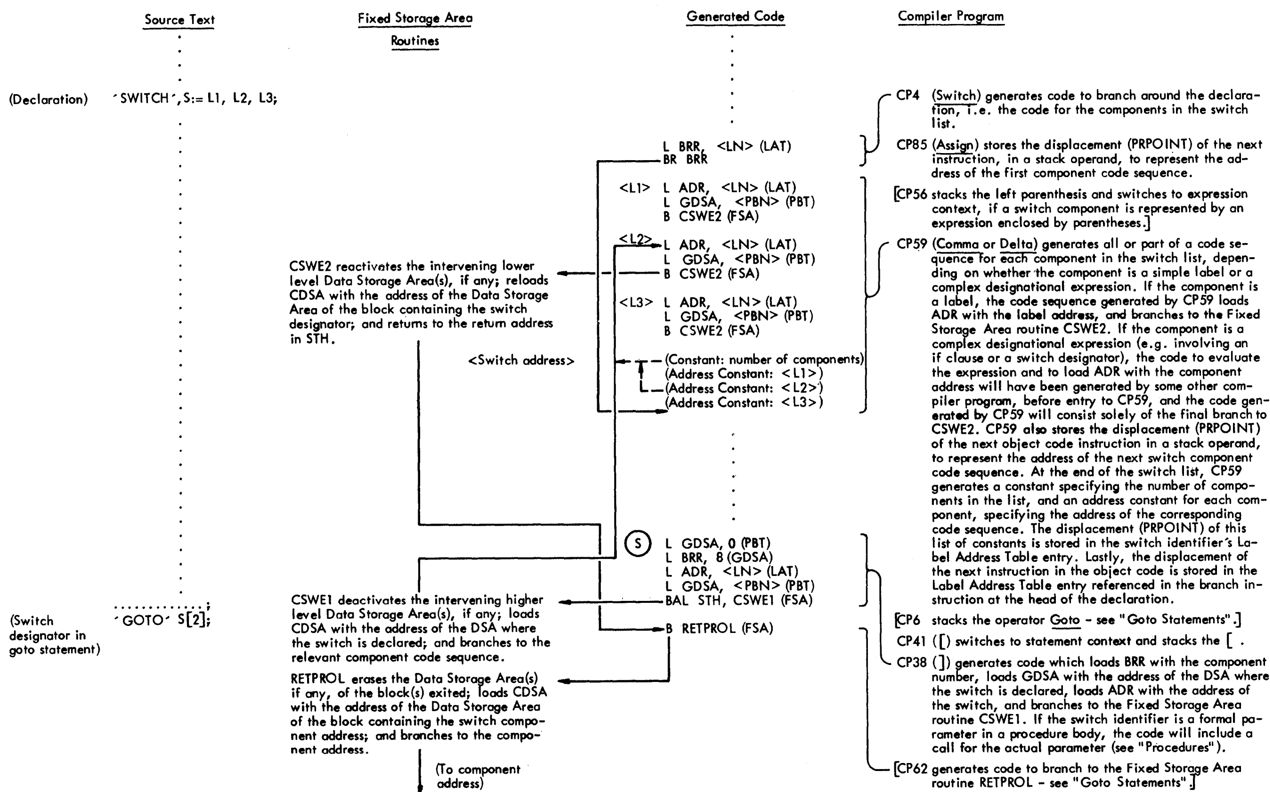


Figure 61. Diagram showing code generated for switch declaration and switch designator

The opening branch instruction ensures that none of the component code sequences is executed until a call for a switch component is actually executed. The length of the code sequence depends on the complexity of the component in the switch list.

The code for a switch designator consists essentially of instructions to load the component number specified in the switch designator, and the switch address (the address of the address constants in the switch declaration), and to branch to the Fixed Storage Area routine CSWE1. If the switch identifier in a designational expression is a formal parameter, the code may be more involved (see "Procedures").

The object time starting point in the execution of the code for a switch is the code corresponding to a switch designator (in Figure 61, the starting point is marked by a circled S).

When the code is executed, registers are loaded with the switch address and the component number, and a branch is then taken (via the Fixed Storage Area routine CSWE1) to the relevant component code sequence in the declaration. The component code sequence loads the component address (in the illustration, the address is a simple label address) in a register and returns (via the Fixed Storage Area routine CSWE2) to the next instruction following the call to CSWE1.

Compiler Program No.4 (CP4)

CASE A CASE B CASE C

Source Operator: Switch, Array, Pi, Phi (all stacked)

Stack Operator: Beta, Pi, Phi or Alpha

CASE A: The source operator Switch marks the beginning of a switch declaration. Code is generated to branch around the object code to be subsequently generated for the components in the switch list. See Figure 61.

CASE B: See "Arrays".

CASE C: See "Prodedures".

Compiler Program No.85 (CP85)

Source Operator: Assign
Stack Operator: Switch

The combination of source and stack operators indicates the start of a switch list in a switch declaration. The operator Switch:= is stacked, replacing Switch, and control is returned to SNOT.

Compiler Program No.56 (CP56)

	CASE A		CASE B
Source Operator:		(	
Stack Operator:	Switch:=	or	Goto

CASE A: The operator (in this context indicates the beginning of a switch component enclosed by parentheses. The operator is stacked and a shift made to the Expression Context Matrix, before control is returned to SNOT.

CASE B: See "Goto Statements".

Compiler Program No.59 (CP59)

Source Operator: Comma or Delta
Stack Operator: Switch:= (released if source operator = Delta)

Either source operator marks the end of a component in a switch declaration. CP59 generates the code sequence indicated in Figure 61 for the component. If the source operator is Delta, indicating the last component in the switch list, a constant, representing the number of components in the list, and a series of address constants, each pointing to one of the preceding component code sequences, are generated.

Compiler Program No.41 (CP41)

	CASE A	CASE B
Source Operator:	[	
Stack Operator:	(See decision matrices -- Appendixes V-a, V-b, V-c)	
Stack Operand:	<Switch Identifier>	<Array Identifier>

CASE A: The operators and the stack operand identify a switch designator, e.g., S[2]. After stacking an operator to specify a return to the current matrix, the Statement Context Matrix is addressed and the source operator is stacked. Control is then returned to SNOT.

CASE B: See "Arrays".

Compiler Program No.38 (CP38)

	CASE A	CASE B
Source Operator:	]	Comma or]
Stack Operator:	[	[
Stack Operand:	<Switch Identifier>	<Array Identifier>

CASE A: The source operator] follows a component number or expression in a switch designator. Code is generated to branch (via the Fixed Storage Area routine CSWE1) to the component code sequence corresponding to the specified component number, as indicated in Figure 61. The stack operators and all operands are released, a shift is made to the decision matrix specified by the next stack operator, and control is returned to SNOT.

CASE B: See "Arrays".

LABELS

Compiler Program No.1 (CP1)

Source Operator: Label Colon
Stack Operator: Alpha, Beta, Pi, Phi, Begin, Semicolon, Then-s, Else-s or Do

The source operator follows a declared label, represented by the operand at the top of the Operand Stack.

The displacement (PRPOINT) of the next instruction in the object module is stored in the Label Address Table entry addressed by the label operand, and the operand is released. If the stack operator is Beta, Pi, or Phi, a Semicolon is stacked in order to ensure that an error is subsequently recorded if a declaration follows.

GOTO STATEMENTS

Compiler Program No.6 (CP6)

	CASE A		CASE B
Source Operator:	Goto	or	For
Stack Operator:	Begin, Do, Semicolon, Then-s, or Else-s		

CASE A: The source operator identifies a goto statement. The operator is stacked, a switch is made to the Statement Context Matrix, and control is returned to SNOT.

CASE B: See "For Statements".

Compiler Program No.56 (CP56)

	CASE A		CASE B
Source Operator:		(	
Stack Operator:	Switch:=	or	Goto

CASE A: See "Switches".

CASE B: The source operator indicates the beginning of a designational expression. The operator is stacked, a switch is made to the Expression Context Matrix, and control is returned to SNOT.

Compiler Program No.62 (CP62)

Source Operator: Epsilon, Eta, Semicolon, End, or Else
Stack Operator: Goto

Any one of the source operators marks the end of a goto statement.

Code is generated to branch to the address designated by the stack operand, either directly or via the Fixed Storage Area routine RETPROL. The stack operand may represent a simple label declared in the current block or in some higher level block, or an address contained in register ADR representing the computed address of a designational expression other than a simple label.

In the last case register GDSA is loaded with the address of the Data Storage Area in which the label was declared.

If the operand represents a simple label declared in the current block or procedure, code is generated to branch directly to the label address, loaded in register BRR. If the operand represents a simple label declared in some higher level block or procedure, the generated code loads the label address in ADR, loads GDSA with the address of the Data Storage Area corresponding to the block in which the label is declared, and branches via RETPROL to the label address. If the operand represents an address in ADR, the code branches to that address via RETPROL. The latter routine releases the Data Storage Area of every block or procedure exited when the branch is taken.

The address of any particular element, $A[s_1,s_2,s_3,\ldots,s_\alpha]$, of the array declared $A[L_1:U_1,L_2:U_2,L_3:U_3,\ldots,L_\alpha:U_\alpha]$, may be expressed by the formula:

(a) address of $A[s_1,s_2,s_3,\ldots,s_\alpha]$

$$= \text{address of } A[L_1,L_2,L_3,\ldots L_\alpha]$$
$$+ (s_1-L_1)\{(U_2-L_2+1)(U_3-L_3+1)\ldots(U_\alpha-L_\alpha+1)P_{\alpha+1}\}$$
$$+ (s_2-L_2)\{(U_3-L_3+1)(U_4-L_4+1)\ldots(U_\alpha-L_\alpha+1)P_{\alpha+1}\}$$
$$+ (s_3-L_3)\{(U_4-L_4+1)(U_5-L_5+1)\ldots(U_\alpha-L_\alpha+1)P_{\alpha+1}\}$$
$$+ \ldots$$
$$+ (s_\alpha-L_\alpha)\{P_{\alpha+1}\},$$

where

α = the number of subscripts in the array,

$L_1 \le s_1 \le U_1,\ L_2 \le s_2 \le U_2,\ L_3 \le s_3 \le U_3,\ \ldots,\ L_\alpha \le s_\alpha \le U_\alpha,$

$A[L_1,L_2,L_3,\ldots,L_\alpha]$ is the address of the first element, and
$P_{\alpha+1}$ is the length in bytes (1, 4, or 8) of each element.

The expressions within braces {...} in formula (a) are called address increment factors, and represent the incremental displacement associated with an increment of 1 in the particular subscript value. The address increment factor, P_{i+1}, for any given subscript position i in the array $A[s_1,s_2,s_3,\ldots,s_\alpha]$ is defined by

$$P_i = P_{\alpha+1} \prod_{j=i}^{\alpha} (U_j - L_j + 1),$$

where $P_{\alpha+1}$ = length of array element = address increment factor of the last subscript position, α .

Replacing the address increment factor expressions in the address formula (a) above, by the notation P_i, the following abbreviated form is obtained:

(b) address of $A[s_1,s_2,s_3,\ldots,s_\alpha]$

$$= \text{address of } A[L_1,L_2,L_3,\ldots,L_\alpha]$$
$$+ (s_1-L_1)P_2+(s_2-L_2)P_3+(s_3-L_3)P +\ldots+(s_\alpha-L_\alpha)P_{\alpha+1}.$$

Expanding and rearranging:

(c) address of $A(s_1,s_2,s_3,\ldots,s_\alpha]$

$$= \text{address of } [L_1,L_2,L_3,\ldots,L_\alpha]$$
$$- (L_1P_2+L_2P_3+L_3P_4+\ldots+L_\alpha P_{\alpha+1})$$
$$+ (s_1P_2+s_2P_3+s_3P_4+\ldots+s_\alpha P_{\alpha+1}).$$

This may be reduced to

(d) address of $A(s_1,s_2,s_3,\ldots,s_\alpha]$

$$= \{\text{address of } A[L_1,L_2,L_3,\ldots,L_\alpha] - \sum_{i=1}^{\alpha} L_i P_{i+1}\}$$
$$+ (s_1P_2+s_2P_3+s_3P_4+\ldots+s_\alpha P_{\alpha+1}).$$

The quantity enclosed by braces {...} represents a constant value, which holds independently of the address of the particular array element. It is called the zero-base address of the array and represents the address of the array element (an imaginary or actual element) with zero subscripts. The zero-base address of an array is computed at object time, and stored in the array's Storage Mapping Function, by subtracting the calculated value of the quantity

$$\sum_{i=1}^{\alpha} L_i P_{i+1}$$

from the absolute address of the first element in the array.

The address of any arbitrary array element $A[s_1,s_2,s_3,...,s_\alpha]$ is determined at object time by adding the calculated value of the expression

$$(s_1P_2+s_2P_3+s_3P_4+...+s_\alpha P_{\alpha+1})$$

to the zero-base address.

Array Declarations

The primary function of the object code generated for every array declared in the source module is:

1. To acquire main storage, sufficient to accommodate all elements of the array;

2. To construct an object time Storage Mapping Function (Figure 62) containing the array address, the zero-base address, and the address increment factors.

See the Fixed Storage Area routine VALUCALL (Chapter 10) concerning the treatment of value-called array parameters of procedures.

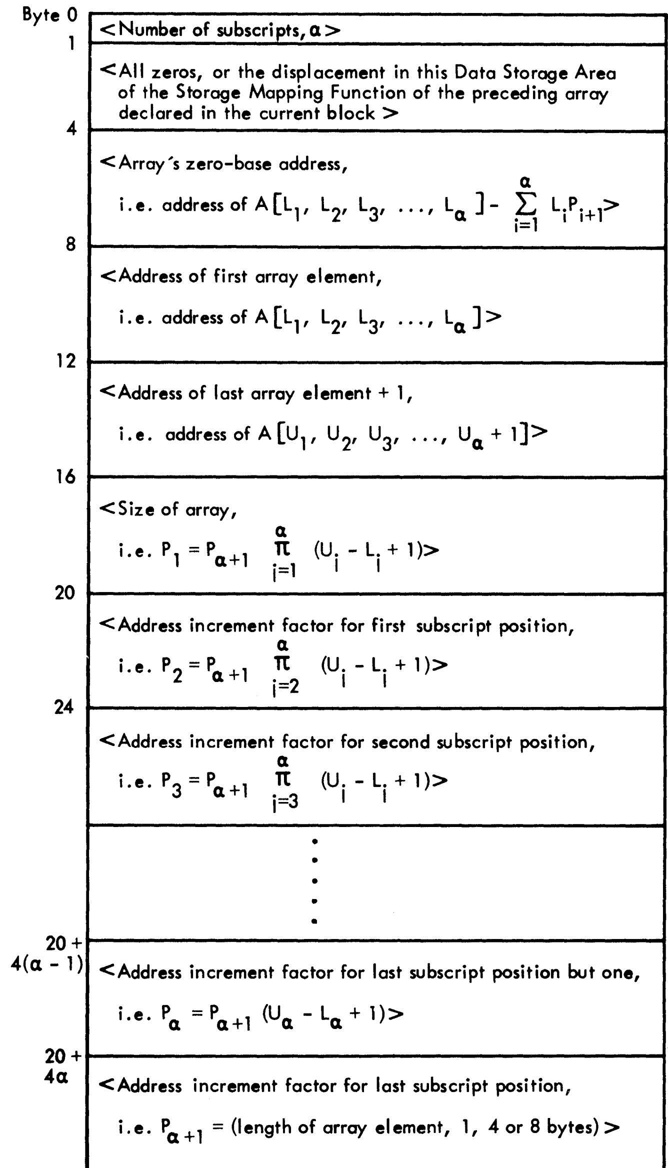


Figure 62. Object Time Storage Mapping Function of an array

The Storage Mapping Function for a particular array is constructed in the object time Data Storage Area of the particular block in which the array is declared. The area acquired for the array, on the other hand, is located outside the Data Storage Area, in any part of main storage provided by the control program. An array's storage area is released, simultaneously with the release of the particular block's or procedure's Data Storage Area, at exit from the block or procedure in which the array is declared.

Array declarations are handled by CP4, CP52, CP36, CP51, and CP54. The object code which issues the GETMAIN instruction for the array storage area and which constructs the array's Storage Mapping Function, is generated by CP51. The steps followed in this object time process are as follows:

1. The range ($U_i - L_i + 1$) of each dimension of the declared array is computed and stored in a corresponding four-byte field of the Storage Mapping Function, beginning at byte 16 for the first dimension.

2. When all of the array dimensions have been processed as in item (1), the array element length $P_{\alpha+1}$ (1, 4, or 8 bytes, depending on whether the array is a boolean, integer or real array) is stored in the last entry of the Storage Mapping Function, to represent the address increment factor P, for the last dimension.

3. The address increment factors for the remaining subscripts, and the array size, are now computed and stored in the Storage Mapping Function. The computation consists in multiplying the dimension range in each entry (beginning with last entry but one and moving upward) by the address increment factor in the entry below, and storing the result in the entry, displacing the previously recorded dimension range. For a three-dimension array, $A[L_1:U_1,L_2:U_2,L_3:U_3]$, with array element length d, the contents of the Storage Mapping Function from byte 16 onwards, after this computation, would be as follows.

Byte		Contents
16-19	Array Size,	$P_1=P_2(U_1-L_1+1)$
20-23		$P_2=P_3(U_2-L_2+1)$
24-27		$P_3=P_4(U_3-L_3+1)$
28-31		$P_4=d$

At the same time, the product $L_i P_{i+1}$ is computed for each dimension and the result added to a cumulative total, so as to calculate the quantity

$$\sum_{i=1}^{\alpha} L_i P_{i+1}$$

This quantity is used in deriving the array's zero-base address (see item 5). The L_i values are saved in consecutive entries of the current Data Storage Area.

4. Main storage equal to the computed array size is acquired, by call to the Fixed Storage Area routine GETMSTO, and the addresses of the first element and the last element + 1 of the array are stored in bytes 8-11 and 12-15 of the Storage Mapping Function. The displacement of the Storage Mapping Function in the current Data Storage Area is recorded in bytes 12-15 of the Data Storage Area (Figure 88), the previous contents of bytes 13-15 being moved to bytes 1-3 of the Storage Mapping Function.

5. The zero-base address of the array

   (= address of $A[L_1,L_2,L_3,\ldots,L_\alpha]$

   $$-\sum_{i=1}^{\alpha} L_i P_{i+1}),$$

   is derived by subtracting the computed value of the quantity

   $$\sum_{i=1}^{\alpha} L_i P_{i+1}$$

   (see item 3) from the address of the first element of the array (item 4). The zero-base address is stored in bytes 4-7 of the Storage Mapping Function.

6. The number of subscripts in the array is stored in byte 0 of the Storage Mapping Function.

<u>Subscripted Variables</u>

Subscripted variables in source module statements are processed by CP41 and CP38. The object code generated by these programs depends primarily on whether the subscripted variable occurs in an embracing for statement, and if so, whether the subscripted variable contains subscripts optimized (precalculated) at entry to the iterated part of the for statement. To be optimizable, a subscript expression must be of the type

$$(\pm F*V\pm A),$$

where F denotes a factor which must be an integer variable or constant, V denotes the for statement's controlled variable, and A denotes an addend which must be an integer variable or constant. F and/or A may be zero constants.

Where a for statement contains a subscripted variable with one or more optimizable subscripts, code is generated (see "For Statements"), at entry to the iterated part of the for statement, to compute (in the next available register named NXTR) the sum of the array's zero-base address and the initial value of the product $s_i P_{i+1}$ for every optimizable subscript in the array, viz.

(e) address of A[0,0,0,...,0]+$\sum s_i' P_{i+1}$,

where s_i' denotes the initial value of the optimizable subscript and P_{i+1} denotes the address increment factor for the subscript position. In addition, code is generated to compute the cyclical address increment for all optimized subscripts, to be added to NXTR in each cycle of the for statement (except the first), viz.

(f) $\sum \pm F * P_{i+1} *$(Step Value),

where F denotes the factor in the subscript, P_{i+1} denotes the address increment factor, and (Step Value) denotes the increment to the controlled variable in each cycle of the for statement. At object time, the contents of NXTR in any given cycle of the for statement will thus be

(g) address of A[0,0,0,...,0]+$\sum s_i' P_{i+1}$

+ (N-1){$\sum \pm F * P_{i+1} *$(Step Value)}

where N denotes the number of executed cycles of the for statement.

CP41 is entered when the opening bracket, [, in a subscripted variable is encountered. If the subscripted variable occurs in an embracing for statement and if the subscripted variable contains one or more optimized (precalculated) subscripts, CP41 generates code to load the precalculated address (expression (g) above) in a register reserved to hold the address of the subscripted variable.

If, however, the subscripted variable contains no optimized subscripts, CP41 generates code which simply loads the reserved register with the array's zero-base address.

CP38 is entered when the <u>Comma</u> separating two subscripts, or the closing bracket,], at the end of an array operand, is encountered. If the array subscript was optimized (i.e., precalculated) in an embracing for statement, CP38 generates no object code for the subscript. If, however, the subscript was not optimized, CP38 operates code to compute the product $s_i P_{i+1}$ for the subscript and to add this product to the address previously loaded (by CP41) in the reserved register.

The result of this treatment at object time is to derive the particular array element address, by adding the contributions of each subscript to the array's zero-base address. If the subscripted variable contains subscripts optimized in an embracing for statement, the sum of the zero-base address and the initial contributions of the optimized subscripts is computed at entry to the iterated part of the for statement; the contributions of the non-optimized subscripts, together with a cyclical address increment for the optimized subscripts, are added in each cycle of the for statement. If the subscripted variable contains no optimized subscripts, on the other hand, the particular array element address must be wholly derived in each cycle by computing the contributions of all subscripts and adding these to the zero-base address.

<u>Compiler Program No.4 (CP4)</u>

	CASE A	CASE B	CASE C
Source Operator:	<u>Switch</u>	<u>Array</u>	<u>Pi</u> or <u>Phi</u>
Stack Operator:	<u>Beta</u>, <u>Pi</u>, <u>Phi</u>, or <u>Alpha</u>		

CASE A: See "Switches".

CASE B: The source operator identifies a following array declaration. The operator is stacked, the Statement Context Matrix is addressed, and control is returned to SNOT.

CASE C: See "Procedures".

<u>Compiler Program No.52 (CP52)</u>

Source Operator: <u>Comma</u> or [
Stack Operator: <u>Array</u>

The source operators represent delimiters in an array declaration, e.g., 'ARRAY' A, B [1:10];. If the source operator is a <u>Comma</u>, a count of the array identifiers associated with the following bound pair list, is incremented. If the source operator is the opening bracket [, the operator ¢ is stacked.

Compiler Program No.36 (CP36)

Source Operator: Colon
Stack Operator: ¢

The source operator represents the colon separating a bound pair in an array declaration, the lower bound being represented by the last entry in the Operand Stack. Code is generated to store the lower bound in the next entry in the Data Storage Area, and the Colon is stacked. If the related stack operand shows that the lower bound is not an integer, a call is first made to the TRREIN subroutine, which generates code to branch to the Fixed Storage Area routine CNVRDI for conversion of the lower bound to integer (fixed point) form.

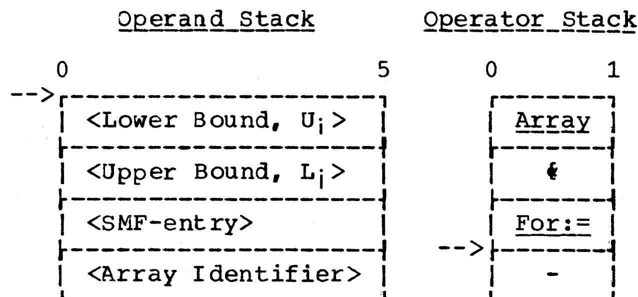
Compiler Program No.51 (CP51)

Source Operator: Comma or]
Stack Operator: Colon

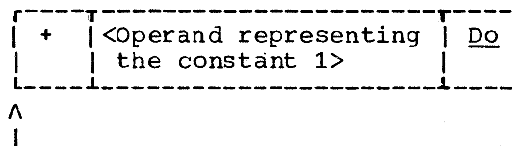
The combination of source and stack operators indicates that the last stack operand represents the upper bound of a bound pair in an array declaration. The operator and operand stacks and the source text pointer are now adjusted, preparatory to entering CP69 (via entry point DHZB1), so as to specify the operation

(Upper Bound, U_i)-(Lower Bound, L_i)+1,

and of storing the resultant in the relevant entry of the array's Storage Mapping Function (Figure 62). The contents of the operator and operand stacks after this adjustment are as follows:



The contents of the source text pointer SOURCE (register 8) are saved, and the pointer is reset to address a special field containing the following dummy Modification Level 2 text:



The sequence of compiler programs and routines subsequently entered, and the actions taken, as a result of these adjustments, is as follows:

Operators at Entry: Source	Stack	Compiler Program/Routine		Operators at Exit: Source	Stack
+	-	CP69	(DHZB1) generates code to perform the operation U_i-L_i, releases the stack operator -, and exits to COMP	+	For:=
+	For:=	COMP	branches to CP66, determined by the operator pair + and For:=	+	+
		CP66	stacks the source operator + and exits to SNOT		
+	+	SNOT	scans the dummy source text to the operator Do, stacks the operand representing the constant 1, and exits to COMP	Do	+
Do	+	COMP	branches to CP69, determined by the operator pair Do and +	Do	For:=
		CP69	generates code to perform the operation $(U_i-L_i)+1$, releases the stack operator +, and exits to COMP		
Do	For:=	COMP	branches to CP70, determined by the operator pair Do and For:=	Do	Assign
		CP70	switches to the Statement Context Matrix, and exits to COMP		
		COMP	branches to CP43, determined by the operator pair Do and For:=		
		CP43	replaces the stack operator For:= by Assign and exits to CP20 (DB1C2)		
Do	Assign	CP20	(DB1C2) generates code to store the dimension range, U_i-L_i+1, in the Storage Mapping Function entry addressed by the relevant stack operand (see above), releases the stack operator Assign, and returns to CP51 (DERE2)	Do	[

At re-entry to CP51, the source text pointer SOURCE is reloaded with the previously saved address of the source operator Comma or] in the current input buffer. If the source operator is Comma, indicating that a further bound pair follows, an entry is reserved in the Data Storage Area for the object time value of the lower bound to be processed next, and control is returned to SNOT. The lower and upper bounds will subsequently be processed by CP36 and CP51, in the manner described above.

If the source operator is the closing bracket], indicating the end of the bound pair list, CP51 now generates code to compute the address increment factors for each subscript position in the array, and to store the factors in the appropriate entries of the array's Storage Mapping Function (Figure 62). Thereafter, the stack operator [is released and code is generated to acquire a storage area for the array, and to fill in the data in the first 16 bytes of the Storage Mapping Function, in the manner outlined at the beginning of this section. A test is then made to determine if the same bound pair list defines a second array (as, for example, in 'ARRAY' a,b[1:10,1:10];). In this case, code is generated to copy the first array's Storage Mapping Function, from byte 16 onwards, into the second array's Storage Mapping Function area. Code is then generated to acquire a storage area for the second array and to complete the remainder of the Storage Mapping Function. This procedure is repeated for every array defined by the bound pair list.

Lastly, the character following the closing bracket is inspected. Logically, the closing bracket may be followed by a Comma (as, for example, in 'ARRAY' a,b[1:10], c[1:100];) or by Delta, representing the semicolon at the end of the

declaration. Any other character following the closing bracket would represent a syntactical error. If the operator is a Comma, control is passed to SNOT. The programs subsequently entered from COMP will process the following array(s) and bound pair list(s). If any character other than a Comma is detected, control is passed to COMP, which then branches to CP54 (if the character is the operator Delta), or to some other compiler program, which will record the syntatical error.

Compiler Program No.54 (CP54)

Source Operator: Delta
Stack Operator: Array [released]

The source and stack operators indicate that the end of an array declaration has been reached. A call is made to the SCHDL subroutine (which updates the semicolon count and generates a call to the Fixed Storage Area routine TRACE, if the TEST option has been specified); the Program Context Matrix is addressed, and control is passed to SNOT.

Compiler Program No.41 (CP41)

	CASE A	CASE B
Source Operator:	[	
Stack Operator:	(See matrices -- Appendixes V-a to V-c)	
Stack Operand:	<Switch identifier>	<Array identifier>

CASE A: See "Switches".

CASE B: A subscripted variable in a statement has been encountered. After stacking an operator to specify a return to the current matrix, the Statement Context Matrix is addressed and the source operator stacked.

A register is reserved to hold the address of the array element. If the subscripted variable contains no subscripts optimized in an embracing for statement (if any), or if the array is a formal parameter, code is then generated to load the register with the array's zero-base address. If, however, the subscripted variable contains one or more subscripts optimized (precalculated) in the embracing for statement, code is generated to load the reserved register with the precalculated address (expression (g) above). The presence of an optimized subscript is determined, in the first instance, by verifying whether SUTABC (Figure 71) contains any entries (representing one or more arrays in the embracing for statement, containing optimized subscripts), and in the second instance, by comparing the position of the current subscripted variable's opening bracket in the input buffer, with the position noted in SUTABC of each of the arrays listed. Where this comparison shows that SUTABC contains an entry for the subscripted variable, the same entry will contain the address of an operand pointing to the object time register (or the Data Storage Area field) which contains the precalculated address. A test is then made of byte 7 of the SUTABC entry to determine if the first subscript has been optimized. If so, the SMT switch in the HCOMPMOD Control Field is turned on, indicating to the arithmetic compiler programs that no code should be generated for any operators in the subscript, and to CP38 that the following subscript has been optimized.

Compiler Program No.38 (CP38)

	CASE A	CASE B
Source Operator:	Comma or]	
Stack Operator:	[	
Stack Operand:	<Switch Identifier>	<Array Identifier>

CASE A: See "Switches".

CASE B: Either source operator is preceded by a subscript expression. The closing bracket,], marks the end of a subscripted variable.

Comma: If the HCOMPMOD switch CMT shows that the subscript was not optimized (CMT=0) in an embracing for statement (if any), code is generated to compute (in register BRR) the product of the subscript value and the address increment factor for the subscript position, viz., $s_i P_{i+1}$, and to add this product to the address. No code is generated if the subscript was optimized (CMT=1). In either case,

byte 7 of the relevant SUTABC entry (Figure 71) is inspected to determine if the next subscript was optimized, and CMT is set accordingly.

]: The same code is generated for the preceding subscript as described above, depending on whether the subscript was optimized or not. In addition, code is generated, if the array identifier is a formal parameter, to verify that the dimensions of the actual and formal parameters are equal. If the TEST option has been specified, code is generated to verify that the address of the array element falls within the reserved storage area for the array. Finally, the operator [and the operands representing the array identifier and the last subscript are released, a switch is made to the decision matrix specified by the next stack operator (also released), and control is passed to SNOTSP.

PROCEDURES

The object code generated for a procedure comprises two or more separate parts, namely:

1. A segment of code representing the declared procedure; and
2. One or more procedure calls, one for each call to the procedure. The procedure call comprises a code sequence for each actual parameter (if any) designated in the call and a branch to the Fixed Storage Area routine PROLOG. When the procedure call is executed, PROLOG acquires a Data Storage Area for the procedure and then stores the addresses of the actual parameter code sequences in the storage fields reserved for the formal parameters in the Data Storage Area. PROLOG then passes control to the first instruction in the procedure.

Figure 63 shows a part of the object code generated for a declared type procedure and for a call to the procedure. The illustration also shows the compiler programs which generate the different parts of the object code.

PROCEDURE DECLARATION

To ensure that a procedure is executed only when specifically called, code is first generated to cause a branch over the procedure body.

The body of a declared procedure begins with a series of two-byte constants (characteristics) indicating the character of the formal parameters (if any) specified in the procedure heading. (Whenever the procedure is called at object time, the characteristics of the actual parameters are compared with the characteristics of the formal parameters, to ensure that the actual parameters satisfy the specifications in the procedure heading.) The characteristic(s) of the formal parameter(s) are followed directly by coding representing the main operative functions of the procedure, unless the procedure contains one or more value-specified parameters. In the latter case, the main body of the procedure is preceded by coding which fetches the value(s) or address(es) of the corresponding value-specified actual parameter(s) and stores the value(s) or address(es) in the storage areas reserved for the value-specified formal parameter(s) in the procedure's Data Storage Area. This operation is accomplished by calling the appropriate actual parameter code sequence(s) in the procedure call (via the Fixed Storage Area routines CAP1 and CAP2) and then calling the Fixed Storage Area routine VALUCALL. The actual parameter code sequence (whose address will have previously been stored by PROLOG in the formal parameter's Data Storage Area field) calculates the value or address of the actual parameter, and loads ADR with the address of the Data Storage Area field containing the value (or, in the case of a specified array with the address of a Storage Mapping Function; or, in the case of a specified label, with the label address). VALUCALL transfers the value (or address) to the formal parameter's Data Storage Area field, displacing the previously stored actual parameter code sequence address.

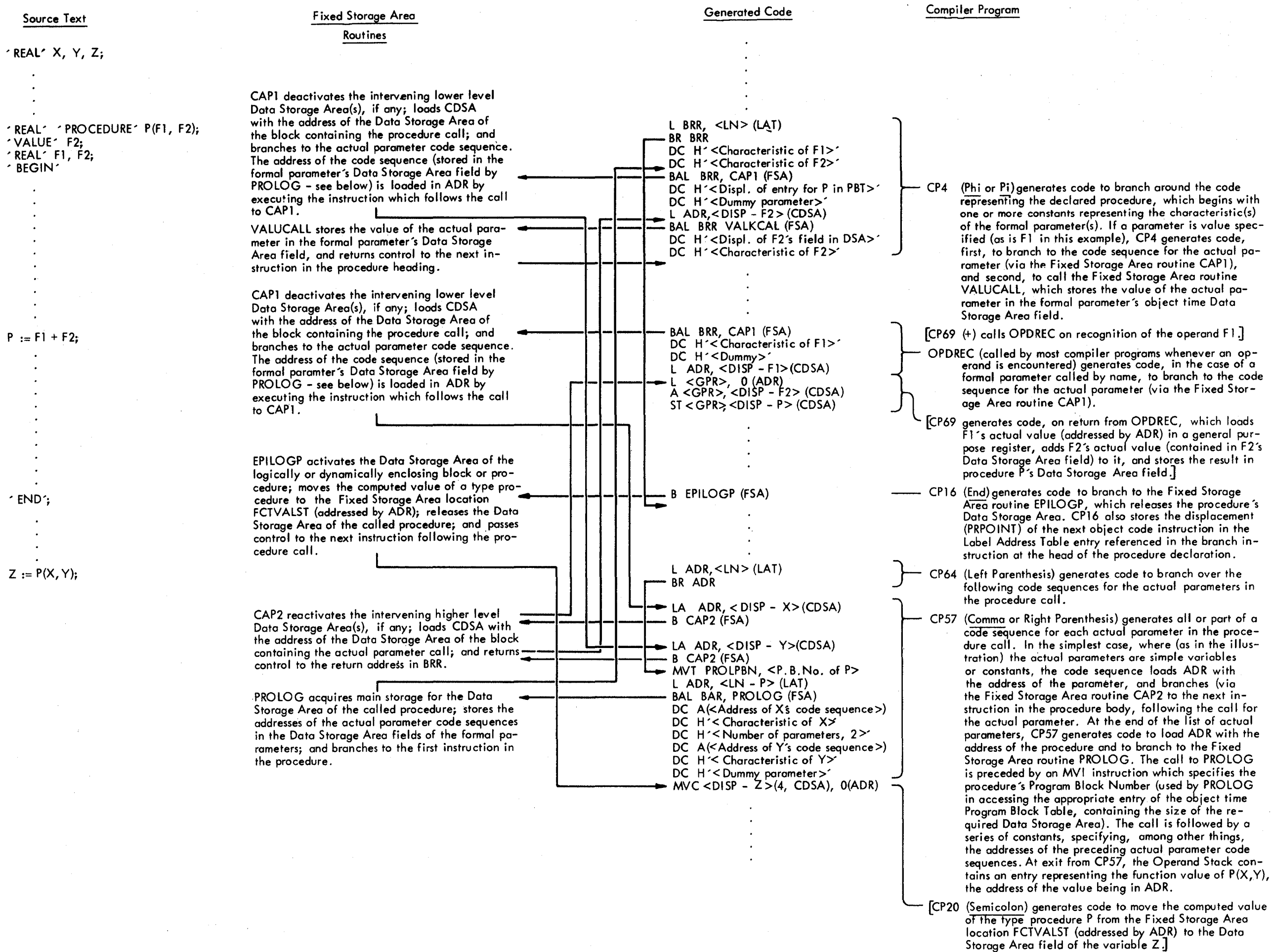


Figure 63. Code generated for declared type procedure and procedure call

Within the procedure body, every formal parameter called by name is represented by a call to the corresponding actual parameter code sequence (via CAP1 and CAP2). The address of the relevant code sequence is obtained from the formal parameter's Data Storage Area field, where it is stored by PROLOG when the particular call for the procedure is executed. In the case of a formal parameter called by value in the procedure body, the actual value or address of the parameter is simply fetched from the formal parameter's Data Storage Area field, where the value or address will have been stored at entry to the procedure.

The close of the procedure body is represented by a branch to the Fixed Storage Area routine EPILOGP. EPILOGP releases the Data Storage Area of the procedure and passes control to the next instruction following the procedure call. If the procedure called is a type procedure, EPILOGP moves the calculated value of the procedure from the appropriate Data Storage Area entry to a standard location in the Fixed Storage Area, before releasing the type procedure's Data Storage Area.

PROCEDURE CALL

The procedure call consists essentially of a call to the procedure by way of the Fixed Storage Area routine PROLOG. Among other things, PROLOG acquires a Data Storage Area for the procedure and then branches to the code representing the procedure. The Program Block Number and the address of the procedure are transmitted to PROLOG by instructions immediately preceding the call. The Program Block Number specifies the appropriate entry in the object time Program Block Table (Figure 84) which contains the size of the Data Storage Area to be acquired by PROLOG for the procedure.

If the procedure call includes any actual parameters, the call to PROLOG is preceded by a code sequence for each actual parameter. The code sequence, which is only executed when called by the procedure, computes the value or the address of the actual parameter (where the actual parameter is not a simple variable or a constant), loads ADR with the address of the actual parameter, and returns control (via the Fixed Storage Area routine CAP2) to the next instruction in the procedure. A branch instruction preceding the actual parameter code sequence(s) ensures that the code sequences are not executed until called by the procedure. The address(es) of the actual parameter code sequence(s) and the characteristic(s) of the actual parameter(s) are stored in a series of constants following the call to PROLOG. The first parameter entry contains the number of actual parameters. PROLOG verifies the compatibility of the formal and actual parameters (by comparing characteristics) and stores each parameter code sequence address and characteristic in the related formal parameter's storage field in the Data Storage Area acquired for the procedure. This enables the appropriate actual parameter code sequence to be accessed and executed whenever the actual parameter is called by the procedure.

Compiler Program No.4 (CP4)

	CASE A	CASE B	CASE C
Source Operator:	Switch	Array	Pi or Phi [stacked]

Stack Operator: Beta, Pi, Phi, or Alpha

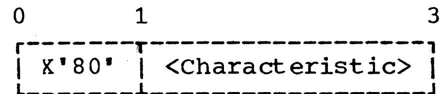
CASE A: See "Switches".

CASE B: See "Arrays".

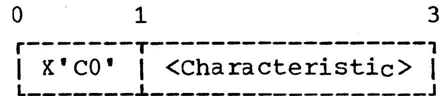
CASE C: The source operator indicates the start of a procedure declaration. Code is first generated to branch around the code subsequently generated for the declared procedure. The branch instruction references a new entry reserved in the Label Address Table, in which the displacement (PRPOINT) of the instruction following the end of the procedure is subsequently inserted by CP16. A scanning operation is then initiated (by call to SNOPDOPR) to locate the operator following the procedure identifier. The operator may be the left parenthesis preceding a formal parameter list, or the operator Delta marking the end of a parameterless procedure heading. In the latter case, control is passed directly to COMP.

If the declared procedure is not parameterless, a series of constants is generated representing the characteristics of the parameters in the formal parameter list. At the same time, an entry is made for each parameter in a table named CBVTAB (Called by Value Table), which accommodates up to 15 three-byte entries. The contents of the CBVTAB entries, for each type of formal parameter are as follows:

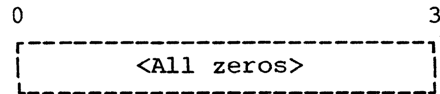
Non-label parameter called by value:



Label parameter called by value:



Parameter called by name:



The characteristics in the entries, which are constructed in the order in which the parameters occur, are copied from the internal names representing the parameters in the Modification Level 2 source text.

When all parameters in the formal parameter list have been processed in the manner indicated, code is generated for every value-called parameter listed in CBVTAB, to fetch the actual parameter value or label address and to store the value or address in the formal parameter's Data Storage Area field. The basic elements of the code generated for a non-label value-called parameter are indicated in Figure 63. In the case of a value-called label parameter, the code consists of a call to the corresponding actual parameter code sequence (via the Fixed Storage Area routines CAP1 and CAP2) followed by instructions which store the actual address (contained in ADR) and the base address (contained in GDSA) of the Data Storage Area of the block where the

label is declared, in the 8-byte storage field reserved for the value-called parameter in the procedure's Data Storage Area. When the end of the declared procedure heading is reached (indicated by the operator Delta), control is passed to COMP.

Operand Recognizer (OPDREC)

See "Subroutine Pool".

Compiler Program No.16 (CP16)

	CASE A	CASE B	CASE C
Source Operator:	Epsilon	End	Epsilon
Stack Operator:	Beta	Begin	Pi or Phi [released]

CASE A and B: See "Blocks and Compound Statements".

CASE C: Epsilon marks the close of a declared procedure.

Code is generated to call the Fixed Storage Area routine EPILOGP. The displacement (PRPOINT) of the next instruction in the object code is stored in the Label Address Table entry referenced by the branch instruction preceding the procedure body, specifying a branch over the procedure (CP4). The procedure type and the number of parameters in the procedure (obtained from the stack operand representing the procedure identifier) are noted in the corresponding entry of Program Block Table III (Figure 60) and a call is made to PBNHDL.

Compiler Program No.64 (CP64)

Source Operator: (
Stack Operator: (See decision matrices -- Appendices V-a, V-b, V-c)

CASE A: The left parenthesis is preceded by an operand representing a procedure identifier and constituting a call for the procedure. An operator is stacked specifying a return to the current decision matrix, the Statement Context Matrix is addressed and code is generated to branch past the code sequence(s) to be subsequently generated (by CP57) for the following actual parameter(s). To specify the address of the first parameter, the displacement (PRPOINT) of the next instruction in the object code is saved in an entry in the Operand Stack. Before control is returned to SNOT, the procedure bracket { is stacked.

CASE B: The left parenthesis is preceded by a standard procedure designator. See "Standard Procedures".

CASE C: The left parenthesis is preceded by an operator. See "Arithmetic Expressions" and "Boolean Expressions".

Compiler Program No.57 (CP57)

Source Operator: Comma or)
Stack Operator: {

Either source operator is preceded by an actual parameter in a procedure call.

CP57 generates all or part of a code sequence for each parameter in the actual parameter list, depending on the type of the parameter. If the actual parameter (an expression) contains any arithmetic or relational operators, the first part of the code sequence (to evaluate the expression) will have been generated before entry to CP57. At the end of the list of actual parameters, CP57 generates a call to the procedure, by way of the Fixed Storage Area routine PROLOG (see Figure 63).

The instructions generated by CP57 in each actual parameter code sequence (each of which terminates with a branch to the Fixed Storage Area routine CAP2) depend on the nature of the actual parameter, represented by the last stack operand.

Type of Actual Parameter	Code Generated (followed in every case by a branch to CAP2)
Formal parameter	Call the actual parameter (call generated by OPDREC).
Integer, real or boolean expression	Load ADR with the address of the value of the expression.
String identifier	Load ADR with the address of the string.
Integer, real or boolean array identifier	Load ADR with the address of the Storage Mapping Function.
Designational expression	Load ADR with the address of the label and GDSA with the address of the Data Storage Area corresponding to the block in which the label is declared.
Switch identifier	Load ADR with the address of the switch and GDSA with the address of the Data Storage Area corresponding to the block in which the switch is declared.
Procedure identifier with no parameters	Call the procedure (call generated by OPDREC).
Procedure identifier with parameters	Load ADR with the address of the procedure. Move the Program Block Number of the procedure to PROLPBN in the Fixed Storage Area. Save registers PBT and LAT in PROLREG.
Standard procedure identifier	Load ADR with the address of a constant in the actual parameter code sequence, representing the last 4 bytes of the standard procedure designator. Move block number 0 to PROLPBN in the Fixed Storage Area.

After each code sequence has been generated, the displacement (PRPOINT) of the next instruction in the object code is stored in an entry reserved in the Label Address Table (addressed by a stack operand) to represent the address of the following code sequence. The addresses thus recorded are stored in the form of address constants in the parameter list following the call to PROLOG (see Figure 63). After the last code sequence has been generated, the displacement of the following instruction is stored in the Label Address Table entry referenced by the branch instruction (generated by CP64) preceding the first code sequence.

CODE PROCEDURES

The term "code procedure" is applied to a declared procedure with a normal procedure heading, but with a procedure body consisting solely of the word 'CODE'. The latter signifies that a precompiled routine with the same name as the procedure identifier in the heading, is to be fetched from the user's library of precompiled procedures.

The generated code corresponding to the heading of a declared code procedure is identical in form to that generated for the heading of any non-code procedure (see Figure 63 and CP4). When the delimiter 'CODE' representing the procedure body is encountered, CP83 generates a call to the Fixed Storage Area routine LOADPP, which loads the precompiled procedure into main storage and records the address of the procedure's entry point in the object time Program Block Table (see Figure 84). The call to LOADPP is executed at entry to the block in which the procedure is declared. This ensures that the precompiled procedure has been loaded into main storage in advance of any call for it in the object module.

When the operator Epsilon, marking the close of the declared code procedure, is encountered, CP16 does not generate a call to EPILOG as in the case of non-code procedures, but simply marks the appropriate entry of Program Block Table III (Figure 60) to show a code procedure and to note the number of parameters. The precompiled procedure is DELETEd (by EPILOG) at exit from the block in which the code procedure is declared.

The object code implementing a call for a code procedure is identical in form with the code for a non-code procedure call. It consists of a call to the Fixed Storage Area routine PROLOG. PROLOG acquires a Data Storage Area for the code procedure, stores the addresses of the actual parameter code sequences in the Data Storage Area fields of the formal parameters, loads ADR with the address of the precompiled procedure (contained in the relevant Program Block Table entry) and branches to the address in ADR. The latter action is taken after determining (by inspection of the Program Block Table entry) that the procedure is a code procedure. The object time registers PBT and LAT are also changed by PROLOG to point to the tables contained in the precompiled procedure load module.

Compiler Program No.83 (CP83)

Source Operator: Gamma

Stack Operator: Pi, Phi, Beta

Gamma represents the body of a declared code procedure. It is followed by an 8-byte unit containing six characters of the code procedure name and two blanks (EBCDIC code).

CP83 generates a call to the Fixed Storage Area routine LOADPP. The call is followed by two parameters: the name of the precompiled procedure and the displacement of the Program Block Table entry for the code procedure.

The displacement (PRPOINT) of the call in the object module is stored in the Label Address Table entry referenced by the branch instruction generated by CP4 (see "Procedures") at the head of the code procedure declaration.

STANDARD PROCEDURES

Compiler Program No.64 (CP64)

Source Operator:
Stack Operator: (See Decision Matrices -- Appendices V-a, V-b, V-c)

CASE A: The source operator is preceded by a procedure identifier. See "Procedures".

CASE B: The source operator is preceded by a standard I/O procedure or mathematical function designator (Appendix III), representing a call for the procedure or function.

To indicate that the standard procedure has been called in the source module, the full word reserved for the address of the standard procedure in the Label Address Table (displacement specified in the last byte of a designator) is flagged, by setting the first bit = 0. Flagging the entry causes an ESD and an RLD record to be generated in the Termination Phase (IEX51) for the called standard procedure or function, ensuring that the Library procedure will be combined with the object module at execution time. After the procedure has been loaded, its entry point address is stored in the Label Address Table entry.

CP64 initiates a count, in a stack operand, of the number of parameters in the standard procedure call. In the same operand the displacement (P) is stored of the next free entry in the current Data Storage Area in which a parameter list will be constructed at object time. The operand is referenced by CP61, which generates the code to construct the parameter list and to call the standard I/O procedure or mathematical function. Before exit to SNOT, the standard procedure bracket ⟨ is stacked.

CASE C: The source operator is preceded by another operator. See "Arithmetic Expressions" and "Boolean Expressions".

Compiler Program No.61 (CP61)

Source Operator: Comma or)
Stack Operator: (

Either source operator is preceded by an actual parameter in a call for a standard I/O procedure or mathematical function.

Except in the case of the standard functions ABS, ENTIER, LENGTH, and SIGN, CP61 generates code which constructs a parameter list (containing an entry for each actual parameter in the procedure call) in the current Data Storage Area, followed by code to load a register with the address of the parameter list and then to branch to the Library procedure concerned, viz.

```
L PARAM, <DISP> (CDSA)
L ENTRY, <LN> (LAT)
BALR RETURN, ENTRY
```

The second instruction fetches the address of the Library procedure from the entry in the Label Address Table whose displacement (<LN>) is specified in the last byte of the standard procedure or function designator (Appendix III). The Library procedure is combined with the object module at execution time, by virtue of the ESD record generated in the Termination Phase (IEX51) for the procedure name (see CP64 above).

The parameter list entry constructed at object time for each actual parameter in a standard procedure or function call consists of a full word containing a code byte and the address of the actual parameter (in the last three bytes). The processing of the actual parameters by CP61 depends, in general, on whether the call is for (1) a standard I/O procedure, or (2) a standard mathematical function.

1. A call for a standard I/O procedure includes two or three actual parameters. The character of these parameters and the parameter list entries constructed at object time are indicated in Chapter 11 under "Input/Output Procedures".

   The I/O operation involved in the standard procedure is noted in the I/O Table (IOTAB -- Figure 64) opposite the data set number (if any) specified by the first parameter in the procedure call. The I/O Table is used in the construction of the Data Set Table (see "Termination Phase" in this chapter).

2. A call for a standard mathematical function includes but one parameter. Execution of the standard function gives an arithmetic result.

   For all standard function calls except those of 'ENTIER', 'ABS', 'LENGTH' and 'SIGN', CP61 generates code to construct a parameter entry in the current Data Storage Area for the actual parameter, and to call the relevant Library routine. In the case of 'ENTIER', the Fixed Storage Area ENTIER routine is called. In the case of 'ABS', 'LENGTH', and 'SIGN', code is generated to perform the function in line.

   Before exit to SNOTSP, the stack operand representing the standard function designator is replaced by an operand representing the function value and pointing to the register which contains the value.

	1 Input	2 Output	2 Sysact 4/13	3 Sysact 8	4 Sysact Undet	5 Sysact Other	6	7
DSN= 0								
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
14								
15								
Undetermined Put/Get								

Figure 64. I/O Table (IOTAB)

FOR STATEMENTS

The logical structure of the code generated for a for statement is governed by the for statement's loop classification (Counting Loop, Elementary Loop or Normal Loop) in the For Statement Table. The For Statement Table, which is constructed in the Scan III Phase (Chapter 6) and transmitted to the Compilation Phase via the Common Work Area, contains a classification byte for every for statement in the source module. The classification byte (Figure 65) not only reflects specific logical characteristics of the for statement, but also specifies (by the pattern of bit-settings in binary positions 0-3) the for statement's loop classification.

Counting Loops

The principal characteristics of the Counting Loop are:

1. The controlled variable does not occur in the iterated part of the for statement (other than in optimizable subscript expressions);
2. The for list is limited to step elements and/or arithmetic elements, and all operands in the for list are constant whithin the for statement;
3. All operands in the for list are of integer type.
4. All subscript expressions contained in the for statement which are functions of the controlled variable are optimized; so also are subscripts consisting of constants or simple variables to which no assignment is made in the for statement. All other subscripts are not optimized.

In the case of a step element, the first two characteristics imply that the loop count (or number of iterations) can be calculated in advance. The formula used in computing the loop count is

$$\text{Loop Count} = \frac{\text{(Test Value-Initial Value+Step Value)}}{\text{Step Value}}$$

Since the loop count can be computed in advance, the iterated statement may be designed as a Branch on Count loop.

Furthermore, since the controlled variable is not a factor in the iterated statement, no assignment need be made to it in each iteration. If the controlled variable occurs in a subscript expression (which must be optimizable), its contribution is pre-calculated in the form of a uniform address increment.

Figures 66 and 68 illustrate the logical structure of the code generated for two Counting Loops, the first containing arithmetic elements, the second containing step elements.

BIT 0=1 if:	the for list contains a while element the real division operator (/) appears in the for list the power operator appears in the for list a real operand appears in the for list the controlled variable appears as a right variable in the iterated part of the for statement (outside optimizable subscript expressions)
BIT 1=1 if:	an assignment is made to the controlled variable in the iterated part of th for statement
BIT 2=1 if:	a label or switch identifier, implying a jump out of the for statement, appears in the iterated part of the for statement
BIT 3=1 if:	an array identifier appears in the for list a procedure identifier or a formal parameter appears in the for statement an assignment is made to any for list identifier in the iterated part of the for statement
(Any one of these conditions qualifies the for statement as a Normal Loop. In this case, all of bits 0,1,2 and 3 are set=1)	
BIT 4=1 if:	the for list contains a while element
BIT 5=1 if:	the for list contains a while element
BIT 6=1	specifies that the for statement contains subscript expressions to be optimized. (The bit is turned on only in the Compilation Phase)
BIT 7=1	specifies that the for list contains two or more elements. (The bit is turned on only in the Compilation Phase)

Loop Classification

COUNTING LOOP:	Bits 0-3 all 0
ELEMENTARY LOOP:	Bits 0-3 mixed 1 and 0
NORMAL LOOP:	Bits 0-3 all 1

Figure 65. For statement classification byte in the For Statement Table

The main features of the code generated for a Counting Loop are:

1. The code sequence representing each for list element is executed once only, and the sequence terminates with a BALR instruction, which branches to the iterated statement and loads the address of the next for list element. In the case of a step element, the code sequence tests for an endless loop and computes the loop count (in register 0), before branching to the iterated statement.

2. The code sequence representing the iterated statement, in the case of a step element, is controlled by a terminal Branch on Count instruction, which returns to the iterated statment, or, if the step element is exhausted, branches to the next for list sequence (or to the exit address).

3. If the for statement contains subscripted variables (arrays), the addresses of the array elements are derived in each iteration (except the first) by the addition of a uniform increment to an initial (or base address, calculated in advance of the first iteration (see "Subscript Optimization" below).

Elementary Loops

The distinguishing characterisitcs of the Elementary Loop are:

1. An assignment may be made to the controlled variable in the iterated statement (the controlled variable may also occur in the iterated statement as a right variable).

2. The for list may contain real operands or expressions containing the real division or power operator.

3. Subscript expressions in the for statement are optimizable, provided no assignment is made to any variable except the controlled variable, in the expression.

If an assignment is made to the controlled variable in the iterated statement, its value after any given number of iterations cannot be predicted without reference to the iterated statement. This implies that, for a step element, the loop count cannot be pre-calculated, and that accordingly, the iterated statement cannot be designed as a Branch on Count Loop. A test for exhaustion of the step element, involving the test value, the step value, and the controlled variable, must be made in each iteration.

This requires that the controlled variable be incremented by the amount of the step value in each iteration. Incrementation of the controlled variable is also required on the ground that the controlled variable may occur in the iterated statement as a right variable as well as a factor in non-optimizable subscript expressions.

Figures 67, 69, 70, and 73 illustrate the logical structure of the code generated for an Elementary Loop, the first containing arithmetic elements, the second step elements, the third containing step elements and an optimizable subscript expression, and the fourth containing while elements.

The main features of the code generated for an Elementary Loop are:

1. The code sequence initiating a step element tests for an endless loop and stores the step and test values in the current Data Storage Area.

2. The controlled variable is incremented once in each cycle of a step element, and a test for exhaustion of the step element, using the stored step and test values and the controlled variable, is made before a branch is taken to the iterated statement. The test is performed by a Fixed Storage Area routine (BCR).

3. If the for statement contains any optimizable subscript expressions, the expressions are optimized by deriving a uniform address increment which is added in each cycle to a pre-calculated base address.

4. No subscript expression is optimizable in the for statement.

Since an assignment may be made to any variable in the for list, the step and test values in a step element may vary between iterations. This implies, first, that the loop count cannot be pre-calculated without reference to the iterated statement; and second, that in each iteration,

a. The step and test values must be calculated,

b. The controlled variable must be incremented, and

c. A test for exhaustion of the step element must be made.

Moreover, since the step value may be a function of the controlled variable, the step value must be calculated twice in each iteration, once when the controlled variable is incremented, and once again, immediately afterwards, in order to determine the sign of the step value. The latter is required in order to perform the test for exhaustion of the step element before branching to the iterated statement, viz:

(Controlled Variable - Test Value)
    *(Sign of Step Value)>0

Figures 67, 72, and 73 illustrate the logical structure of the code generated for a Normal Loop, the first containing arithmetic elements, the second containing step elements, and the third containing while elements.

The main features of the code generated for a Normal Loop are:

1. The step and test values are computed in each iteration, the step value being computed twice (once for incrementing the controlled variable, and once for determining the sign of the step value).

2. The controlled variable is incremented and a test for exhaustion of the for list element is performed in each iteration.

3. Array element addresses are computed by evaluating the full subscript expression(s) in each cycle (subscript optimization is not possible).

Normal Loops

The principal characteristics of the Normal Loop are:

1. An assignment may be made in the iterated statement to any variable in the for list.

2. The step value may be a function of the controlled variable.

3. The for statement may contain a procedure statement (which may change the values of any one or more of the for list variables).

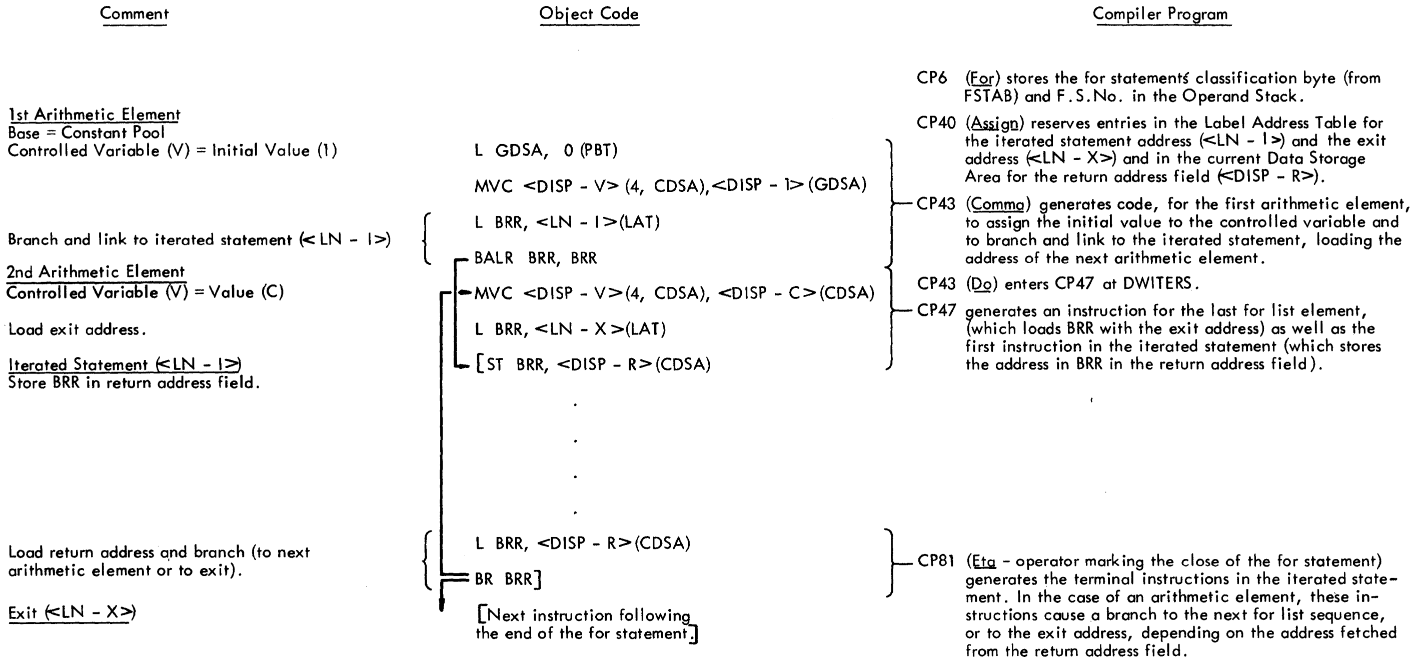


Figure 66. Logical structure of the code generated for a Counting Loop containing arithmetic elements

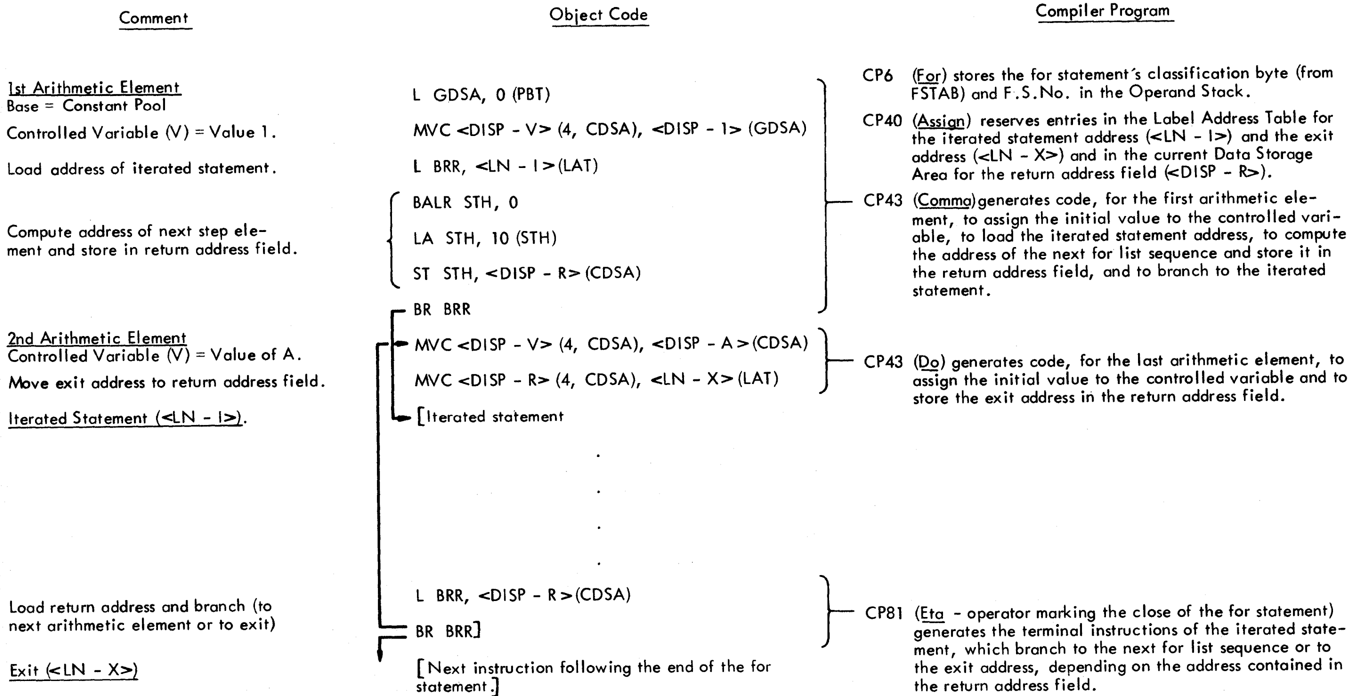


Figure 67. Logical structure of the code generated for an Elementary Loop or Normal Loop containing arithmetic elements

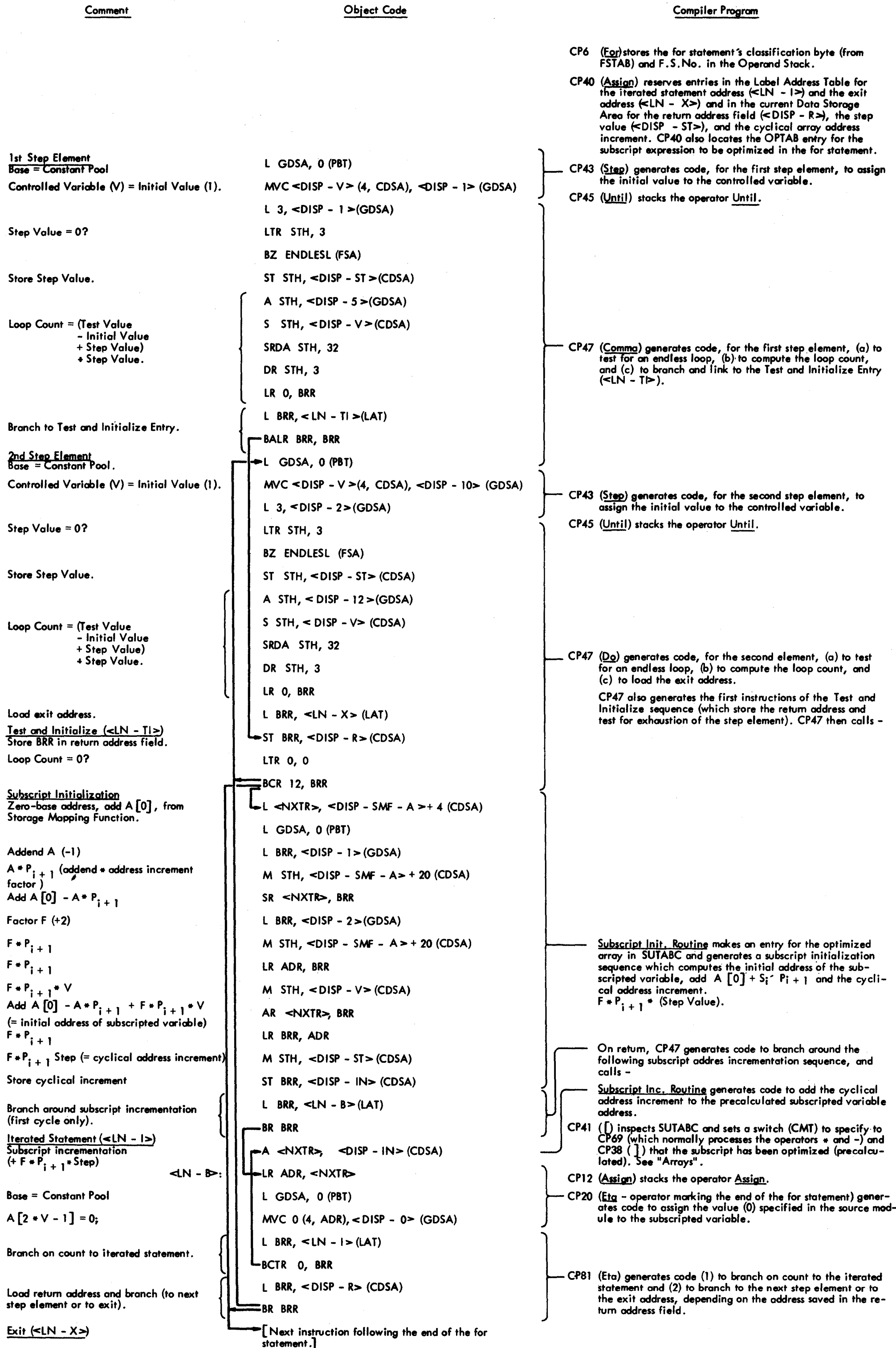


Figure 68. Logical structure of the code generated for a Counting Loop containing step elements and optimizable subscript expression

Subscript Optimization

A subscript expression of an array identifier contained in the iterated part of an embracing for statement is defined to be optimizable in that for statement if the expression is of the form

$$\pm F*V\pm A,$$

where F(Factor) is an integer variable or constant, V is the controlled variable of the embracing for statement, and A(Addend) is an integer variable or constant. Two conditions for optimization of a subscript expression of the above type are:

1. That the embracing for statement be a Counting Loop or an Elementary Loop; and
2. That no assignment be made in the iterated statement to any variable in the subscript expression.

In the general case, the address of any given array element, $A[s_1,s_2,s_3]$ is given by

$$addA[s_1,s_2,s_3]=addA[0,0,0]+s_1P_2+s_2P_3+s_3P_4$$

where addA[0,0,0] is the array's zero-base address and $s_i P_{i+1}$ is the product of the subscript and the address increment factor for the subscript position. The zero-base address and the address increment factors are obtained from the array's Storage Mapping Function (Figure 62 - see "Arrays"). The product $s_i P_{i+1}$ represents the contribution of the particular subscript to the displacement of the array element from the zero-base address.

The displacement contribution of any linear (optimizable) subscript of the form (F*V+A) is

$$s_i P_{i+1}=\{F*V+A\}P_{i+1}.$$

The change in the displacement contribution associated with a change (or "step") in the value of the controlled variable V is

$$\begin{aligned}\Delta s_i P_{i+1} &= \\ &=\{F*(V+Step)+A\}P_{i+1} \\ &-\{F*V+A\}P_{i+1} \\ &=F*Step*P_{i+1}.\end{aligned}$$

If the controlled variable V changes by a constant step value in a succession of iterations, the change in the subscript's displacement contribution, $F*Step*P_{i+1}$, is constant in each iteration. If $s_i' P_{i+1}=(F*V'+A)P_{i+1}$ is the subscript's displacement contribution in the first iteration (where V' is the initial value of the controlled variable), the displacement contribution in the nth iteration is

$$s_i' P_{i+1}+(n-1)\Delta s_i P_{i+1} =(F*V'+A)P_{i+1}+(n-1)(F*Step*P_{i+1}).$$

An equivalent form is

$$\text{(a)}\quad \begin{aligned}&\{s'P_{i+1}+(n-2)\Delta s_i P_{i+1}\}+\Delta s_i P_{i+1} \\ &=\{(F*V'+A)P_{i+1}+(n-2)(F*Step*P_{i+1})\} \\ &+F*Step*P_{i+1}.\end{aligned}$$

Equation (a) expresses the subscript optimization formula, which states that, for an optimizable subscript:

1. the change in the subscript's displacement contribution is constant in each iteration, if the change (or step) in the controlled variable is constant, and is given by $\Delta s_i P_{i+1}=F*Step*P_{i+1}$ (called the cyclical address increment).
2. the subscript's displacement contribution in each iteration is obtained by adding the cyclical address increment, $F*Step*P_{i+1}$, to the subscript's displacement contribution in the preceding iteration, viz:

   $$(F*V'+A)P_{i+1}+(n-2)(F*Step*P_{i+1}).$$

The address of the array element $A[s_1,s_2,s_3]$ in the nth iteration, where subscripts s_1 and s_2 are optimizable and subscript s_3 is non-optimizable, may be expressed as

$addA[s_1,s_2,s_3]$

in nth iteration

$$=\{addA[0,0,0]+s_1'P_2+s_2'P_3+(n-2) *(\Delta s_1P_2+\Delta s_2P_3)\}+\Delta s_1P_2+\Delta s_2P_3+s_3P_4.$$

This states that the address of a subscripted variable containing one or more optimized subscripts is obtained in each iteration of a step element, by adding the cyclical address increments of the optimized subscripts, viz. $\Delta s_i P_{i+1}=F*Step*P_{i+1}$, together with the displacement contributions of the non-optimizable subscripts, viz. $s_i P_{i+1}$, to a pre-calculated address element, viz. the expression in braces {...}. The latter represents the sum of the array's zero-base address, addA[0,0,0], plus the displacement contributions of the optimized subscripts in the first iteration, $s_i' P_{i+1}=(F*V'+A)P_{i+1}$, plus the cumulative total of the cyclical address increments, $\Delta s_i P_{i+1}=F*Step*P_{i+1}$, added in

the preceding iterations for all optimized subscripts.

In the generated object code, the optimization of subscript expressions comprises two phases: Subscript Initialization and Subscript Incrementation.

Subscript Initialization (illustrated in Figures 65 and 70) is performed before entry to the iterated statement. It consists in computing (in any available general purpose register <NXTR>) the sum of the array's zero base address and the displacement contributions of the optimized subscripts for the first iteration, thus (continuing the example above)

<NXTR>=addA[0,0,0]+$s_1' P_3 + s_2' P_3$;

and in deriving and storing (in a field in the current Data Storage Area, <DISP-IN>) a cyclical address increment, representing the sum of the cyclical displacement increments of all optimized subscripts, to be added to <NXTR> in each subsequent iteration, thus

<DISP-IN>=$\Delta s_1 P_3 + \Delta s_2 P_3$.

Subscript Incrementation consists in adding the cyclical address increment to <NXTR>, thus

A <NXTR>, <DISP-IN>(CDSA).

Where the subscripted variable contains a non-optimized subscript (as in the example above), the displacement contribution for the non-optimized subscript is added to <NXTR> after the code to evaluate the product of the full expression and the subscript increment factor, viz. $s_i P_{i+1}$, is executed inside the iterated statement.

<u>Compiler Program No.6 (CP6)</u>

	CASE A	CASE B
Source Operator:	<u>Goto</u>	<u>For</u>
Stack Operator:	<u>Begin</u>, <u>Semicolon</u>, <u>Then-s</u> or <u>Else-s</u>	<u>Do</u>,

CASE A: See "Goto Statements".

CASE A: <u>For</u> marks the beginning of a for statement. The operator is stacked and the Statement Context Matrix addressed. Three Operand Stack entries are reserved, in the last of which the for statement's classification byte (OPTBYTE-Figure 65) and For Statement Number are stored.

<u>Compiler Program No.40 (CP40)</u>

Source Operator: <u>Assign</u>
Stack Operator: <u>For</u>

The <u>Assign</u> operator follows the controlled variable, whose internal name has been entered in the Operand Stack.

CP40 stacks the operator <u>For:=</u> and reserves two entries in the Label Address Table and one or more storage fields in the current Data Storage Area (depending on the for statement's loop classification, indicated by the classification byte stored by CP6 in the stack), and stores the displacements of these entries in the stack operands reserved by CP6. The Label Address Table entries, in which the relative address of the iterated statement and the exit address are subsequently inserted, will be referenced by instructions generated subsequently by other compiler programs (see Figures 66-70 and 72, 73). The Data Storage Area fields reserved will be used at object time for storing the return address and the conditional entry address.

CP40 also searches the Optimization Table to determine if the table contains any entries for optimizable subscript expressions contained in the for statement (an entry is identified by comparing the For Statement Number previously entered in the stack by CP6, with the For Statement Number in the first byte of the Optimization Table entry - Figure 50). If an entry is found, bit 6 (OPTB) of the classification byte in the stack is turned on, to indicate that code to optimize the subscript expression is to be generated.

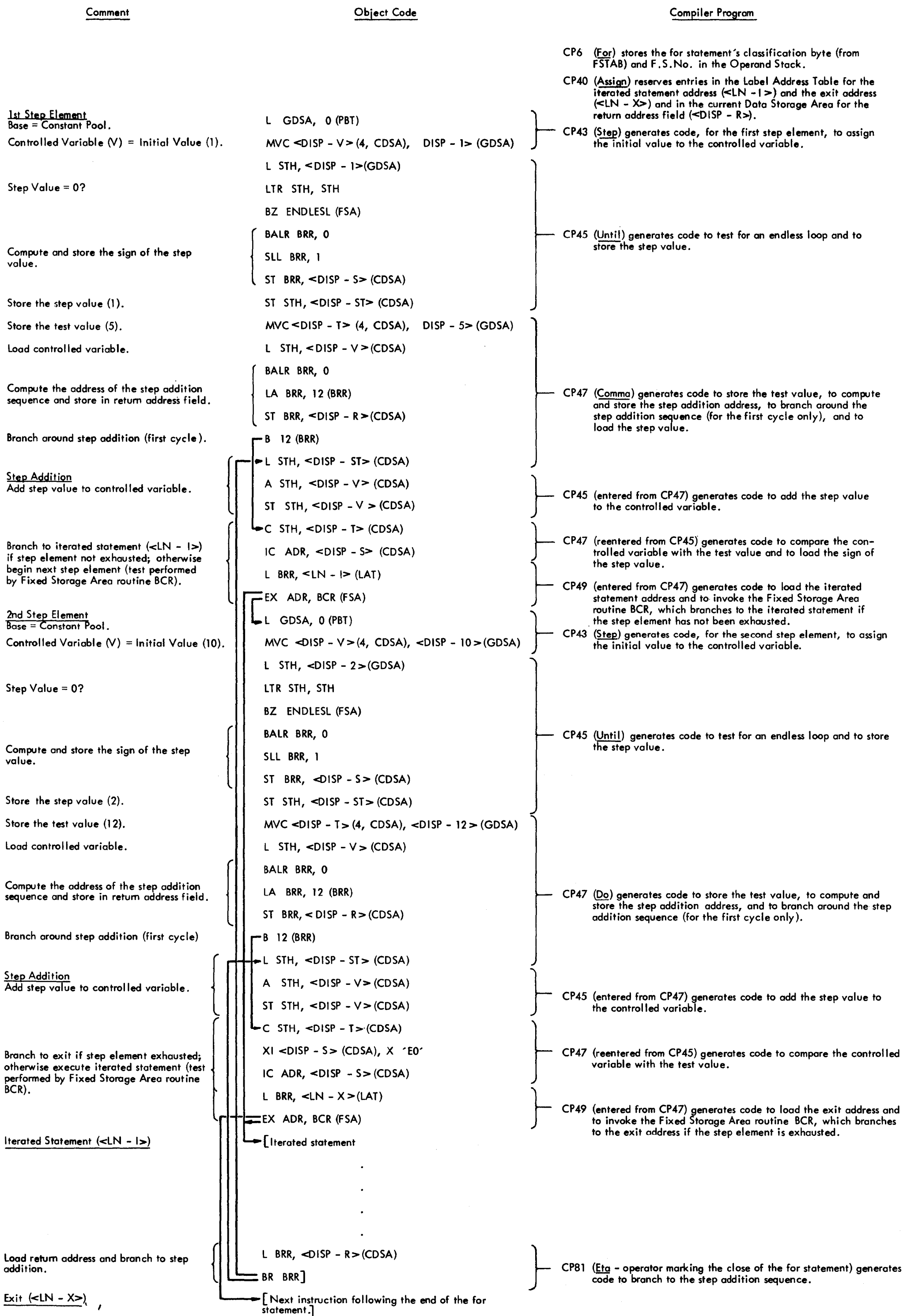


Figure 69. Logical structure of the code generated for an Elementary Loop containing step elements

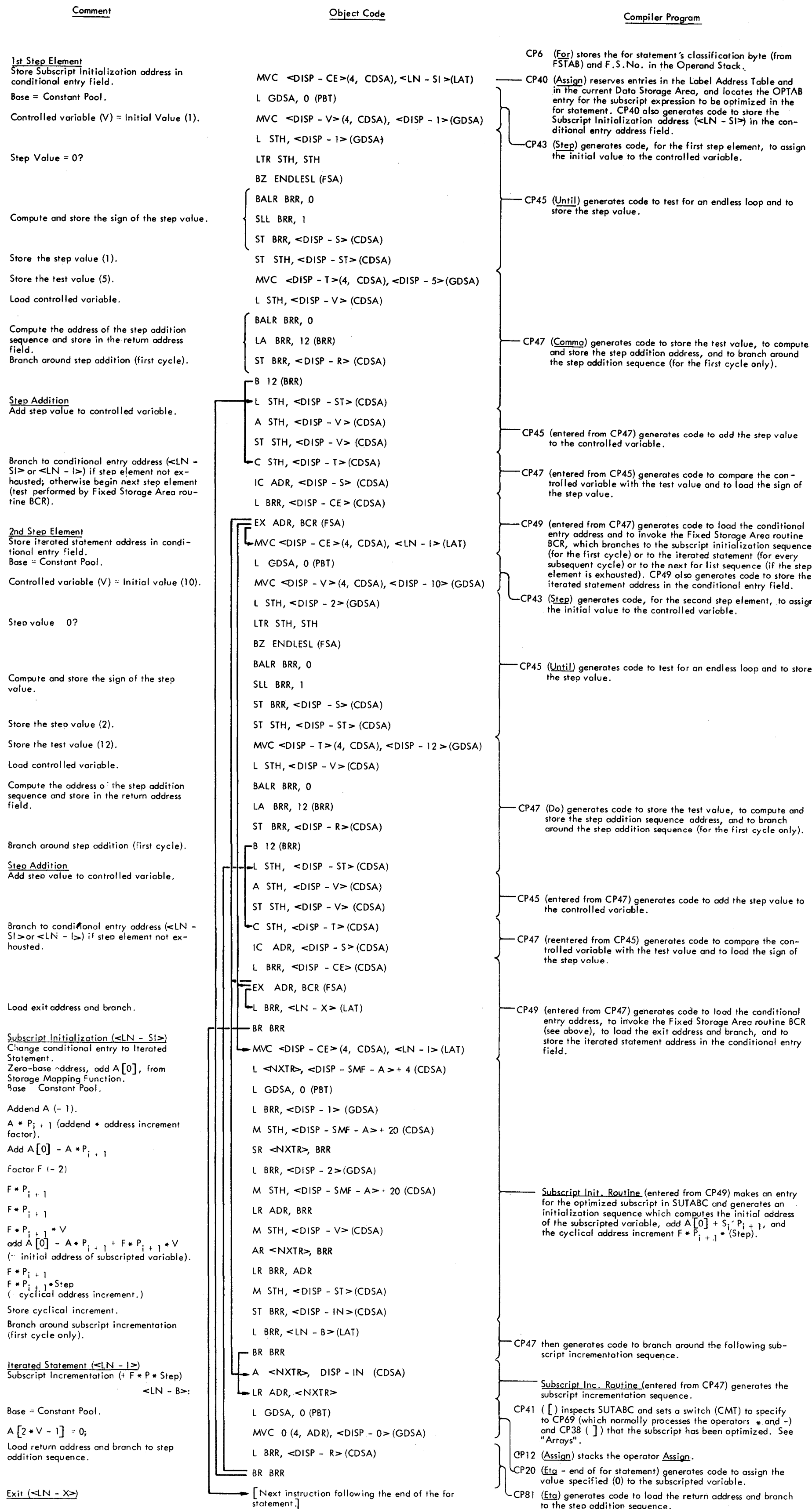


Figure 70. Logical structure of the code generated for an Elementary Loop containing step elements and an optimizable subscript expression

Compiler Program No.43 (CP43)

Source Operater: Step, While, Do, or Comma
Stack Operater: For:=

The source operator follows the initial value (represented by the last stack operand) to be assigned to the controlled variable.

Except in the case of a Counting Loop containing no subscript expressions (indicated by bit 6, OPTB, in the for statement's classification byte), code is generated (by branching to CP20) to assign the initial value to the controlled variable (Figures 67-70 and 72, 73).

Depending on the source operator and the loop classification, code is then generated as illustrated in the figures indicated below.

Comma (end of an arithmetic element)
- Counting Loop: Figure 66.
- Elementary or Normal Loop: Figure 67.
- Elementary Loop (with optimization): Figure 70.

Do (end of an arithmetic element and of the for list)
- Counting Loop: CP47 is entered at DWITERS. Figure 66.
- Elementary Loop: Figure 67.

Step
- Counting or Elementary Loop: Figure 68-70 (no code).
- Normal Loop: Figure 72.

While
- Elementary or Normal Loop: Figure 73.

The stack operand representing the initial value of the controlled variable is released, and, except in the case of the Comma, the source operator is stacked.

Compiler Program No.45 (CP45)

Source Operator Until
Stack Operator Step

Until is preceded by the step value, represented by the last operand in the stack.

Depending on the for statement's loop classification, code is generated as illustrated in the figures indicated below:

Counting Loop: Figure 68 (no code).
Elementary Loop: Figures 69 and 70.
Normal Loop: Figure 72.

In every case, the operator Until is stacked, replacing the stack operator Step.

CP45 is also entered (at DVE2 and DVH3) from CP47. See Figures 69 and 70.

Compiler Program No.47 (CP47)

Source Operator: Comma or Do
Stack Operator: Until

The source operator is preceded by the test value, represented by the last stack operand.

Depending on the for statement's loop classification, code is generated as illustrated in the figures indicated below:

Counting Loop: Figure 68. The figure illustrates a Counting Loop containing step elements and an optimized subscript expression. As indicated in the figure, the subscript initialization and subscript incrementation sequences are generated at the end of the for list (indicated by Do) by entry to the Subscript Initialization Routine (DWG3) and the Subscript Incrementation Routine (UVA1) -- see below. Where subscript optimization is not required (Bit 6 of the classification byte = 0), these routines are not entered.

Elementary Loop: Figures 69 and 70. Both figures illustrate a Counting Loop containing step elements, but Figure 70 shows a Counting Loop containing in addition an optimizable subscript expression. As indicated in the figures, CP47 enters CP45 (at DVE2 or DVH3, depending on whether the controlled variable is integer or real) and exits to CP49 (at EMG1). The latter calls the Subscript Initialization and Incrementation routines, where necessary.

Normal Loop: Figure 72. The figure shows that CP47 exits to CP49 (at EMG1).

CP47 is also entered (at DWITERS) from CP43 (Figure 66).

Compiler Program No.49 (CP49)

Source Operator: Comma or Do
Stack Operator: While

The source operator is preceded by a boolean expression, representing the condition specified in the while element. The for statement must be an Elementary or Normal Loop. Figure 73 illustrates the code generated for either of these loop classifications, where the source operator is Do, marking the end of the for list, and where the for statement (an Elementary Loop) contains no optimizable subscript expressions. The code generated in the case of the Comma operator is identical, except that the address loaded before the conditional branch is that of the iterated statement. Where an Elementary Loop contains optimizable expressions, the code generated by the Subscript Initialization and Incrementation routines (USA1 and UVA1 -- see below), on call from CP49 is similar to that illustrated in Figure 70.

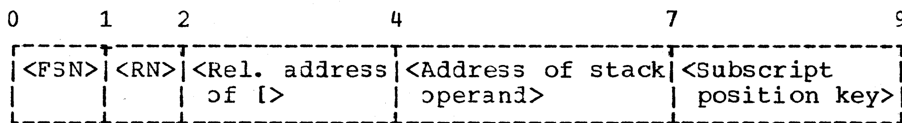
CP49 is also entered (at EXITERS) from CP43 and (at EMG1) from CP47 (see Figures 69, 70 and 72).

Subscript Initialization Routine (DWG3 or USA1)

This routine is entered from CP47 and CP49 at the close of a for list (indicated by the source operator Do), when it is determined (by inspection of bit 6 of the for statement's classification byte 'OPTBYTE' entered in the operand stack by CP6) that the iterated statement contains a subscript expression to be optimized. On recognition of the operator For, CP6 will have located the first of one or more entries in the Optimization Table (Figure 50) representing the subscript expression(s) to be optimized in the for statement.

The Subscript Initialization Routine constructs an entry in the Subscript Table-C (SUTABC), Figure 71, for every subscripted variable containing optimizable subscript expressions represented by entries in the Optimization Table, provided no previous entry was made for the same subscript in an enclosing for statement or in the current for statement, and generates a subscript initialization sequence (see "Subscript Optimization" above, and Figures 68 and 70).

Subscript Table-C is referenced by CP41 and CP38 (see "Arrays"), which are entered whenever the operators [and Comma in a subscripted variable are encountered. Its function is to enable CP41 and CP38 to identify the subscript expressions (if any) in a subscripted variable which have been optimized, and, if any subscripts have been optimized, to enable CP38 to locate the stack operand which specifies the object time register (<NXTR>) containing the pre-calculated array element address.



0	1	2	4	7	9
<FSN>	<RN>	<Rel. address of [>	<Address of stack operand>	<Subscript position key>	

<FSN> = <For Statement Number>
<RN> = <Number of the Modification Level 2 text record containing the operator [which precedes the first subscript of the subscripted variable in the iterated statement>
<Rel. address of [> = <Relative address of the operator [in the text record specified by <RN> above>
<Address of stack operand> = <Address of the stack operand representing the pre-calculated array element address>
<Subscript position key> - (16 bits representing subscript positions 0-15. Bit=1 if the subscript has been optimized.)

Figure 71. Entry in Subscript Table-C (SUTABC)

Source Text: 'FOR' V := 1 'STEP' 1 'UNTIL' 5, 10 'STEP' 2 'UNTIL' 12 'DO' 'BEGIN'..........'END

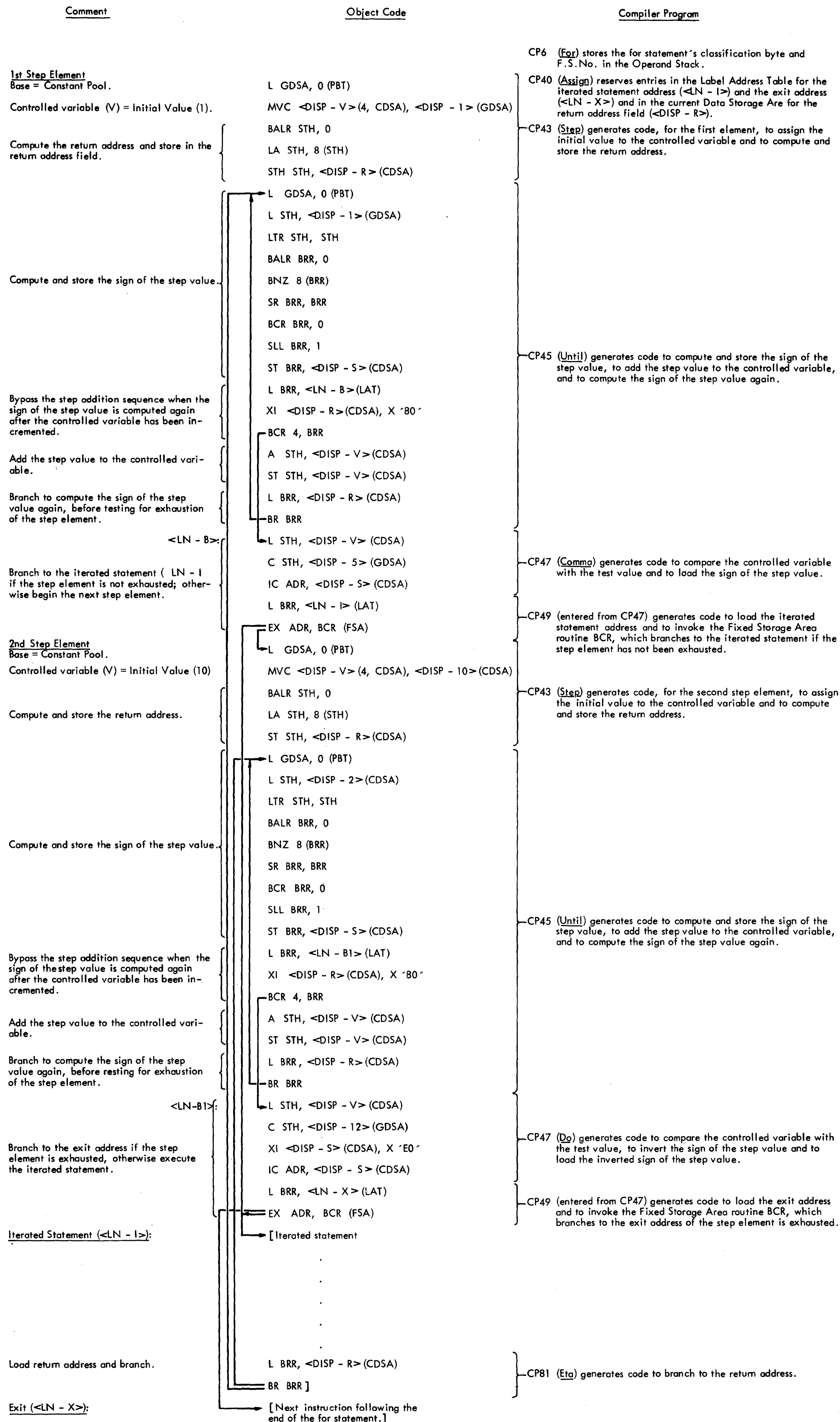
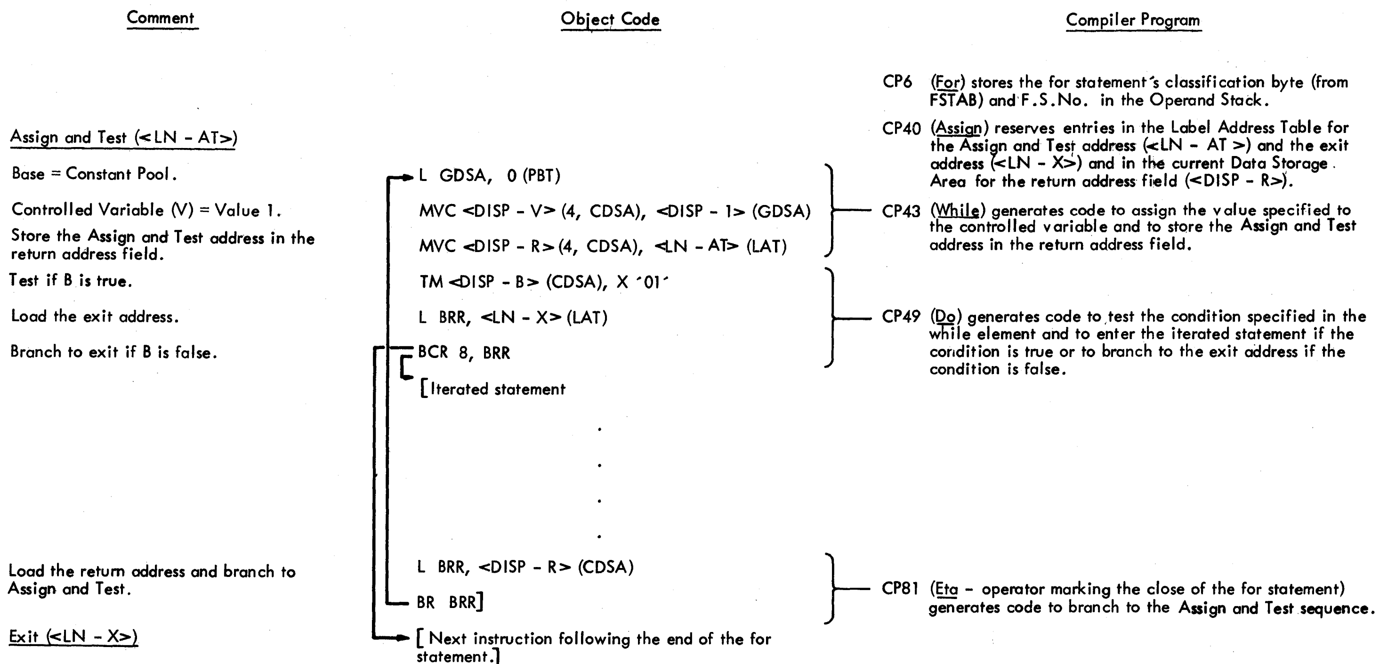


Figure 72. Logical structure of the code generated for a Normal Loop containing step elements



•Figure 73. Logical structure of code generated for Elementary Loop or Normal Loop containing a while element

Initially, a search is made to determine if SUTABC contains any entries (indicating one or more optimized subscripts in an enclosing for statement) and, in this event, if there is an entry for the same subscripted variable (determined by comparing the record number and relative address in bytes 11-13 of the OPTAB entry previously located by CP40, with bytes 1-4 of the SUTABC entry). The action taken depends on the result of this test:

1. No entry for the subscripted variable is found in SUTABC.

   A new entry is constructed in SUTABC, the contents of bytes 11-13 of the OPTAB entry being copied into bytes 1-3 of the SUTABC entry, the current For Statement Number into byte 0. An object time register (<NXTR>) is reserved in which the pre-processed array element address will be calculated, and the address of a stack operand representing the pre-processed address is entered in the SUTABC entry.

   Code is generated to load <NXTR> with the array's zero-base address (addA[0,0,0]), the address of the array's Storage Mapping Function (Figure 62) being obtained from the OPTAB entry.

For every OPTAB entry which contains the same address data in bytes 11-13 (all such entries representing optimizable subscript expressions of the same subscripted variable), the corresponding bit in bytes 7 and 8 of the SUTABC entry is turned on (to specify the optimized subscript position) and code is generated to add the product (Addend) * (Address Increment Factor), $A*P_{i+1}$ to <NXTR> and to add the product (Factor)*(Address Increment Factor), $F*P_{i+1}$ to ADR. When all of the OPTAB entries for the same subscripted variable have been processed in this way, code is generated to multiply the contents of ADR by the initial value of the controlled variable and to add the result to NXTR, which now contains the quantity $addA[0,0,0]+ A*P_{i+1}+\sum(F*P_{i+1})*V'$. Code is then generated to multiply the contents of ADR by the step value and to store the result, $(F*P_{i+1})*Step$, representing the

cyclical address increment, in the current Data Storage Area.

If any other OPTAB entries are found relating to another subscripted variable in the current for statement, a new SUTABC entry is constructed and the pre-processed address and cyclical address increment are computed in the manner described.

2. An entry for the same subscripted variable is found in SUTABC. This indicates that a subscript initialization sequence was generated in an enclosing for statement for one or more subscripts of the same subscripted variable. In this case, the same SUTABC entry is used, the contents of byte 0 being overwritten with the current For Statement Number, and code is generated to load <NXTR> with the previously calculated array element address, namely

   add $A[0,0,0]+\Sigma A*P_{i+1}+\Sigma(F*P_{i+1})*V'$.

   The object time location of this pre-processed address is determined with the aid of the operand address in the SUTABC entry.

Subscript Incrementation Routine (UVA1)

This routine is entered from CP47 and CP49 when address incrementation is required for one or more optimized subscripts (the for statement contains a step element). Code is generated to add the cyclical address increment, $\Sigma(F*P_{i+1})*Step$, to the pre-processed array element address. See Figures 68 and 70.

Compiler Program No.81 (CP81)

Source Operator: Eta
Stack Operator: Do

Eta marks the close of the current for statement. CP81 generates the terminal instructions of the iterated statement (see Figures 66-70 and 72, 73) and deletes all entries in SUTABC. All stack operands relating to the current for statement are released and the operator Do is released.

ASSIGNMENT STATEMENTS

An assignment statement is implemented essentially by a MOVE instruction or a STORE instruction, whose effect is to transfer the value of the expression to the right of the assignment operator to the Data Storage Area field of the operand to the left of the assignment operator. The expression on the right may be

1. a simple variable or a constant whose object time value is contained in a Data Storage Area field, or
2. a complex expression, whose value may be contained in a register or a Data Storage Area field.

Compiler Program No.12 (CP12)

Source Operator: Assign
Stack Operator: Begin, Semicolon, Then-s, Else-s, or Do

The source operator identifies the beginning of an assignment statement.

CP12's function is to test the characteristic of the left variable (represented by the stack operand) for assignability, and to stack the Assign operator. If the operand is a formal parameter, in which case assignability can only be determined at object time, CP12 generates code to check for assignability by inspecting the characteristic in the relevant actual parameter code sequence (see "Procedures"). Thereafter, a call is made to OPDREC which generates code to call the actual parameter.

Compiler Program No.21 (CP21)

Source Operator: Assign
Stack Operator: Assign

The operators identify a multiple assignment, e.g., a:=b:=c.

Unless one or both of the last two stack operands are all-purpose operands, control is passed to CP12, in which the last operand is tested for assignability. The operand before last will have been tested previously.

Compiler Program No.20 (CP20)

Source Operator: Semicolon, Epsilon, Eta, End or Else
Stack Operator: Assign

The combination of source and stack operators indicates the end of an assignment statement. The last two stack operands represent the operands to the left and right of the assignment operator.

CP20's function is to determine if the operands are compatible (i.e., real-real, integer-integer, real-integer, or integer-real, or boolean-boolean) and, if one is real and the other integer, to generate a call to the appropriate Fixed Storage Area routine to convert the right operand to the same type as that of the left operand. The call, where required, is generated by TRINRE or TRREIN.

If both operands are boolean, and if the right operand is a boolean constant, the assignment of the value of the right operand to the left variable is implemented by an MVI instruction. In all other cases (i.e., where the operands are a combination of real and/or integer, or where both operands are boolean, the right operand being a boolean variable), the assignment is implemented by entry to the Real-Real or Integer-Integer routine of Compiler Program No.69 (CP69). The latter routines generate code to store (or move) the value of the right operand (depending on whether the latter is contained in a register) to the Data Storage Area field of the left variable (the object-time address of which is contained in the stack operand before last). A boolean assignment is handled by the Real-Real routine, which generates the necessary move instruction (in the object code, boolean operands are at no point loaded into registers).

At re-entry to CP20 from CP69, the Assign operator is released, and unless the preceding stack operator is For, ₵, or Assign, control is passed to COMP, after the last two stack operands have been released. The operator For indicates that CP20 was entered from CP43 for the special case of an assignment to the controlled variable in a for statement (see CP43 under "For Statements"). The operator ₵ indicates another special case, in which CP20 is used in the generation of code for an array declaration (see CP51 under "Arrays"). The operator Assign indicates a multiple assignment, e.g., a:=b:=c, where the assignment a:=b remains to be implemented. The remaining assignment is generated by branching back to a point (BIE4) within CP20, after moving the last operand downward (replacing b by c) so as to specify the equivalent assignment a:=c.

CONDITIONAL STATEMENTS

The implementation of a conditional statement in the code generated by the compiler may be demonstrated by the following example:

```
...;'IF' B>C 'THEN' A:=B+C 'ELSE' A:=B-C;...
```

Source Operator		Object Time Action	Compiler Program
'IF'		Store occupied registers	CP8 stacks If-s and switches to EXC
>			CP67 stacks >
'THEN'		Evaluate (B>C). Store True or False in Data Storage Area field.	CP69 releases >
		Branch to E (below) if B>C is False	CP78 replaces If-s by Then-s and switches to PGC
:=			CP12 stacks Assign
+			CP22 switches to EXC
			CP66 stacks +
'ELSE'		Compute (B+C)	CP69 releases +
			CP70 switches to STC
			CP71 switches to PGC
		Store (B+C) at A	CP20 releases Assign
		Branch to F (below)	CP17 replaces Then-s by Else-s
:=			CP12 stacks Assign
-			CP22 switches to EXC
			CP66 stacks - (minus)
;	E:	Compute (B-C)	CP69 releases - (minus)
			CP70 switches to STC
			CP70 switches to PGC
		Store (B-C) at A	CP20 releases Assign
	F:	[Next instruction]	CP18 releases Else-s

The symbols EXC, PGC and STC represent, respectively, the Expression Context Matrix, the Program Context Matrix, and the Statement Context Matrix (Appendix V). The special operators If-s, Then-s, and Else-s (see Appendix I-d) identify the delimiters 'IF', 'THEN', and 'ELSE' as relating to a conditional statement, as opposed to the same delimiters occurring in boolean or conditional expressions.

Compiler Program No.8 (CP8)

Source Operator: If
Stack Operator: Begin, Semicolon, Else-s, or Do

In the context identified by the stack operator, If marks the beginning of a conditional statement.

After stacking the operator If-s and addressing the Expression Context Matrix, a call is made to the CLEARRG subroutine, which generates code to store all object-time registers in use.

Compiler Program No.78 (CP78)

	CASE A	CASE B
Source Operator:	Then	
Stack Operator:	If-s	If

CASE A: Then marks the end of an if clause ('IF' (boolean expression) 'THEN') in a conditional statement.

Code is generated to test the value of the preceding boolean expression (which may be a boolean variable or constant, a boolean function designator, a relation or a more complex boolean expression) and, if the value is False, to branch to the first instruction representing the alternative statement following 'ELSE' (or if there is no alternative statement, to the first instruction representing the next sequential statement). The branch instruction references an entry in the Label Address Table (reserved by CP78) in which the relative branch address will be stored by CP17. Unless the boolean expression preceding 'THEN' is a simple boolean variable or constant, code will have been generated by other compiler programs, before entry to CP78, to evaluate the expression.

At entry to CP78, the stack operand addresses a Data Storage Area field in which the value of the boolean expression will be stored at object time.

Before control is returned to SNOT, the stack operator If-s is replaced by Then-s and the Program Context Matrix is addressed.

CASE B: See "Conditional Expressions".

Compiler Program No.17 (CP17)

Source Operator: Else
Stack Operator: Then-s

Else precedes the second alternative in a conditional statement.

CP17 replaces the stack operator Then-s by Else-s and generates code to branch around the immediately following sequence representing the second alternative statement. The code references a new entry in the Label Address Table in which the relative branch address will be stored by CP18. CP17 also stores the relative address (PRPOINT) of the second alternative code sequence in the Label Address Table entry (addressed by a stack operand) previously reserved by CP78.

Compiler Program No.18 (CP18)

Source Operator: Semicolon, Epsilon, Eta, or End
Stack Operator: Then-s or Else-s

The source operator marks the end of a conditional statement.

CP18 releases the stack operator, stores the displacement (PRPOINT) of the next object code instruction in the Label Adress Table entry reserved by CP17 (or CP78), and exits to COMP.

CONDITIONAL EXPRESSIONS

The implementation of a conditional expression in the code generated by the Compiler may be demonstrated by the following example:

```
...; A:=B+('IF'B>C'THEN'C'ELSE'-C);...
```

Source Operator	Object Time Action	Compiler Program
:=		CP12 stacks *Assign*
+		CP22 switches to EXC
		CP66 stacks +
(		CP64 stacks (
'IF'	Store occupied registers	CP80 stacks *If*
>		CP67 stacks >
'THEN'	Evaluate (B>C). Store *True* or *False* in Data Storage Area field	CP69 releases >
	Branch to E(below) if (B>C) is *False*	CP78 replaces *If* by *Then*
'ELSE'	Load C. Branch to F(below)	CP87 replaces *Then* by *Else*
-(minus)		CP66 stacks - (minus)
)	E: Load -C	CP69 releases - (minus)
	Transfer -C to same register as C	CP79 releases *Else*
;	F: Compute (B+/-C)	CP68 releases (
		CP69 releases +
		CP70 switches to STC
		CP71 switches to PGC
	A:=(B+/-C)	CP20 releases *Assign*

The abbreviations EXC, STC, and PGC represent the Expression Context Matrix, the Statement Context Matrix and the Program Context Matrix, respectively.

Compiler Program No.64 (CP64)

Source Operator:
Stack Operator: (See decision matrices)

CASE A: See "Procedures".

CASE B: See "Standard Procedures".

CASE C: The source operator precedes a conditional, boolean or arithmetic expression. The source operator is stacked.

Compiler Program No.80 (CP80)

Source Operator: If
Stack Operator: (

The combination of operators indicate that If opens a conditional expression enclosed by parentheses.

Code is generated, by call to CLEARRG, to store all occupied object time registers, and If is stacked.

Compiler Program No.34 (CP34)

Source Operator: If
Stack Operator: (See Statement Context Matrix -- Appendix V-b)

If opens a conditional expression. If is stacked and the Expression Context Matrix is addressed.

Compiler Program No.65 (CP65)

	CASE A	CASE B
Source Operator:	If or	Not
Stack Operator:	If or If-s	(See Expression Context Matrix -- App. V-c)

CASE A: If opens a conditional expression inside an if clause. If is stacked.

CASE B: See "Boolean Expressions".

Compiler Program No.78 (CP78)

	CASE A	CASE B
Source Operator:		Then
Stack Operator:	If-s	If

CASE A: See "Conditional Statements".

CASE B: Then follows a boolean expression in a conditional expression.

CP78 generates code to test the value of the immediately preceding boolean expression and to branch to the first instruction representing the second alternative expression following 'ELSE', if the value is False. Unless the boolean expression consists solely of a boolean variable or constant, code will have been generated, before entry to CP78, to evaluate the expression and to store the value in a Data Storage Area field. The object time location of the stored value is addressed by the stack operand. The generated code references a new entry in he Label Address Table, in which the relative address of the alternative statement will be subsequently stored. Before exit to SNOT, the stack operator If is replaced by Then.

Compiler Program No.87 (CP87)

Source Operator: Else
Stack Operator: Then

Else follows a designational, arithmetic or boolean expression representing the first alternative in a conditional expression. CP87's function is to ensure (by generating the requisite object code), that:

for designational expressions, the address value of the expression is loaded in ADR.

for arithmetic expressions, the value of the expression is loaded into a fixed point or floating point register, depending on whether the value is integer or real.

for boolean expressions, the value (True or False) of the expression is stored in a field in the current Data Storage Area.

Where the expression is complex, code will have been generated, before entry to CP87, to evaluate the expression, and in this case, the value or address will already be contained in the appropriate register or Data Storage Area field. If however, the expression is a simple label, an arithmetic constant or variable, or a boolean constant or variable, CP87 generates a Load or Move instruction. In all cases, the expression is represented by the stack operand pointing to a Label Address Table entry, a Data Storage Area field, or a register.

CP87 also generates an unconditional branch around the second alternative expression which follows 'ELSE'. The code references a new entry in the Label Address Table, in which the relative branch address will subsequently be stored by CP79. In addition, CP87 stores the displacement (PRPOINT) of the next object code instruction in the Label Address Table entry previously reserved by CP78, representing the address of the second alternative expression. Before exit to SNOT, the stack operator Then is replaced by Else.

Compiler Program No.79 (CP79)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: Else

The source operator marks the end of a conditional expression. It is preceded by a designational, arithmetic or boolean expression representing the second alternative expression.

CP79's function is:

1. to generate the necessary object code such that, if the condition following 'IF' is False, the address or value of the second alternative will be loaded in the same register (ADR in the case of a designational expression), or moved to the same Data Storage Area field as that specified in the coding for the first alternative expression (see CP87 above); and

2. to generate, if necessary, a call to the Fixed Storage Area integer-real conversion routine, in the event one of the alternative expressions is real and the other is integer.

The two alternative expressions are represented by the last two stack operands. At exit from CP79, these operands are replaced by a single operand which addresses the object time register or Data Storage Area field in which the address or value of the particular alternative expression (depending on the condition identified at object time) will be contained after evaluation of the complete conditional expression.

Finally, the displacement (PRPOINT) of the next object code instruction is stored in the Label Address Table entry specified by an operand previously stacked by CP87, representing the address of the unconditional branch following the first alternative expression, and the stack operator Else is released.

BOOLEAN EXPRESSIONS

Object time boolean operations (specified in the source module by the operators 'AND', 'OR', 'EQUIV', and 'IMPL') are performed in fields reserved for intermediate results in the current Data Storage Area (in the listing these fields are referred to as "Object Stack entries"). When code to implement a boolean operator is to be generated, a test is first made to determine if the first operand constitutes:

1. a logical constant or a declared boolean variable, or

2. an intermediate boolean value.

If the operand is a logical constant ('TRUE' or 'FALSE') or a declared boolean variable (as in X'AND'Y), a field is reserved in the current Data Storage Area (by incrementing pointer P -- see Figure 54) and code is generated to move the operand to the reserved field and to perform the specified boolean operation in that field. If, however, the operand is an intermediate logical value, representing the value, say, of a relation (as in A>B'AND'C>D), the generated code will execute the specified boolean operation in the Data Storage Area field containing the intermediate value.

The operators 'AND' and 'OR' are implemented directly by the corresponding machine instructions. 'EQUIV' is implemented by the combination Exclusive Or (inversion) and Or. 'IMPL' is implemented by interchanging the operands and by Exclusive Or (inversion) and Exclusive Or. Where the second operand is a logical constant (whose value is known at compile time), the object code utilizes immediate instructions.

Compiler Program No.64 (CP64)

Source Operator:
Stack Operator: (See decision matrices -- Appendix V)

CASE A: See "Procedures".

CASE B: See "Standard Procedures".

CASE C: The source operator precedes an arithmetic, boolean or conditional expression. The Expression Context Matrix is addressed and the source operator stacked.

Compiler Program No.65 (CP65)

	CASE A	CASE B
Source Operator:	If	Not
Stack Operator:	If or If-s	(See Expression Context Matrix --Appendix V-c)

CASE A: See "Conditional Expressions".

CASE B: The logical operator Not identifies a boolean expression. Not is stacked.

Compiler Program No.67 (CP67)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: (See Expression Context Matrix -- App. V-c)

The source operator (which may be an arithmetic or relational operator or any one of the logical operators And, Or, Equiv, or Impl) is stacked.

Compiler Program No.76 (CP76)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: And, Or, Equiv, or Impl

The source operator indicates that the operation specified by the stack operator, between the boolean operands represented by the last two stack operands, may be implemented.

CP76 generates code to perform the specified operation in a current Data Storage Area field, and releases the stack operator. At exit to COMP, the stack operand addresses the Data Storage Area field in which the result (True or False) of the operation will be contained at object time.

Compiler Program No.77 (CP77)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: Not

The source operator is preceded by a boolean operand (a constant, a variable or a complex expression) to be operated on by the stack operator Not. CP77 generates code to invert the logical value of the operand in a current Data Storage Area field, and releases the operator Not.

ARITHMETIC EXPRESSIONS AND RELATIONS

Compiler Program No.64 (CP64)

Source Operator: (
Stack Operator: (See decision matrices -- Appendix V)

CASE A: See "Procedures".

CASE B: See "Code Procedures".

CASE C: The source operator precedes an arithmetic, boolean or conditional expression. The Expression Context Matrix is addressed and the source operator stacked.

Compiler Program No.66 (CP66)

Source Operator: + or -
Stack Operator: (See Expression Context Matrix -- App. V-c)

If the source operator was preceded by an operand, the operator is stacked. If, however, the source operator was preceded by an operator, and the source operator is -(minus), the operator Monadic Minus is stacked.

Compiler Program No.67 (CP67)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: (See Expression Context Matrix -- App. V-c)

The source operator cannot be implemented before the following expressions and operators are known. The source operator is stacked.

Compiler Program No.63 (CP63)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: Monadic Minus

The monadic minus operator is implemented by object code which loads the complement of the last stack operand. If the operand was not previously loaded into a register, a load instruction is generated before the load complement instruction is generated. The stack operator is released.

Compiler Program No.68 (CP68)

Source Operator:)
Stack Operator: (

The source operator marks the end of an arithmetic, boolean, or conditional expression. The stack operator is released.

Compiler Program No.69 (CP69)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: (See Expression Context Matrix -- App. V-c)

The priority rules specify that the arithmetic, relational or power operator in the Operator Stack shall be implemented.

CP69 handles the generation of code for all of the following:

Arithmetic operators: +,-,*,/, and ÷

Relational operators: <,≥,>,≤,= and ≠

Power operator (Power)

Assignment operator (Assign). Assignment statements are processed initially by CP12 and CP20 (see "Assignment Statements"), but the object code to implement an assignment is generated in most cases by a subprogram of CP69 (Real-Real Routine or Integer-Integer Routine), entered from CP20.

The stack operator specifies an operation between the operands on either side of the operator (both of which must be arithmetic), represented by the last two operands in the Operand Stack. Each operand is first inspected by the OPDREC subroutine, which determines if the operand is a formal parameter or a parameterless procedure, and if so, generates code to call the actual parameter code sequence or the parameterless procedure (see "Procedures").

Depending on the stack operator and the character of the operands (real or integer), control is passed to one of several major subprograms of CP69:

Integer-Integer Routine (DHZB1):
Both operands integer.
Operator: +, -, or any relational operator.

Integer Division Routine (ISB1)
Both operands integer.
Operator: ÷ .

Integer Multiplication Routine (IPB1)
Both operands integer.
Operator: *.

Integer Power Routine (IUB1)
Both operands integer.
Operator: Power.

Real-Real Routine (DHEB2):
First operand real, second operand real or integer.
Operator: any relational operator. If the second (or last) operand is integer, a call is generated (by the TRINRE subroutine) to the Fixed Storage Area routine CNVIRD for integer-to-real conversion.

Real-Integer Power Routine (I1B1)
First operand real, second operand integer.
Operator: Power.

Real Power Routine (HOB1)
Second operand real, first operand real or integer.
Operator: Power. If the first operand is integer, a call is generated (by the TRINRE subroutine) to the Fixed Storage

Area routine CNVIRD for integer-to-real conversion.

After code to implement the indicated operation has been generated, the last operator and operand in the Stack are released, and (except in the case of an assignment) the stack operand originally representing the operand to the left of the stack operator is modified to specify the object time register or Data Storage Area field containing the result of the operation implemented.

Integer-Integer Routine (DHZB1)

This routine generates code, on call from CP69, to implement the arithmetic operators + and -, and the relational operators <, ≥, >, ≤, =, and ≠, connecting two integer operands. It is also entered from CP20 for normal assignments (see "Assignment Statements") and from CP51 (see "Arrays").

Except in the case of relational operators, object time operations are performed in registers, and the routine handles the generation of object code to load an operand (where neither operand is contained in a register) by calling the appropriate subroutine in the Subroutine Pool.

In the case of relational operators, a compare instruction is generated first and a call is then made to the Relational subroutine (IMB1), which generates code to move the value _True_ or _False_ (X'01' or X'00'), depending on the condition code set, to a field in the current Data Storage Area.

In the case of an assignment operator, a store or move instruction is generated.

Integer Division Routine (ISB1)

This routine generates code to implement the operator ÷ connecting two integer operands, on call from CP69.

Before generating code to implement the division operator, tests are made and appropriate object code generated, to ensure that the first operand is loaded into an even-numbered register and that the next odd-numbered register is free.

Integer Multiplication Routine (IPB1)

This routine generates code to implement the operator * connecting two integer operands, on call from CP69. Before generating code to implement the multiplication operator, tests are made and appropriate object code generated, to ensure that one of the operands is loaded into an odd-numbered register and that the other operand is loaded into the preceding even-numbered register.

Integer Power Routine (IUB1)

This routine implements the power operator connecting two integer operands, on call from CP69, by generating a call to the standard Power function (Load Module IHIFII) in the ALGOL Library (Chapter 10). The code generated consists in part of instructions which store the object time addresses of the two operands (base and exponent) in a parameter list in the current Data Storage Area, in part of a calling sequence, which loads the address of the parameter list and branches to the standard Power function.

Real-Real Routine (DHEB2)

This routine generates code, on call from CP69, to implement the arithmetic operators +, -, *, and / and the relational operators <, ≤, >, ≥, =, and ≠ connecting two operands, one (or both) of which is real (before entry, code will have been generated to convert the non-real operand, if any). The routine also generates code to implement an assignment, on call from CP20 (see "Assignment Statements"). The implementation of operators is similar to that of the Integer-Integer Routine, as regards arithmetic as well as relational operators, except that floating point registers are used.

Real-Integer Power Routine (I1B1)

This routine implements the power operator connecting a real operand and an integer operand, on call from CP69, by generating a call to the standard Power function (Load Module IHIFRI or IHIFDI, depending on whether the precision of the base is short or long) in the ALGOL Library. The code generated is similar to that generated by the Integer Power Routine (see above).

Real Power Routine (HOB1)

This routine implements the power operator connecting two real operands, on call from CP69, by generating a call to the standard Power function (Load Module IHIFRR or IHIFDD, depending on whether the precision of the base is short or long) in the ALGOL Library. The code generated is similar to that generated by the Integer Power Routine (see above).

Compiler Program No.68 (CP68)

Source Operator:)
Stack Operator: (

The source operator marks the close of an arithmetic, boolean or conditional expression enclosed by parentheses. The stack operator is released.

SEMICOLON HANDLING

Compiler Program No.24 (CP24)

Source Operator: Delta
Stack Operator: Beta, Pi or Phi

Delta represents the semicolon terminating a declaration or a specification. A call is made to the SCHDL subroutine, which updates the Semicolon Count at SCSC and, if the TEST option is specified, generates a call to the Fixed Storage Area TRACE routine.

Compiler Program No.25 (CP25)

Source Operator: Semicolon
Stack Operator: Beta, Pi, Phi or Begin

The Semicolon marks the end of the first statement in a block, procedure or compound statement. The Semicolon is stacked (to ensure that, if a further declaration follows, an error will be recorded by CP28) and the SCHDL subroutine is called (see CP24 above). If the Semicolon was preceded by an operand (in which case the operand logically represents a call for a parameterless procedure), a call is made to the PLPRST subroutine (which generates the appropriate procedure call or records an error) and the operand is released.

Compiler Program No.23 (CP23)

Source Operator: Semicolon, Epsilon, Eta or End
Stack Operator: Semicolon

The source operator ends a statement in a block, a procedure, a for statement or a compound statement. If the source operator is a Semicolon, the SCHDL subroutine is called; otherwise the Semicolon in the stack is released.

Compiler Program No.7 (CP7)

Source Operator: For, Goto, If, [, (or Assign
Stack Operator: Beta, Pi, Phi

The source operator identifies the first statement in a block or a procedure. A Semicolon is stacked to ensure that, if a declaration is subsequently encountered, an error will be recorded by CP28.

CONTEXT SWITCHING

Each of the three decision matrices (Appendix V) specifies the set of compiler programs to be entered for all possible pairs of source-stack operators within a particular context of the source module, identified as a program context, a statement context and an expression context. As soon as a change in context occurs (signified by one or more critical source operators), a corresponding change in decision matrix is indicated. The appropriate change in matrix is effected by a particular compiler program specified in the currently operative matrix. (See "Decision Matrices" in this chapter). After the change has been effected, control is in every case passed to COMP, which branches to the compiler program specified in the new matrix for the original operator pair.

Compiler Program No.19 (CP19)

Source Operator: If
Stack Operator: Assign

A change from program to statement context is indicated. The Statement Context Matrix is addressed.

Compiler Program No.22 (CP22)

Source Operator: (Any arithmetic, logical or relational operator)
Stack Operator: Assign

A change from program to expression context is indicated. The Expression Context Matrix is addressed.

Compiler Program No.33 (CP33)

Source Operator: (Any arithmetic, logical or relational operator)
Stack Operator: (See Statement Context Matrix)

A change from statement to expression context is indicated. The Expression Context Matrix is addressed.

Compiler Program No.70 (CP70)

Source Operator: (See Expression Context Matrix -- App. V-c)
Stack Operator: (See Expression Context Matrix -- App. V-c)

A change from expression to statement context is indicated. The Statement Context Matrix is addressed.

Compiler Program No.71 (CP71)

Source Operator: (See Statement Context Matrix -- App. V-b)
Stack Operator: (See Statement Context Matrix -- App. V-b)

A change from statement to program context is indicated. The Program Context Matrix is addressed.

LOGICAL ERROR RECOGNITION

The following compiler programs are entered on detection of operator/operand sequences which are logically or syntactically incorrect. Their function is

1. to record the error in the Error Pool;

2. to switch the Compiler to Syntax Check Mode (see Chapter 9), or, in one case (CP84), to terminate compilation; and

3. to make appropriate adjustments to the Operator/Operand Stack, so as to permit syntax checking to proceed. Errors are recorded by a branch to the Error Recording Routine, which also switches to Syntax Check Mode.

Compiler Program No.26 (CP26)

Source Operator: Array, Switch, Pi or Phi
Stack Operator: Then-s, Else-s, Assign or Semicolon

The source operator identifies a declaration outside the block head, i.e., following or inside a statement. Error No. 166 is recorded and all stack operators and operands relating to the statement (if any) in which the declaration occurs, are released. The declaration will be processed (syntax checked) by the compiler program subsequently entered.

Compiler Program No.27 (CP27)

Source Operator: (See decision matrices -- Appendix V)
Stack Operator: (See decision matrices -- Appendix V)

The stack and source operators represent an improper operator sequence. Error No. 194 or 195 (depending on whether the operators are separated by an operand) is recorded, and the Operator/Operand Stack is adjusted, after release of the last stack operator and operand (if any), according to the next stack operator, to permit syntax checking to proceed.

Compiler Program No.28 (CP28)

Source Operator: Delta
Stack Operator: (See decision matrices -- Appendix V)

Delta, which normally terminates a declaration, has been encountered outside the block head. Error No. 166 is recorded, and the Operator/Operand Stack is adjusted to permit syntax checking to proceed.

Compiler Program No.29 (CP29)

Source Operator: Array, Switch, Pi or Phi
Stack Operator: Begin or Do

A declaration outside a block head has been encountered. Error No. 166 is recorded and control passed to CP4 for syntax checking of the declaration.

Compiler Program No.30 (CP30)

Source Operator: Label Colon
Stack Operator: (See Program Context Matrix -- App. V-a)

A label has been incorrectly declared inside a statement. If the Label Colon was preceded by an operand (representing the label), error No. 169 is recorded and the Label Colon and the label are disregarded. Otherwise, an internal compilation error is indicated; control is passed to CP84, which records error No. 173 and terminates compilation.

Compiler Program No.31 (CP31)

Source Operator: (See Program Context Matrix -- App. V-a)
Stack Operator: (See Program Context Matrix -- App. V-a)

The stack and source operators represent an invalid operator sequence in the program context. Error No. 160 or 161 is recorded, depending on whether the operators are separated by an operand, and the Expression Context Matrix is addressed.

Compiler Program No.72 (CP72)

Source Operator: Else
Stack Operator: Else

The combination Else...Else is valid only in a conditional expression (indicated if the stack operator above Else is Assign). In this case, control is passed to CP79; otherwise, control is passed to CP75, which records error No. 194 or 195.

Compiler Program No.73 (CP73)

Source Operator: If
Stack Operator: (See Expression Context Matrix -- App. V-c)

The operator combination indicates that the opening parenthesis which should enclose the expression beginning with 'IF' is missing. Error No. 160 or 161 is recorded, and the operator (is stacked.

Compiler Program No.74 (CP74)

Source Operator: (Any relational operator)
Stack Operator: (Any relational operator)

A sequence of relational operators is invalid. Error No. 160 or 161 is recorded and the operator * is arbitrarily stacked, in place of the source operator, in order to permit syntax checking to proceed.

Compiler Program No.75 (CP75)

Source Operator: (See decision matrices -- Appendix V)
Stack Operator: (See decision matrices -- Appendix V)

The operator sequences represented by the stack and source operators is invalid. Error No. 194 or 195 is recorded.

Compiler Program No.84 (CP84)

Source Operator: (See decision matrices -- Appendix V)

Stack Operator: (See decision matrices -- Appendix V)

The invalid operator sequence represented by the stack and source operators indicates that the source module contains one or more fundamental logical defects. Error No. 173 is recorded and the Compiler is terminated, by exit to CPERR1.

Compiler Program No.86 (CP86)

Source Operator: Else
Stack Operator: If or If-s

The operator Then is missing in the source text. Error No. 160 or 161 is recorded and the operator Then or Then-s is stacked, in place of Else.

CLOSE OF SOURCE MODULE

Compiler Program No.3 (CP3)

Source Operator Omega
Stack Operator: Alpha

Omega marks the end of the source module. Providing the source module is not a precompiled procedure (determined by inspection of the HCOMPMOD Control Field-Appendix IV), code is generated to branch to the Termination Routine TERMIN in the Fixed Storage Area. Control is then passed to CPEND, the normal exit from IEX51. (See "Subroutine Pool".)

SUBROUTINE POOL

The Subroutine Pool, contained in Control Section IEX50000, comprises the subroutines used by most compiler programs in common, as well as a short initialization routine and the scanning and decision-making routines Scan to Next Operator (SNOT) and Compare (COMP). The latter two routines are described in an earlier section of this chapter.

Initialization

Scan to Next Operator (SNOT)

Compare (COMP)

These routines are described elsewhere in this Chapter.

Change Input Buffer (JBUFFER)

Re-sets a pointer (SOURCE-register 6) to address a buffer containing a new record of the Modification Level 2 text and READs a further record from SYSUT2 into the alternate buffer. Called by SNOT, CP4, CP51, and CP66 on recognition of the record-end character Zeta.

Next OPTAB Entry (NXTOPT)

Increments a pointer (AOPTABE) by 14 bytes to address the next entry in the Optimization Table, and re-sets the pointer to a new record where the end of the current record has been reached. Called by CP40, CP47, and CP49, when the Optimization Table is being searched for subscript expressions to be optimized in a for statement.

Error Recording (SERR)

Stores an error pattern in the Error Pool and sets the Syntax Check Mode switch on in the HCOMPMOD Control Field, on call from most compiler programs and subroutines. The error pattern is described in Chapter 9. There are five entry points, depending on the length of source text to be included in the ultimate diagnostic message.

Conversion Integer-Real (TRINRE)

Generates code to call the Fixed Storage Area routine CNVIRD (which converts an integer operand to real form in floating point register 0) and modifies the stack operand to represent a real operand contained in FPR0. Called by CP20, CP47, CP61, CP69 and CP79.

Conversion Real-Integer (TRREIN)

Generates code to call the Fixed Storage Area routine CNVRDI (which converts a real operand to integer form in fixed point register STH). Called by CP20, CP36, CP38, and CP51.

Generate Object Code (GENERATE)

Constructs TXT and RLD records of object module instructions specified by the calling compiler program, and outputs the records on the SYSLIN and/or SYSPUNCH data sets, depending on the options (LOAD and/or DECK) specified. The calling sequence in the calling compiler program is of the form

```
BAL INFORM, GENTXT4(0,SBR)
<4-byte object instruction>
```

where INFORM (register 2) loads the address of the object instruction to be generated.

There are seven entry points (GENTXT2, GENTXT4, GENTXT6, GENTXTS, GENTXTP2, GENTXTP4, and GENRLD), each entry point corresponding to a particular instruction length. GENTXTP2 and GENTXTP4 are used for floating point instructions. GENTXTS is used for object code sequences of varying length. The call to GENTXTS is of the form

```
LA   INFORM,   <Label   of   instruction
sequence>
BAL LENGTH, GENTXTS(0,SBR)
DC H 'Length of instruction sequence'
```

The GENERATE subroutine is also included in the Scan I/II and Scan III Phases (IEX11 and IEX30).

Store Object Time Registers (CLEARRG)

Generates code (by calling ROUTIN13 and ROUTINE9) to store fixed point and floating point registers currently in use in the object module, and updates the object time register control fields (Figures 56 and 57). Called by CP0, CP1, CP8, CP34, CP38, CP40, CP57, CP61, CP64, and CP80.

Operand Recognizer (OPDREC)

Inspects the stack operand addressed by OPDK (register 3) to determine if the operand represents a formal parameter or a parameterless procedure and generates code as follows:

Operand	Code generated
Formal parameter called by name	Store occupied registers. Branch to actual parameter code sequence (see Figure 63).
Formal parameter label called by value	Load ADR with label address (contained in formal parameter's Data Storage Area field specified in the stack operand).
Formal parameter array called by value	Load ADR with address of array (contained in formal parameter's Data Storage Area field specified in the stack operand). Load GDSA with Data Storage Area base address.
Parameterless procedure	Store occupied registers. Call procedure (see Figure 63).

No code is generated if the operand is not a formal parameter or a parameterless procedure.

OPDREC is called by CP12, CP20, CP30, CP36, CP38, CP40, CP41, CP45, CP47, CP49, CP51, CP57, CP59, CP61, CP62, CP69, CP76, CP77, CP78, CP79, and CP87.

Update DSA Pointer (MAXCH)

Records the current value of the displacement pointer P (register 8) in the Program Block Table III entry (Figure 60) corresponding to the Program Block Number of the current block or procedure, if the value of P exceeds the displacement recorded. This records the size required, up to the particular point, for the block's or procedure's object-time Data Storage Area.

Semicolon Handling (SCHDL)

Updates the semicolon count (by storing at SCSC the semicolon number following a Semicolon or Delta operator) and, if the TEST option is specified, generates a branch to the Fixed Storage Area TRACE routine (Chapter 10). Called by CP23, CP24, CP25, CP28, CP54, and CP59 on detection of the Semicolon or Delta operator.

ROUTINE1

Inspects a non-address operand in the Stack and stores the operand's base register number (6 or 7) and displacement at VPLACE and WPLACE, respectively. Generates a load instruction to load GDSA (register 6) with the appropriate Data Storage Area base address if the operand's storage field is contained in a Data Storage Area other than that currently addressed by CDSA or GDSA. Called by most compiler programs

preparatory to editing an object code instruction.

ROUTINE2

Generates code to load an address operand into a floating-point register. Adjusts the stack operand and the object-time register control fields (Figure 57). Called by CP63, CP69, and CP87.

ROUTINE3

Inspects an address operand in the Stack and stores the operand's register number (9-ADR) and displacement (0) at VPLACE and WPLACE, respectively. Generates a load instruction, if the address is not in ADR, and updates the object-time register control field (Figure 56). Called by a large number of compiler programs.

ROUTINE4

Generates code to load a non-address operand into a floating-point register. Adjusts the stack operand and the object-time register control fields (Figure 57). Called by CP63, CP69, and CP87.

ROUTINE5

Generates code to load a non-address operand into a general purpose register. Adjusts the stack operand and the object-time register control fields (Figure 56). Called by CP63, CP69, and CP87.

ROUTINE6

Generates code to load an address operand into a general purpose register. Adjusts the stack operand and updates the object-time register control fields (Figure 56). Called by CP63, CP69, and CP87.

ROUTINE7

Reserves an object-time general purpose register and calls ROUTINE9, which generates a store instruction if the register is in use.

ROUTINE8

Releases the last reserved general purpose register as well as the related Data Storage Area field.

ROUTINE9

Generates code to store a general purpose register and adjusts the stack operand.

ROUTIN10

Generates code to store the contents of ADR and adjusts the stack operand. Called by CP69 and CP87.

ROUTIN11

Reserves a floating-point register and calls ROUTIN13, which generates a store instruction if the register is in use. Reserves a storage field for the register and updates the object-time register control fields (Figure 57).

ROUTIN12

Releases the last reserved floating-point register and the related Data Storage Area field.

ROUTIN13

Generates code to store a floating-point register and adjusts the stack operand. Called by CP69.

ROUTIN14

Generates code to store Floating Point Register 0 and adjusts the stack operand. Called by CP61, CP69, and CP87.

ROUTIN15

Inspects the stack operand and stores the operand's base register (CDSA-6 or GDSA-7) and displacement at VPLACE and WPLACE, respectively. Called by several compiler programs preparatory to editing an object code instruction.

Program Block Number Handling (PBNHDL)

Called at entry to or exit from a block or procedure. Stores the current Data Storage Area displacement value in register P in the Program Block Table III entry of the block exited and re-loads P with the displacement contained in the entry for the entered block. Called by CP0, CP4, and CP16.

Parameterless Procedure Statement (PLPRST)

Inspects the stack operand to determine if the operand is a parameterless procedure, and if so, generates a call to the procedure (see Figure 63). In any other case, error No. 183 or 187 is recorded.

TERMINATION PHASE (IEX51)

The Termination Phase represents a logical extension of the Compilation Phase but constitutes a seperate load module (IEX51) which is loaded and executed (by XCTL in IEX50) in succession to the Compilation Phase.

The main functions of the Termination Phase are:

1. To construct Program Block Table IV (PBTAB4) and the Data Set Table (DSTAB).
2. To generate TXT and RLD records of the object Label Address Table (LAT), Program Block Table (PBT), and Data Set Table (DSTAB).
3. To generate ESD records for all Library procedures called in the source module.
4. To generate RLD records and an ESD record for the Address Table, containing the relative addresses of the object time Program Block, and Label Address Tables and of the entry point of the object module.
5. To print a statement of object time storage requirements (unless an error has been recorded).
6. To print diagnostic messages for the record errors, if any.
7. To return control to the Final Exit Routine in the Directory.

The foregoing functions are discussed below under appropriate headings, in the approximate order in which the functions are performed in the Termination Phase.

Program Block Table IV (PBTAB4)

Program Block Table IV (PBTAB4) is constructed from Program Block Table III (PBTAB3), transmitted from the Compilation Phase via the Common Work Area. PBTAB3 is described in this chapter under "Phase Initialization" (see also Figure 60).

The Termination Phase constructs PBTAB4 by copying each four-byte entry of PBTAB3 into a new eight-byte entry, the first four bytes of which are reserved for the object time address of each block's or procedure's Data Storage Area, or for the entry point of a code procedure. The content of the PBTAB4 entry is identical with that of the object time Program Block Table (PBT) (see chapter 11 and Figures 84 and 83) except for the first four bytes, which are filled in at object time only.

After construction of PBTAB4, TXT records of the table are generated. In addition, RLD records are generated for the address of Constant Pool No. 0 in the first entry (and for any other Constant Pool(s) in the last entries) of PBTAB3.

Label Address Table (LAT)

The compile time Label Address Table (LAT), first constructed at initialization of the Compilation Phase, is transmitted to the Termination Phase in main storage. The table is described elsewhere in this chap-

ter (see "Phase Initialization" and Figure 59).

The Termination Phase generates TXT and RLD records of the Label Address Table. By virtue of the RLD records generated, the relative addresses of declared labels, switches, and procedures, as well as Compiler-generated branch addresses, are translated to absolute addresses in the object time Label Address Table (see "Label Address Table" in Chapter 11 and Figures 85 and 83).

For each of the standard I/O procedures and mathematical functions called in the source module, an ESD record is generated. The standard I/O procedures and mathematical functions called in the source module are identified by a 0 in bit 0 of the first 28 full-words of the Label Address Table (for any standard procedure not called, bit 0 in the corresponding full-word is equal to 1). The ESD record generated contains the name of the ALGOL Library module, obtained from a corresponding entry in one of two tables (SHRTAB or LNGTAB), depending on the precision specified. The address of the standard procedure in the object module is stored by the control program in the corresponding full-word of the object time Label Address Table. Appendix III lists the internal names of all standard procedures, in which the displacement of the corresponding Label Address Table entry is specified.

Data Set Table (DSTAB)

The object time Data Set Table, constructed in the Termination Phase, is described in Chapter 11 (see also Figures 86 and 83).

The Termination Phase constructs a 36-byte entry for data sets Nos. 0 and 1 and for every other data set for which an object time input, output and/or SYSACT operation is specified in the I/O Table (Figure 64). A 28-byte entry (called the PUT/GET Control Field) is also constructed at the end of the table, if a PUT or GET operation is specified for any of the data sets. The first full word (APGCF) in the Data Set Table contains the address of the PUT/GET Control Field or, if the latter is not present, the address of the first byte following the end of the table (in the latter case, bit 0 in the first byte of APGCF is set=1).

The symbolic names and function of the various fields in the data set entry are indicated in Figure 86. The following list shows the initial contents of each field in the entry after the Data Set Table has been constructed in the Termination Phase.

Data Set Entry (36 bytes)

Field Name	Bytes	Initial Setting
ADCB	0-3	0
R	4-7	0
RE	8-11	0
NBB	12-15	0
BB	16-19	0
S	20-21	1
P	22-23	80
K	24	2
Q	25	0
DSF	26-27	(see below)
NOTEADR	28-31	0
BL	32-33	0
DSF	34-35	(copy of bytes 26-27 -- see below)

The settings of the 2-byte DSF field depend on the data specified for the data set in the I/O Table (IOTAB -- Figure 64), as follows:

IOTAB specifies:	DSF Setting
No output, SYSACT 4/13 or SYSACT - undetermined	X'0000'
Output, but no input, SYSACT 4/13 or SYSACT - undetermined	X'2000'
SYSACT 4/13 or SYSACT. undetermined, but no output	X'4000'
Input and Output or SYSACT 4/13 or SYSACT - undetermined and Output	X'4200'

The symbolic names and function of the various fields in the PUT/GET Control Field are indicated in Figure 86. The following list shows the initial contents of each field in the PUT/GET Control Field as inserted by the Termination Phase.

PUT/GET Control Field (28 bytes)

Field Name	Bytes	Initial Setting
ADCB	0-3	0
R	4-7	0
RE	8-11	0
BB	12-15	0
BE	16-19	2048
NOTEADR	20-23	0
S	24-25	0
TYP	26	0
PG	27	0

After construction of the Data Set Table, TXT records of the table are generated. The position of the Data Set Table in the object module is indicated in Figure 83.

Address Table and END Record

The Termination Phase constructs an address table containing the following relative address constants:

Address of Program Block Table (PBT)

Address of Label Address Table (LAT)

Address of first instruction in the generated object module (following the end of Constant Pool 0)

The relative addresses are obtained from the object module displacement (length) pointer (PRPOINT), augmented, where appropriate, by the length of the preceding table (see Figure 83). An ESD record for the Address Table, and RLD records for the three address constants, are generated.

The relative address of the entry point (IHIFSAIN) in the combined object module (the Fixed Storage Area Initialization routine) is recorded in the END record and in an ESD record. The relative address is obtained by adding a known displacement to the object module displacement pointer.

Statement of Object Time Storage Requirements

A statement is printed of the main storage required by the object module, unless a compile time error was detected in IEX40, IEX50, or IEX51. The printed statement, discussed in the ALGOL Programmers Guide, indicates the length of the generated instructions, the Constant Pool(s) and the Data Storage Area required by each block and procedure (obtained from PBTAB4).

Diagnostic Output

Diagnostic messages reflecting the errors, if any, recorded in IEX40, IEX50, or IEX51 are printed out by the Error Message Editing Routine (see Chapter 9).

End of Compilation

At the close of compilation, the Common Area of main storage is released and control is passed (by a RETURN macro) to the Final Exit Routine in the Directory (Chapter 2).

CHAPTER 9: COMPILE TIME ERROR DETECTION AND DIAGNOSTIC OUTPUT

ERROR DETECTION

The Compiler detects syntactical or logical errors in the source module as well as procedural errors involving Compiler options and DD statements. In addition, the Compiler may be terminated as a result of program interrupts, unrecoverable I/O errors, and unopened data sets.

An error is recorded by the operative phase in the form of an error pattern, stored in the Error Pool. The recorded error patterns are processed and diagnostic messages printed out by the Error Message Editing routine in each of Load Modules IEX21, IEX31, and IEX51. (See "Diagnostic Output").

Every error detected by the Compiler (or occurring during compilation) is identified with one of three degrees of severity. According to the severity of each detected error, the Compiler may:

1. Continue normal processing and compilation;
2. Terminate object code generation, but continue to scan the source module for other errors; or
3. Terminate all processing immediately after diagnostic messages have been printed out for errors recorded in the Error Pool.

The degree of severity of each detected error is indicated by the severity code (W, S, or T) in the printed diagnostic message.

The diagnostic messages printed out for all detected errors are listed in OS ALGOL Programmer's Guide. Appendix F in this manual lists the errors detected in each of the several phases.

WARNING ERRORS (SEVERITY CODE W)

A warning error (code W) constitutes a (normally) minor syntactical defect or procedural error. Where an error of this kind is detected, the Compiler takes action to disregard, correct, or otherwise compensate for the error. Except for such corrective action, the detection of a warning error has no effect on the subsequent processing and compilation action of the Compiler, unless a serious or terminating error is subsequently detected. The detection of warning errors is confined to the Initialization, Scan I/II, and Scan III Pases.

SERIOUS ERRORS (SEVERITY CODE S)

A serious error (Code S) represents a serious logical or syntactical defect in the source module, that cannot be disregarded or corrected by the Compiler without, in most cases, making an arbitrary assumption concerning the logic of the source module. When a serious error is detected, the Compiler enters Syntax Check Mode. In this mode of operation, all object code generation is terminated, but the search continues, until a terminating error occurs, for other possible defects in the source module. Entry to Syntax Check Mode is specified by the CMT switch in the HCOMPMOD Control Field (Appendix IV).

Entry to Syntax Check Mode has the following effects in the several phases.

Scan I/II Phase (IEX11): Processing is unaffected, except that, where a formal parameter is unspecified (a serious error), the parameter is assigned an all-purpose internal name in the Identifier Table, so as to enable syntax checking functions to proceed in the later phases.

Identifier Table Manipulation Phase (IEX20): Processing is in no way affected.

Scan III Phase (IEX30): All processing involving the subscript Table is terminated and the storage of constants in the Constant Pool is inhibited, but the search for faults and errors in the source module continues. Undefined identifiers and defective constants are replaced, in the Modification Level 2 text, by all-purpose internal names.

Subscript Handling Phase (IEX40): All processing is inhibited, and control is passed directly to the Compilation Phase.

Compilation Phase (IEX50): The generation of object code is terminated. The stacking and inspection of operators and operands continues, however, to permit the detection of further syntactical or logical errors.

Termination Phase (IEX51): Processing is limited to the construction of the Data Set Table. All generation of TXT and RLD records for object time tables is inhibited.

TERMINATING ERRORS (SEVERITY CODE T)

Terminating errors include (a) logical or syntactical defects in the source module, whose presence severely prejudices the subsequent interpretation of the logic of the source module; (b) logical errors in the Compiler; (c) mechanical or logical defects involving data sets; (d) errors identified with the exhaustion of work areas owing to the length or complexity of the source text; and (e) errors arising from the defective invokation of the Compiler.

When a terminating error is detected (or occurs), all processing and/or compilation is immediately suspended and the Compiler is terminated after the recorded error(s) have been processed; more specifically, on detection of a terminating error, the operative phase stores an appropriate error pattern in the Error Pool and then passes control (by XCTL) to the next successive Diagnostic Output Module (IEX21, IEX31, or IEX51). The latter prints out diagnostic messages for the error(s) recorded in the Error Pool, and exits to the terminating routine in the Termination Phase. The detection of a terminating error is registered in the operative phase by means of the TERR switch in the HCOMPMOD Control Field.

DIAGNOSTIC OUTPUT

The Error Message Editing routine edits the error patterns stored by the several phases in the Error Pool, and prints out diagnostic messages on the SYSPRINT data set by calling the PRINT subroutine in the Directory.

The routine is normally executed at three points in the course of compilation:

1. After execution of the Identifier Table Manipulation phase
2. After execution of the Scan III phase
3. After execution of the Compilation phase.

The Error Message Editing routine constitutes a control section, named IEX60000, of the three load modules IEX21, IEX31, and IEX51. Figure 74 indicates the composition and execution sequence of Load Modules IEX21, IEX31, and IEX51.

In addition to the Error Message Editing routine, each module contains a control section comprising a Message Pool. The Message Pool in each load module contains an entry for each type of error that may be edited in the particular editing phase. The entries govern the composition of the messages printed out.

The Error Pool containing recorded error patterns is transmitted between phases in main storage.

When all error patterns have been processed, the Error Message Editing routine transfers control to the next phase in sequence, unless a terminating error is identified (indicated by the TERR switch in the HCOMPMOD Control Field). In the latter case, control is passed directly to the terminating routine in the Termination Phase (IEX51).

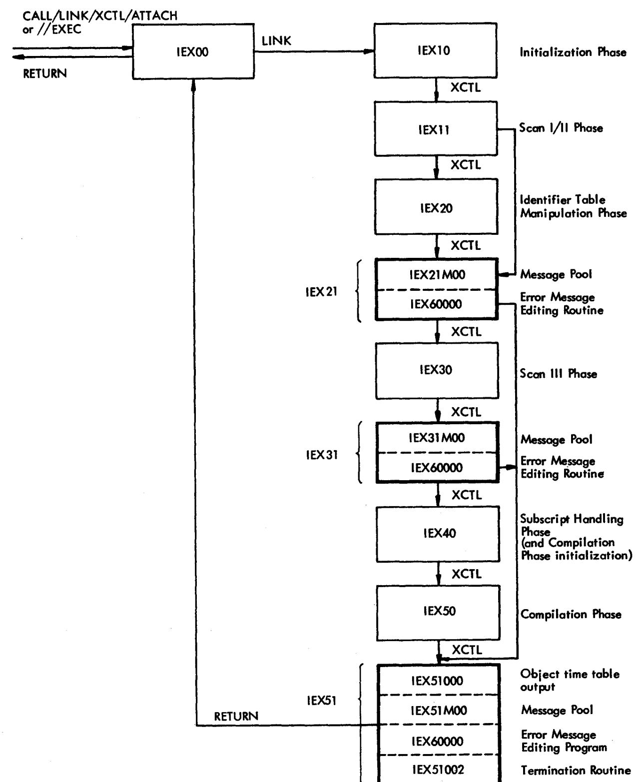
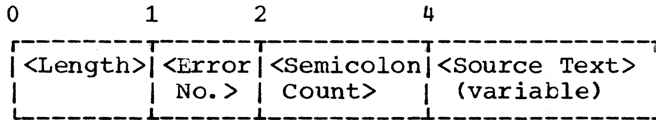


Figure 74. Composition and execution sequence of Load Modules IEX21, IEX31, and IEX51, containing the Error Message Editing Routine (IEX60000)

ERROR POOL

The Error Pool is an area of main storage acquired by the Initialization Phase (IEX10), and used by all phases in common for the storage of error patterns.



0	1	2	4
<Length>	<Error No.>	<Semicolon Count>	<Source Text> (variable)

Figure 75. Error pattern stored in Error Pool

The length of the error pattern is variable but is in no case less than four bytes. The error number identifies the error detected.

The semicolon count represents the number of the semicolon preceding the statement in which the error was detected, counted from the first semicolon in the source module. The error pattern may or may not contain characters found in the source module. If it does, the source text may consist of a maximum of six characters of an erroneous identifier or an erroneous delimiter, a maximum of twelve digits of a constant, one or two delimiters, or a maximum of 16 EBCDIC characters. In all cases except the last, the source text stored in the error pattern is in the form of the internal code (Appendices I-b and I-c); valid delimiters are represented by single one-byte symbols. When the error message is edited and printed out, the source text (if any) is translated back to the external code, delimiters being represented by one or more characters according to their original representation in the source module.

MESSAGE POOL

The Message Pool contains a set of variable length entries, which govern the editing of each message printed out by the Error Message Editing routine.

The Message Pool forms a control section of the three load modules containing the Error Message Editing routine. In the normal chronological order in which the three modules are loaded, the control sections containing the Message Pool are IEX21M00, IEX31M00, and IEX51M00. The entries in the Message Pool vary between control sections.

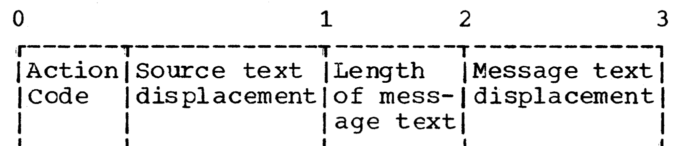
Figure 76 indicates the content of the entries in the Message Pool.

The length of the Message Pool entry is variable, depending on the number of Insertion Codes contained in the entry and on the message text. The entry may contain no Insertion Code, or it may contain one or more Insertion Codes up to a maximum of five. The actual number of Insertion Codes in the entry is specified in the second byte of the entry. The Insertion Codes, each of which occupy three bytes, are described below.

The severity code (W, S, or T) specifies the seriousness of the particular error. See "Error Detection".

The last bytes in the Message Pool entry contain the text of the message that is printed out for the particular error detected in the source module. The message is stored in EBCDIC characters. Where an error pattern contains source text, the source text is inserted in the appropriate place(s) in the printed message. The insertion of source text is controlled by the Insertion Codes in the Message Pool entry.

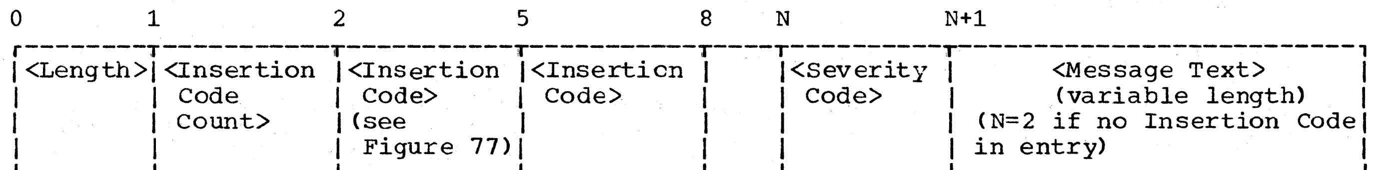
Figure 77 indicates the composition of the Insertion Code in the Message Pool entry.



0		1	2	3
Action Code	Source text displacement	Length of message text	Message text displacement	

Figure 77. Three-byte Insertion Code in the Message Pool entry (see Figure 76)

The Action Code in the first four bits specifies one of five alternative routines which handle the transfer (and, where required, translation) of message or source



0	1	2	5	8	N	N+1
<Length>	<Insertion Code Count>	<Insertion Code> (see Figure 77)	<Insertion Code>		<Severity Code>	<Message Text> (variable length) (N=2 if no Insertion Code in entry)

Figure 76. Message Pool entry

text to a work area where the error message is assembled before being output on the SYSPRINT data set. There are five Action Codes:

Action Code

1111 New Line (COT37)

The work area pointer (named WPT AREA) is shifted to the first position in the work area field that represents a new print line.

1000 Move Source Text (COT07)

The source text is transferred from the error pattern (in the Error Pool) to the next free byte in the work area, specified by the work area pointer. The location of the source text in the error pattern is specified by the displacement data contained in the second four bits of the Insertion Code (in this case, the displacement is invariably four bytes - - see Figure 74). The length of the source text is computed from the length indicator in the error pattern, minus four. Action Code 1000 is used mainly in connection with errors involving parameter options. When such an error is detected, the parameter name (e.g. DDname) is stored in the error pattern in lieu of source text. Since the parameter name is never translated to internal code, the transfer of the source text to the work area calls for no translation.

0100 Move ALGOL Delimiter (COT10)

The one-byte ALGOL symbol in the error pattern source text field is converted to its external word form representation and transferred to the next free byte in the Work Area, specified by the Work Area pointer. The location of the one-byte symbol in the error pattern is specified by the displacement data in the second four bits of the Insertion Code. The displacement of the symbol in the error pattern may be four or five bytes. In some cases, the source text may consist of a single one-byte ALGOL symbol, and accordingly, the displacement will be four bytes. In other cases, the text may consist of two delimiter words (represented by two one-byte symbols in the error pattern) which are to be inserted in different positions in the output message. In the latter case, the transfer of the delimiter words to their appropriate locations in the work area is governed by two separate Insertion Codes (separated by one or more others), both specifying the 0100 Action Code. In the first Insertion Code, the displacement of the first symbol in the error pattern will be specified as four bytes, while in the second, the displacement of the second symbol in the error pattern will be specified as five bytes.

Translation of a one-byte ALGOL symbol to its external representation is accomplished with the aid of two tables, named WSYMBSTK and WORDSEBC (or WSYMBSRC and WORDSISO, if the external character set specified is ISO). The tables named WSYMBSTK and WSYMBSRC contain a list of all one-byte internal symbols of ALGOL delimiters, together with the displacement addresses of corresponding entries in the WORDSEBC (or WORDSISO) table. The WORDSEBC and WORDSISO tables both contain a list of the external representations of ALGOL delimiters, and the character code in both tables is the EBCDIC code. The tables differ only to the extent that certain delimiters are represented in one table by symbols and in the other by whole words. Thus, for example, the relational operators < and > are represented in the WORDSEBC table as < and > while in the WORDSISO table they are represented as 'LESS' and 'GREATER'. The translation of a given internal symbol is obtained by first finding the entry in the WSYMBSTK (or WSYMBSRC) table containing the identical symbol and then taking the external translation from the entry in the WORDSEBC or WORDSISO table, specified by the indicated displacement address in the entry. The particular table used is governed by the SET60 switch in the HCOMPMOD Control Field.

0010 Move and Translate Source Text (COT33)

The source text (displacement: four bytes in error pattern; length: length - 4) is transferred to the next free byte in the work area specified by the work area pointer, and translated to EBCDIC representation. The translation is accomplished by means of a TRANSLATE instruction, specifying the translation table named WINTEBC.

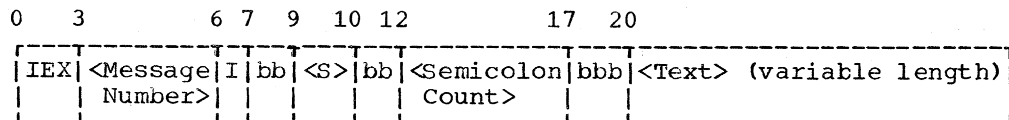


Figure 78. Format of the printed error message

0000 Move Partial Message Text (COT30)

A segment of the message text in the Message Pool entry, specified by the length and displacement data in the second and third bytes of the Insertion Code, is transferred to the work area.

As indicated above, the Message Pool entries may contain up to a maximum of five Insertion Codes or they may contain no Insertion Code at all. If the error message to be printed out occupies more than one printed line, or if the error message includes message text as well as source text, the assembly of the message in the work area consists of several individual transfers or editing actions, specified by a corresponding number of Insertion Codes in the Message Pool entry. The Insertion Codes are preset with the correct text lengths and displacements, to insure that the text is divided at the appropriate places where source text is to be inserted, or where a new line is to be started.

If, however, the error message consists entirely of message text occupying not more than a single line, the assembly of an error message involves only a single transfer operation. In this case, the Message Pool entry contains no Insertion Code, and the Message Pool entry in effect specifies a sixth editing action, handled by the Move Full Message Text routine (COT31). This routine transfers the entire message text, consisting of no more than a full line, from the Message Pool entry, to the work area. Since the entry contains no Insertion Code, the displacement of the text in the entry is invariably three bytes, and the text length is given by the entry length, less three (see Figure 76).

ERROR MESSAGE

The format of the printed error message is indicated in Figure 78.

The significance of the various symbols in the figure is as follows:

IEX — (The name of the ALGOL Compiler) identifies the program generating the message.

Message No. — The number of the error detected.

I — Signifies that the message is informative only and requires no response on the part of the operator.

bb — Indicates blank space.

S — The Severity Code, which may be W, S, or T.

Semicolon Count — The number of the semicolon preceding the statement or declaration in which the error was detected.

Text — The message text (and source text, if any), describing the error detected.

A list of the compile time error messages generated by the Compiler is provided in OS ALGOL Programmer's Guide. Appendix F in this manual lists the errors detected in each of the several phases.

LOGIC OF THE ERROR MESSAGE EDITING ROUTINE

The logical flow of the Error Message Editing routine is outlined in Flowcharts 040-043 in the Flowchart Section.

Error patterns are processed in sequence. With the aid of the error number in the second byte of the error pattern and an address table, the corresponding Message Pool entry is found.

A test is made of the Insertion Code Count (COT31) in the Message Pool entry (Byte 1). If the count is zero (indicating that the entry contains no Insertion Codes), the Move Full Message Text routine is activated. If the Insertion Code count is not zero, the count is loaded into a register, and the Action Code in the first Insertion Code is tested. The appropriate routine corresponding to the Action Code

(COT06) is then activated. After execution of the routine, the Insertion Code count is decreased by 1, and if the count is still nonzero, the next Insertion Code in the Message Pool entry is inspected and the routine specified by the Action Code is activated. This procedure is repeated until the Insertion Code count has been reduced to zero.

When the diagnostic message has been completely assembled in the Work Area, the entire message is processed by a TRANSLATE instruction, using a translation table named WEBCDIC. This translation operation is designed to enable the user to make desired character substitutions, that is, to replace certain characters in the standard EBCDIC character set by alternative optional characters. Until changed by the user, the translation table reflects the standard EBCDIC code, and the translation is equivalent to a straight EBCDIC to EBCDIC translation, having no effect on the diagnostic message.

The message is printed out on SYSPRINT by call to the PRINT subroutine in the Directory. Before PRINT is called, a test is made of the PRT switch in the HCOMPMOD Control Field to determine if the printer is operative. If the printer is not operative, diagnostic message No. 210 is printed out on the console typewriter, and control is passed to the Compiler termination routine (IEX51002). In the event a program interrupt occurs during execution of control section IEX60000, diagnostic message no. 211 is printed out and the Compiler is terminated.

The ALGOL Library is a partitioned data set named SYS1.ALGLIB. The Library consists of the following principal components.

1. The Fixed Storage Area, containing administrative routines and a pre-assembled work area required by the object module.

2. I/O procedures

3. Mathematical standard functions

The Fixed Storage Area and those I/O procedures and standard functions which are called in the source module, are linked to the object module by the Linkage Editor to form an executable load module.

This chapter also describes the Object Time Error routine, which is not part of the ALGOL Library, but forms a module of the SYS1.LINKLIB data set. The Error routine, whose function is to record object time errors, is loaded only if and when an error is detected during execution of the object program.

FIXED STORAGE AREA (IHIFSA)

The Fixed Storage Area is a single module (IHIFSA) containing housekeeping routines, a Common Data Area, and routines for conversion between real and integer representation. The housekeeping routines perform the functions of initialization and terminination, acquiring and releasing Data Storage Areas at entry to, and exit from, blocks and procedures, and branching between segments of generated code when procedures and actual parameters are called. The Common Data Area contains address constants and working tables referenced by the object module.

COMMON DATA AREA

The principal fields used by the object module in the Common Data Area are:

1. Two register save areas of 72 bytes each.

2. Address constants, including:
   Pointers to the top and bottom of the Return Address Stack (RAS) and to the last entries in the upper and lower sections of the Stack
   The address of the Data Set Table (DSTAB)
   The address of the Note Table (NOTTAB)
   The address of a location (FCTVALST) where values of user-defined type procedures are stored.

The Return Address Stack, Data Set Table, and Note Table are described in Chapter 11.

3. Counters, switches and storage locations, including:
   Program Block Number Counter (PROLPBN).
   Semicolon Counter (SCRCS)
   Option switches (OPTSW)-see Chapter 11.
   Storage location for values of user-defined type procedures (FCTVALST).
   Program Block Number Counter (PROLPBN).

4. Branch Instruction Table. The table consists of a set of branch instructions to routines in the Fixed Storage Area. The table is designed to permit deletions or changes in the Fixed Storage Area, without necessitating a multiplicity of corrections to addresses in the object code. The relevant branch instruction in the table is executed whenever a call is made to a Fixed Storage Area routine.

FIXED STORAGE AREA ROUTINES

The administrative routines of the Fixed Storage Area are listed and described below.

In the object module, the call to a Fixed Storage Area routine usually has the following form:

BAL REG,<DISPL>(FSA)

where REG represents register 8(ADR) or register 15 (BRR), and DISPL is the displacement of the branch instruction in the Branch Instruction Table (see item 4 in the Common Data Area, above).

Initialization (ALGIN)

ALGIN is the first routine to be entered at execution of the object module. ALGIN executes the SPIE macro instruction, specifying PIEROUT (see below) as the program interrupt exit; acquires main storage for, and initializes the Return Address Stack; sets the Option Switches (OPTSW) to reflect the options specified for the object module; and branches to the object module. The entry point to ALGIN is named IHIFSAIN. This is the location specified in the END card as the initial entry point of the object module. See Chapter 11. Error exits: 41, 42

Prologue (PROLOG)

PROLOG is invoked when a new block is entered and when a procedure is called. In the case of a block, PROLOG acquires a Data Storage Area (Figure 88) for the new block, initializes the Data Storage Area, and stores the address of the Data Storage Area in the corresponding entry of the Program Block Table (Figure 84). The latter entry specifies the required size of the Data Storage Area. An entry containing the same Data Storage Area address is also made in the Return Address Stack (Figure 90).

In the case of a procedure call, PROLOG acquires a Data Storage Area for the procedure and, after testing for compatibility between formal and actual parameters, stores the addresses of the actual parameter code sequences and the actual parameter characteristics, in the storage fields of the formal parameters in the new Data Storage Area. See Figure 63. An entry is also made in the Return Address Stack, containing the address of the acquired Data Storage Area and the return address, i.e. the address of the next instruction following the procedure call.

In both cases PROLOG passes control to the entered block or the called procedure. Base register CDSA addresses the new Data Storage Area.
Error exits: 20, 21, 36.

Epilogue (EPILOG)

EPILOG is invoked at the close of every block or procedure.

EPILOG releases the Data Storage Area (addressed by base register CDSA) of the exited block or procedure, and loads CDSA with the address of the Data Storage Area of the enclosing (or calling) block or procedure, contained in the second full word of the released Data Storage Area (Figure 86). At the same time, the address of the dynamically preceding Data Storage Area (if any) of the exited block or procedure, contained in the first full word of the released Data Storage Area, is stored in the corresponding entry of the Program Block Table, and the last entry of the Return Address Stack (Figure 90) is released.

If any precompiled procedure was called in the exited block or procedure, the precompiled procedure, whose address will have been entered in the Return Address Stack by LOADPP, is deleted. Similarly, if main storage was acquired for any arrays declared in the exited block (or value-specified in the exited procedure), the array storage area(s) and Storage Mapping Function area(s), if any, are released. The presence of any declared or specified arrays is determined by inspection of bytes 10 and 14 of the released Data Storage Area, while the address(es) of the array storage area(s) are obtained from the related Storage Mapping Function(s). See Chapter 11 and Figure 62.

In the case of an exited block, EPILOG passes control to the next instruction following the end of the block. In the case of a procedure, control is passed to the return address contained in the Return Address Stack.

Call Actual Parameter, Part 1 (CAP1)

CAP1 is invoked when a formal parameter called by value is encountered in a procedure heading, or when a formal parameter called by name is encountered in the body of a procedure.

CAP1 makes an entry in the Return Address Stack (Figure 90), containing the address of the current Data Storage Area and the return address (i.e., the address of the next instruction following the actual parameter call); stores the address of the dynamically preceding Data Storage Area (if any), contained in the first full word of the current Data Storage Area (Figure 88), in the corresponding Program Block Table entry (Figure 84); loads CDSA with the address of the enclosing (or calling) block, contained in the second full word of the current Data Storage Area; and branches to the actual parameter code sequence. See Figure 63.

Where the actual parameter call is enclosed by one or more intermediate blocks inside the procedure, an additional entry is made in the Return Address Stack for every intermediate block, containing the address of the related Data Storage Area and a zero return address, and the Program Block Table is updated to show the address of the dynamically preceding Data Storage Area, if any, for every intermediate block. The number of entries made in the Return Address Stack is governed by a comparison between the Program Block Number of the procedure and the Program Block Number contained in the Data Storage Area(s) of each intermediate block (if any) enclosing the parameter call inside the procedure.

Call Actual Parameter, Part 2 (CAP2)

CAP2 is entered after the code sequence of an actual parameter has been executed as a result of an actual parameter call in a procedure. See Figure 63.

CAP2 loads base register CDSA with the address (contained in the last Return Address Stack entry) of the Data Storage Area of the procedure calling the actual parameter; stores CDSA in the related Program Block Table entry (Figure 84); releases the Return Address Stack entry (Figure 90); and branches to the return address (the address of the instruction following the actual parameter call) indicated in the Return Address Stack.

Where the actual parameter call is enclosed by one or more intermediate blocks inside the procedure, CAP2 stores the Data Storage Area address listed in each of the corresponding Return Address Stack entries, in the related Program Block Table entries, and releases the entries. The number of entries released is governed by the return address in each entry. A non-zero return address identifies the last entry to be released. The latter entry corresponds to the block or procedure immediately encompassing the actual parameter call.

Value Call (VALUCALL)

VALUCALL is invoked when a formal parameter called by value is encountered in a procedure heading. Immediately before VALUCALL is entered, the actual parameter code sequence will have been executed, and the address of the actual parameter will have been loaded in register ADR. VALUCALL stores the actual parameter's value in the formal parameter's storage field (Figure 89) and returns control to the return address in the procedure heading (contained in register BRR). Where necessary, the Fixed Storage Area conversion routines, CNVIRD or CNVRDI, may be called in order to convert the actual parameter to a real or integer value, according to the type of the formal parameter.

If the parameter is a value-specified array, the Fixed Storage Area routine GETMSTO is called, which allocates space for the array and its Storage Mapping Function. The Storage Mapping Function (Figure 62) is then copied from the block in which it is declared (and modified to indicate the new location), and the array is transferred (and converted if necessary) to the new location.

Return Routine (RETPROG)

RETPROG is invoked by the object module when a goto statement (which may lead out of the current block) is executed. RETPROG compares the address of the Data Storage Area of the target block (previously loaded in base register CDSA) with the Data Storage Area address contained in the last Return Address Stack entry (Figure 90) and, if they differ (indicating that the goto statement implies a branch out of the current block), calls the EPILOG routine, which releases the Data Storage Area addressed by the Return Address Stack entry, and then releases the entry. This procedure is repeated until the Return Address Stack entry is found which corresponds to the target block. At this point, RETPROG passes control to the branch address in the target block (previously loaded in ADR).

Call Switch Element, Part 1 (CSWE1)

CSWE1 is invoked by the object module when a switch designator is encountered. See Figure 61.

CSWE1 (whose function is analogous to that of CAP1, see above) makes one or more entries in the Return Address Stack (Figure 90), to reflect the intervening block structure between the switch designator and the switch declaration; updates the Program Block Table; and branches to the switch element in the switch declaration.

Call Switch Element, Part 2 (CSWE2)

CSWE2 is invoked by the object module after the code sequence representing a switch element in a switch declaration has been executed, in response to a switch designator. See Figure 61.

CSWE2 (whose function is analogous to that of CAP2, see above) releases one or more entries in the Return Address Stack; updates the Program Block Table; and returns control to the next instruction following the switch designator (whose address is specified in the Return Address Stack).

Trace (TRACE)

TRACE is invoked by the object module at the end of the code representing each source statement, provided the compile time TEST option is specified. TRACE copies the semicolon count (recorded in the object module following the call to TRACE)into the Fixed Storage Area field named SCRCS and (provided one of the execution time options TRACE, TRBEG, or TREND is specified) into the program trace list. If the trace list buffer is filled, the contents of the buffer are written on SYSUT1.

Load Precompiled Procedure (LOADPP)

LOADPP is invoked by the object module when a precompiled procedure is declared. This routine loads the precompiled procedure by issuing a LOAD macro, and returns to the instruction following the procedure declaration. The Program Block Table and Return Address Stack are updated to show the entry point and name of the precompiled procedure.
Error exit: 36

Standard Procedure Declaration (SPDECL)

SPDECL is invoked by the object module if the name of a standard I/O procedure or mathematical function is an actual parameter in another procedure. This routine constructs a parameter list for the parameter(s) to be operated on, and branches to the standard procedure or function called. The parameter list is constructed after executing the actual parameter code sequences.

Get Main Storage (GETMSTO)

GETMSTO is invoked by VALUCALL or by the object module when space for an array is to be allocated. This routine gets the main storage required for the array.
Error exit: 18

Program Interrupt (PIEROUT)

PIEROUT is invoked by the operating system when a program interrupt occurs in the object program. This routine handles the program status word, and then branches to the FSA error routine FSAERR.
Error exits: 28,29,30,31,33

FSAERR

Invoked by the object program or other routines in the ALGOL library when an error occurs. This routine stores the number of the error in a storage area, FSAERCOD, before invoking the error routine, IHIERR (in SYS1.LINKLIB) by means of a LINK macro.
Error exit: None

Termination (ALGTRMN)

Invoked when the end of the program is reached, or by FSAERR. This routine prints out the program trace list, closes all data sets, and frees the Return Address Stack. It then restores registers and returns to the calling program.
Error exit: None

Integer to Real Conversion (CNVIRD)

Invoked by other routines in the ALGOL library and by the object program. This routine converts an integer number into a real number of either short or long precision.
Error exit: None

Real to Integer Conversion (CNVRDI/ENTIER)

Invoked by other routines in the ALGOL library and by the object program. This routine has two entry points. CNVRDI is used internally by the other routines. ENTIER is used when 'ENTIER' is encountered

in the object program. The routine converts a real number of either short or long precision into an integer number.
Error exit: 40

INPUT/OUTPUT PROCEDURES

The Library contains the following I/O procedures:

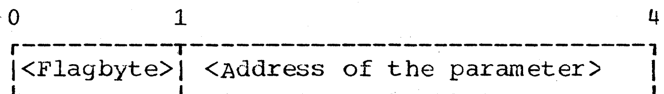
Module Name	I/O Procedure
IHIGPR	GET, PUT
IHIIAR	INARRAY, INTARRAY
IHIBA	INBARRAY
IHIIBO	INBOOLEAN
IHIIDE	INREAL, ININTEGER
IHIISY	INSYMBOL
IHILOR	OUTREAL (Long precision)
IHISOR	OUTREAL (Short precision
IHIOAR	OUTARRAY
IHIOBA	OUTBARRAY
IHIOBO	OUTBOOLEAN
IHIOIN	OUTINTEGER
IHIOST	OUTSTRING
IHIOSY	OUTSYMBOL
IHIOTA	OUTTARRAY
IHISYS	SYSACT
IHIPTT	Power of Ten Table

All I/O procedures utilize one or more subroutines contained in a common subroutine pool comprising load module IHIIOR.

In the object code generated by the Compiler, a call to an I/O procedure has the following form

```
BALR   RETURN,  ENTRY
```

where RETURN and ENTRY are the symbolic names of registers 14 and 15, respectively. PARAM (register 1) addresses a parameter list consisting of two or three full-word entries.


0 1 4
<Flagbyte>	<Address of the parameter>

<Flagbyte>: Bit 0: <Parameter to be converted (1) or not (0)>
Bit 1: <Parameter may be assigned a value (0) or not (1)>
Bits 2-7: (Not used)

Figure 79. Four-byte parameter list entry for a standard procedure call

Each parameter entry references a parameter which specifies the operation to be performed by the I/O procedure, as follows:

SYSACT: parameter 1: Data Set Number
parameter 2: Function No. 1-15
parameter 3: Quantity

GET/PUT: parameter 1: Record Number
parameter 2: Address of LIST procedure

INSYMBOL/ OUTSYMBOL: Parameter 1: Data Set Number
parameter 2: String
parameter 3: Destination or Source

OUTSTRING: parameter 1: Data Set Number
parameter 2: String

All other procedures: parameter 1: Data Set Number
2: Destination or Source

PUT/GET (IHIGPR)

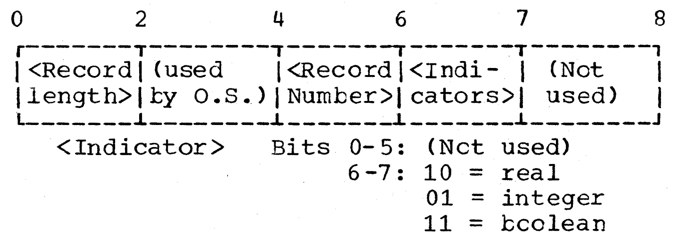
This module handles the transfer of data between the ALGOL program and the SYSUT2 data set without conversion to external format. It contains two primary and eight subroutines, as follows:

Primary Routines: PUT
GET

Subroutines: OUTPUT
INPUT
OPENGP
CLOSEGP
OPENEXIT
CAP1GP
THUNKOUT
THUNKIN

PUT

Invoked by the PUT I/O procedure. This routine transfers the data specified in the LIST procedure in the ALGOL program to the SYSUT2 data set. All data handled by a single PUT is stored as one record (maximum length 2K bytes). The first eight bytes of each record have a standard format (see Figure 80).


0 2 4 6 7 8
<Record length>	(used by O.S.)	<Record Number>	<Indicators>	(Not used)

<Indicator> Bits 0-5: (Not used)
6-7: 10 = real
01 = integer
11 = boolean

Figure 80. Label of a PUT/GET record

If the last PUT/GET operation was PUT, then the current PUT routine checks this operation (with the CHECK macro) and makes an entry for it in the Note Table (Figure 87).
Error exits: 39,43

GET

Invoked by the GET I/O procedure. This routine retrieves the data from the SYSUT2 data set to the parameters specified in the LIST procedure in the ALGOL program.
Error exits: 10,14,39,43

OUTPUT

Invoked by the LIST procedure via PROLOG. This routine transfers, one parameter at a time, data from the ALGOL program to the output buffer.
Error exits: 20, 21, 38, 43

INPUT

Invoked by the LIST procedure via PROLOG. This routine transfers, one parameter at a time, data from the input buffer to the ALGOL program.
Error exits: 20,21,38,43

OPENGP

Invoked by the PUT routine if SYSUT2 is not open. This routine opens the SYSUT2 data set. The DCB field is the same as for the other data sets used at execution time (see OPEN) except:

```
RECFM=V
EXLST=A(AOPENEXIT)
[AOPENEXIT=address of OPENEXIT routine]
```

Error exit: 41

CLOSEGP

Invoked by CLOSEPE if SYSUT2 is open. This routine closes the SYSUT2 data set.

OPENEXIT

Invoked by OPENGP. This routine tests the blocksize if it was specified in the DD card for SYSUT2. The default value is 2K.
Error exit: None

CAP1GP

Invoked by INPUT and OUTPUT. This routine calls the actual parameter for INPUT and OUTPUT. It is a copy of the CAP1 routine in the Fixed Storage Area.
Error exit: 36

THUNKOUT

Invoked by the LIST procedure for PUT calling an actual parameter. This routine assigns OUTPUT as an actual parameter to the LIST procedure.
Error exit: None

THUNKIN

Invoked by the LIST procedure for GET when calling an actual parameter. This routine assigns INPUT as an actual parameter to the LIST procedure.
Error exits: None

INARRAY/INTARRAY(IHIIAR)

This module consists of a single routine, INARRAY/INTARRAY, with two entry points. It repeatedly invokes INREAL/ININTEGER until the end of the array is reached.
Error exit: None

INBARRAY (IHIIBA)

This module consists of a single routine, INBARRAY, which repeatedly invokes INBOOLEAN until the end of the array is reached.
Error exit: None

INBOOLEAN (IHIBO)

This module consists of a single routine, INBOOLEAN, which reads the next Boolean value from the record and assigns it to DESTINATION. A 6-byte field, MB, is used to store characters following an apostrophe to see if they specify a Boolean value, i.e. 'TRUE' or 'FALSE'. If they do not, then the characters after the next apostrophe are examined until a Boolean value is found.
Error exit: 2,3,5

INREAL/ININTEGER (IHIIDE)

This module consists of a single routine, INREAL/ININTEGER, with three entry points. It reads the next real or integer value from the record and assigns it to DESTINATION. A flagbyte FKT, is set to 0 if the INREAL entry point is used and to 4 if the ININTEGER entry point is used.
Error exits: 2,3,5,6

The subroutine CONVERT (see subroutine pool below) may be called if a conversion from real to integer is required. If a conversion from integer to real is required, the Fixed Storage Area routine CNVIRD is called. For transforming numbers to floating-point or fixed-point representation a power-of-ten table (contained in module IHIPTT) is used.

INSYMBOL (IHIISY)

This module compares the next character on the data set specified by the first parameter, with each of the characters in the character string specified in the second parameter, and, when a match is identified, assigns a number to the variable specified in the third parameter, designating the position of the matching character in the string. If no match is identified, the variable is assigned the value 0.
Error exits: 2,3,5

OUTREAL (IHISOR)

This module consists of a single routine, OUTREAL-SHORT, which transfers a short-precision real number from SOURCE to a data set. It dissects the number into its mantissa and exponent parts by a series of diminishing approximations (making use of an internal power-of-ten table) and converts to external format.
Error exit: 38

OUTREAL (IHILOR)

This module consists of a single routine, OUTREAL-LONG, which transfers a long-precision real number from SOURCE to a data set. It dissects the number into its mantissa and exponent parts by a series of diminishing approximations (making use of IHIPTT) and converts to external format.
Error exit: 38

OUTARRAY (IHIOAR)

This module consists of a single routine, OUTARRAY, which repeatedly invokes OUTREAL-SHORT or OUTREAL-LONG until the end of the array is reached.
Error exit: None

OUTBARRAY (IHIOBA)

This module consists of a single routine, OUTBARRAY, which repeatedly invokes OUTBOOLEAN until the end of the array is reached.
Error exit: None

OUTBOOLEAN (IHIOBO)

Invoked by OUTBOOLEAN I/O procedure. This module consists of a single routine, OUTBOOLEAN, which transfers a Boolean value to a data set, depending on the value of SOURCE.
Error exit: 38

OUTINTEGER (IHIOIN)

This module consists of a single routine, OUTINTEGER, which transfers an integer number from SOURCE to a data set.
Error exit: 38

OUTSTRING (IHIOST)

This module consists of a single routine, OUTSTRING, which transfers a string from STRING to a data set.
Error exit: None

OUTSYMBOL (IHIOSY)

This module selects a character from the character string specified in the second parameter, according to the positional number designated by the third parameter, and transfers the character selected to the data set specified by the first parameter.
Error exit: 8

OUTTARRAY (IHIOTA)

This module consists of a single routine, OUTTARRAY, which repeatedly invokes OUTINTEGER until the end of the array is reached.
Error exit: None

SYSACT (IHISYS)

This module consists of the main routine, SYSACT, and 15 subroutines, SYSACT1 to SYSACT15. They influence the execution of an I/O operation by either assigning a value from DSTAB to QUANTITY, or using the value of QUANTITY to reposition pointers and respecify indicators in the Data Set Table (Figure 86).

SYSACT

This routine calls the appropriate subroutine according to the number specified in FUNCTION.
Error exit: 9

SYSACT1

This routine assigns the number of the current character in the record (R-RE+P+1) to QUANTITY (Figure 86).
Error exits: 10,12

SYSACT2

This routine skips forward to the character whose number is indicated by QUANTITY (R=QUANTITY+RE-P-1); either in the current record when QUANTITY is greater than the number of the current character; or in the next record when QUANTITY is less or equal to the number of the current character. The skipped positions are filled with blanks if the last I/O operation performed output.
Error exits: 2,5,10,13

SYSACT3

This routine assigns the number of the current record (S) to QUANTITY.
Error exits: 10,12

SYSACT4

This routine positions the record pointer at the record whose number is specified by the third parameter. For backward repositioning, an entry in NOTTAB (Figure 87) is required. For forward repositioning, the skipped records are filled with blanks if the last I/O operation performed output.
Error exits: 2,5,10,13,14

SYSACT5

This routine assigns the value of the record length indicator (P) to QUANTITY.
Error exits: 10,12

SYSACT6

This routine assigns the value of QUANTITY to the record length indicator (P).
Error exits: 11,13

SYSACT7

This routine assigns the value of the indicator for the number of records per section (Q) to QUANTITY.
Error exit: 12

SYSACT8

This routine assigns the value of QUANTITY to the indicator for the number of records/section (Q).
Error exits: 0,2,11,13

SYSACT9

This routine assigns the value of the indicator for the number of blanks recognized as a delimiter (K) to QUANTITY.
Error exit: 12

SYSACT10

This routine assigns the value of QUANTITY to the indicator for the number of blanks recognized as a delimiter (K).
Error exit: 13

SYSACT11

This routine assigns a value to QUANTITY according to the status of the data set.
Error exit: 12

SYSACT12

This routine opens a closed data set when QUANTITY=1, and closes an open data set when QUANTITY=0.
Error exit: 13

SYSACT13

This routine assigns the number of the current record (S) to QUANTITY, and also sets flagbit (DS4) which causes the current record to have an entry put in NOTTAB when NEXTREC is called.
Error exits: 2,10,12

SYSACT14

For a sectioned data set, this routine passes over the number of records (inclusive of current one) specified by QUANTITY, or to beginning of the next section, if the number of records left in the section is less than QUANTITY. For an unsectioned data set, this routine passes over the number of records (inclusive of current one) specified by QUANTITY.

The skipped positions and records are filled with blanks if the last I/O operation performed output.
Error exits: 2,5,10,13

SYSACT15

For a sectioned set, this routine passes to the record, in the next section, whose number is specified by QUANTITY. The skipped positions are filled with blanks.

For an unsectioned data set, this routine invokes SYSACT14 in order to pass over the number of records (inclusive of current one) specified by QUANTITY.
Error exits: 10,13

SUBROUTINE POOL (IHIIOR)

This module comprises eleven routines which perform common functions required by all I/O modules, and by IHIERR and IHIFSA. The routines are:

CLEARNOTTAB
CLOSE
CLOSEPE
CONVERT
DCBEXIT
ENDOFDATA
ENTRYNOTTAB
EVDSN
NEXTREC
OPEN
SYNAD

CLEARNOTTAB

Invoked by NEXTREC, CLOSE. This routine deletes the entries in NOTTAB (Figure 87) for records which are no longer positioned behind the current record, when output has occurred.
Error exit: None

CLOSE

Invoked by CLOSEPE, SYSACT12, and TERMNTE (in IHIFSA) and IHIERR. This routine closes a data set.
Error exit: None

CLOSEPE

Invoked via TERMNTE at end of the ALGOL program execution. This routine closes all data sets by repeatedly invoking CLOSE for any that are open (CLOSEGP for SYSUT2).
Error exit: None

CONVERT

Invoked by I/O routines when a conversion from real to integer is required. This routine is a copy of CNVRDI/ENTIER contained in the Fixed Storage Area.
Error exit: 1

DCBEXIT

Invoked by OPEN. This routine establishes the Record Format, Record Length and Blocksize DCB information for an execution time data set.
Error exit: None

ENDOFDATA

Invoked by CHECK macro during NEXTREC, CLOSE, SYSACT4, or OPEN, if the last record has been read. This routine stops the retrieval of another record.
Error exits: 5

ENTRYNOTTAB

Invoked by NEXTREC, ENDOFDATA, PUT, or GET. This routine makes an entry in NOTTAB by inserting the data set number (from register DSN), record identification number (from DSTAB) and feedback address (stored by a previous NOTE macro).
Error exit: 4

EVDSN

Invoked by the modules that handle all I/O procedures except PUT and GET. This routine evaluates the data set number specified for the I/O procedure.
Error exit:

NEXTREC

Invoked by the modules that handle all I/O procedures except INARRAY/INTARRAY, INBARRAY, OUTARRAY, OUTTARRAY, OUTBARRAY, PUT, and GET. It is also invoked by IHIERR and TERMNTE. For blocked records, this routine gets the next record in the block. If the end of the block has been reached it reads or writes the block and gets a new block. For an unblocked record, this routine reads or writes the record, makes an entry in NOTTAB (Figure 87) if required, and gets the next record.
Error exits: 2, 3

OPEN

Invoked by the modules that handle all I/O procedures except PUT and GET and by TERMNTE and IHIERR. This routine opens a data set and gets main storage for NOTTAB (Figure 87). It assigns initial addresses to pointers in DSTAB (Figure 86), and stores the start address of NOTTAB in the fixed storage area. The standard DCB model is:

```
    DSORG  = PS
    MACRF  = RP, WP
    DDNAME = SYSIN when DSN = 0
             SYSPRINT when DSN = 1
             ALGLDDbb when DSN = 2
             to 15 (bb is replaced by DSN)
    NCP    = 1
    EODAD  = A (ENDOFDATA)
    EXLST  = A (DCBEXIT)
    SYNAD  = A (SYNAD)
```

Error exit: 2, 7, 37, 41

SYNAD

Invoked by CHECK macro during OPEN, PUT, GET, NEXTREC, CLOSE, or SYSACT4 if an unrecoverable I/O error occurs. This routine passes to error exit 32.
Error exit: 32

MATHEMATICAL STANDARD FUNCTIONS

The table in Figure 81 lists the modules contained in the Library which perform the standard mathematical functions indicated. The routines have been taken from the FORTRAN IV Library, and minor modifications have been incorporated to suit ALGOL requirements. The routines are described in OS FORTRAN IV Library.

OBJECT TIME ERROR ROUTINE (IHIERR)

The Object Time Error Routine (IHIERR) is a module of the SYS1.LINKLIB data set. It is loaded into main storage (on call from FSAERR), only if an error is detected in the object program. IHIERR constructs and prints out the error message on the SYSPRINT data set, before entering TERMNTE, which terminates the object program. If the DUMP option is specified, IHIERR also edits and prints out the contents of the existing Data Storage Areas. The addresses of a series of dynamically or logically enclosing blocks or procedures are obtained by calling the EPILOG routine.

Function	Precision	ALGOL# Library Module Name	ALGOL# Library Entry Name	FORTRAN IV (E) Library Module Name	FORTRAN IV (E) Library Entry Name
LN	Long	IHILLO	IHILLO	IHCILOG	DLOG
	Short	IHISLO	IHISLO	IHCLOG	ALOG
EXP	Long	IHILEX	IHILEX	IHCLEXP	DEXP
	Short	IHISEX	IHISEX	IHCSEXP	EXP
SQRT	Long	IHILSQ	IHILSQ	IHCLSQRT	DSQRT
	Short	IHISSQ	IHISSQ	IHCSSQRT	SQRT
ARCTAN	Long	IHILAT	IHILAT	IHCLATAN	DATAN
	Short	IHISAT	IHISAT	IHCSATAN	ATAN
SIN		IHILSC	IHILSCS	IHCLSCN	DSIN
COS	Long	IHILSC	IHILSCC	IHCLSCN	DCOS
SIN		IHISSC	IHISSCS	IHCSSCN	SIN
COS	Short	IHISSC	IHISSCC	IHCSSCN	COS
POWER:	1)	IHIFII	IHIFII	IHCFIXPI	FIXPI#
	2)	IHIFRI	IHIFRI	IHCFRXPI	FRXPI#
	3)	IHIFDI	IHIFDI	IHCFDXPI	FDXPI#
	4)	IHIFRR	IHIFRR	IHCFRXPR	FRXPR#
	5)	IHIFDD	IHIFDD	IHCFDXPD	FDXPD#

Notes:
1) Base and Exponent: fixed point
2) Base: floating point (short) ; Exponent: fixed point
3) Base: floating point (long); Exponent: fixed point
4) Base: floating point (short) ; Exponent: floating point (short)
5) Base: floating point (long); Exponent: floating point (long)

Figure 81. Module names of mathematical standard functions contained in the ALGOL Library

The output of the Compiler is an object module, consisting of machine language instructions, constants, and object time tables. The object module is output as a card deck on SYSPUNCH and/or transferred to the SYSLIN data set (in card-image ESD, TXT, RLD and END records) depending on the options specified in the invoking EXEC statement. The object module contains an ESD record entry for each Library procedure called in the source module and for the Fixed Storage Area, as well as RLD record entries for all address constants.

The object module is processed by the Linkage Editor, which combines the Fixed Storage Area and the called Library routines with the generated code, to form a single load module. The RLD records provided in the object module are used by the Linkage Editor to change relative address constants to absolute address constants.

OBJECT MODULE

Figure 82 indicates the composition of the object module generated by the Compiler.

IHIFSAIN is the symbolic address of the entry point of the object module. IHIFSAIN is specified in the END record if the compilation option is PROGRAM. It designates the address of the initial entry routine in the Fixed Storage Area.

If the compilation option is PROCEDURE the entry point specified inthe END record is IHIENTIF.

IHIENTIF is the symbolic address of an Address List, containing theaddresses of the object time Program Block Table, the Label Address Table and the start address of the generated instructions. The Address list is referenced by the Initialization routine and by other routines in the Fixed Storage Area.

Object Module

Record Type	Content
ESD	Control section definition. Contains procedure name if the PROCEDURE option is specified.
TXT	Constant Pool(s)
TXT	" "
"	" "
TXT	Generated object code
TXT	" " "
"	" " "
RLD	(Param. and switch list addresses)
RLD	" " " " "
TXT	
TXT	
"	
TXT	Label Address Table (LAT)
TXT	(Declared procedure, switch and label addresses)
RLD	(Internal branch addresses)
RLD	" " "
"	" " "
ESD	(Library routines called)
ESD	" " "
"	" " "
RLD	" " "
RLD	" " "
"	" " "
TXT	Program Block Table (PBT)
TXT	" " " "
"	" " " "
RLD	(Constant Pool addresses)
RLD	" " "
"	" " "
TXT	Data Set Table (DSTAB)
TXT	" " " "
"	" " " " } *
RLD	Put/Get Control Field address } *
IHIENTIF:	Address Table
TXT	Addresses of Program Block Table
TXT	Addresses of Label Address Table
TXT	Addresses of first instruction in object module
ESD	IHIFSAIN - Initial entry routine * in Fixed Storage Area
RLD	Address of Program Block Table
RLD	Address of Label Address Table
RLD	Address of first instruction
ESD	IHIENTIF - address of Address Table } *
ESD	DSTAB: addr. of Data Set Table } *
END	IHIFSAIN if compilation option is PROGRAM. (IHIENTIF if compilation option is PROCEDURE)

*generated only if the PROGRAM option is specified

Figure 82. Composition of the object module

LOAD MODULE

Figure 83 pictures the organization of the load module constructed by the Linkage Editor. The layout of routines and tables is indicated in detail in the storage map in Appendix IX-b.

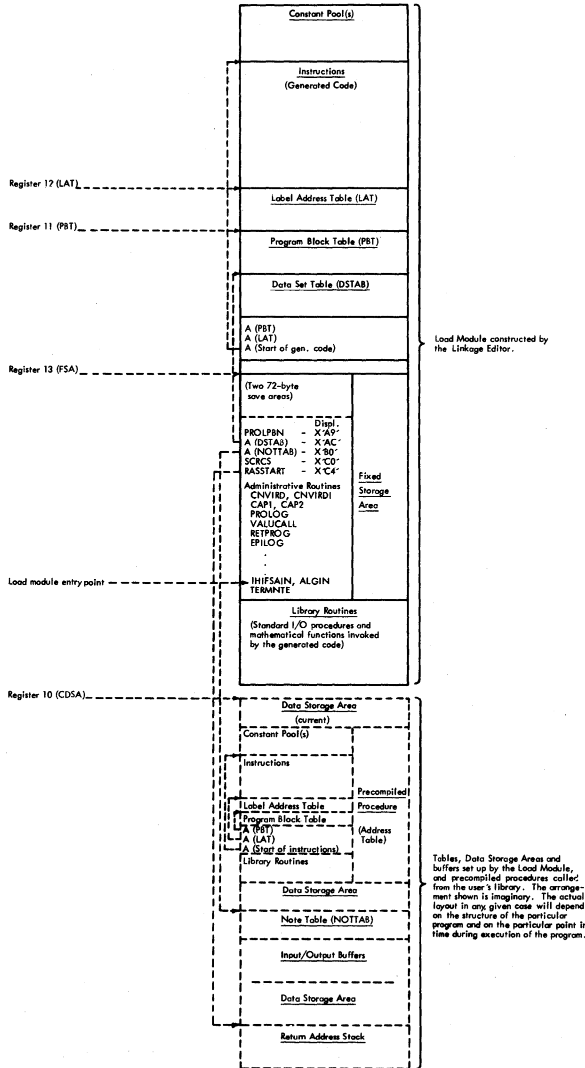


Figure 83. Sketch showing the organization of the load module and the use of main storage by a hypothetical load module.

The logical structure of the code generated for specific logical features of the source module is discussed under **various** headings in Chapter 8. Figure 58 lists the principal sections of Chapter 8. Each section indicates the part played by the related Fixed Storage Area routines in the load module.

OBJECT TIME TABLES

The tables used by the load module at execution time are discussed below. Figure 83 indicates the location of the various tables in the load module.

PROGRAM BLOCK TABLE (PBT)

The Program Block Table (Figure 84) is the object time version of Program Block Table IV, constructed **by the Termination** Phase (IEX51 - Chapter 8).

The Program Block Table has three main functions:

1. To identify the character of every scope (block, procedure, type procedure, or code procedure) in the program. The character of the scope determines the particular action taken by the Fixed Storage Area routine PROLOG, which **is called when** a block is entered or a procedure **called.** See below.

2. To specify the size of the Data Storage Area required **by every** block and procedure. Main **storage** for the required Data Storage Area is acquired by PROLOG as soon as the block is entered or the procedure is called.

3. To provide a record of the base address of the Data Storage Area. The base address, which is recorded immediately after acquisition of the Data Storage Area, is used in addressing any variable declared or specified in the particular block or procedure.

Every block and procedure is identified by a serial Program Block Number. The Program Block Number designates a corresponding entry in the Program Block Table. It is specified in every call to the Fixed Storage Area routines PROLOG and EPILOG, which are invoked at entry to and at exit from a block or procedure.

The particular action taken by PROLOG at entry to a given scope depends on the character of the scope, as indicated below.

<u>Block</u> A Data Storage Area is acquired for the entered block, its address being stored in the corresponding Program Block Table entry and loaded into base register 10 (CDSA). Control is then passed to the first instruction in the block.

<u>Procedure-Call</u> After a Data Storage Area has been acquired, as in the case of a block, the addresses and characteristics of the actual parameters specified in the procedure call are moved into the storage fields reserved for the formal parameters in the new Data Storage Area. The procedure is then entered. See Figure 63.

<u>Type-Procedure-Call</u> The action taken is the same as in the case of a normal procedure. There is a difference (of 8 bytes), however, in the displacement of the formal parameter storage fields in the Data Storage Area.

<u>Code Procedure Call</u> In the case of a code procedure, the Program Block Table entry contains the address of an address list within the corresponding precompiled procedure. See Figure 83. (The address is stored in the entry by the Fixed Storage Area routine LOADPP, after loading the precompiled procedure into main storage from the user's library. The precompiled procedure is loaded when the code procedure is declared.) The address list contains the addresses of the precompiled procedure's own Program Block and Label Address Tables as well as the entry point address. PROLOG loads registers 11 (PBT), 12 (LAT) and 8 (ADR) with the addresses in the address list, and then acquires a Data Storage Area for the precompiled procedure, using the size specified in the first entry of the precompiled procedure's own Program Block Table. The addresses and characteristics of the actual parameters are then moved into the new Data Storage Area, and the precompiled procedure is activated.

The Program Block Table is consulted and updated by the following Fixed Storage Area routines, in addition to PROLOG, EPILOG and LOADPP: CAP1, CAP2, CSWE1 and CSWE2.

Entry for:	0	4		8
Constant Pool 0:	<Address of Constant Pool 0>	<Job step name or procedure name>*		
P.B. No. 1:	<Adress of DSA/Address of precompiled procedure/All zeros>	<Size of DSA>	<Scope Code>	<No. of param.>
P.B. No. 2:	<Address of DSA/Address of precompiled procedure/All zeros>	<Size of DSA>	<Scope Code>	<No. of param.>
P.B. No. 3:	<Address of DSA/Address of precompiled procedure/All zeros>	<Size of DSA>	<Scope Code>	<No. of param.>
. . . .	. . . .	. . . .		
P.B. No. N:	<Address of DSA/Address of precompiled procedure/All zeros>	<Size of DSA>	<Scope Code>	<No. of param.>
Constant Pool 1 (if any):	<Address of Constant Pool 1>	<All zeros>		
Constant Pool 2 (if any):	<Address of Constant Pool 2>	<All zeros>		

<Scope Code> - X'00' = Block
X'04' = Non-type procedure
X'08' = Type procedure
X'10' = Code procedure

<No. of param.> = <Number of parameters (zero for a block)>

*<Job step name> if the module is a program,
<Procedure name> if the module is a precompiled procedure

•Figure 84. Object time Program Block Table (PBT)

Label Address Table LAT)

The Label Address Table contains a maximum of 256 four-byte entries, containing absolute addresses needed by the load module. The first 28 entries are reserved for the addresses of ALGOL Library modules representing standard I/O procedures and mathematical functions which have been combined with the load module. If a given Library module has been combined with the load module, the corresponding full-word entry in the Label Address Table will contain the entry point address of the Library module; otherwise, the entry will contain the value X'80000000'. The full-word entry which corresponds to each standard I/O procedure or mathematical function is indicated by the displacement data in the internal names of the standard procedures listed in Appendix III.

Displacement	Entry
00	<Address of SYSACT or X'80000000'>
04	<Address of SQRT or X'80000000'>
08	<Address of SIN or X'80000000'>
0C	<Address of COS or X'80000000'>
10	<Address of ARCTAN or X'80000000'>
14	<Address of LN or X'80000000'>
18	<Address of EXP or X'80000000'>
1C	<Address of INSYMBOL or X'80000000'>
20	<Address of INREAL or X'80000000'>
24	<Address of ININTEGER or X'80000000'>
28	<Address of INBOOLEAN or X'80000000'>
2C	<Address of INARRAY or X'80000000'>
30	<Address of INTARRAY or X'80000000'>
34	<Address of INBARRAY or X'80000000'>
38	<Address of OUTSYMBOL or X'80000000'>
3C	<Address of OUTREAL or X'80000000'>
40	<Address of OUTINTEGER or X'80000000'>
44	<Address of OUTBOOLEAN or X'80000000'>
48	<Address of OUTARRAY or X'80000000'>
4C	<Address of OUTTARRAY or X'80000000'>
50	<Address of OUTBARRAY or X'80000000'>
54	<Address of OUTSTRING or X'80000000'>
58	<Address of PUT or X'80000000'>
5C	<Address of GET or X'80000000'>
60	<Address of POWER-FiFi* or X'80000000'>
64	<Address of POWER-FIFi* or X'80000000'>
68	<Address of POWER-FIFI* or X'80000000'>
6C	<X'80000000'>
70	(Addresses of labels, procedures and switches declared in the source module)
	(Branch addresses generated by the Compiler)
Max. 0400	

*Fi = Fixed Point; FI = Floating Point. The abbreviations designate the representation of the base and exponent, respectively.

•Figure 85. Object time Label Address Table (LAT)

Data Set Table (DSTAB)

The Data Set Table is constructed by the Termination Phase (IEX51 - Chapter 8). The table is consulted and updated at execution time by the standard I/O procedures invoked by the load module, and indicates the status of all data sets. Each entry provides a record, among other things, for the address of the relevant Data Control Block, for other specific data required to execute and I/O operation, as well as for the temporary storage of variable address information.

The Data Set Table contains a 36-byte entry for data sets Nos. 0 and 1 and for every other data set used by the load module, as well as a 28-byte entry (called the PUT/GET Control Field) at the end of the table, if the load module executes a PUT or GET macro instruction. The first full-word in the table, named APGCF, contains the address of the PUT/GET Control Field, or, if the latter is not present, the hexadecimal value X'80000000'.

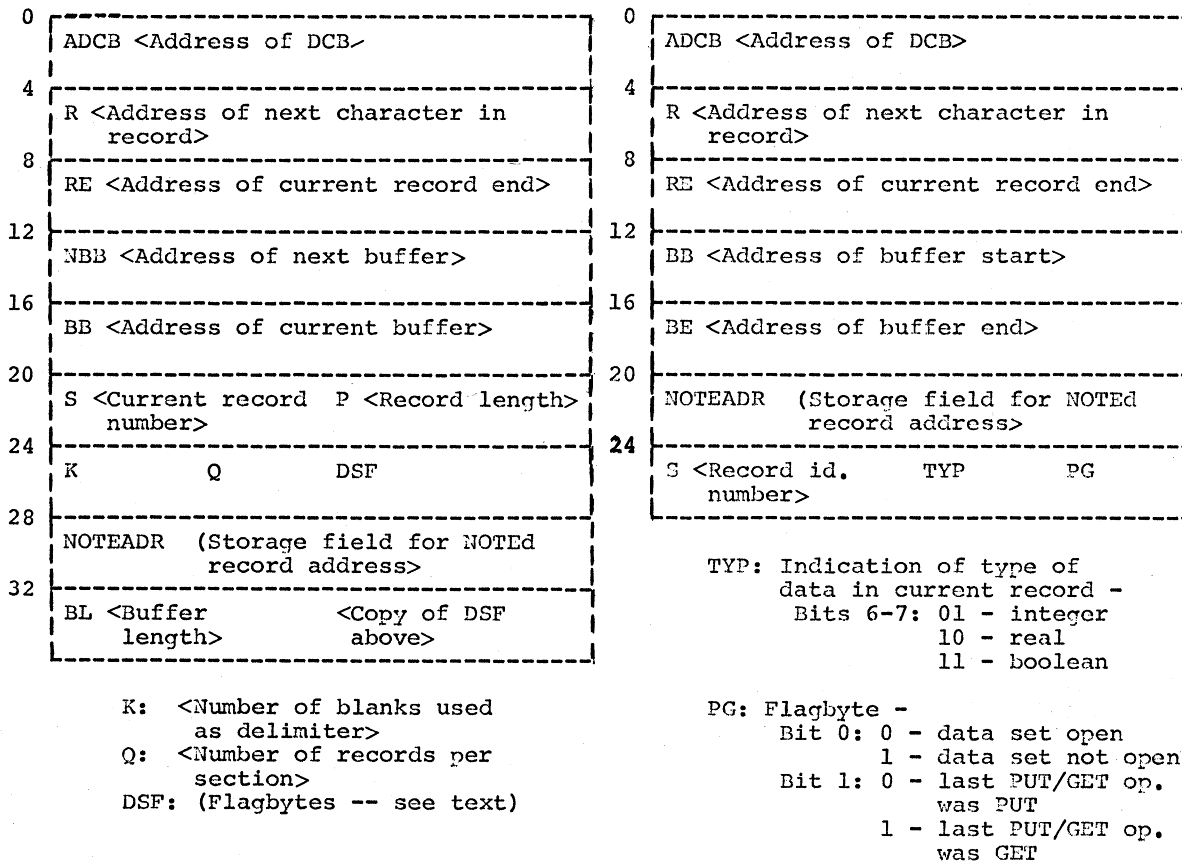


Figure 86. Content of the data set entries and the PUT/GET Control Field in the Data Set Table

Bytes 26 and 27 of the data set entry (named DSF) are used as flagbytes. The significance associated with each of the binary positions used is as follows:

DSF-

byte 26-bit 0= 0 - Data set is not open
- 1 - Data set is open
- 1= 0 - Records are blocked
- 1 - Records are unblocked
- 2= 0 - Last I/O operation was input
- 1 - Last I/O operation was output
- 3= 0 - In the current block no output has occurred
- 1 - In the current block output has occurred
- 4= 0 - -
- 1 - An entry in NOTTAB is required when the next record is addressed
- 5= 0 - Backward repositioning was not performed
- 1 - Backward repositioning was performed
- 6= 0 - Last block was not output
- 1 - Last block was output
- 7= 0 - End of data has not been reached
- 1 - End of data has been reached

byte 27-bit 0= 0 - -
- 1 - Data set has been repositioned backward after an input operation or data set is being closed.
- 1= 0 - Record does not contain a control character
- 1 - Record contains a control character.
- 2 (not used)
- 3= 0 - SYSPRINT is not to be opened for diagnostics or termination messages
- 1 - SYSPRINT is to be opened for diagnostics or termination messages
- 4= 0 - -
- 1 - Data set is being opened
- 5= 0 - -
- 1 - An unrecoverable I/O operation has occurred.
- 6= 0 - Data set has never been opened
- 1 - Data set has been opened
- 7= 0 - Data set 1 is not to be closed
- 1 - Data set 1 is to be closed

The initial values inserted in the various fields of the data set entries and the PUT/GET Control Field are indicated in Chapter 8 under "Data Set Table".

Note Table (NOTTAB)

The Note Table is an index which is consulted and updated by the ALGOL Library procedures PUT, GET, SYSACT(13) and SYSACT(4). The table provides space for up to 127 8-byte entries containing a code number identifying a particular PUT/GET operation or a record on a given data set, and a data set address specifying the location on the data set where the PUT/GET operation was started or where the record is stored. The entries serve to facilitate the retrieval of data from a data set.

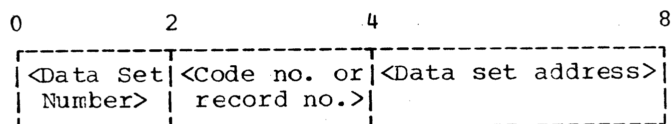
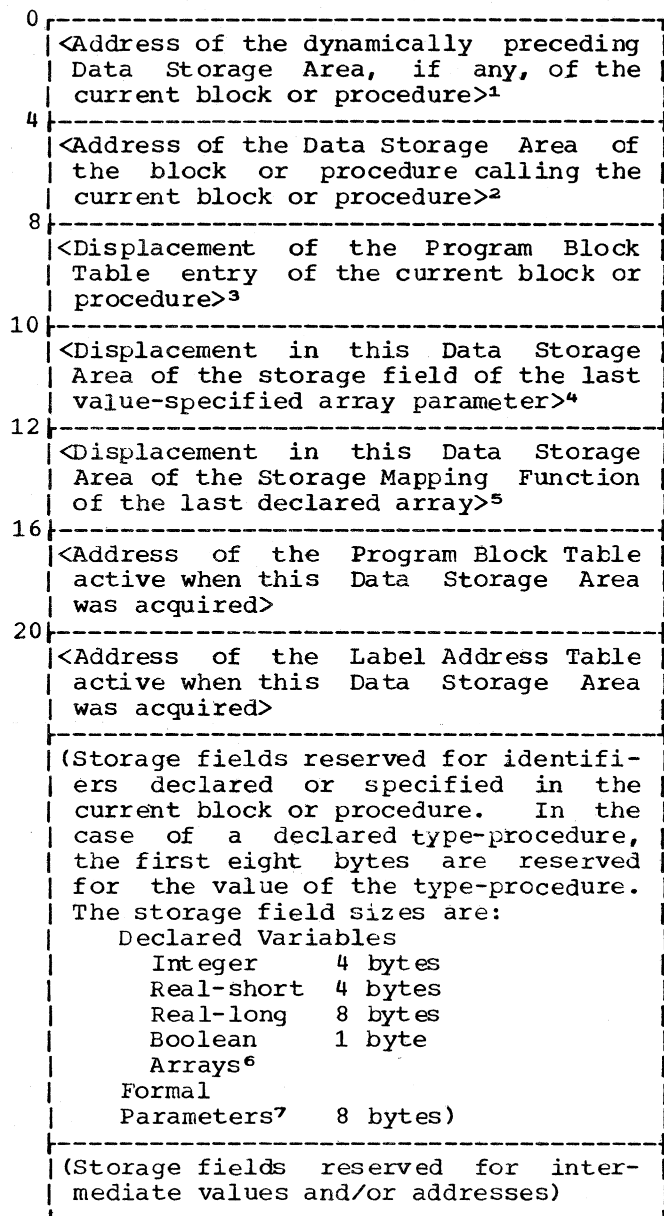


Figure 87. Entry in the object time Note Table (NOTTAB)

Data Storage Area (DSA)

The Data Storage Area comprises the total area of main storage required for all identifiers declared or specified in a block or a procedure and for intermediate results and/or addresses which must be temporarily stored. A Data Storage Area is acquired (by the Fixed Storage Area routine PROLOG) when a block is entered or a procedure is called. The Data Storage Area is released (by the Fixed Storage Area routine EPILOG) at exit from the particular block or procedure. The size of a given block's or procedure's Data Storage Area is specified in the Program Block Table. When a Data Storage Area is acquired, its address is stored in the corresponding entry of the Program Block Table. Base register CDSA (register 10) at all times addresses the Data Storage Area of the current (or immediately embracing) block or procedure.



Offset	Content
0	<Address of the dynamically preceding Data Storage Area, if any, of the current block or procedure>[1]
4	<Address of the Data Storage Area of the block or procedure calling the current block or procedure>[2]
8	<Displacement of the Program Block Table entry of the current block or procedure>[3]
10	<Displacement in this Data Storage Area of the storage field of the last value-specified array parameter>[4]
12	<Displacement in this Data Storage Area of the Storage Mapping Function of the last declared array>[5]
16	<Address of the Program Block Table active when this Data Storage Area was acquired>
20	<Address of the Label Address Table active when this Data Storage Area was acquired>
	(Storage fields reserved for identifiers declared or specified in the current block or procedure. In the case of a declared type-procedure, the first eight bytes are reserved for the value of the type-procedure. The storage field sizes are: Declared Variables Integer 4 bytes Real-short 4 bytes Real-long 8 bytes Boolean 1 byte Arrays[6] Formal Parameters[7] 8 bytes)
	(Storage fields reserved for intermediate values and/or addresses)

Figure 88. Content of the Data Storage Area

Notes on Figure 88 (the numbers refer to the superscripts in the figure):

[1] A preceding Data Storage Area for the current block or procedure will exist only where the current block or procedure is entered as a result of a recursive procedure call. In this case, the present Data Storage Area will relate either to the recursively called procedure or to a block or procedure enclosed by a recursively called procedure.
The address of the preceding Data Storage Area, if any, is obtained from the Program Block Table entry of the current block or procedure. The latter entry will immediately afterward contain the address of the current Data Storage Area.

[2] The address of the Data Storage Area of the calling block or procedure is obtained from the corresponding entry in the Program Block Table.

[3] At exit from the current block or procedure, the address in bytes 0-4 of the dynamically preceding Data Storage Area of the recursively calling block or procedure is stored in the corresponding entry of the Program Block Table. Bytes 8-9 specify the displacement of the entry.

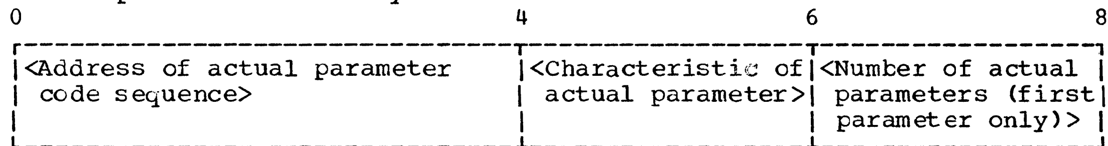
[4] During an active procedure call, the storage field of a value-specified array parameter contains the address of a temporarily constructed Storage Mapping Function (a copy of the Storage Mapping Function of a declared array) located outside any existing Data Storage Area, adjacent to the array storage area. The area occupied by the storage Mapping Function and the array must be released at exit from the current block or procedure.

[5] At exit from the current block, the storage area(s) acquired for any array(s) declared in the current block is (are) released. Bytes 12-16 specify the displacement of the Storage Mapping Function of the last declared array (if any), containing the address of the array's storage area. Where two or more arrays are declared in a block the Storage Mapping Function (Figure 62) contains the displacement in the current Data Storage Area of the Storage Mapping Function of the previously declared array.

[6] See Chapter 5 ("Storage Allocation").

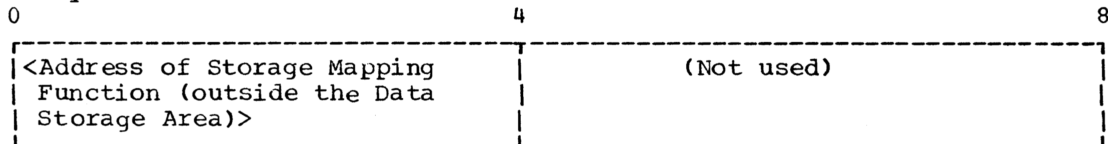
[7] The contents of the eight-byte storage field of a formal parameter called by name and called by value is indicated in Figure 89.

Formal parameter called by name.

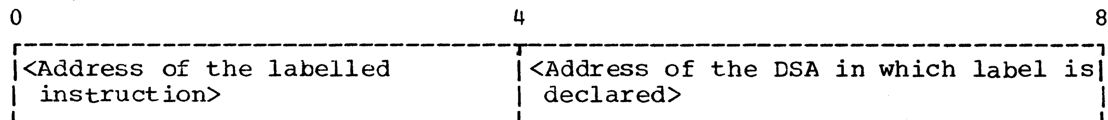


Formal parameter called by value:

Array:



Label:



Variable:

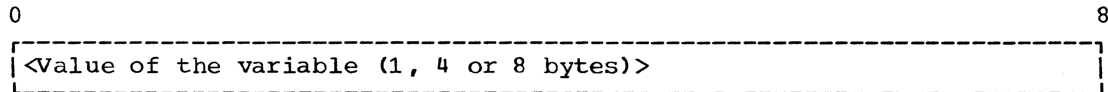


Figure 89. Content of the 8-byte storage field of a formal parameter called by name and called by value during an active procedure call

Storage Mapping Function (SMF)

The Storage Mapping Function (Figure 62), which is constructed at object time, specifies, among other things, the size of the area required for a particular array and (after the area has been acquired) the address of the array's storage area as well as the array's zero-base address. The Storage Mapping Function is described in detail in Chapter 8 under "Arrays".

In the case of a declared array, the Storage Mapping Function is constructed, at entry to the block in which the array is declared, in the storage area reserved for it at compile time in the block's Data Storage Area. Immediately after construction of the Storage Mapping Function, a storage area is acquired for the array outside the Data Storage Area, the address being stored in the appropriate entry of the Storage Mapping Function.

In the case of a value-specified array parameter of a procedure, the actual array specified in the procedure call is copied (by the Fixed Storage Area routine VALUCALL) into another storage area, together with a modified copy of the related Storage Mapping Function.

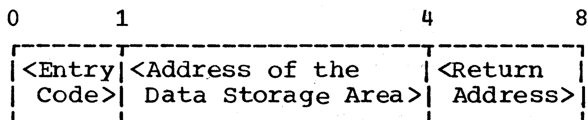
Return Address Stack (RAS)

The Return Address Stack has three main functions:

1. to provide a record of the encompassing block structure at every point in the object module. This record, which consists of the addresses of the Data Storage Areas of all embracing blocks and procedures, is used mainly in releasing the related Data Storage Area(s) when a goto statement, implying an exit from one or more embracing blocks or procedures, is executed.

2. to provide a record of the return address, following a call to a procedure, to an actual parameter code sequence, or to a switch element.

3. to provide a record of the names and entry point addresses of load modules representing precompiled procedures called by the object module. The entry point addresses are used when, at exit from the embracing block, the loaded module is deleted.



<Entry Code>:

X'00' - Signifies that the entry relates to a call for an actual parameter or a switch element.

X'FE' - Signifies that the entry relates to a declared code procedure, that the DSA address is the address of the DSA of the block in which the code procedure is declared, and that the return address is the entry point of the load module representing the precompiled procedure. Specifies to RETPROG that the load module is to be deleted at exit from the block in which the code procedure is declared. In this case, the load module name is stored in an entry at the opposite end of the Return Address Stack.

X'FF' - Signifies that the entry relates to a block entered or a procedure called, and specifies to RETPROG that the Data Storage Area addressed in bytes 0-3 is to be released, if a goto statement implies a branch out of the related block or procedure.

<Return Address>:

(Entry code X'00') - <address of next instruction following a call for an actual parameter or a switch element, or unspecified>

(Entry code X'FE') - <entry point address of the precompiled procedure load module>

(Entry code X'FF') - <address of next instruction following a procedure call, or unspecified (in the case of a block)>

Figure 90. Entry in the object time Return Address Stack (RAS)

Return Address Stack entries are constructed and released by the following Fixed Storage Area routines (see Chapter 10):

Entries constructed by	Entries released by
PROLOG	EPILOG
LOADPP	RETPROG
CAP1	CAP2
CSWE1	CSWE2

A total of 2048 bytes are provided for the Return Address Stack.

OBJECT TIME REGISTER USE

In the generated object module, certain general purpose registers are used exclusively as base registers (e.g. to address the current Data Storage Area or the Program Block Table), while the remaining general purpose registers and all floating point registers are used for variable computational and addressing use. The compile-time control and assignment of variable-use registers is described in Chapter 8 under "Control of Object Time Registers".

The function of the base registers is illustrated in Figure 83.

Register No.	Name	Use
General Purpose Registers		
0	(Variable)	Variable computational use.
1	(Variable)	Variable computational use.
2	(Variable)	Variable computational use.
3	(Variable)	Variable computational use.
4	(Variable)	Variable computational use.
5	(Variable)	Variable computational use.
6	(Variable)	Variable computational use.
7	(Variable)	Variable computational use.
8	ADR	Variable addressing use
9	GDSA	Base register - addresses the Data Storage Area of a block (other than the current block or procedure)
10	CDSA	Base register - addresses the current Data Storage Area (i.e. of the immediately encompassing block or procedure).
11	PBT	Base register - addresses the Program Block Table.
12	LAT	Base register - addresses the Label Address Table
13	FSA	Base register - addresses the Fixed Storage Area.
14	STH	Variable, short term use, e.g. in communication with Fixed Storage Area routines.
15	BRR	Branch register - used in branching within the generated code and in branching to Fixed Storage Area routines.
Floating Point Registers		
0		Variable computational use
2		Variable computational use
4		Variable computational use
6		Variable computational use

Figure 91. Object time register use

ALGOL (F) Compiler Flowcharts

The following section contains a complete set of summary flowcharts of the ALGOL Library routines.

The organization of the flowcharts reflects the logical division of the Compiler into ten individual phases, as well as the normal sequence of execution of the phases. (A deviation from the normal execution sequence occurs only if a terminating error is detected by a given phase. The possible deviations are indicated in the overall flowchart and are discussed in Chapter 9). The organization of the flowcharts is as follows.

	Chart Nos.	Pages
ALGOL Compiler - Overall Flow	001	197
Directory (IEX00)	002-006	198-200
Initialization Phase (IEX10)	007-010	201-202
Scan I/II Phase (IEX11)	011-034	203-214
Identifier Table Manipulation Phase (IEX20)	035-039	215-217
Diagnostic Output (IEX21)	040-043	218-219
Scan III Phase (IEX30)	044-070	220-233
Diagnostic Output (IEX31)	071	234
Subscript Handling Phase (IEX40)	072-076	235-237
Compilation Phase (IEX50)	077-123	238-261
Termination Phase (IEX51)	124-128	262-264
ALGOL Library	129-142	265-271

An aid in quickly locating the flowchart and/or descriptive text for any given routine is provided in the Index of Routines in Appendix XI.

With a few exceptions, the flowcharts have been condensed, so as to accomodate two flowcharts on every page. Each flowchart is identified by a serial flowchart number, which appears at the top of the chart, together with the related microfiche identification. It is important to note that connections between flowcharts (indicated by the standard off-page connector symbol) are represented in terms of the flowchart number and not the page number.

The following special conventions have been implemented in connection with the use of the processing symbol to represent complex logical program units, i.e. modules (or phases), routines and subroutines.

1. The striped symbol identifies a phase or a closed subroutine. The symbolic name of the phase or subroutine, and the related detailed flowchart (if any), are identified above the stripe. The absence of a flowchart number above the stripe indicates that no detailed flowchart exists.

2. Routines and major components of routines are represented by the standard (unstriped) symbol. The symbolic name (if any) of the routine and the detailed flowchart (if any) are shown in the upper left and right corners of the symbol, separated by a dashed line.

Chart 001: ALGOL Compiler Overall Flow

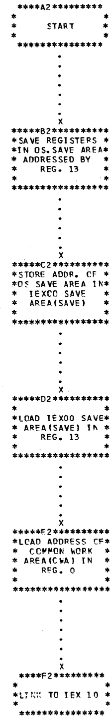
```
CALL/LINK/XCTL
EXEC/ATTACH
..........................X*****A2**********         *****A3**********
                          *IEX00       002*          *IEX10       007*
                          *----------------*LINK     *----------------*
                          *   COMPILER     *........X*                *.....
                          *   DIRECTORY    *         *INITIALIZATION  *    .
X.........................*                *         *                *    .
RETURN                    ******************         ******************    .
                                 X                          .              .
                                 .                          . XCTL         .
                                 .                          X              .
                                 .                   *****B3**********     .
                                 .                   *IEX11       011*     .
                                 .                   *----------------*    .
                                 .                   *                *...X.
                                 .                   *SCAN I/II PHASE*     .
                                 .                   ******************    .
                                 .                          . XCTL         .
                                 .                          X              .
                                 .                   *****C3**********     .
                                 .                   *IEX20       035*     .
                                 .                   *----------------*    .
                                 .                   *IDENTIFIER TABLE     .
                                 .                   * MANIPULATION   *    .
                                 .                   *     PHASE      *    .
                                 .                   ******************    .
                                 .                          . XCTL         .
                                 .                          .X.............
                                 .                          X
                                 .                   *****D3**********
                                 .                   *IEX21     040  *
                                 .                   *----------------*
                                 .                   *                *.....
                                 .                   DIAGNOSTIC OUTPUT     .
                                 .                   ******************    .
                                 .                          . XCTL         .
                                 .                          X              .
                                 .                   *****E3**********     .
                                 .                   *IEX30       044*     .
                                 .                   *----------------*    .
                                 .                   *SCAN III PHASE *     .
                                 .                   ******************    .
                                 .                          . XCTL         .
                                 .                          X              .
                                 .                   *****F3**********     .
                                 .                   *IEX31     071  *     .
                                 .                   *----------------*    .
                                 .                   *                *...X.
                                 .                   DIAGNOSTIC OUTPUT     .
                                 .                   ******************    .
                                 .                          . XCTL         .
                                 .                          X              .
                                 .                   *****G3**********     .
                                 .                   *IEX40       072*     .
                                 .                   *----------------*    .
                                 .                   *   SUBSCRIPT    *    .
                                 .                   *HANDLING PHASE  *    .
                                 .                   ******************    .
                                 .                          . XCTL         .
                                 .                          X              .
                                 .                   *****H3**********     .
                                 .                   *IEX50       077*     .
                                 .                   *----------------*    .
                                 .                   *  COMPILATION   *    .
                                 .                   *     PHASE      *    .
                                 .                   ******************    .
                                 .                          . XCTL         .
                                 .                          .X.............
                                 .                          X
                                 .                   *****J3**********
                                 .                   *IEX51     124  *
                                 .          RETURN   *----------------*
                                 ....................*  TERMINATION   *
                                                     *     PHASE      *
                                                     ******************

                                                     *****K3**********
                                                     *          129  *
                                                     *----------------*
                                                     *    LIBRARY     *
                                                     ******************
```

```
CALL/LINK/XCTL/EXEC/ATTACH       (FROM OS ON PROGRAM INTERRUPT)     (FROM OS AT END OF DATA)
FROM OS INVOKER
        .                                 .                                  .
        X                                 X                                  X
*****B1**********                *****B3**********                 *****B5**********
*IEX00000    003*                *PIROUT      004*                 *EODAD1/2/3/IN 005
*---------------*                *---------------*                 *---------------*
*COMPILER ENTRY *                *    PROGRAM    *                 *  END OF DATA  *
*    ROUTINE    *                *   INTERRUPT   *                 * EXIT ROUTINES *
*               *                *    ROUTINE    *                 *               *
*****************                *****************                 *****************
        .                                 .                                  .
        X                                 X                                  X
 ****C1*********                  ****C3*********                   ****C5*********
*               *                *  TERMINATION  *                 *EOD ROUTINE IN *
* LINK TO IEX10 *                *  ROUTINE IN   *                 * CURRENT PHASE *
*               *                * CURRENT PHASE *                 *               *
 ***************                  ***************                   ***************


 (RETURN FROM IEX51)
       *****
       *002*
       * E1*
        * *
         *
         .
         X
*****E1**********
*            003*
*---------------*
* COMPILER EXIT *
*    ROUTINE    *
*               *
*****************                                                  (FROM OS
         .                       (FROM OS ON I/O ERROR)              ON I/O ERROR
         X                                 .                         ON SYSPRINT
 ****F1*********                           X                                  .
*   RETURN TO   *                *****F3**********                            X
*    INVOKER    *                *SYNAD       005*                 *****F5**********
*               *                *---------------*                 *SYNPR     005  *
 ***************                 *   I/O ERROR   *                 *---------------*
                                 *    ROUTINE    *                 * SYSPRINT I/O  *
                                 *               *                 * ERROR ROUTINE *
                                 *****************                 *               *
                                           .                       *****************
                                           X                                  .
                                  ****G3*********                             X
                                 *  TERMINATION  *                  ****G5*********
                                 *  ROUTINE IN   *                 *  TERMINATION  *
                                 * CURRENT PHASE *                 *  ROUTINE IN   *
                                  ***************                  * CURRENT PHASE *
                                                                    ***************

                                                                   CIB(IEX11)
                                                                   PRINTITB(IEX20)
                                                                   COT21(IEX21,-31,-51)
                                                                    ****H5*********
                                                                   *               *
                                                                   *     PRINT     *
                                                                   *               *
                                                                    ***************
                                                                              .
                                                                              X
                                                                   *****J5**********
                                                                   *PRINT      006 *
                                                                   *---------------*
                                                                   * PRINT SOURCE  *
                                                                   *PROGRAM AND/OR *
                                                                   *DIGNOSTIC MESS.*
                                                                   *****************
                                                                              .
                                                                              X
                                                                    ****K5*********
                                                                   *               *
                                                                   *    RETURN     *
                                                                   *               *
                                                                    ***************
```

Chart 003: Directory - IEX00 Microfiche IEX00-1

Initial Entry and Final Exit

(FROM INVOKER)



(RETURN FROM IEX51)

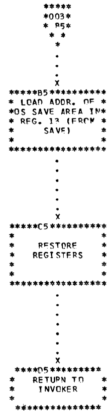


Chart 004: Directory - IEX00 Microfiche IEX00-1

Program Interrupt - PIROUT

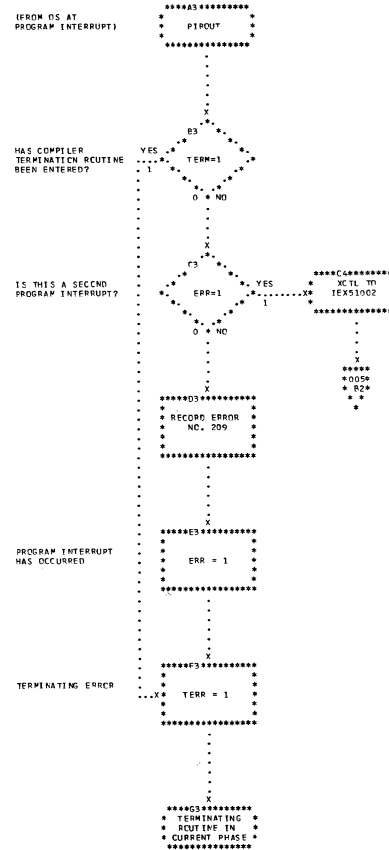


Chart 005: Directory - IEX00 — Microfiche IEX00-1

I/O Error and EOD - SYNPR, SYNAD, EODAD1/-2/-3/-IN

005 B2

FROM OS ON I/O ERROR

B2: SYNPR

C2: PRT=1 (PRINTER DOWN)

FROM OS ON I/O ERROR

B3: SYNAD

C3: CLOSE DCB AFFECTED

D3: RECORD ERROR NO. 210

E3: TERR = 1 (TERMINATING ERROR)

F3: TERMINATING ROUTINE IN CURRENT PHASE

FROM OS AT EOD

B5: EODAD 1/2/3/IN

C5: LOAD ADDRESS IN EODUT1/ EODUT2/ EODUT3/ EODIN

D5: EOD ROUTINE IN CURRENT PHASE

Chart 006: Directory - IEX00 — Microfiche IEX00-1

PRINT and PRINTP

CIB(IEX11)
PRINTITB(IEX20)
COT27(IEX21,IEX31,IEX51)

A1: PRINT

B1: SAVE REGISTERS ADRESS SYSPRINT DCB

(IS THIS A NEW PAGE) PRINTSW
C1: HEADING(S) TO BE PRINTED — YES (PRINTSW =ONSW) → C2; NO (PRINTSW =OFFSW) → D1

PRINTA
C2: MOVE TEXT FROM PRINT BUFFER TO SAVELINE → D2

PRINTH
D2: PNTSW=OFFSW (HEAD.PRINTED) LINCNTA+1=0 (NOT TITLE PAGE)

(LINCNTA+1= ='FF')
D1: IS THIS THE TITLE PAGE — YES → G1; NO → E1

PRINTF
(IS THIS THE LAST LINE OF THIS PAGE)
E1: LINE COUNT=50 — NO → H1; YES → F1

E2: ADD 1 TO PAGE COUNT AND MOVE TO PAGEHEAD

(BUFFER CONTAINS A LINE OF TEXT)
F1: PUT CONTROL CHAR FOR PAGE SHIFT IN FIRST BYTE OF PRINT BUFFER

F2: REG 9=ADDR OF PAGEHEAD. REG 10=DISP TO NEXT HEADL. REG11 =END PAGEHEAD

(PAGEHEAD ACCOMODATES UP TO THREE LINES OF TEXT)

G1: PRINTSW=ONSW (HEADING TO BE PRINTED NEXT CALL)SEE C1.

PRINTC
G2: THIS HEADLINE TO BE PRINTED — YES → H2; NO → J3

PRIPRINTD
H1: PRINTP 006 — PRINT THE LINE GR1=ADDR OF NEW BUFFER

H2: MOVE HEADLINE FROM PAGEHEAD TO PRINTBUFFER

(LINCNT=X'7F')
J1: CALL FOR NEW PAGE — YES → C2; NO → K1

J2: PRINTP 006 — PRINT THE HEAD-LINE GR1 = ADDR OF NEW BUFFER

PRINTB
J3: (ADDR. NEXT HEAD-LINE). ALL HEAD-LINES PROC — YES → J4; NO → G2

J4: CALL FOR NEW PAGE — NO → J5; YES → K1

J5: RESTORE TEXT FROM SAVELINE TO PRINTBUFFER → H1

PRINTE
K1: CLEAN OUT NEW BUFFER AND INSERT SINGLE SPACE CONTROL CHARACTER

K2: STORE LINE COUNT. GR1=ADDR OF NEW BUFFER. RESTORE REGS.

K3: RETURN

PRINT
A4: PRINTP

B4: GR1=PREVLIN (ADDR.THE CURR.PRINT BUFFER)

C4: INCREMENT LINE COUNT BY K* CONTROL CHAR IN FIRST BYTE OF TEST LINE

D4: PUT THE LINE OF TEXT ON SYSPRINT

E4: STORE ADDR.OF NEW BUFFER AT PREVLIN

F4: RETURN

Chart 007: Initialization Phase - IEX10 Microfiche IEX10-1

Overall Flow

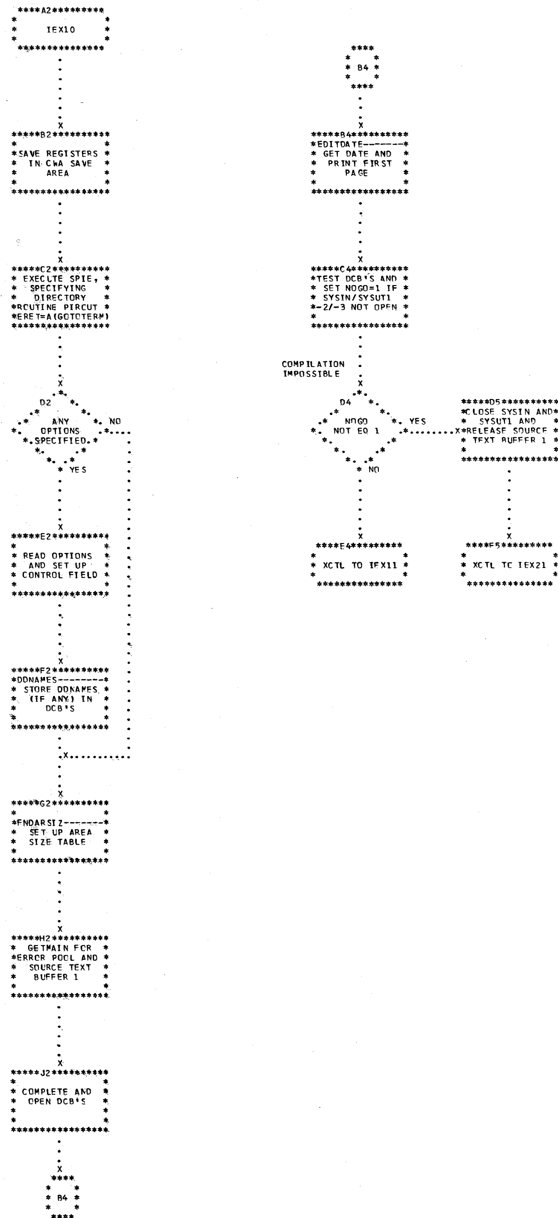


Chart 008: Initialization Phase - IEX10 Microfiche IEX10-1

Option Processing

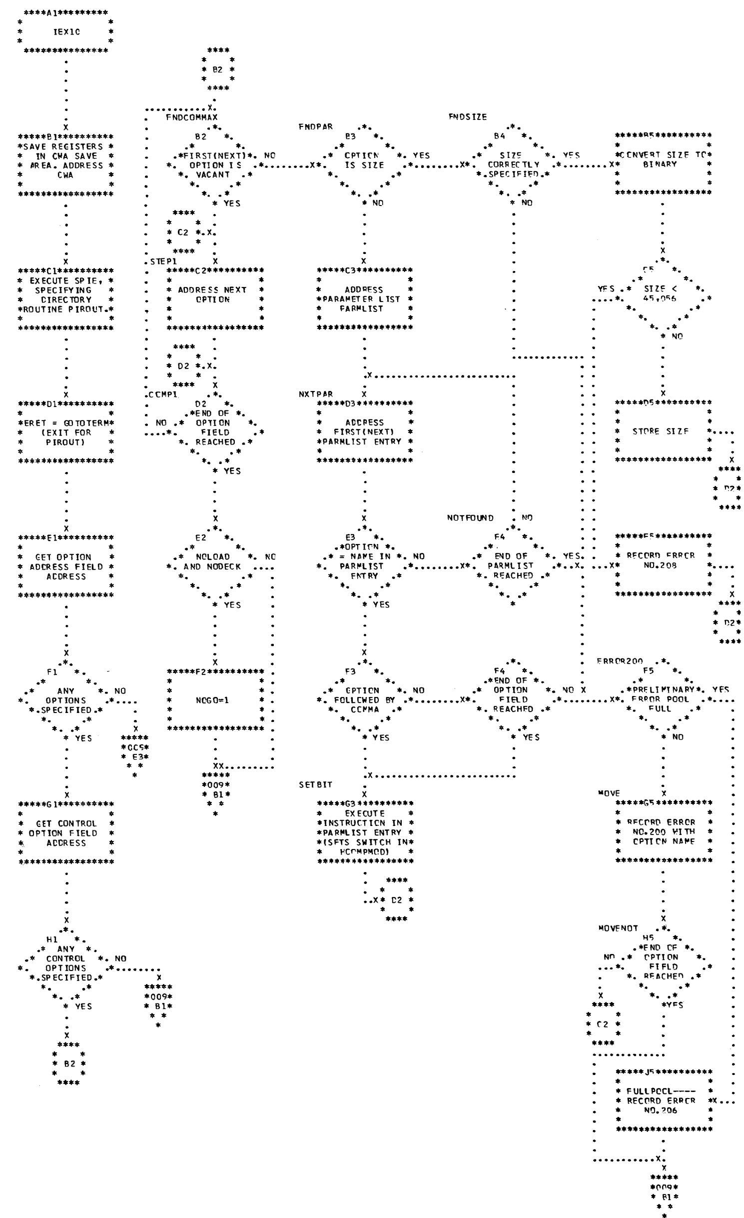


Chart 009: Initialization Phase - IEX10 — Microfiche IEX10-1

DDname Processing and Opening Data Sets

DDNAMES 009 B1

B1: ANY DD NAMES OR PAGE NO. SPECIFIED — NO: B3; YES: C1

C1: GET DD NAME FIELD ADDRESS

D1: ANY DD NAMES SPECIFIED — NO: B3; YES: E1

E1: MORE THAN 10 DD NAMES — YES: E2; NO: F1

E2: RECORD ERROR NO.207

F1: GET FIRST ADDRESS IN DCB ADDRESS FIELD

G1: DCB ADDRESS = 0 — YES: NXTDDNAM K1; NO: H1

H1: NEXT DD OPTION VACANT — YES: NXTDDNAM K1; NO: J1

J1: MOVE DD NAME TO DCB

NXTDDNAM K1: GET NEXT DD OPTION AND DCB ADDRESS

K2: END OF DD OPTION FIELD — YES: B3; NO: G1

B3 — HEADINFO B3: GET HEADING OPTION FIELD ADDRESS

C3: ANY PAGE NO. SPECIFIED — YES: D3; NO: 009 E3

D3: MOVE PAGE NO. TO PAGECNT

009 E3 — OPEN E3: STORE OPEN EXIT ROUTINE ADDRESS IN SYSPRINT DCB

FNDARSIZ F3: GET NEXT AREA SIZE TABLE. GET SIZE

G3: SIZE > = SIZE IN AREA SIZE TABLE — YES: F3; NO: H3

H3: MOVE AREA SIZE TABLE TO CWA

J3: GETMAIN FOR ERROR POOL AND SOURCE TEXT BUFFER 1 — B5

E4: ERET = OPEXERR (PROG. INTERR. EXIT FOR PIROUT)

G4: PRTNO = 1

H4: RECORD ERROR NO.201, NOGO =1 — 010 H1

B5 — MOVEPOOL B5: MOVE ERRORS FROM PRELIMINARY ERROR POOL TO MAIN ERROR POOL

C5: STORE OPEN EXIT ROUTINE ADDRESSES IN SYSLIN,SYSIN SYSPUNCH DCB'S

D5: STORE SOURCE TEXT BUFFER LENGTH IN SYSUT1 AND SYSUT2 DCB'S

E5: OPEN SYSLIN, SYSPRINT, SYSPUNCH, SYSUT2 AND SYSUT3

F5: OPEN SYSUT1 AND SYSIN (DCB'S IN CWA)

G5: SYSPRINT OPENED — NO: G4; YES: H5

EDITDATEX H5: GET DATE AND STORE IN TITLE FOR PRINTED OUTPUT

J5: SKIP TO NEW PAGE AND PRINT TITLE ON SYSPRINT — 010 B1

Chart 010: Initialization Phase - IEX10 — Microfiche IEX10-1

Opening Data Sets

010 B1

B1: ERET = GOTOEDIT (PROG. INTERR. EXIT FOR PIROUT)

C1: LOAD SPECIFIED — NO: E1; YES: D1

D1: SYSLIN OPENED — NO: D2; YES: E1

D2: NLOAD =1. RECORD ERROR NO.202

TSTPUNCH E1: DECK SPECIFIED — NO: G1; YES: F1

F1: SYSPUNCH OPENED — NO: F2; YES: G1

F2: NDCK =1.RECORD ERROR NO.203

TSTIN G1: SYSIN, SYSUT1-2-3 OPENED — NO: G2; YES: 010 H1

G2: RECORD ERROR NO.201. NOGO =1

COMPILATION IMPOSSIBLEH1 — H1: NOGO NOT EQ 1 — NO: H2; YES: J1

GOTOEDIT H2: CLOSE — CLOSE SYSIN AND SYSUT1. RELEASE BUFFER 1

H3: TERR = 1

J1: BRANCH TO IEX11

J3: XCTL TO IEX21

GOTOTERM (FROM DIRECTORY ROUTINE PIROUT IN PROGRAM INTERRUPT) 010 B4

B4: CLOSE — CLOSE SYSIN AND SYSUT1 REALEASE BUFFER

C4: XCTL TO IEX 51002

Chart 011: Scan I/II Phase - IEX11 Overall Flow

Microfiches IEX11-1, IEX11-2 and IEX11-3

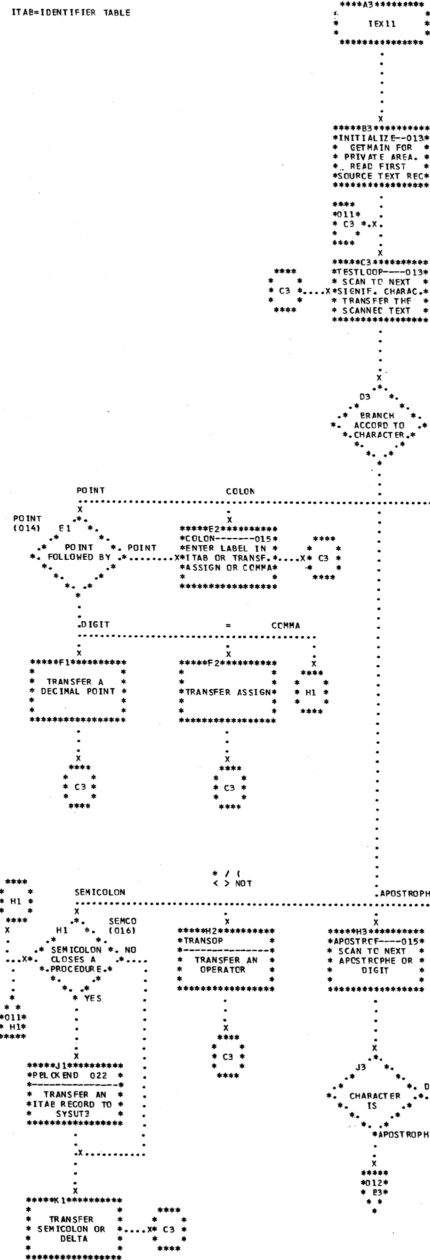


Chart 012: Scan I/II Phase - IEX11 Overall Flow

Microfiches IEX11-1, IEX11-2 and IEX11-3

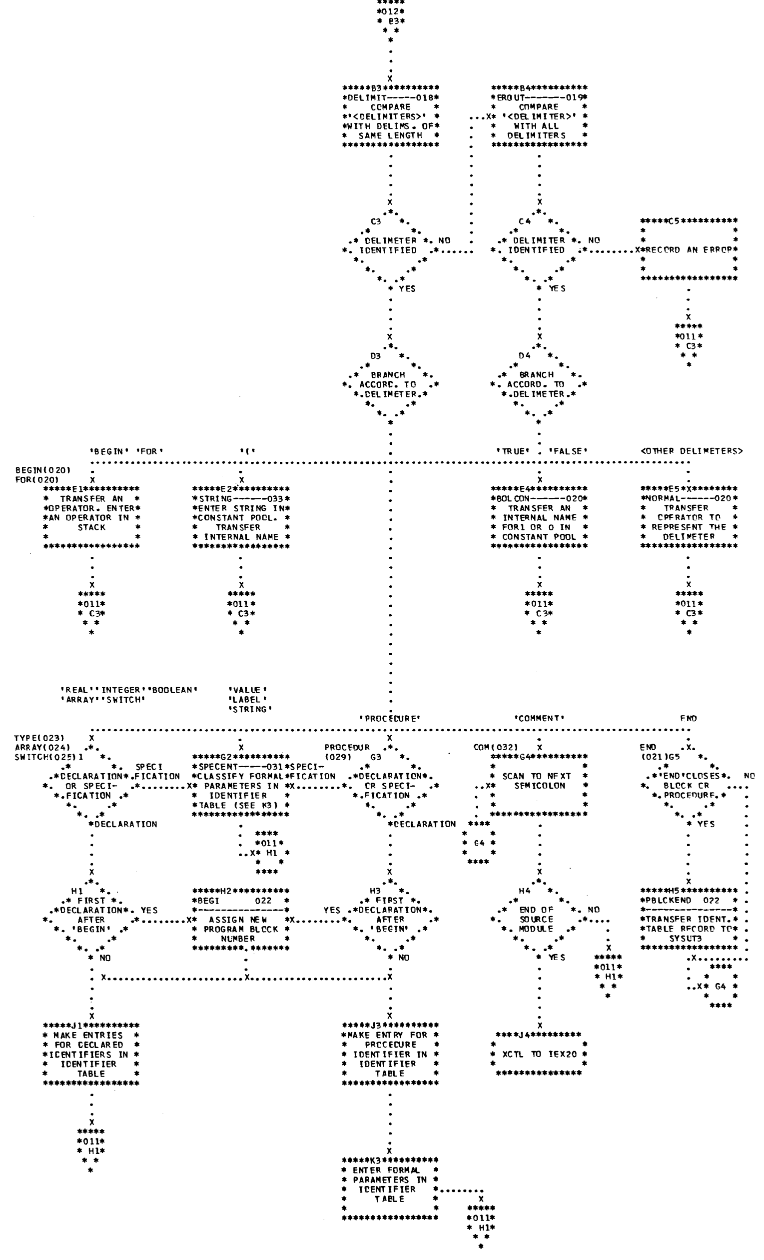


Chart 013: Scan I/II Phase - IEX11 Microfiche IEX11-1

Initialization and TESTLOOP

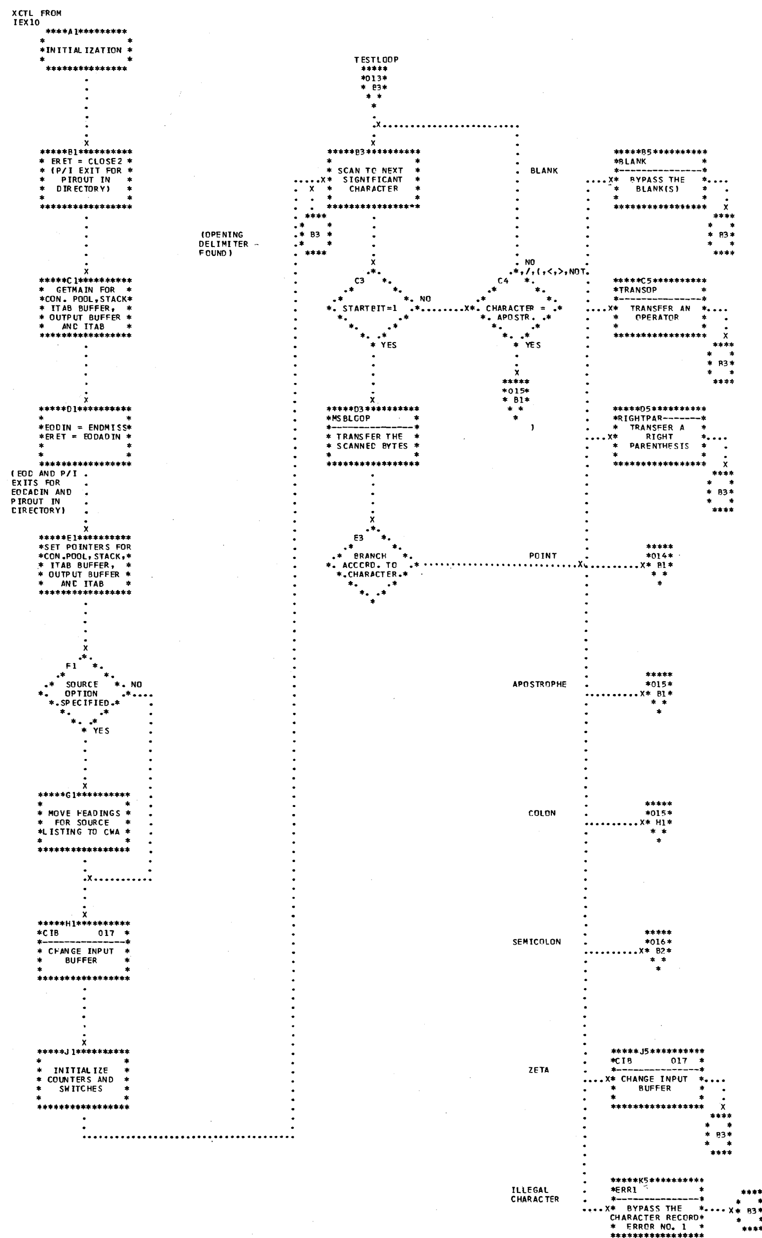


Chart 014: Scan I/II Phase - IEX11 Microfiche IEX11-1

POINT and STATE

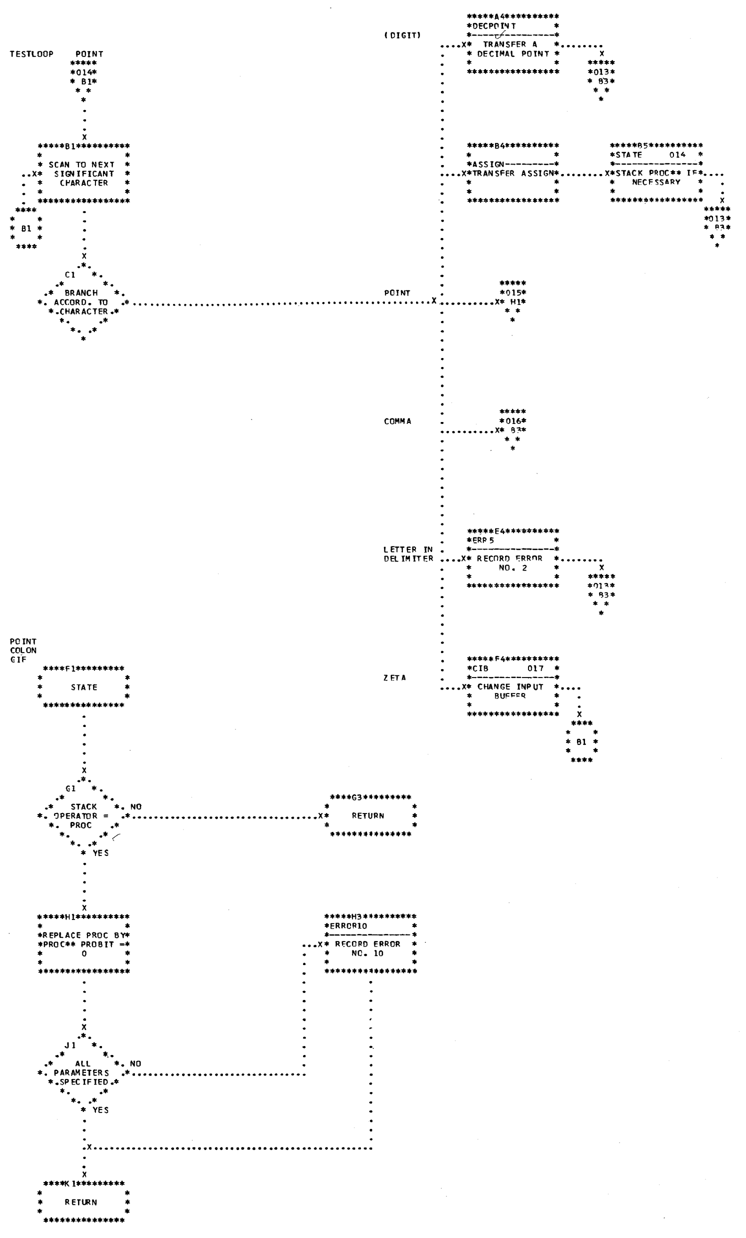


Chart 015: Scan I/II Phase - IEX11 — APOSTROF and COLON — Microfiche IEX11-1

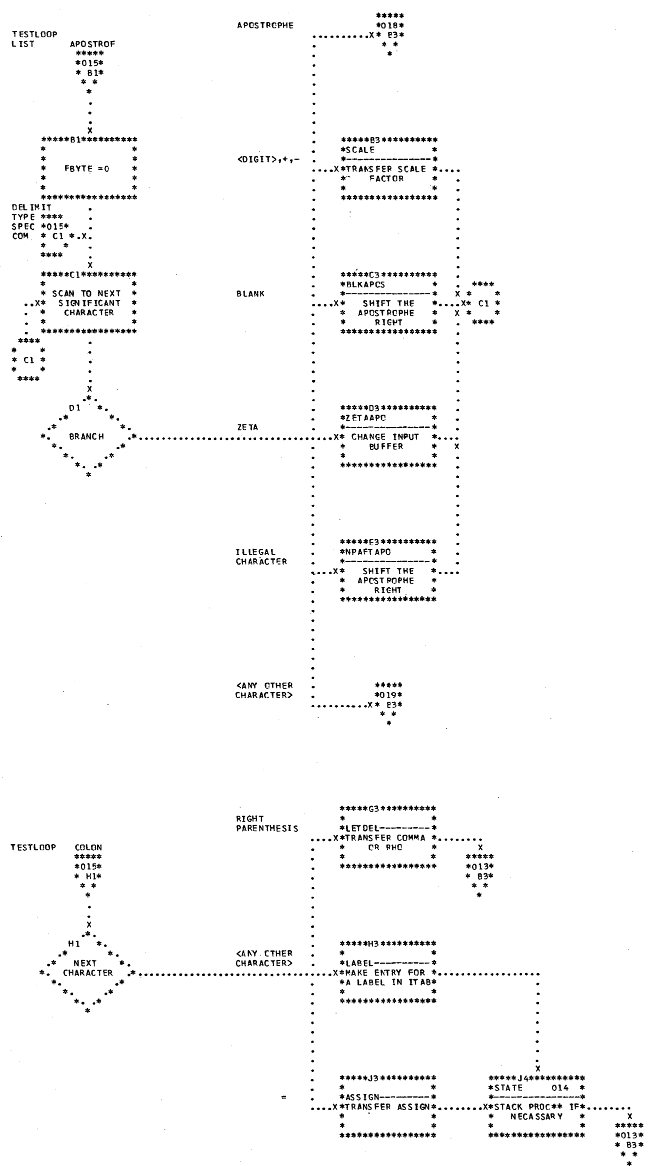


Chart 016: Scan I/II Phase - IEX11 — SEMCO — Microfiche IEX11-1

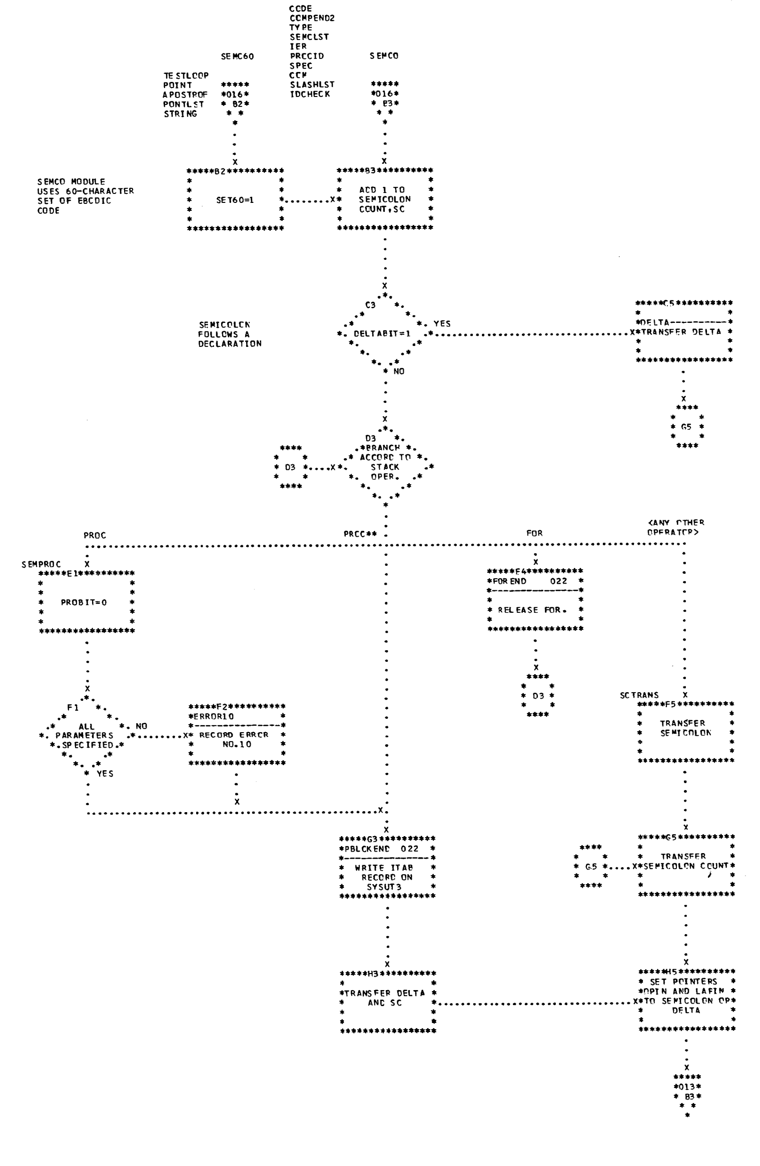


Chart 017: Scan I/II Phase - IEX11 Microfiche IEX11-1

CIB,COB and COBSPEC

ALL ROUTINES WHICH SCAN SOURCE TEXT

A1: CIB

B1: MOVE LAST 7 CHARACTERS TO FRONT OF WORK AREA

EAP(REG.3)=BYTE POINTER IN OUTPUT BUFFER
APE POINTS TO LAST BYTE IN OUTPUT BUFFER
REG0(REG.0) CONTAINS THE CURRENT ADDRESS IN EAP PLUS A SPECIFIC NUMBER OF BYTES SPECIFIED BY THE CALLING ROUTINE

C1: SOURCE SPECIFIED? — NO / YES

D1: PRINT CC6 — PRINT PREVIOUS RECORD IN PRINT AREA

E1: GETREC — GET NEW RECORD

F1: ISO SPECIFIED? — YES / NO

F2: ISOTRANS — TRANSLATE (,), =, 8, AND APOS. TO EBCDIC CODE

G1: SOURCE SPECIFIED? — NO / YES

G2: MOVE NEW RECORD TO DUMMY PRINT AREA(SAVEPRINT)

H1: MOVE SEMICOLON COUNT (CONVERTED) AND NEW RECORD TO PRINT AREA

J1: TRANSLATE NEW RECORD IN WORK AREA. INSERT ZETA

K1: RETURN

ALL ROUTINES WHICH TRANSFER CHARACTERS TO OUTPUT BUFFER

A3: COBSPEC

A4: COB

B3: REG0 'GRTR OR EQ' APE? — YES / NO

B4: EAP=APE? — NO / YES

B5: RETURN

C3: RETURN

C4: TRANSFER ZETA

D4: CHECK LAST WRITE OPERATION

E4: WRITE THIS RECORD ON SYSUT1

F4: SET POINTERS EAP AND APE TO ALTERNATE BUFFER

G4: INCREMENT OUTPUT RECORD COUNT(ONC)

H4: RETURN

Chart 018: Scan I/II Phase - IEX11 Microfiche IEX11-1

DELIMIT, TYPESPEC and COMSPEC

APOSTROF DELIMIT — 018 B3

B3: SCANNED CHARACTERS = 0? — YES / NO

B4: ERR 6 — RECORD ERROR NO. 6 → 015 C1

C3: SCANNED CHARACTERS > 10? — YES → 019 B3 / NO

(SEE CHAPTER 4-SWITCHES)

TYPESPEC

D2: DELIMITER = 'PROCEDURE'? — YES → 029 B1 / NO

D3: FBYTE? — F0 → 018 D2 / FF → 018 D4 / 00

COMSPEC

D4: DELIMITER = 'ELSE'? — YES → 020 B4 / NO

E2: DELIMITER = 'ARRAY'? — YES → 024 B1 / NO

E4: DELIMITER = 'END'? — YES → 021 B1 / NO → 032 C2

F2: IERSPEC 028 — DISREGARD THE DECL. → 031 C3

F3: INITIALIZE POINTERS TO DELIMITER TABLE

G1: ADDRESS NEXT DELIMITER TABLE ENTRY

G2: ALL DELIMITER ENTRIES COMPARED? — NO / YES → 019 B3

G3: DELIMITER ENTRY FOUND? — NO / YES

OPENING DELIMITER FOUND

G4: STARTBIT=1 — NO → 019 B4 / YES

H3: BRANCH ACCORD. TO DELIMITER

'/' 'OR' 'AND' 'NOT' 'STEP' 'LESS' 'IMPL' 'UNTIL' 'WHILE' 'POWER' 'EQUAL' 'EQUIV' 'NOTLESS' 'GREATER' 'NOTEQUAL' 'NOTGREATER' → 020 B1

'DO' 'ELSE' 'THEN' → 020 B4

'IF' 'GOTO' → 020 B3

'TRUE' 'FALSE' → 020 B2

'BEGIN' → 020 F1

'FOR' → 020 F3

'CODE' → 020 F5

'END' → 021 B1

'REAL' 'INTEGER' 'BOOLEAN' → 023 B3

'ARRAY' → 024 B3

'SWITCH' → 025 B3

'PROCEDURE' → 029 B3

'LABEL' 'STRING' → 031 B1

'VALUE' → 031 B5

'COMMENT' → 032 B3

'(' → 033 B3

Chart 019: Scan I/II Phase - IEX11 Microfiche IEX11-1

EROUT and STARTDEL

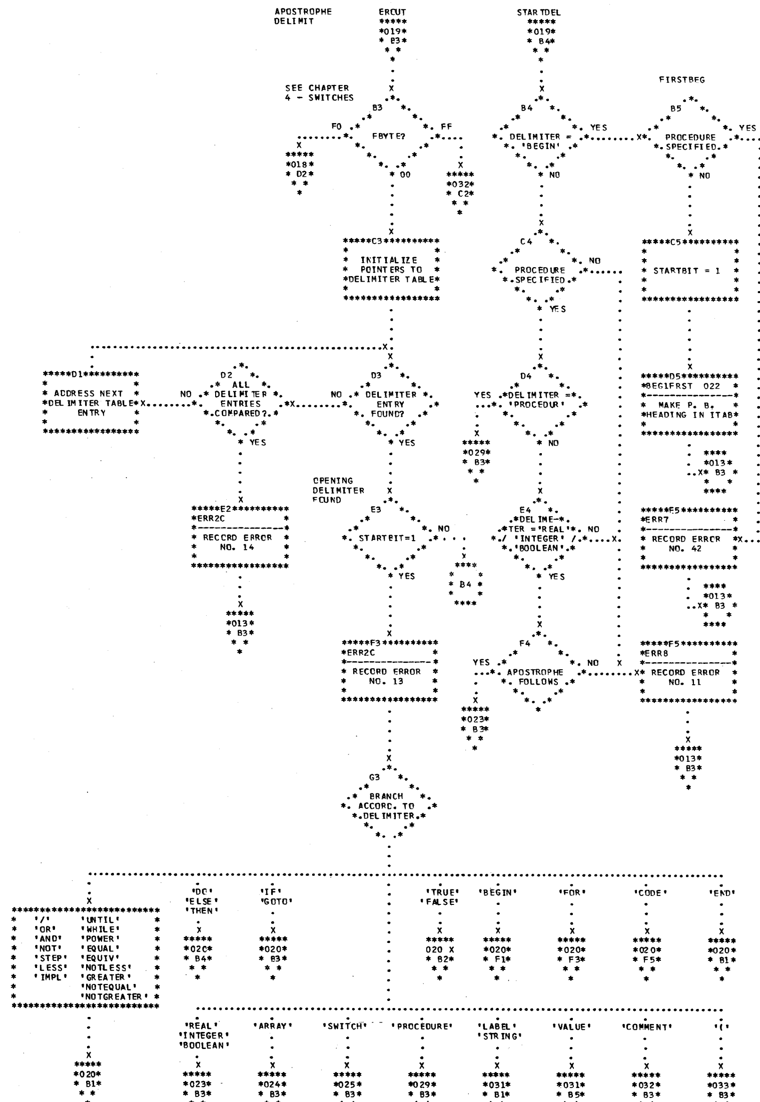


Chart 020: Scan I/II Phase - IEX11 Microfiche IEX11-1

NORMAL, BOLCON, GIF, TED BEGIN, FOR and CODE (CODE and FOR: IEX11-2)

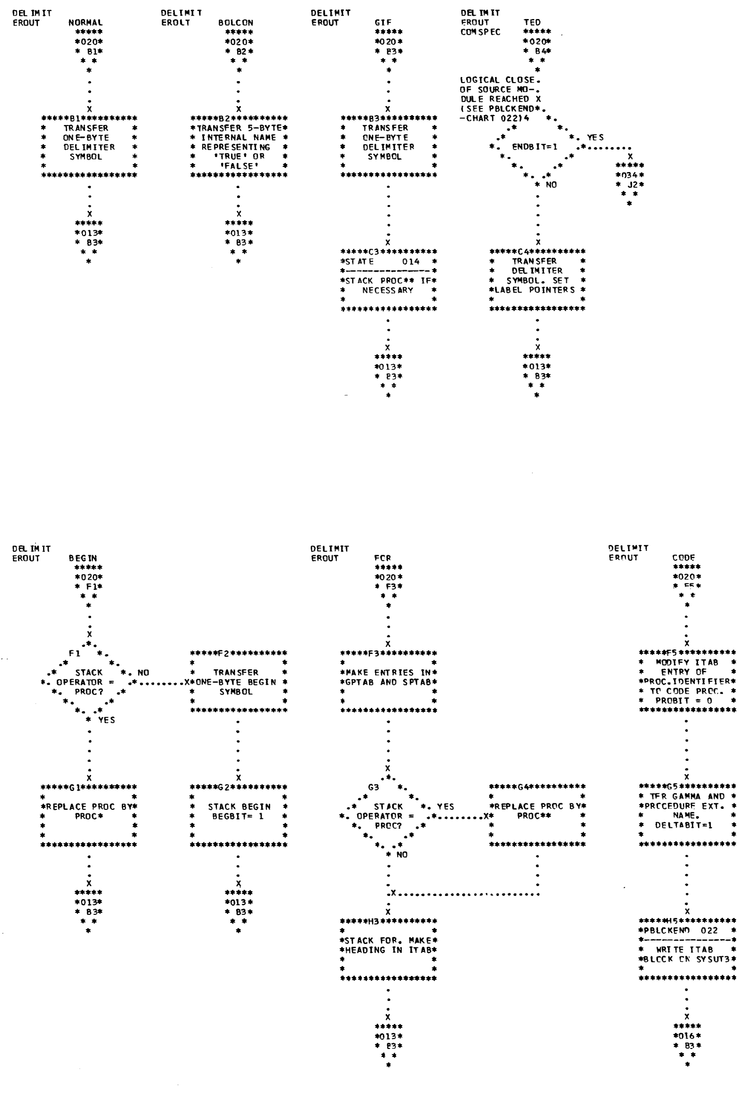


Chart 021: Scan I/II Phase - IEX11 Microfiche IEX11-2
END and COMPEND2

DELIMIT COMSPEC — END *021* B1

LOGICAL CLOSE OF SOURCE MODULE REACHED

B1: ENDBIT=1 — 1: *034* J2; 0: continue; *B1*

C1: STACK OPERATOR? — ALPHA: C2 *ERR8* RECORD ERROR NO. 11 OR 18 → *013* B3

D2 *FOREND 022* — FOR: RELEASE FOR. TRANSFER ETA. → *B1*

E2 *PBLCKEND 022* — PROC**: WRITE ITAB BLOCK ON SYSUT3. RELEASE PROC** → *B1*

F2 *PBLCKEND 022* — PROC.: WRITE ITAB BLOCK ON SYSUT3. RELEASE PROC → *B1*

G2 *PBLCKEND 022* — PROC*: WRITE ITAB BLOCK ON SYSUT3. RELEASE PROC* → *032* B4

H2 — BEGIN.: *COMPDEND* RELEASE BEGIN. TRANSFER END → J3

J2 *PBLCKEND 022* — BETA.: WRITE ITAB BLOCK ON SYSUT3. RELEASE BETA → J3

J3: STACK OPERATOR? — PROC*, FOR: J4 ENDELSEBIT = 1 → *032* B4; <ANY OTHER OPERATOR> → *032* B

COMMEND — COMPEND2 *021* B4

B4: ENDELSEBIT = 0 — YES

C4: STACK OPERATOR = FOR? — NO: B5 DELTABIT = 1 → C5 *PBLCKEND 022* WRITE ITAB BLOCK ON SYSUT3 → *016* B3; YES: *C4*

D4 *FOREND 022*: RELEASE FOR TRANSFER ETA

E4: STACK OPERATOR? — FOR: *C4*; PROC**: F4 DELTABIT = 1; <ANY OTHER OPERATOR>: *016* B3

G4 *PBLCKEND 022*: WRITE ITAB BLOCK ON SYSUT3 → *016* B3

ALPHA → *022* G3

Chart 022: Scan I/II Phase - IEX11 Microfiche IEX11-2
BEG1, PBLCKEND and FOREND

TYPE, TYPEARRAY, ARRAY, SWITCH → A1: BEG1

(BEG1FRST) B1: BEGBIT = 0. REPLACE BEGIN BY BETA IN STACK

C1: TRANSFER BETA AND NEXT HIGHER P.B.NO. AND I.G.NO.

D1: MAKE PROGRAM BLOCK HEADING IN ITAB

E1: RETURN

SEMCO, CODE, END, COMPEND2 → A2: PBLCKEND

B2: RESET POINTERS TO PRECEDING P.B. AND F.S. HEADINGS

C2: WRITE CLOSED ITAB BLOCK ON SYSUT3

CALL FROM EODADIN

D2: PBOBIT=1 — YES: *034* D4; NO: continue

E2: TFR EPSILON AND ENCLOSING P.B. AND I.G. NOS. RELEASE STACK OPERATOR

CALL FROM ENDMISS

F2: TERBIT=1 — YES: *034* C1; NO: continue

G2: STACK OPERATOR = ALPHA — YES: G3 CHANGE EOD ADDRESS TO EODADIN. ENDBIT = 1 → *032* P4; *022* G3; NO: continue

H2: RETURN

SEMCO, END, COMPEND2 → A4: FOREND

B4: PRECEDING ITAB ENTRY?

A FOR STATEMENT HEADING ENTRY → C3: DELETE THE HEADING ENTRY

AN IDENTIFIER ENTRY → C4: MAKE A CONTINUATION ENTRY (I.G. NO. FROM CURRENT HEADING)

A FOR STATEMENT CONTINUATION ENTRY → C5: OVERLAY THE I.G.NO. IN THE ENTRY WITH THE CURRENT I.G. NO.

D4: TRANSFER ETA AND ENCLOSING I.G.NO. RELEASE FOR.

E4: RETURN

Chart 023: Scan I/II Phase - IEX11 Microfiche IEX11-2

TYPE

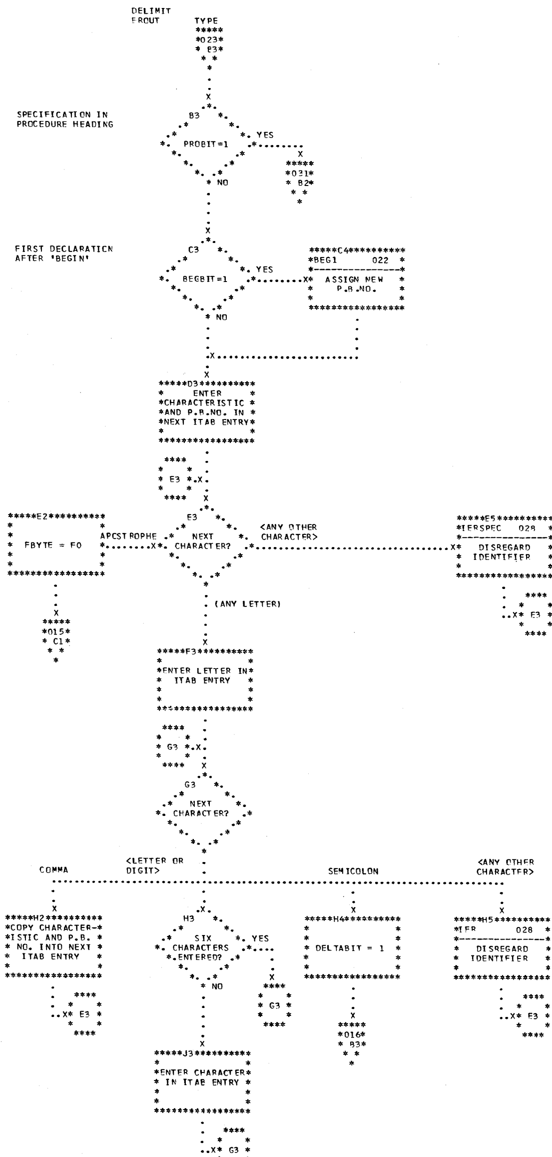


Chart 024: Scan I/II Phase - IEX11 Microfiche IEX11-2

TYPEARRY and ARRAY

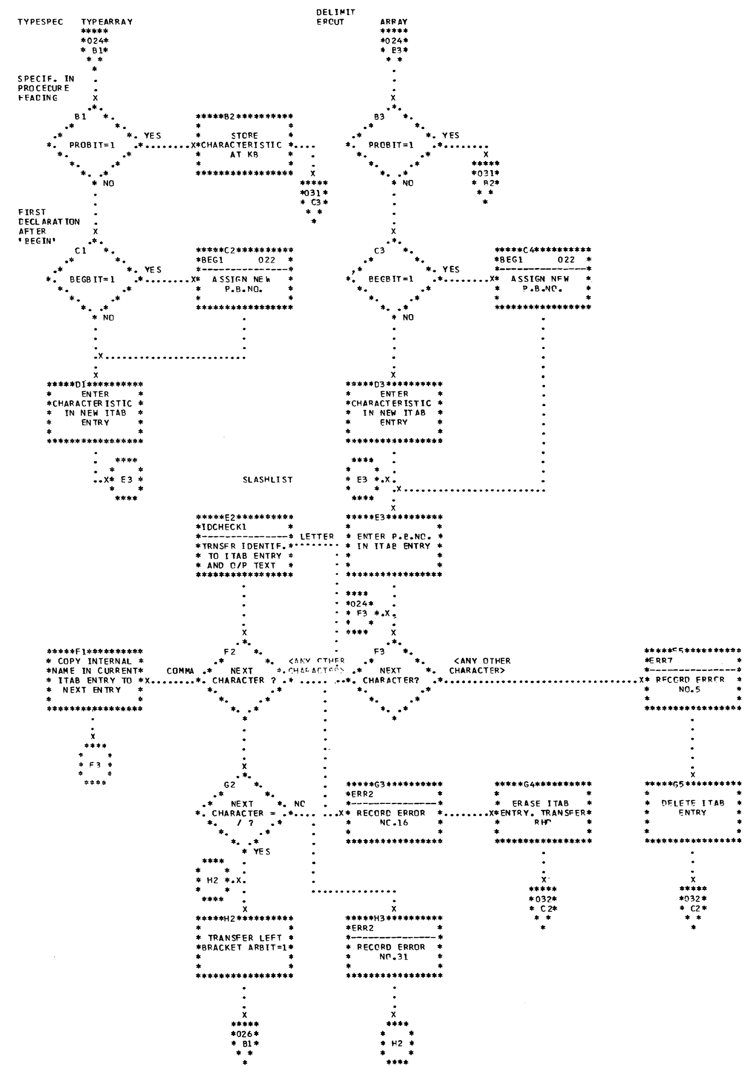


Chart 025: Scan I/II Phase - IEX11 Microfiche IEX11-2

SWITCH

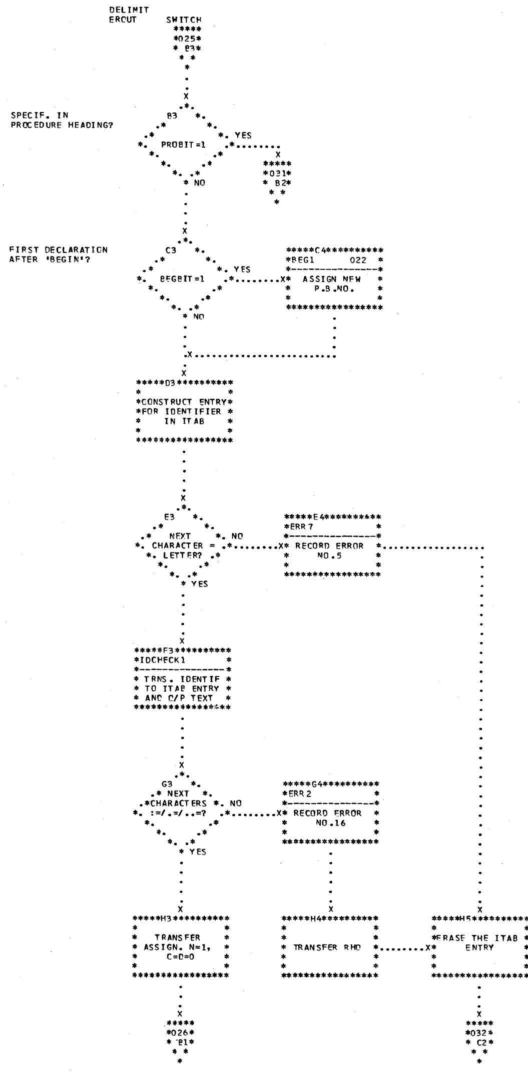


Chart 026: Scan I/II Phase - IEX11 Microfiche IEX11-2

LIST

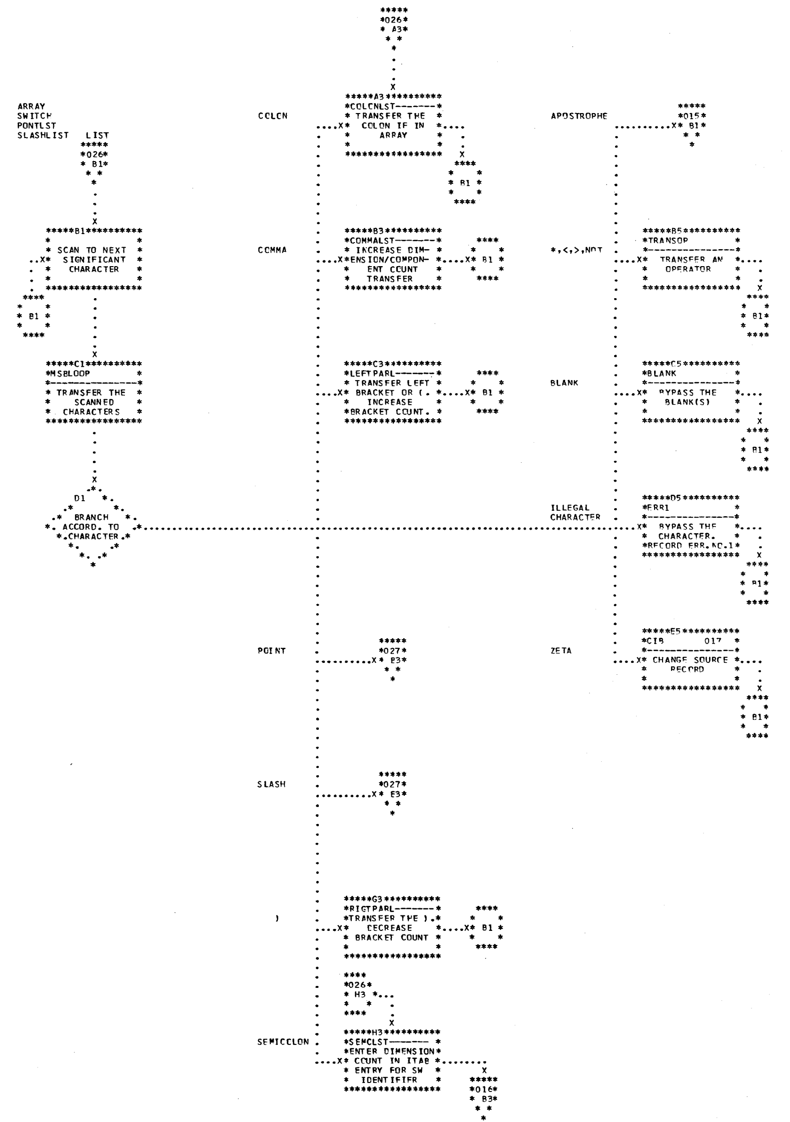


Chart 027: Scan I/II Phase - IEX11 — Microfiche IEX11-2

POINTLST

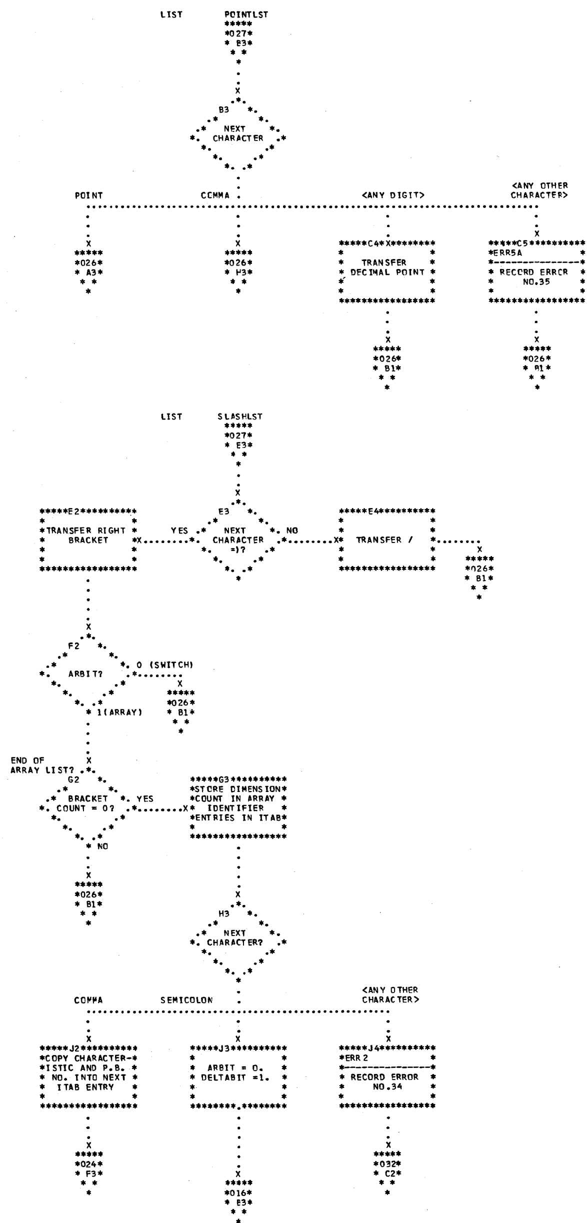


Chart 028: Scan I/II Phase - IEX11 — Microfiche IEX11-2

IERSPEC and IER

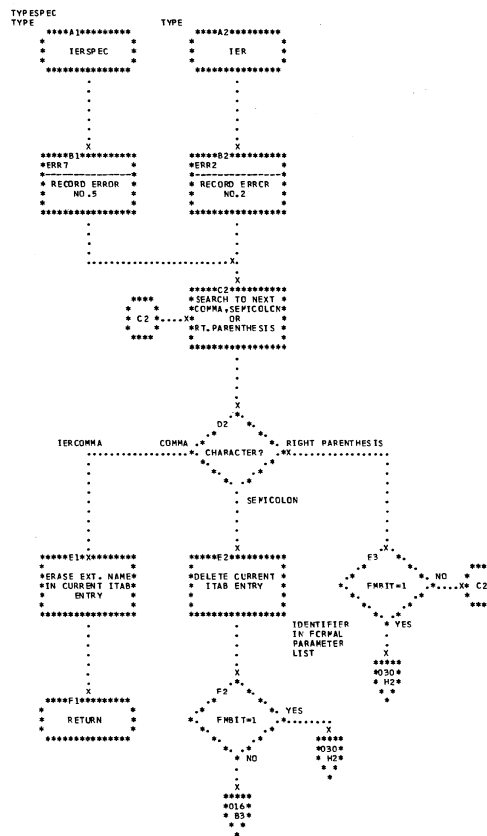


Chart 029: Scan I/II Phase - IEX11   Microfiche IEX11-2

TYPPROC and PROCEDUR

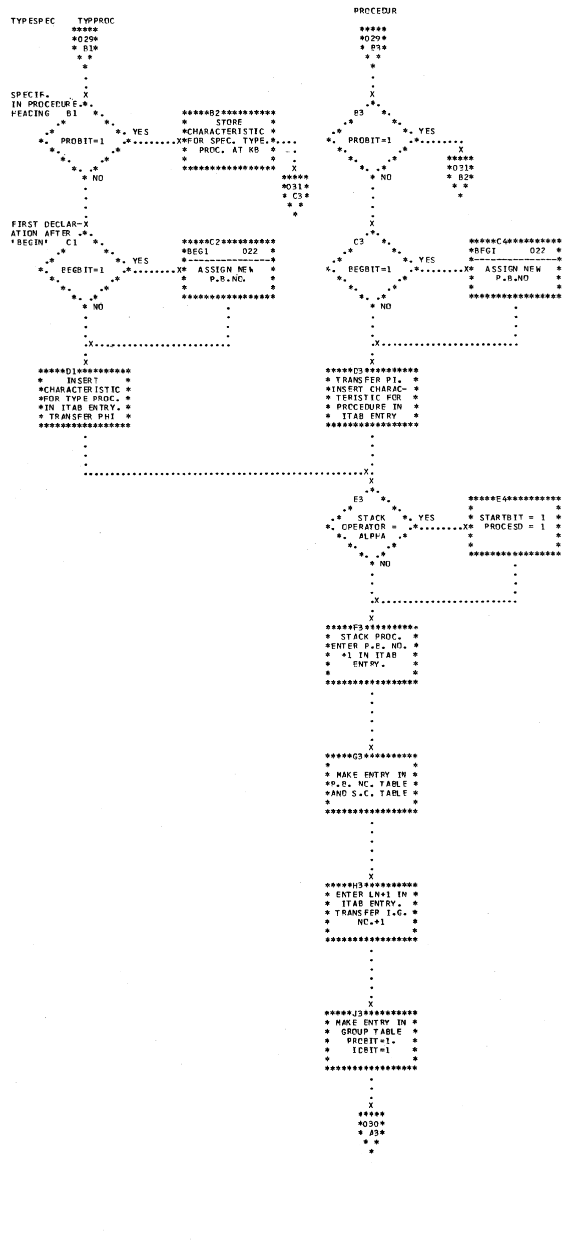


Chart 030: Scan I/II Phase - IEX11   Microfiche IEX11-2

PROCID

030 A3
A1 IDCHECK1 — TRANSFER IDENT. TO ITAB ENTRY AND O/P TEXT
A3 NEXT CHARACTER? — ANY LETTER; <ANY OTHER CHARACTER>
A3
B1 IDBIT=1 — YES (PROCEDURE IDENTIFIER); NO (FORMAL PARAMETER)
B4 IERSPEC 028 — DISREGARD IDENTIFIER
A3
NOLETTERX
B5 IDBIT=1 — 0 (FORMAL PARAMETER); YES (PROCEDURE IDENTIFIER)
C3 NEXT CHARACTER? — SEMICOLON; (; <ANY OTHER CHARACTER>
C5 ERR7 — RECORD ERROR NO.5
D2 DELTABIT =1
D2
D3 PROCESD=1? (PRECOMPILED PROCEDURE) — NO; YES
D3
D4 TRANSFER RHO
D5 SEARCH FOR SEMICOLON OR LEFT PARENTHESIS
E3 MOVE PROCEDURE NAME(EXTERNAL CODE) TO ESDPARAM
E4 ERR 2 — RECORD ERROR NO.16
E5 CHARACTER? — (; SEMICOLON
D2, D3
F3 DELTABIT=1 — YES; NO
F5 TYPE PROCEDURE? — NO; YES
EDA1
G2 NEXT CHARACTER? — COMMA;); ANY OTHER CHARACTER
G3 TRANSFER (
G4 MAKE HEADING ENTRY IN ITAB
G5 MAKE SECOND ENTRY FOR PROCEDURE IDENTIFIER
C30 H2
PROCFIN
H1 TRANSFER COMMA. INCREASE PARAMETER COUNT
A3
H2 NEXT CHARACTER — SEMICOLON; <LETTER>; <ANY OTHER CHARACTER>
H3
H3 TRANSFER RHO
H4 IER 028 — DISREGARD IDENTIFIER
H2
H5 FMBIT = 1
SCYES3
J1 ENTER PARAM. CT IN PROC. ID. ENTRY. FMBIT=0
PROCDEL
J2 SEARCH TO NEXT NONLETTER CHARACTER
J3 LETTERS END WITH :(— YES; NO
J4 REPLACE LETTER DELIMITER)LETTERS:(BY COMMA
A3
J5 DELTABIT =1 — YES; NO
016 B3
K1 TYPE PROCEDURE? — YES; NO
K2 COPY PARAM. COUNT INTO SECOND ITAB ENTRY (SEE H5)
016 B3
K4 ERR 28 — RECORD ERROR NO.37
032 C2
K5 IDBIT=0
A3

Chart 031: Scan I/II Phase - IEX11 Microfiche IEX11-2

SPEC, SPECENT, IDCHECK and VALUE

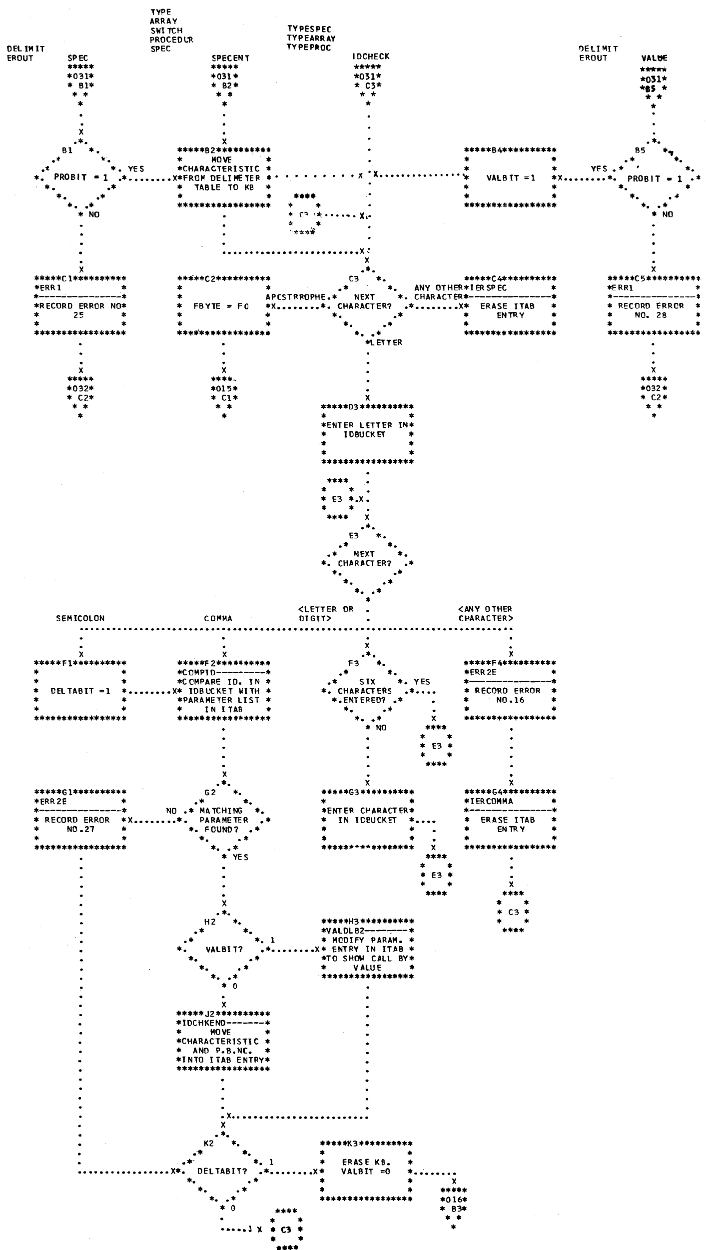


Chart 032: Scan I/II Phase - IEX11 Microfiche IEX11-2

COM, COMERR, COMMEND

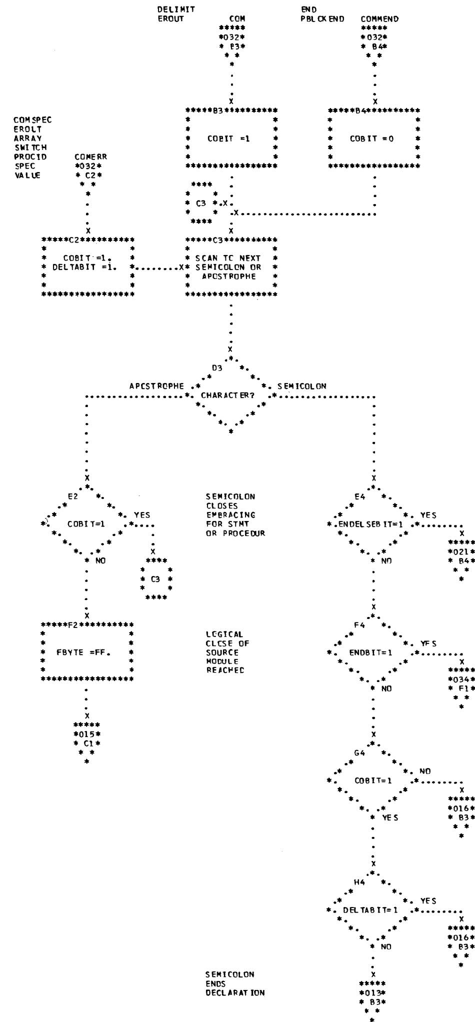


Chart 033: Scan I/II Phase - IEX11 — Microfiche IEX11-2

STRING

DELIMIT EROUT → STRING 033 B3

B3: TRANSFER AN APOSTROPHE AND 5-BYTE INTERNAL NAME FOR STRING (CHARACTERISTIC-X'C900' CONSTANT POOL-X'00' DISPLACEMENT-CONSTANT POOL POINTER-SPCLT) BREAK

C3: SCAN TO NEXT APOSTROPHE

D3: MOVE SCANNED CHARACTERS FROM PRINT AREA TO CONSTANT POOL

E3: NEXT CHARACTER? — (→ E2; <ANY OTHER CHARACTER> → E4;) → F3

E2: STORE (AT REPL. (EXTERNAL CODE)

E4: REGN =1 → K3

(REPL: 0 1 2 3)

F3: STORE) AT REPL+1 (EXTERNAL CODE)

G3: NEXT CHARACTER = APOS.? — NO → G4; YES → H3

G4: REGN =2 → K3

H3: CHARACTER AT REPL+1 =? — (→ H2;) → J3

H2: INCREASE SQC. REGN =3 (SQC=STRING QUOTE COUNT)

J3: DECREASE SQC. REGN =3

J4: SQC = 0? — YES → J5; NO → K3

J5: STORE STRING LENGTH IN CONSTANT POOL ENTRY → 013 B3

K3: MOVE REPL (LENGTH=REGN) TO CONSTANT POOL ENTRY → C3

Chart 034: Scan I/II Phase - IEX11 — Microfiche IEX11-2

ENDMISS, READROUT, COMPFIN EODADIN and CLOSE2 — (COMPFIN: IEX11-1)

OS AT EOD → ENDMISS 034 B1

B1: TERBIT =1 — FOR → A2; BETA.PROC. PR → B2

A2: FOREND 022 — RELEASE FOR → C1

B2: PBLCKEND 022 — RELEASE STACK OPERATOR → C1

C1 (PBLCKEND): STACK OPERATOR? — BEGIN → C2; ERROR35 ALPHA → D2

C2: TRANSFER END. RELEASE BEGIN → C1

D2: ERROR1 — RECORD ERROR NO.39 → B4

COMMEND → READROUT 034 F1

F1: SCAN TEXT FOR ANY NON-BLANK CHARACTER

G1: CHARACTER = ZETA? — YES → G2; NO → J2

G2: CIB — READ NEXT RECORD → F1 (AT END, VIA CS1 → B1)

ERR9 TED END 034 J2 → J2: ERR7 — RECORD ERROR NO.39 → B4

COMPFIN ERR4 034 K2 → K2: TERM =1 → B4

OS (AT PROGRAM INTERRUPT) ENDMISS COMPFIN ERR9 → EODADIN 034 B4

B4: TERMINATING ERROR — TERR=1 — YES → F4; NO → C4

C4: ANY ENTRIES FOR PROGRAM BLOCK NO.0 IN ITAB — P.B.C IN ITAB? — YES → C5; NO → D4

C5: PBOBIT =1

D4: TRANSFER OMEGA (PBLCKEND)

D5: PBLCKEND 022 — WRITE LAST RECORD ON SYSUT3

E4: MORE THAN ONE OUTPUT TEXT RECORD — ONC =0? — YES → E5; NO → F4

E5: COB — WRITE LAST RECORD ON SYSUT1

F4: STORE CONSTANTS 0-15 IN CONST. POOL. UPDATE POINTER

PIROUT IN DIRECTORY → CLOSE 2 034 G3

G3: NOFREE =1

G4: DECK/LOAD? — YES → G5; NO → H4

G5: PRECOMPILED PROCEDURE? — NO; YES → H5

H4: CLOSE SYSUT3, SYSIN AND SYSUT1

H5: GENESD — GENERATE ESD RECORD FOR P

J4: FREEMAIN TO RELEASE PRIVATE AREA. (IF NOFREE =0)

J5: GENTXTS 127F1 — GENERATE TXT RECORDS FOR CONSTANT POOL

K3: IEX21

K4: TERMINATING ERROR — TERR=1 — YES → K3; NO → K5

K5: IEX20

Chart 035: Identifier Table Manipulation Phase - IEX20 Microfiche IEX20
Overall Flow

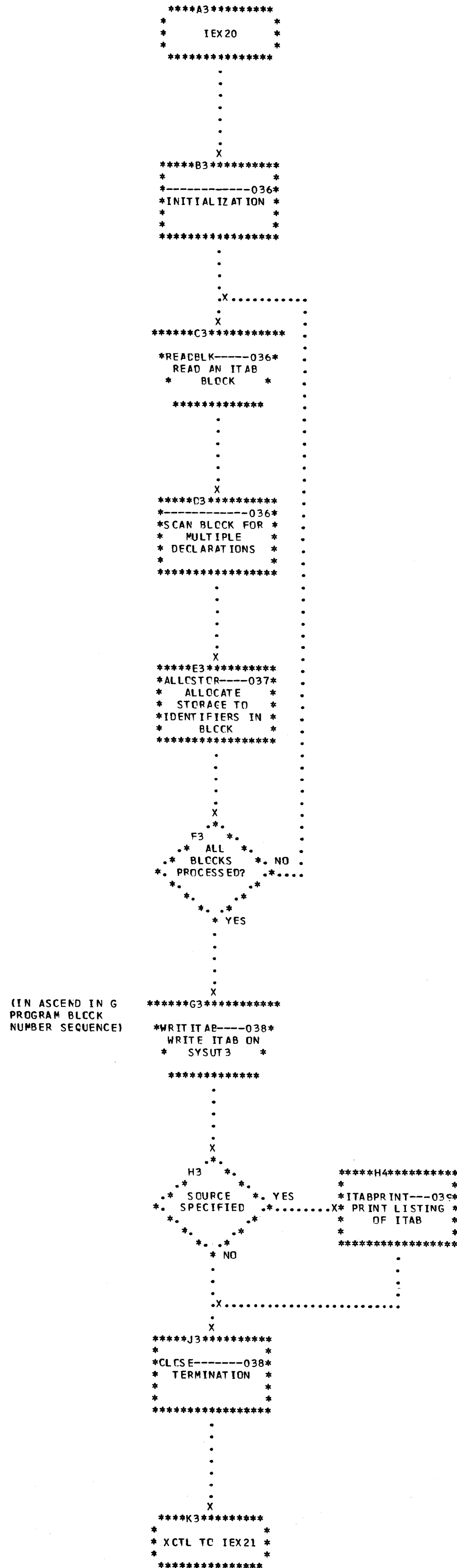


Chart 036: Identifier Table Manipulation Phase - IEX20 Microfiche IEX20
Initialization and READBLK

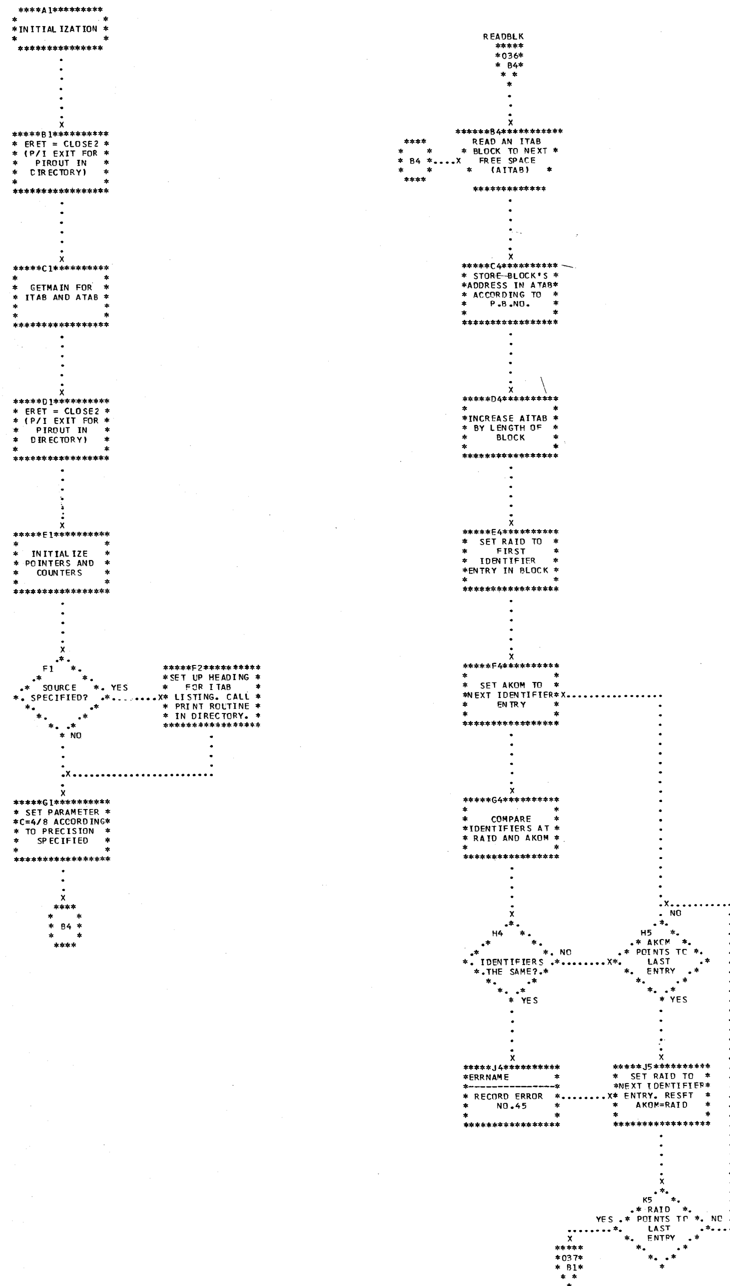


Chart 037: Identifier Table Manipulation Phase - IEX20 Microfiche IEX20
ALLOSTOR

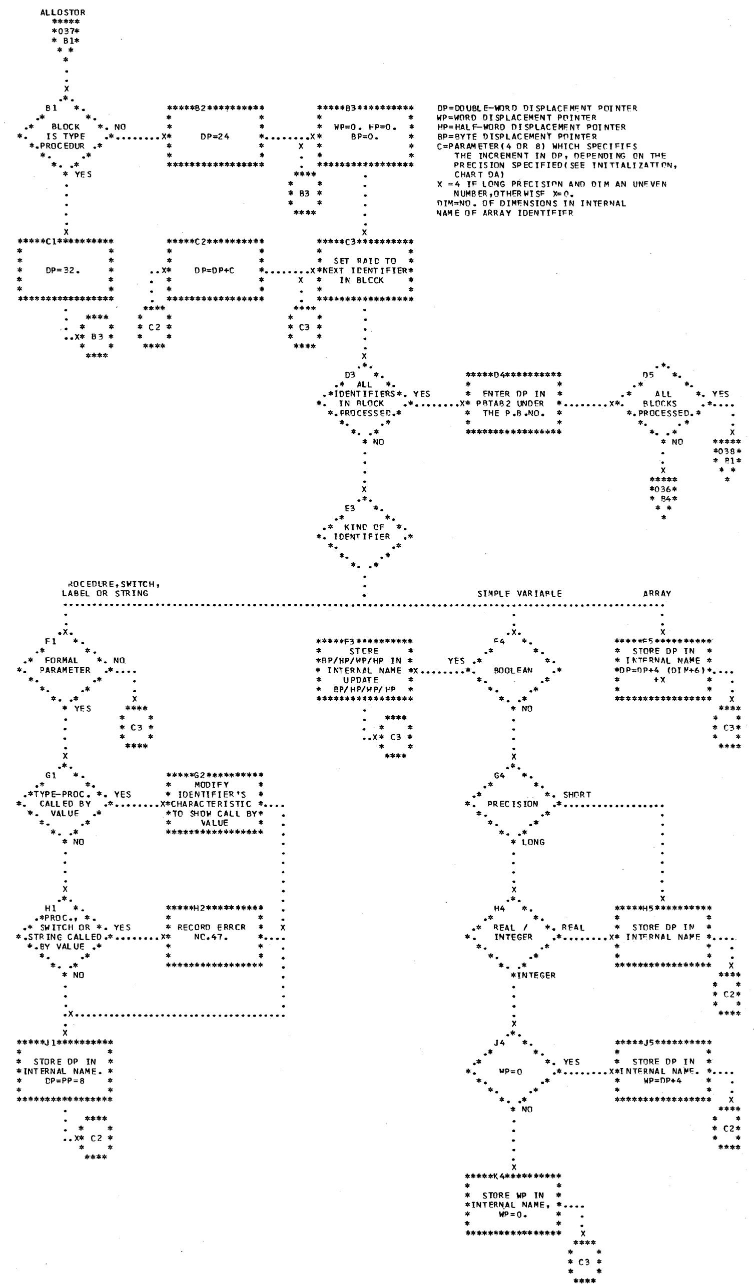


Chart 038: Identifier Table Manipulation Phase - IEX20 — WRITITAB and CLOSE

Microfiche IEX20

WRITITAB (038 B1)

- B1: ADDRESS FIRST ATAB ENTRY
- C1: CLOSE SYSUT3
- D1: GET ADDRESS OF NEXT ITAB BLOCK FROM ATAB. GET BLOCK LENGTH FROM HEADING
- E1: ALL BLOCKS WRITTEN OUT? — YES: 039 B1; NO: F1
- F1: WRITE THE BLOCK ON SYSUT3
- G1: ADDRESS NEXT ATAB ENTRY
- H1: ALL BLOCKS WRITTEN OUT? — NO: D1; YES: 039 B1

CLOSE (038 B4)

- B4: FREEMAIN FOR PRIVATE AREA (ITAB AND ATAB)
- CLOSE2
- C4: XCTL TO IEX21

Chart 039: Identifier Table Manipulation Phase - IEX20 — ITABPRNT

Microfiche IEX20

ITABPRNT (039 B1)

- B1: SOURCE SPECIFIED? — NO: 038 B4; YES: C1
- C1: ADD 1 TO P.B.NO. (PBN=0 OR 1 AT START)
- D1: USING ATAB ADDRESS THE CORRESPONDING BLOCK IN ITAB
- E1: PRINTITB — PRINT BLANK LINE (CALL PRINT)
- F1: CONVERT — CONVERT PBN TO EBCDIC
- F2: MOVE PBN TO PRINT AREA
- G1: CONVERT — CONVERT SURROUNDING PBN (FROM PBTAB1)
- G2: MOVE SURROUNDING PBN TO PRINT AREA
- H1: CONVERT SC (FROM HEADING ENTRY) AND MOVE TO PRINT AREA
- J1: ANY IDENTIFIERS IN BLOCK? — NO: J2; YES: B2
- J2: PRINTITB — PRINT PBN, SC AND SURROUNDING PBN

GETNVAR

- B2: ALL ENTRIES IN BLOCK PROCESSED? — YES: B4; NO: C2
- C2: SET RAID TO NEXT ITAB ENTRY IN ALPHABETICAL ORDER.

TESTVAR

- D2: IS IDENTIFIER ALL-PURPOSE IDENTIF. — NO: D3; YES: H3
- D3: IDENTIFIER TYPE SPECIFIED — YES: D4; NO: E4
- D4: MOVE SYMBOL IR OR B TO WORK TO SHOW IDENTIFIER TYPE
- E4: KIND OF IDENTIFIER — SWITCH ARRAY, PROCEDURE OR CODE PROCEDURE: F3; SIMPLE VARIABLE: H4; STRING OR LABEL: F5
- F3: MOVE SYMBOL S, A, P OR C TO WORK TO SHOW IDENTIFIER KIND
- F5: MOVE SYMBOL T OR L TO WORK TO SHOW IDENTIFIER TYPE
- G3: CONVERT2 — CONVERT PARAM/ COMP/DIMEN.COUNT AND MOVE TO WORK

ENDBLOCK

- B4: LAST LINE PRINTED — NO: B5; YES: C4
- B5: PRINTITB — PRINT LAST TIME (CALL PRINT)
- C4: LAST ITAB BLOCK PROCESSED — NO: C1; YES: 038 B4

ENDVAR

- H3: TRANSLATE EXTERNAL NAME, CONVERT DISPL. OR LN AND MOVE TO WORK
- H4: IDENTIFIER = FORMAL PARAMETER — NO: H3; YES: H5
- H5: MOVE SYMBOL N OR V TO WORK TO SHOW CALL BY NAME/VALUE — H3
- J3: MOVE DATA IN WORK TO PRINT AREA
- J4: LINE FILLED? — NO: J5; YES: K4
- J5: ENDVAR2- SHIFT ITAB BLOCK UPWARD TO DELETE IDENTIF. PROCESSED — B2
- K4: PRINTITB — PRINT THE LINE (CALL PRINT)

Chart 040: Diagnostic Output - IEX21 — Microfiches IEX21-1 and IEX21M-1

Overall Flow - Error Message Editing Routine (IEX60000)

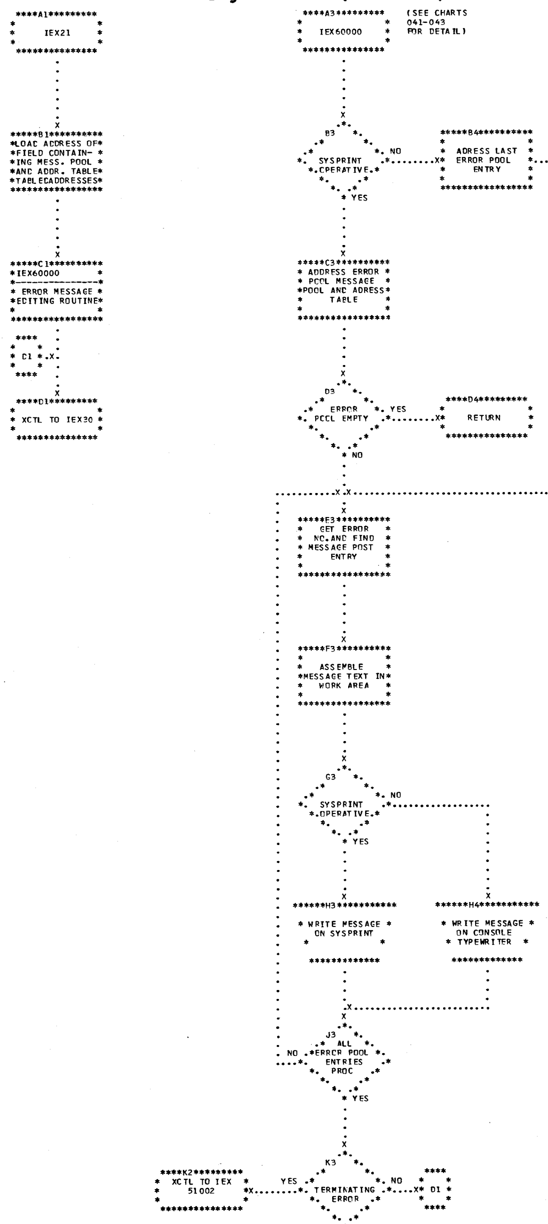


Chart 041: Diagnostic Output - IEX21 — Microfiche IEX21-1

Error Message Editing Routine

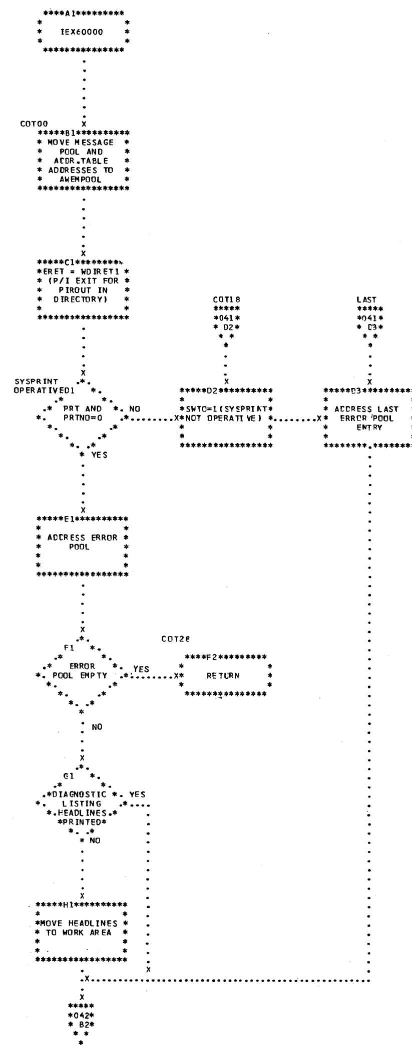


Chart 042: Diagnostic Output - IEX21 Microfiche IEX21-1
Error Message Editing Routine

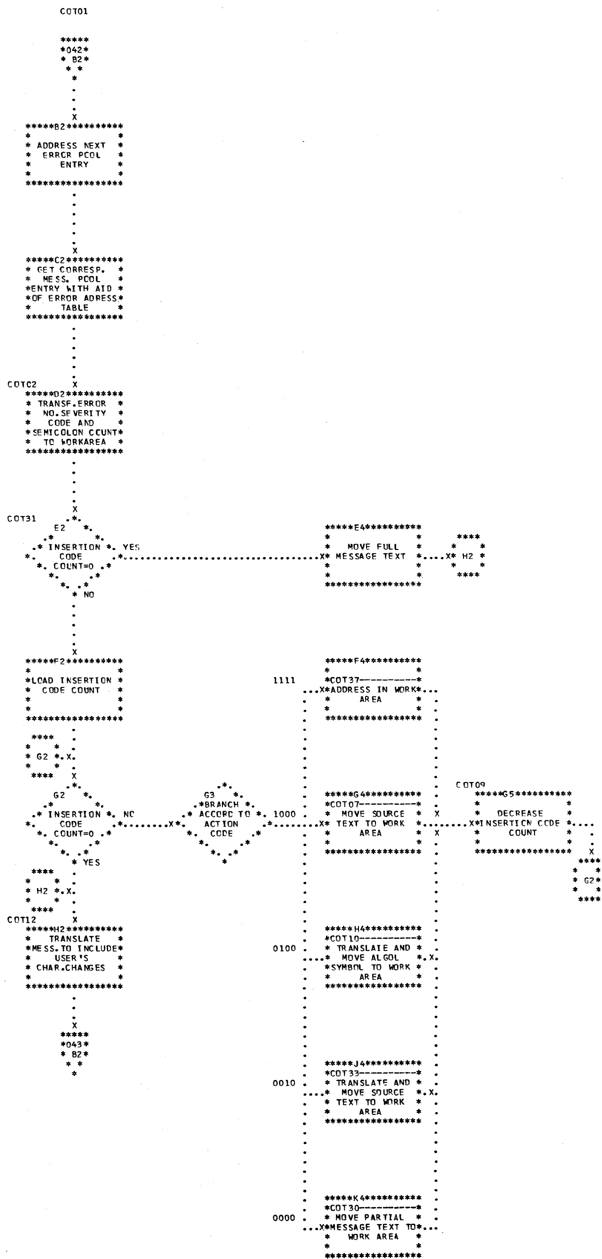


Chart 043: Diagnostic Output - IEX21 Microfiche IEX21-1
Error Message Editing Routine

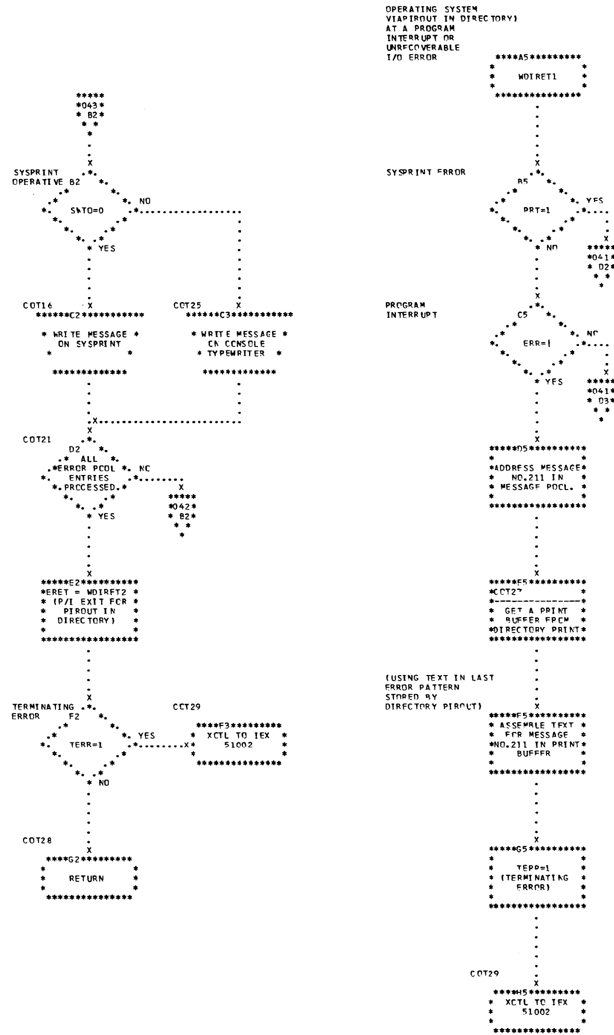


Chart 044: Scan III Phase - IEX30 Overall Flow

Microfiches IEX30-1, IEX30-2 and IEX30-3

NOTE: ALL OPERATORS EXCEPT ZETA ARE TRANSFERRED TO THE MODIFICATION LEVEL 2 TEXT

ITAB = IDENTIFIER TABLE
SUTAB = SUBSCRIPT TABLE
ARIDSTAB = ARRAY IDENTIFIER STACK
CRIDTAB = CRITICAL IDENTIFIER TABLE
LVTAB = LEFT VARIABLE TABLE

A3: IEX30

B3: INITIATE----046 — GET PRIVATE AREA. READ FIRST ITAB AND SOURCE RECORDS

044 C3

C3: GENTFST-----046 — SCAN TO NEXT SIGNIFICANT CHARACTER

D3: BRANCH ACCORDING TO CHARACTER

BETA,PI,PHI (OPENING OF A BLOCK OR PROCEDURE)
E1: BETA--------047 / PIPHI-------047 — READ NEXT ITAB BLOCK FROM SYSUT3 → C3

EPSILON (CLOSE OF A BLOCK OR PROCEDURE)
E2: EPSILON-----047 — DELETE THE CORRESPONDING ITAB BLOCK IN CORE → C3

FOR
E4: FOR---------047 — SET SWITCH TO INDICATE ENTRY TO FOR LIST → C3

STEP,WHILE
E5: STEP--------047 / WHILE------047 — MARK FSTAB ENTRY TO SHOW OP. IN FOR LIST → C3

DIVIDE,POWER
G1: DIPOW-------047 — MARK ESTAB ENTRY IF OP. IN FOR LIST → C3

DO
G2: DO----------049 — SET SWITCH TO INDICATE EXIT FROM FOR LIST → C3

ETA (END OF FOR STMT.)
G4: ETA---------049 — DELETE FOR LIST IDENTIFIERS IN CRIDTAB → C3

GAMMA
G5: GAMMA-------049 — TRANSFER CODE PROCEDURE IDENTIFIER FROM INPUT TEXT → C3

APOSTROPHE
J1: QUOTE-------049 — TRANSFER STRING INTERNAL NAME FROM INPUT TEXT → C3

ZETA
J2: ICHA 049 — READ NEXT MOD. LEV.1 RECORD FROM SYSUT1 → C3

→ 045 A3

OMEGA (SOURCE MODULE END)
J4: OMEGA-------050 — WRITE LAST LVTAB,SUTAB AND TEXT RCRDS. FREEMAIN

K4: XCTL TO IEX31

RHO
J5: RHO---------051 — REPLACE LAST) BY A COMMA → C3

Chart 045: Scan III Phase - IEX30 Overall Flow

Microfiches IEX30-1, IEX30-2 and IEX30-3

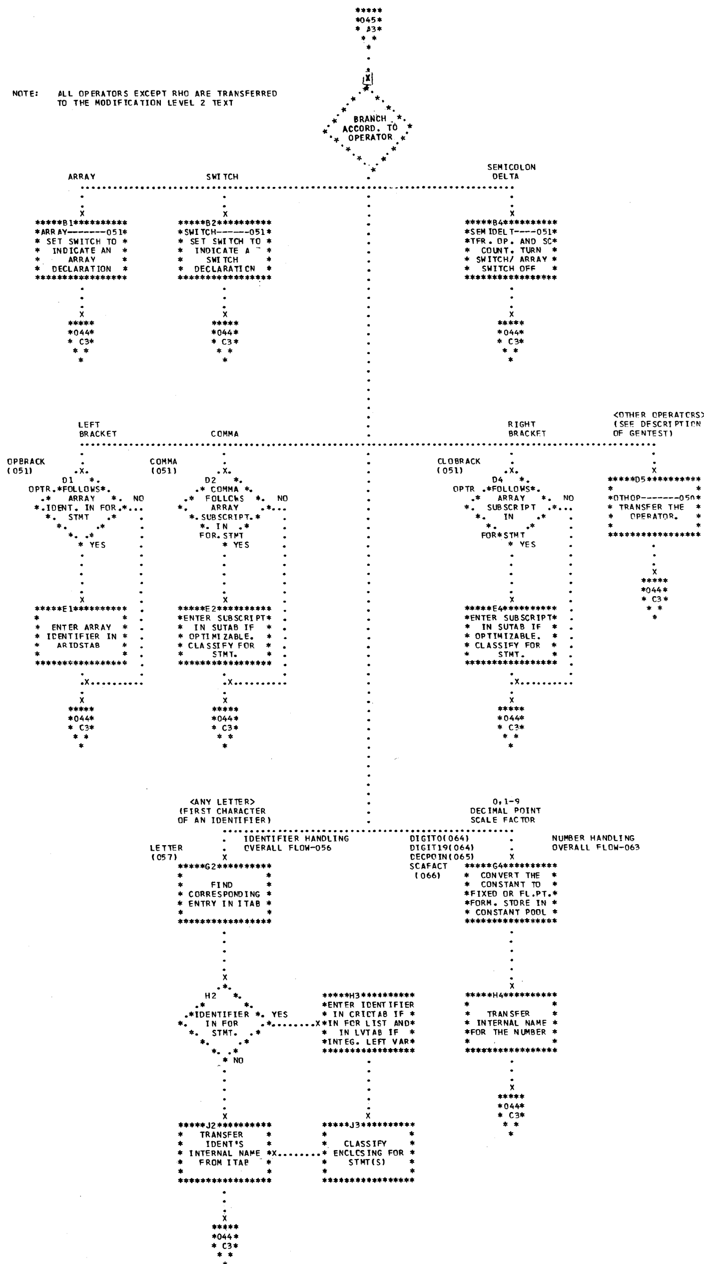


Chart 046: Scan III Phase - IEX30 INITIATE and GENTEST

Microfiche IEX30-1

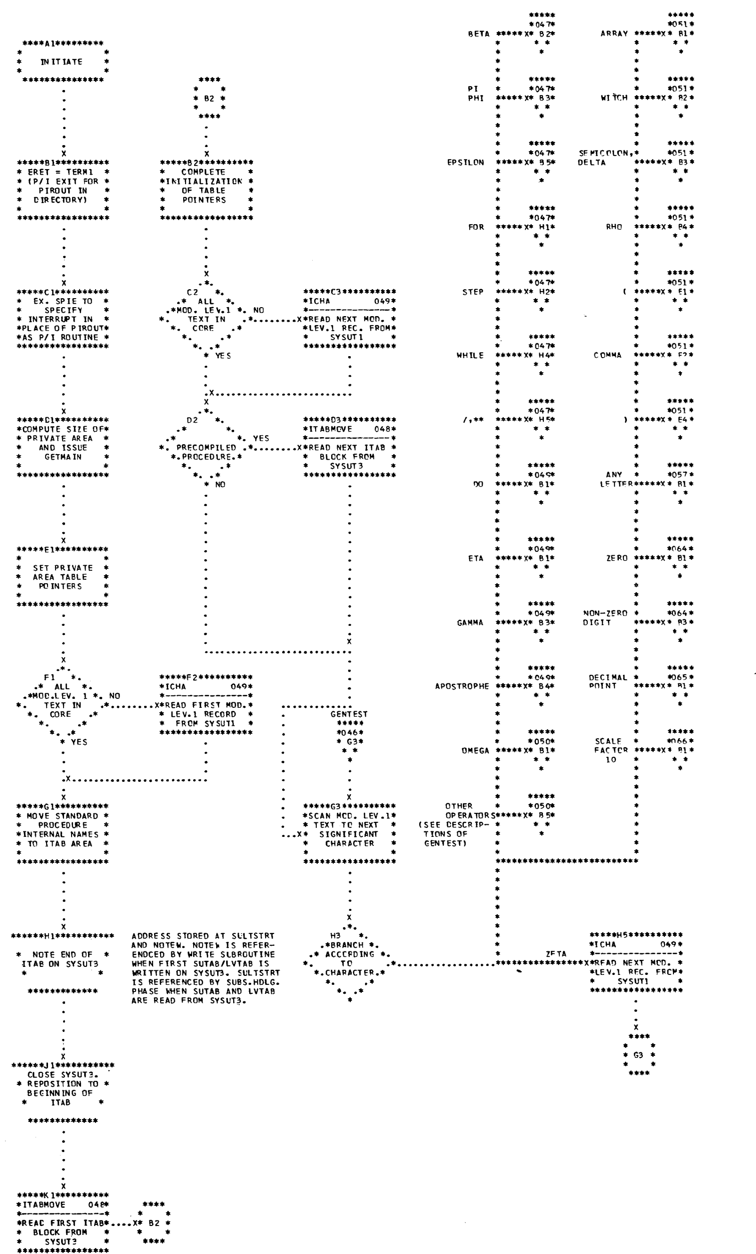
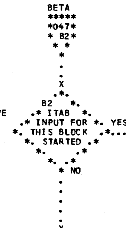
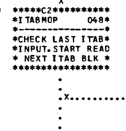
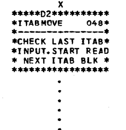
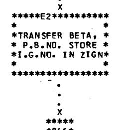
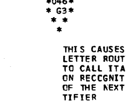
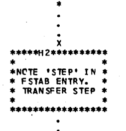
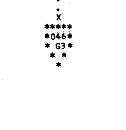
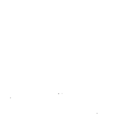
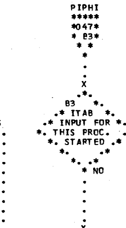
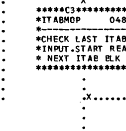
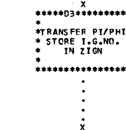
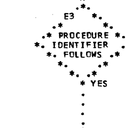
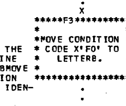
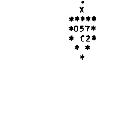


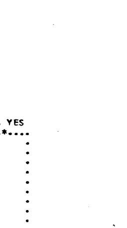
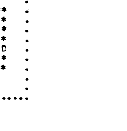
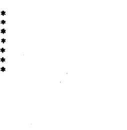
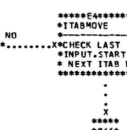
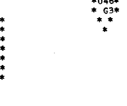
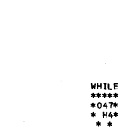
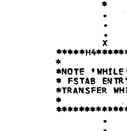
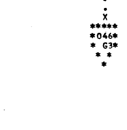
Chart 047: Scan III Phase - IEX30 — Microfiche IEX30-1

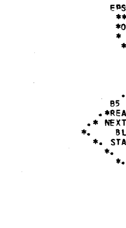
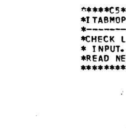
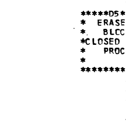
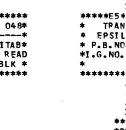
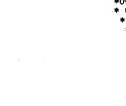
BETA, PIPHI, EPSILON, FOR, STEP, WHILE and DIPOW

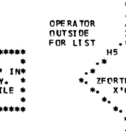
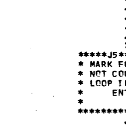
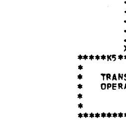






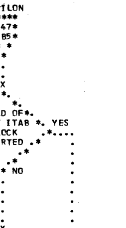
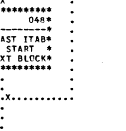
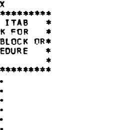



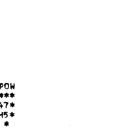
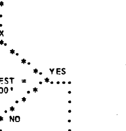
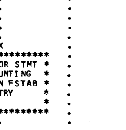
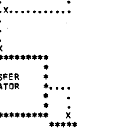
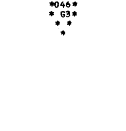
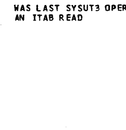
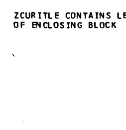
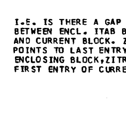






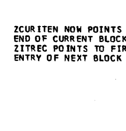
Chart 048: Scan III Phase - IEX30 — Microfiche IEX30-2

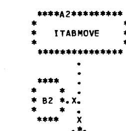
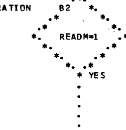
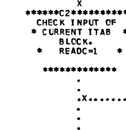
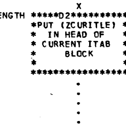
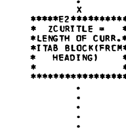
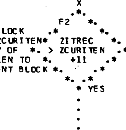
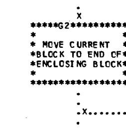
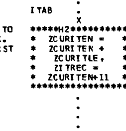
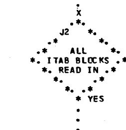
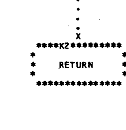
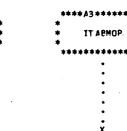
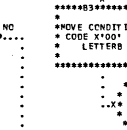
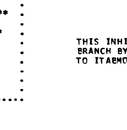
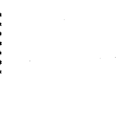
ITABMOVE and ITABMOP

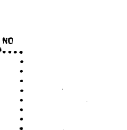
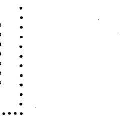
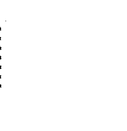
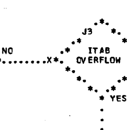
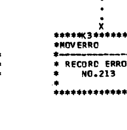
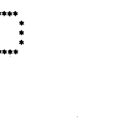
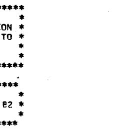
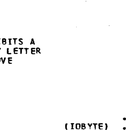
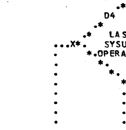
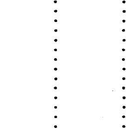
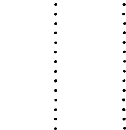
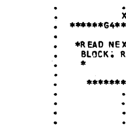
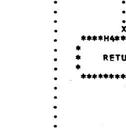
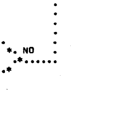
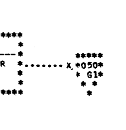


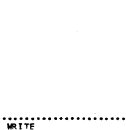
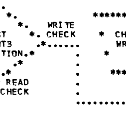
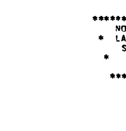
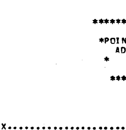
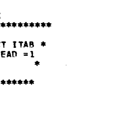
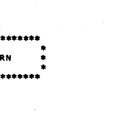






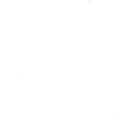
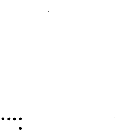
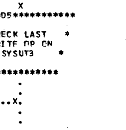
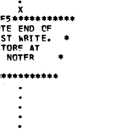
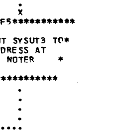
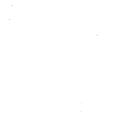







Chart 049: Scan III Phase - IEX30 — Microfiche IEX30-1

DO, ETA, GAMMA, QUOTE, ICHA and OUCHA (ICHA and OUCHA: IEX30-2)

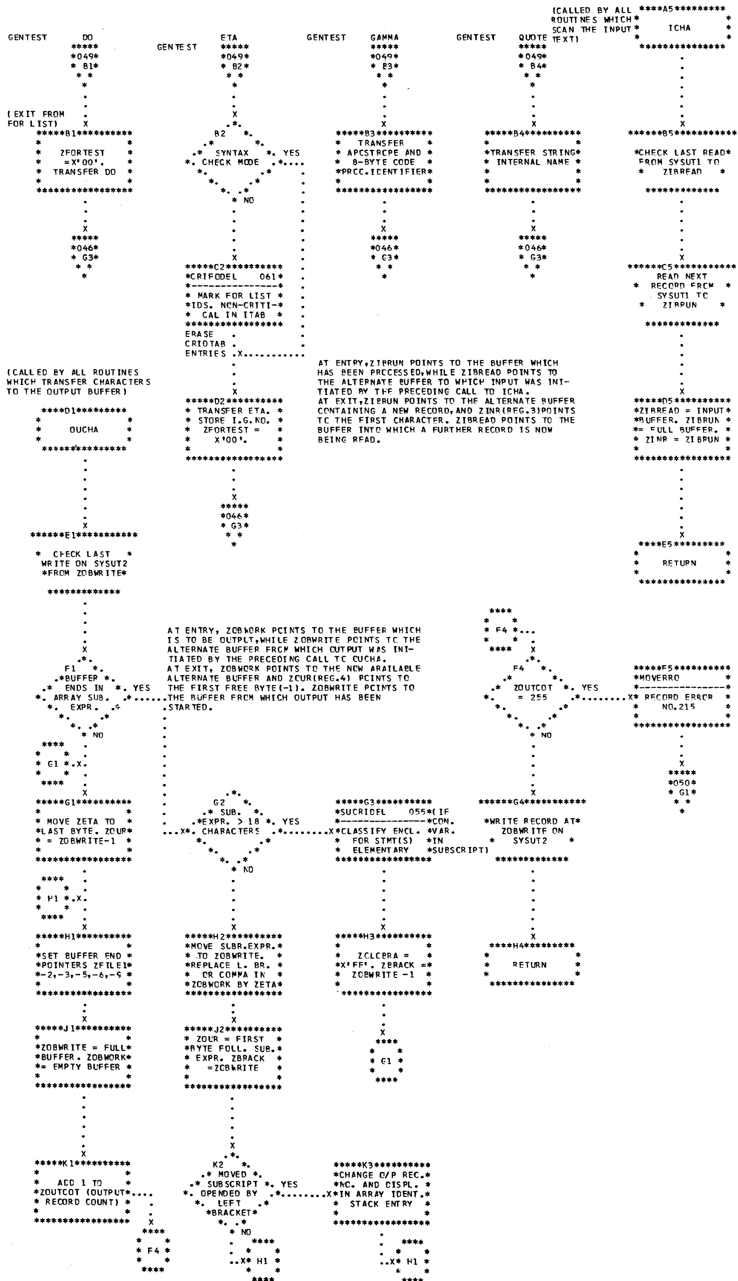


Chart 050: Scan III Phase - IEX30 — Microfiche IEX30-1

OMEGA, OTHOP

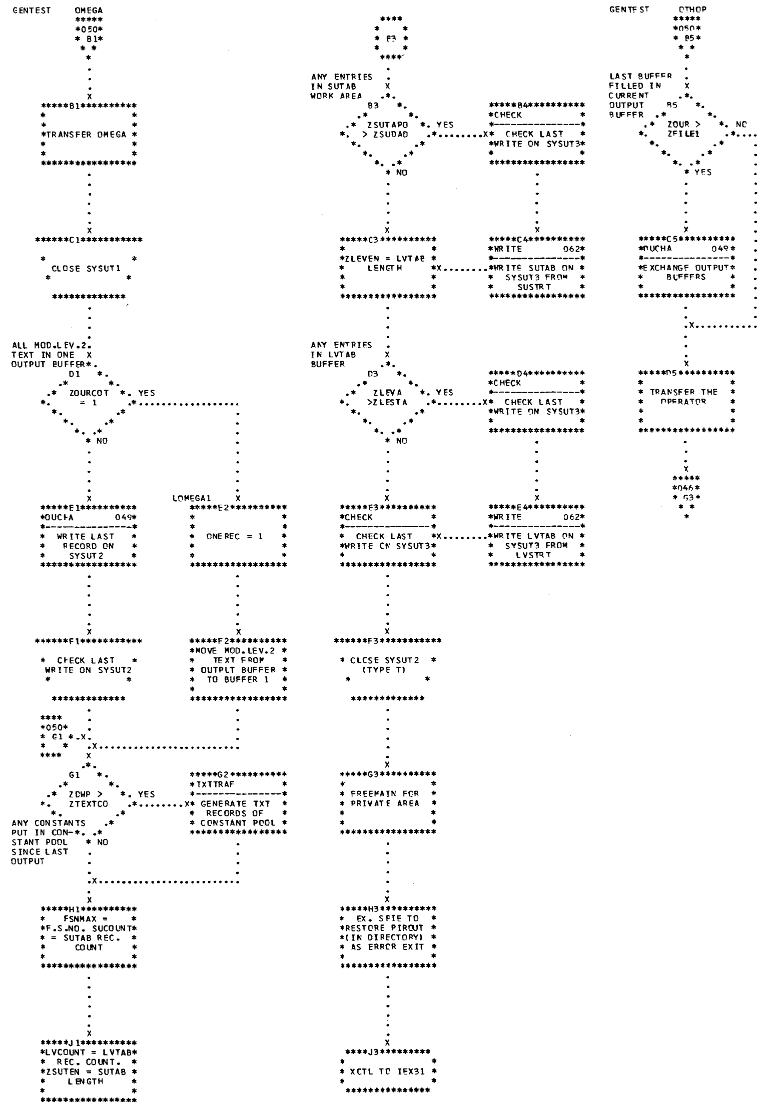


Chart 051: Scan III Phase - IEX30 — Microfiche IEX30-1

ARRAY, SWITCH, SEMIDELT, RHO, OPBRACK, COMMA and CLOBRACK

ALL ROUTINES ENTERED FROM GENTEST

- ARRAY (051 B1) → B1: SARRAY = 1. TRANSFER ARRAY → 046 G3
- SWITCH (051 B2) → B2: SSWITCH = 1 TRANSFER SWITCH → 046 G3
- SEMIDELT (051 B3) → B3: TRANSFER SEMICOLON OR DELTA AND SEMICOLON COUNT → 046 G3
- RHO (051 B4) → B4: LAST TRANSFERRED CHARACTER =) ? YES → B5: REPLACE) BY A COMMA → 046 G3; NO → 046 G3

OPBRACK (051 D1)
- D1: LEFT BRACKET IN ARRAY DECL. ? YES → K1; NO → E1
- E1: SYNTAX CHECK MODE ? YES → K1; NO → F1
- F1: ARRAY IN FOR STMT ? NO → K1; YES → G1
- G1: ARRAY IN SUBSCRIPT EXPRESS. ? NO → J1; YES → H1
- H1: SUCRIDEL 055 — CLASSIFY ENCL. FOR STMT(S) ELEMENTARY (IF CON VAR. IN ENCL. SUBSCRIPT) → J1
- J1: ENTER ARRAY IDENT. IN ARIDSTAK ZBRACK=ADDR OF LEFT BRACKET
- K1: TRANSFER LEFT BRACKET → 046 G3

COMMA (051 D2)
- D2: SYNTAX CHECK MODE ? YES → K2; NO → E2
- E2: COMMA FOLLOWS ARRAY SUB. IN FOR STMT ? NO → K2; YES → F2
- F2: SUBSCRIPT CONTAINS AN ARRAY ? (ZCLOBRA =X'FF') YES → F3: ZCLOBRA = X'00' → G3; NO → G2
- G2: SUBSCRIPT > 18 CHAR. ? YES → G3; NO → H2
- G3: SUCRIDEL 055 — CLASSIFY ENCL. FOR STMT(S) ELEMENTARY → J2
- H2: SUSCRITE 052 — ENTER SUBSCRIPT IN SUTAB IF OPTIMIZABLE → J2
- J2: ZBRACK = ADDRESS OF COMMA IN OUTPUT BUFFER
- K2: TRANSFER THE COMMA → 046 G3

CLOBRACK (051 D4)
- D4: RIGHT BRACKET IN ARRAY DECL. ? YES → K5; NO → E4
- E4: SYNTAX CHECK MODE ? YES → K5; NO → F4
- F4:) FOLLOWS ARRAY SUB. IN FOR-STMT ? NO → K5; YES → G4
- G4: SUBSCRIPT CONTAINS AN ARRAY ? (ZCLOBRA =X'FF') YES → H5; NO → H4
- H4: SUBSCRIPT > 18 CHAR. ? YES → H5; NO → J4
- H5: SUCRIDEL 055 — CLASSIFY ENCL. FOR STMT(S) ELEMENTARY (IF CON. VAR. IN ENCL. SUBSCRIPT) → J5
- J4: SUSCRITE 052 — ENTER SUBSCRIPT IN SUTAB IF OPTIMIZABLE → J5
- J5: DELETE ARRAY IDENTIFIER IN ARIDSTAK → K3
- K3: IS THERE AN EMBRACING SUBSCRIPT ? YES → K4; NO → K5
- K4: ZCLOBRA =X'FF'. ZBRACK = ADDR. OF OF RT. BRACKET IN OUTPUT BUFFER → K5
- K5: TRANSFER) → 046 G3

Chart 052: Scan III Phase - IEX30 — Microfiche IEX30-1

SUSCRITE

OPBRACK, COMMA, CLOBRACK

- A1: SUSCRITE → A2
- A2: SUBR=ZBRACK (BEGINNING OF SUBSCRIPT) → A3
- A3: OPERAND 054 (FIRST OPERAND ADDRESSED BY SUBR) — OPPTR=ADDR. OF CRIDTAB ENTRY IF OPD.CON.VAR. (IF NON-INTEGER OPERAND OR OPERATOR → F2) → B3

NOTE: OPPTR, CVR AND ADDENDR ARE REGISTERS WHICH ADDRESS ENTRIES IN CRITICAL IDENTIFIER TABLE (CRIDTAB). FACTOR AND ADDEND ARE LOCATIONS TO WHICH OPERANDS ARE MOVED FROM THE OUTPUT BUFFER.

- B3: OPERATOR FOLLOWING 1ST. OPERAND ? + OR − → SUBPM (C1); <ANY OPERATOR OTHER THAN +,−OR *> → F2; * → SUBAST (C3); <END OF SUBSCRIPT> → SUBONE (C5)
- C1 (SUBPM): MOVE FIRST OPD. AND SIGN TO ADDEND. ADDENDR=OPPTR → D1
- C3 (SUBAST): SUBMULT 054 (FIRST AND SECOND OPERANDS) — CVR=ADDR. CRIDTAB ENTRY IF OPD. CON. VAR. (2ND OPD. NON-INTEGER OR OPERATOR → F2) → D3
- C4: CVR = OPPTR. MOVE +/−1 TO FACTOR (SIGN OF OPD) → D4
- C5 (SUBONE): 1ST OPD. IS CONTROLLED VARIABLE ? YES → C4; NO → D5
- D1: OPERAND 054 (SECOND OPERAND) — OPPTR=ADDR. OF CRIDTAB ENTRY IF OPD.CON.VAR. (IF NON-INTEGER OPERAND OR OPERATOR → F2) → E1
- D3: OPERATOR FOLLOWING 2ND. OPERAND ? <ANY OPERATOR OTHER THAN + OR −> → F2; END OF SUBSCRIPT → C4; + OR − → E4
- D4: MOVE 0 TO ADDEND → 053 D1
- D5: CVR = OPPTR. MOVE OPERAND TO ADDEND. MOVE 0 TO FACTOR → E5 (F.S.NO. FROM CRIDTAB ENTR)
- E1: OPERATOR FOLLOWING 2ND. OPERAND ? * → F1 (SECOND AND THIRD OPERANDS); ANY OPERATOR OTHER THAN * → SUBNOOP (F2); END OF SUBSCRIPT → SUBPM1 (G3)
- E4: OPERAND 054 (THIRD OPERAND) — OPPT=ADDR.OF CRIDTAB ENTRY IF OPD.CON.VAR. → F4
- E5: ADDRESS FSTAB ENTRY OF NEXT OUTERMOST ENCL. FOR STMT. → F5
- F1: SUBMULT 054 — CVR=ADDR. CRIDTAB ENTRY IF OPD. CON.VAR. → 053 B1
- F2 (SUBNOOP): SUCRIDEL 055 — CLASSIFY ENCL. FOR STMT(S) ELEMENTARY (IF CON. VAR. IN SUBSCRIPT → C52 F2) → G2
- F4: MOVE OPERAND TO ADDEND. ADDENDR=OPPTR → 053 B1
- F5: OPTIMIZATION POSSIBLE ? YES → G5; NO → J5
- G2: RETURN
- G3 (SUBPM1): ADDENDR = OPPTR ? ADDENDR>OPPTR (1ST OPD.IS CON.VAR.) → H2; ADDENDR < OPPTR (2ND OPD.IS CON.VAR.) → H3; ADDENDR=OPTTR. (NEITHER OPD. IS CON.VAR., OR BOTH OPDS.THE SAME XCON.VAR<) → H4 (ADDEND NOT= 0)
- G5: ARRAY AND ADDEND DECL. OUTSIDE FOR STMT ? YES → H5; NO → J5
- H2: CVR = ADDENDR. ADDENDR =OPPTR. MOVE SIGN OF 1ST OPD. TO FACTOR → J2
- H3: CVR = OPPTR MOVE SIGN OF 2ND OPD. TO FACTOR → J3
- H4: FIRST OPERAND IS CONTROLLED VARIABLE ? YES → 053 D4; NO → J4
- H5: SUTABENT 053 — ENTER SUBSCRIPT IN SUTAB UNDER F.S.NO. → J5
- J2: MOVE 2ND OPD TO ADDEND. MOVE 1 TO FACTOR → 053 B1
- J3: MOVE 1 TO FACTOR → 053 B1
- J4: RETURN
- J5: LAST CRIDTAB ENTRY PROCESSED ? NO → E5; YES → K5
- K5: RETURN

Chart 053: Scan III Phase - IEX30 — Microfiche IEX30-1

SUSCRITE and SUTABENT

(FROM CHART 052)

053 B1

B1: ADDENDR IS CONTROLLED VARIABLE? — YES → B3; NO → C3 path

B3: CLASSIFY FOR STATEMENT ELEMENTARY LOOP IN FSTAB

C3: CVR:ADDENDR
- CVR > ADDENDR ENDR (CVR IS CON.VAR. OF INNERMOST EMBRACING FOR-STMT) → D1
- CVR < OR = ADDENDR (OPTIMIZATION IMPOSSIBLE- IF CVR=ADDENDR = 0,NEITHER OPD. IS CON. VAR. IF CVR=ADDENDR NOT = 0,BOTH OPDS. ARE THE SAME CONTROL VAR. IF CVR < ADDENDR,ADDENDR IS CON.VAR. OF A NESTED FOR STMT) → D4

(FROM CHART 052)

SUBFIN1

D1: OPTIMI. POSSIBLE FOR STMT OF CVR (FROM FSTAB) — NO → D4; YES → E1

053 D1

053 D4

SUBFIN4

D4: CLASSIFY THE FOR-STATEMENT ELEMENTARY LOOP IN FSTAB

E1: ARRAY FACTOR AND ADDEND DECL. OUTSIDE FOR STATEMENT — NO → D4; YES → F1

E4: RETURN

F1: SUTABENT 053 — ENTER SUBSCRIPT IN SUTAB UNDER F.S. NO.

G1: RETURN

SUSCRITE

F3: SUTABENT

G3: WRITE SUBROUTINE CALLED — YES → G4; NO → H3

WAS SUTAB BUFFER FILLED BY PRECEDING ENTRY AND WRITE SUBROUTINE CALLED (INDICATED IF POINTER ZSLTAPO WAS SET TO SUTAB START)?

G4: CHECK — CHECK LAST WRITE ON SYSUT3

H3: ENTER SUBSCRIPT IN SUTAB

H4: MAX. SUTAB LENGTH REACHED — NO → H3; YES → H5

H5: TABOFLO — MARK ENCL. FOR STATEMENTS NOT OPTIM.

(DELETE ARIDSTAB AND CRIDTAB)

J3: BUFFER FILLED — YES → J4; NO → K3

J4: CHECK — CHECK LAST LVTAB WRITE ON SYSUT3

J5: MOVERRO — RECORD ERROR NO.87

D4

K3: RETURN

K4: WRITE 062 — WRITE SUTAB ON SYSUT3

Chart 054: Scan III Phase - IEX30 — Microfiche IEX30-1

OPERAND and SUBMULT

SUSCRITE SUBMULT

A1: OPERAND

B1: SUBR POINTS TO INTEGER OPERAND — NO (NON-INTEGER OPERAND OR OPERATOR) → 052 F2; YES → C1

C1: OPERAND A CRITICAL IDENTIFIER — NO → C2; YES → D1

C2: OPPTR =0 (OPERAND NOT CONTROLLED VARIABLE)

D1: SET OPPTR TO CORRESPONDING ENTRY IN CRIDTAB

D2: RETURN

E1: IDENTIFIER IS CONTROLLED VARIABLE — NO → E2; YES → F1

E2: OPPTR = 0 (OPERAND NOT CONTROLLED VARIABLE)

F1: RETURN

SUSCRITE

A3: SUBMULT

ADDRESS NEXT OPERAND

B3: MOVE OPERAND TO FACTOR. CVR=OPPTR SUBR=SUBR+6

C3: OPERAND 054 — OPPTR=ADDR. OF CRIDTAB ENTRY IF OPD.CON.VAR.

D3: OPPTR:CVR
- OPPTR < CVR (FIRST OPERAND IS CON. VAR.) → E3
- OPPTR > CVR (SECOND OPERAND IS CONTROLLED VARIABLE) → E4
- OPPTR = CVR (NEITHER OPERAND IS CON.VAR.,OR BOTH OPERANDS THE SAME CON.VAR.) → 052 F2

E3: MOVE SECOND OPERAND TO FACTOR

E4: CVR =OPPTR OPPTR =CVR

F3: OPERAND AT FACTOR ALSO CON.VAR. — YES → F4; NO → G3

F4: CLASSIFY FOR STATEMENT ELEMENTARY IN FSTAB

G3: RETURN

Chart 055: Scan III Phase - IEX30 Microfiche IEX30-1

SUCRIDEL

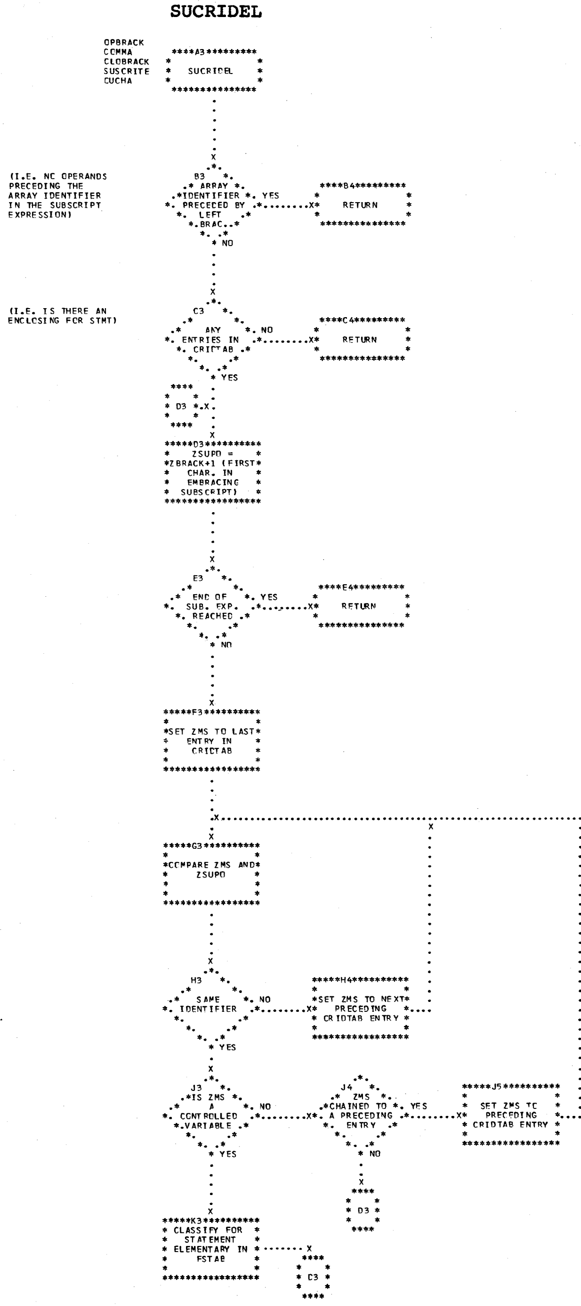


Chart 056: Scan III Phase - IEX30 Microfiche IEX30-1 and IEX30-2

Overall Flow - Identifier Handling

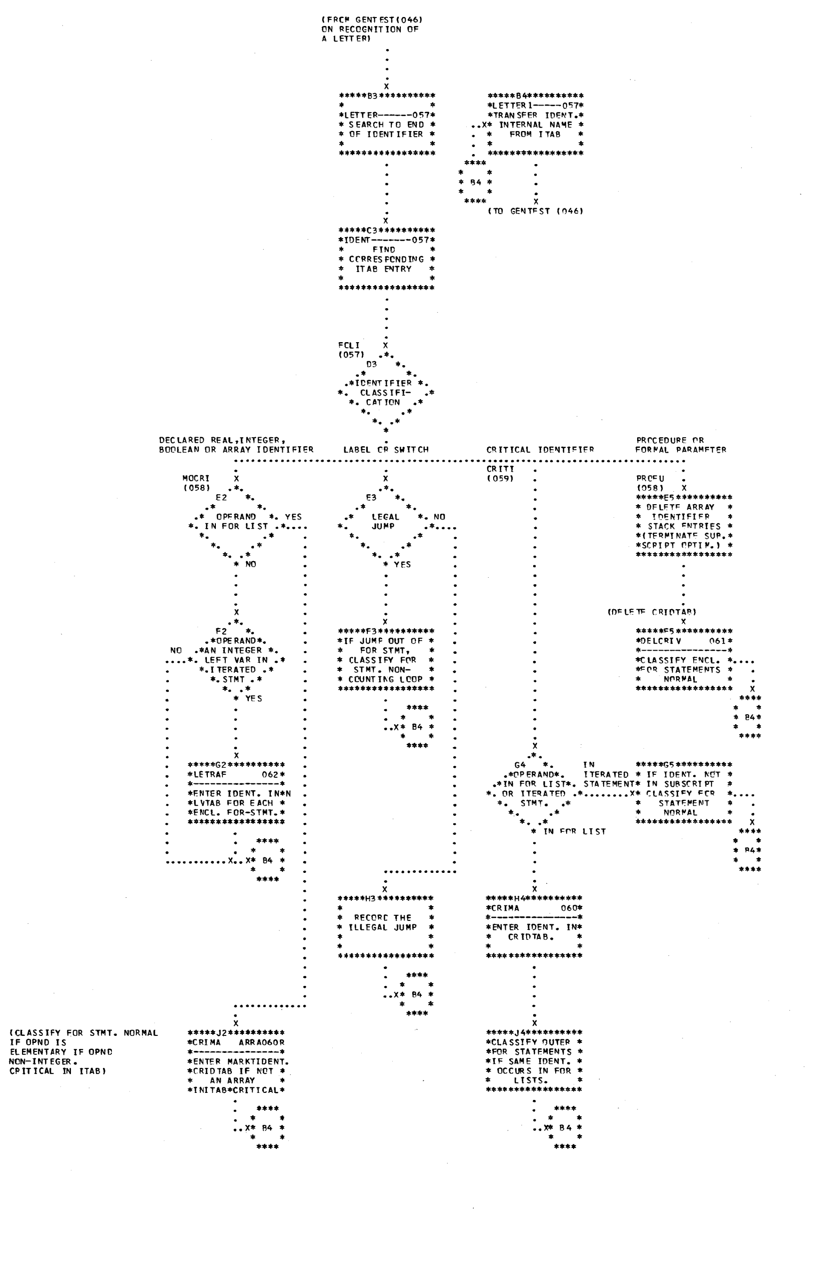


Chart 057: Scan III Phase - IEX30 Microfiche IEX30-1

LETTER, IDENT and FOLI

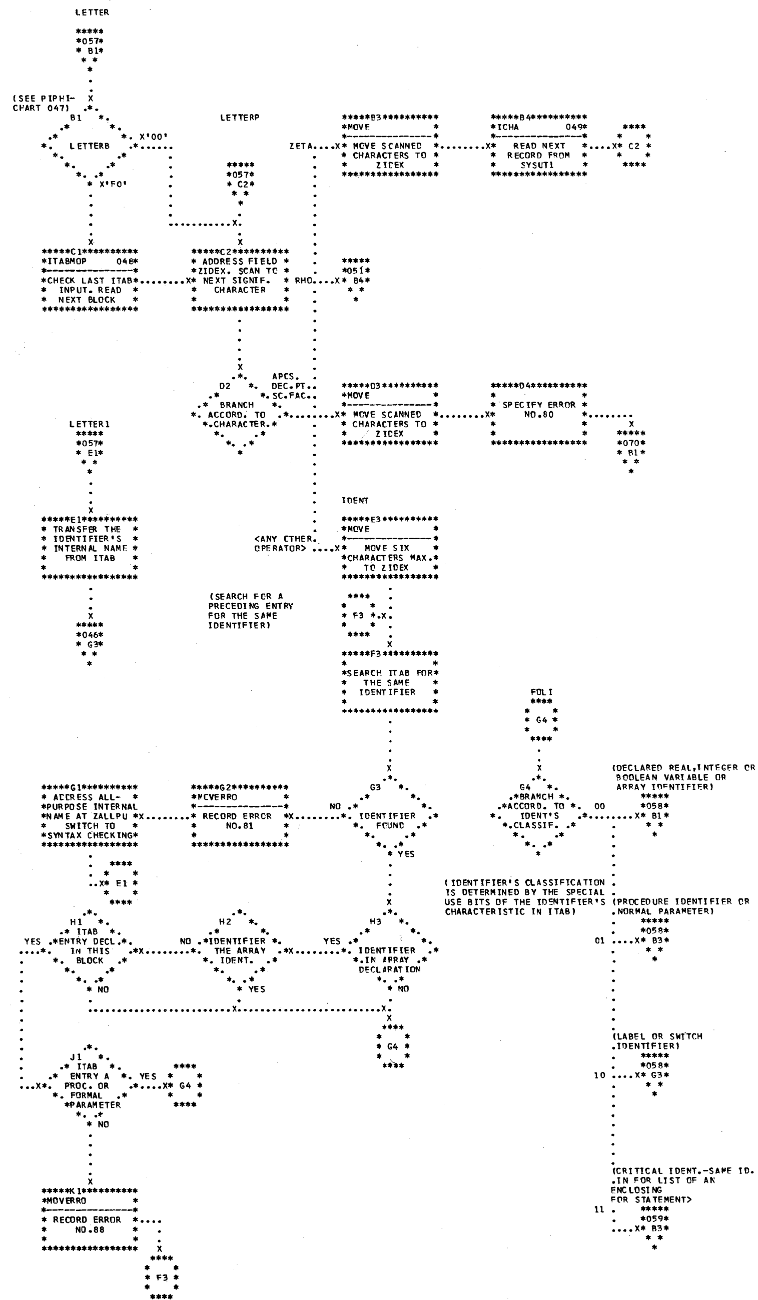


Chart 058: Scan III Phase - IEX30 Microfiche IEX30

NOCRI, PROFU and SWILA

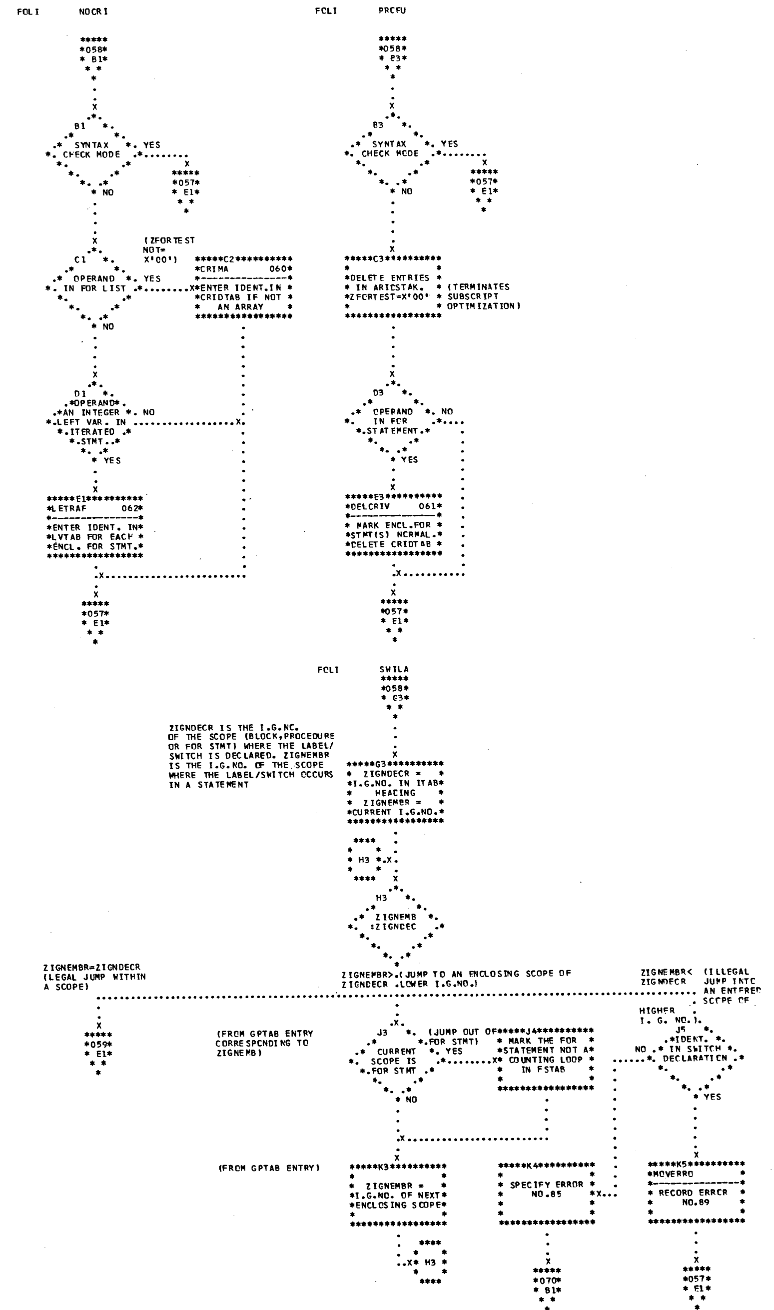


Chart 059: Scan III Phase - IEX30 CRITI

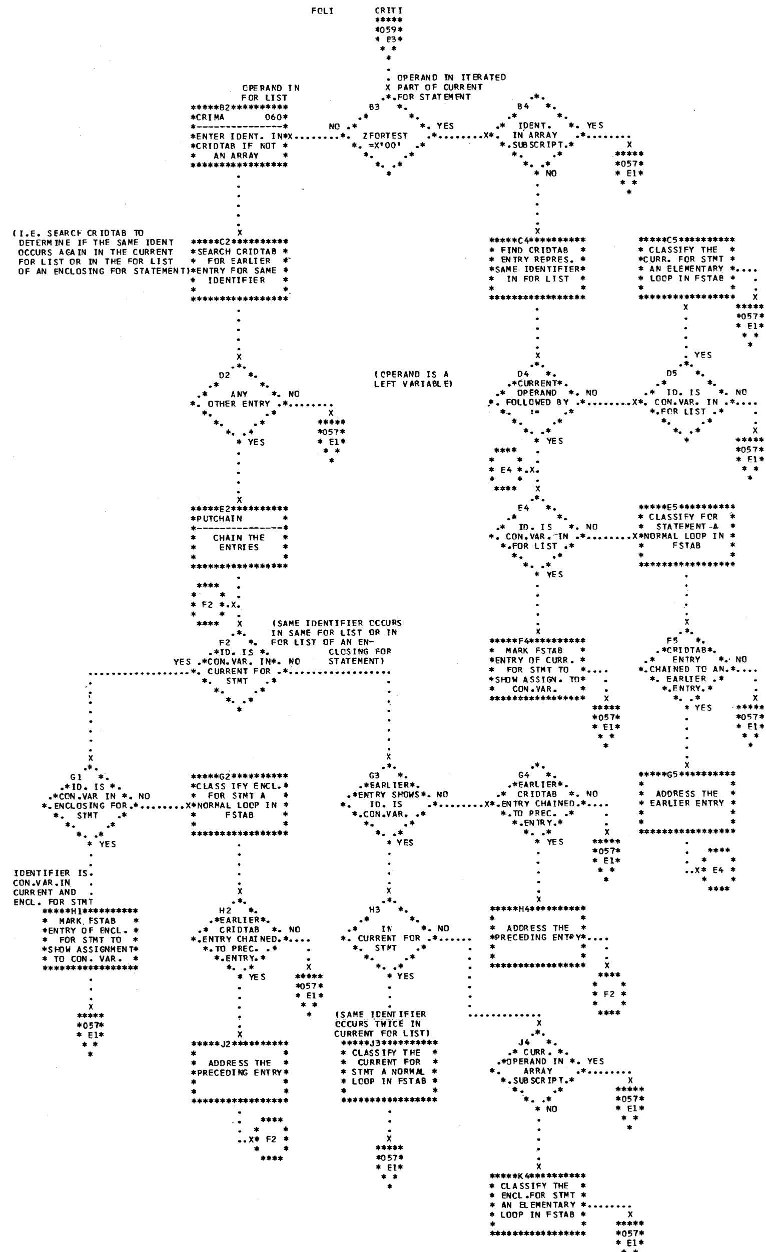


Chart 060: Scan III Phase - IEX30 CRIMA

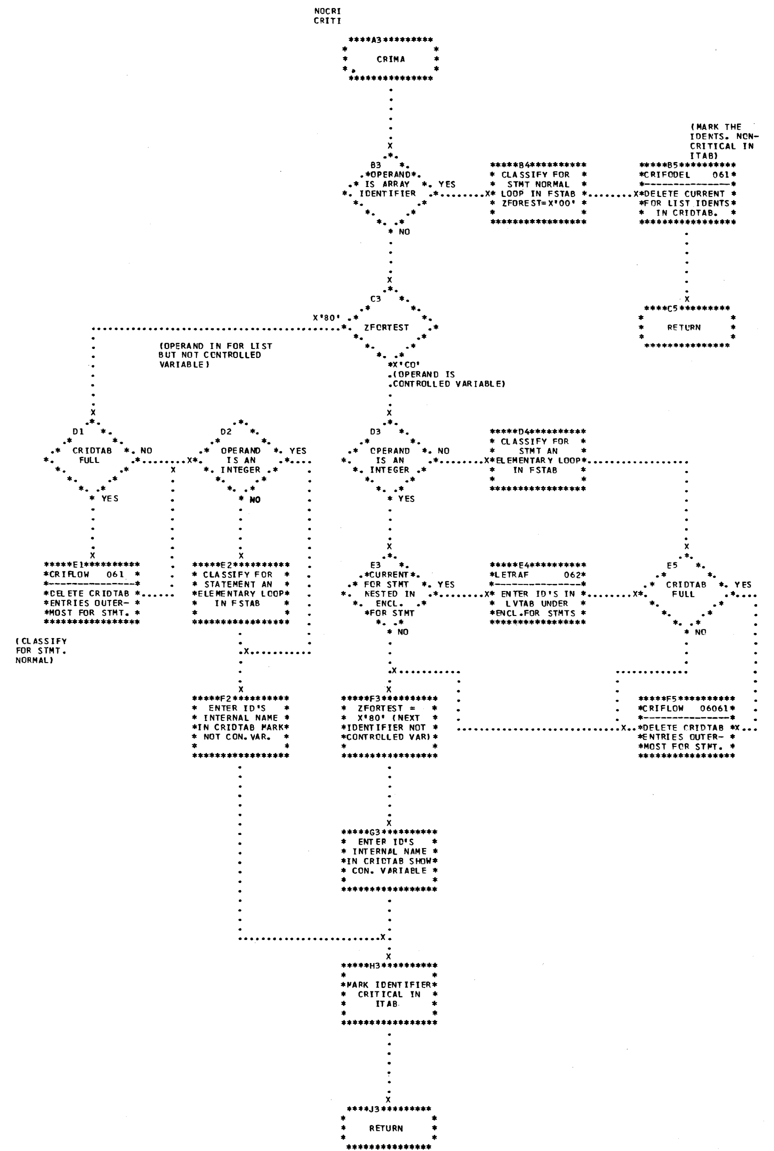


Chart 061: Scan III Phase - IEX30 CRIFODEL, CRIFLOW and DELCRIV — Microfiche IEX30-1

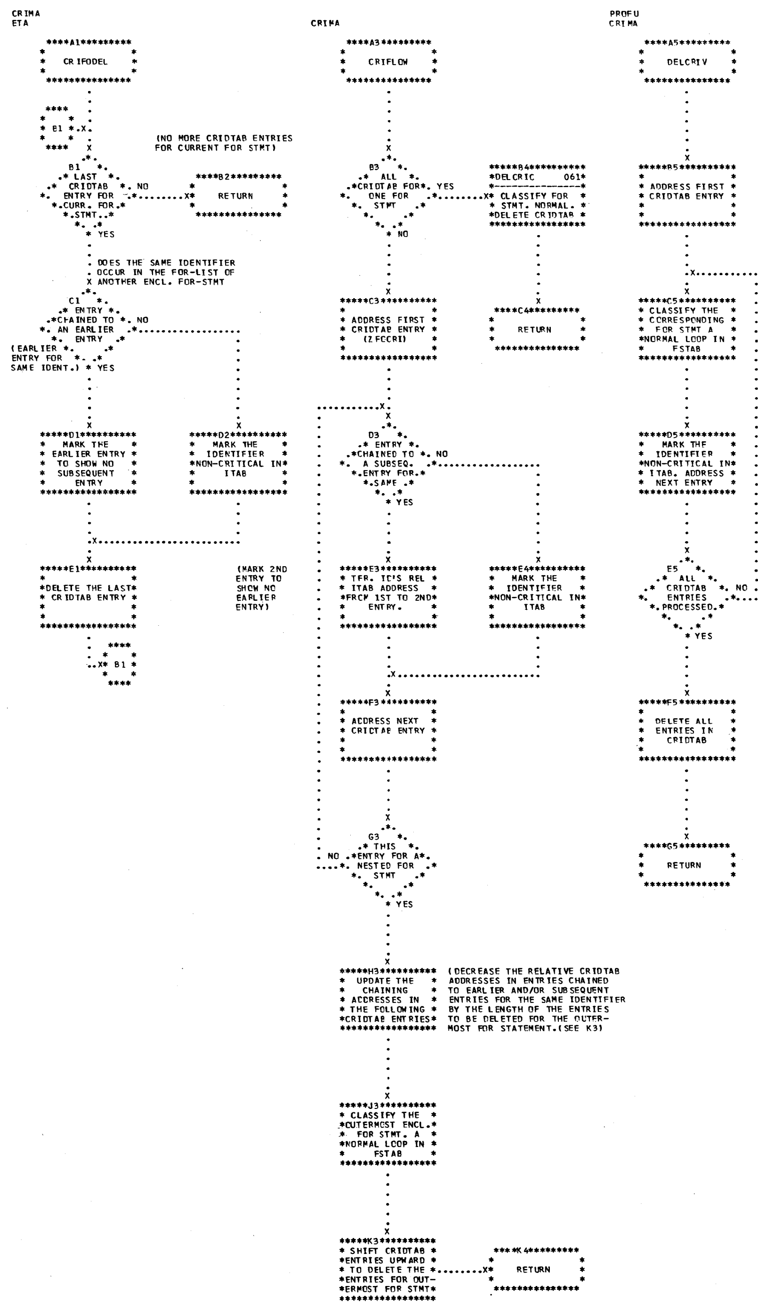


Chart 062: Scan III Phase - IEX30 LETRAF and WRITE — Microfiche (LETRAF: IEX30-1 and WRITE: OEX30-2)

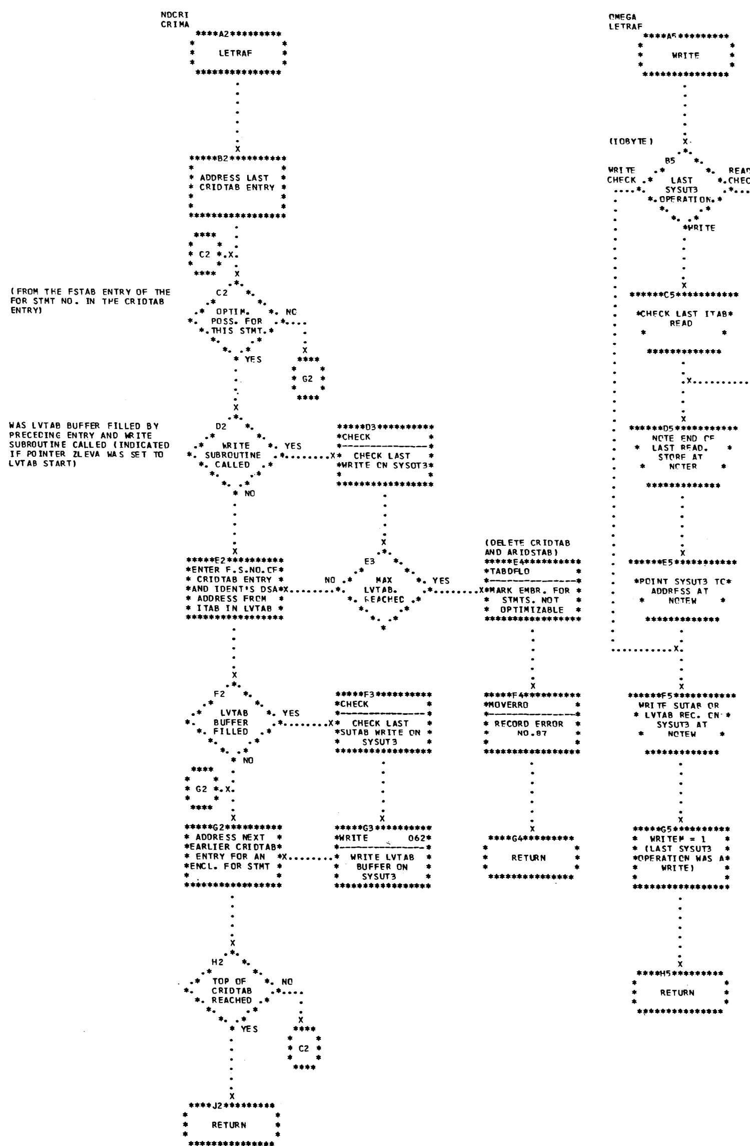


Chart 063: Scan III Phase - IEX30 — Microfiche IEX30-2

Overall Flow and Number Handling

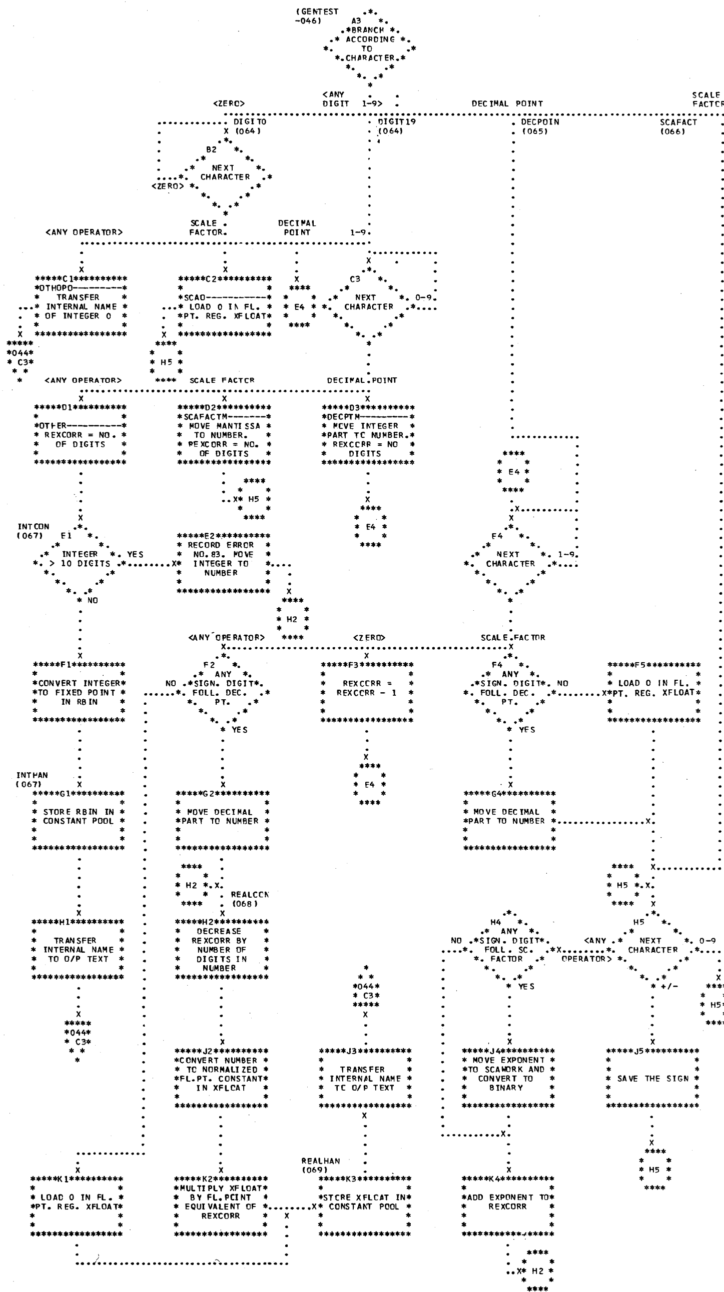


Chart 064: Scan III Phase - IEX30 — Microfiche IEX30-2

DIGIT0 and DIGIT19

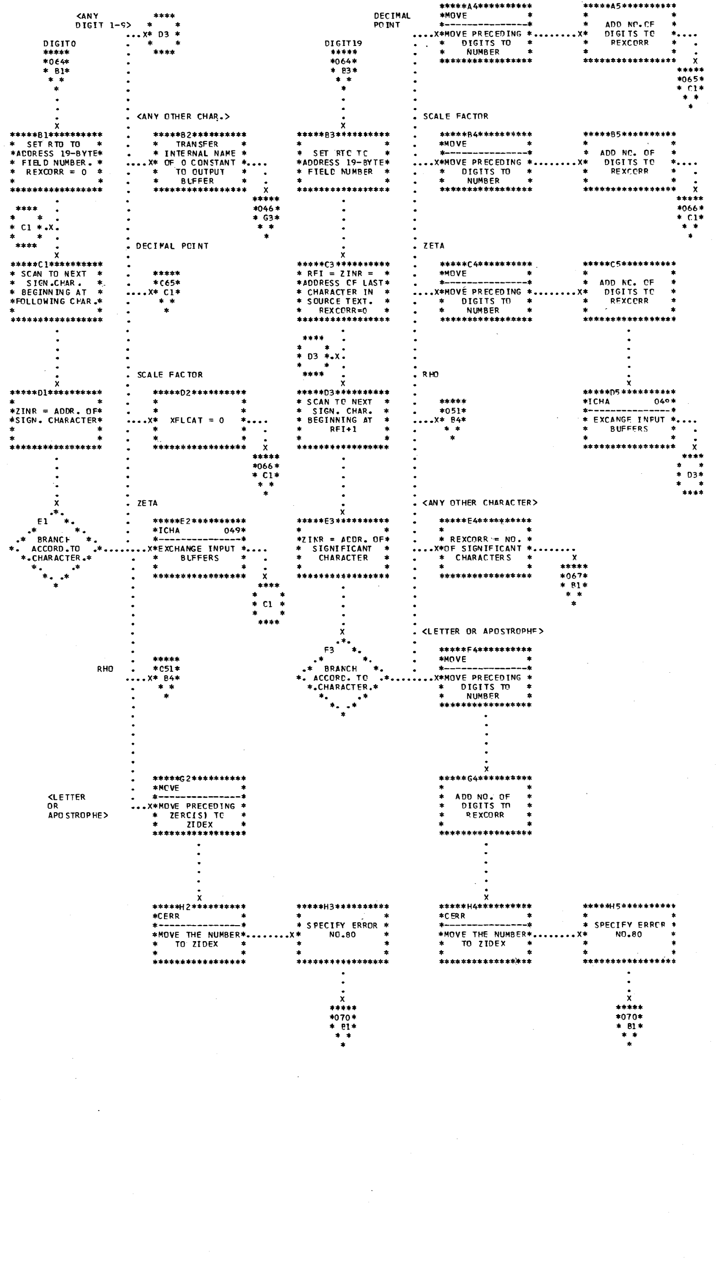


Chart 065: Scan III Phase - IEX30 Microfiche IEX30-2

DECPOIN

(E.G. 0.0055,REXCORR IS DECREMENTED ONCE FOR EACH 0 FOLLOWING THE DECIMAL POINT)

DECPOIN 065 B1

- B1: ADDRESS 19-BYTE FIELD NUMBER. REXCORR=0
- 065 C1
- C1: RFI = ADDR. OF NEXT SOURCE CHARACTER
- D1
- D1: SCAN TO NEXT SIGNIFICANT CHARACTER
- E1: ZINR = ADDR. OF SIGNIFICANT CHARACTER
- F1: BRANCH ACCORDING TO CHARACTER

<ZERO>
- A2: ZINR:RFI — ZINR =RFI → A3; ZINR >RFI → D1
- A3: ANY SIGN. DIGIT AT NUMBER — NO → A4; YES → D1
- A4: REXCORR = REXCORR-1 → C1

ZETA
- C2: MOVE — MOVE PRECEDING DIGITS TO NUMBER
- C3: ICHA 049 — EXCHANGE INPUT BUFFERS
- C4: RFI =ZINR =ADDRESS OF SIGNIFICANT CHARACTER → D1

<ANY LETTER> DECIMAL POINT OR APOSTROPHE
- D2: MOVE — MOVE PREC.DIGS. TO NUMBER IF NO SYNTAX ERROR
- D3: CERR — MOVE THE NUMBER TO ZIDEX
- D4: SPECIFY ERROR NO.8C → 070 B1

SCALE FACTOR
- E2: MOVE — MOVE PREC.DIGS. TO NUMBER IF NO SYNTAX ERROR
- E3: ANY SIGN. DIGITS AT NUMBER — YES → 066 C1; NO → F3
- F3: XFLOAT = 0 → 066 C1

<ANY OTHER OPERATOR./
- G2: MOVE — MOVE PRECEDING DIGITS TO NUMBER IF NO SYNTAX*ERROR
- G3: ANY SIGN. DIGITS AT NUMBER — YES → 068 B1; NO → H3
- H3: XFLOAT =0 → 069 B1

Chart 066: Scan III Phase - IEX30 Microfiche IEX30-2

SCAFACT

SCAFACT 066 B1

- B1: SET RTO TO 19-BYTE FIELD NUMBER. XFLOAT=1. REXCORR=0
- 066 C1
- C1: SAVE REXCORR AND RTO
- D1: SET RTO TO ADDRESS 9-BYTE FIELD SCAWORK. SF=1
- E1
- E1: RFI = ADDRESS OF NEXT CHARACTER IN SOURCE TEXT
- F1
- F1: SCAN TO NEXT SIGNIFICANT CHARACTER IN SOURCE TEXT
- G1: ZINR = ADDRESS OF SIGNIFICANT CHARACTER
- H1: BRANCH ACCORD. TO CHARACTER

<ANY DIGIT 1-9>
- A2: SF19 = 1 → F1

ZERO.
- B2: ANY SIGN. DIGIT. FOLLOWING SCA.FAC. — NO → B3; YES (SF19=1) → F1
- B3: SFLO = 1 → F1

ZETA.
- D2: MOVE — MOVE PRECEDING DIGITS TO SCAWORK
- D3: ICHA 049 — EXCANGE INPUT BUFFERS
- D4: RFI = ADDRESS OF FIRST BYTE → F1

<LETTER>. DEC.PT. SC.FACT. APOS.
- E2
- E2: MOVE — MOVE PRECEDING DIGITS TO SCAWORK
- E3: ZTCSCA = LENGTH OF EXPONENT
- E4: CERR — MOVE THE NUMBER TO ZIDEX
- E5: SPECIFY ERROR NO.80 → 070 B1

+/-.
- F2: LEADING SIGN — YES (SFSIGN=1 SELO=1 SF19=1) → F3; NO → G2
- F3: SCALE +1 = SIGN
- F4: SFSIGN = 1 → E1

<ANY OTHER CHAR.>.
- G2: ANY DIGIT FOLLOWS SC.FACT. (SF19=1 SFLO=1) — YES → G3; NO → E2
- G3: ANY SIGN. DIGIT FOLLOWING SCA.FACT. (SF19) — YES → G4; NO → J5
- G4: MOVE — MOVE PRECEDING DIGITS TO SCAWORK
- G5: CONVERT EXPONENT IN SCAWORK TO BINARY IN ZPACK
- H5: ADD EXPONENT TO REXCORR
- J5: ZTOSCA = LENGTH OF EXPONENT. RTO = ADDR. OF NUMBER → 068 B1

Chart 067: Scan III Phase - IEX30 — Microfiche IEX30-2

INTCON and INTHAN

INTCON

067 B1

B1: INTEGER > MAX — YES → B2

B2: MOVERRO — RECORD ERROR NO. 83 → 046 G3

NO → C1: CONVERT CONSTANT AT RFI TO BINARY IN RBIN → A3

C2: MOVE — MOVE CONSTANT TO NUMBER → 046 G3

067 INTHAN → A3: TRANSFER 2-BYTE INTEGER CONSTANT CHARACTERISTIC → A3

INTHANB

B3: CONSTANT > 15 — NO → B4: TRANSFER 3-BYTE ADDRESS OF CONSTANT IN CONSTANT POOL NO. 0. → 046 G3

YES → C3: SYNTAX CHECK MODE — YES → K4

NO → INTHAN2 / INTHAN6 / INTHAN4

D3: COMPARE CONSTANT IN RBIN WITH ENTRIES IN CONSTANT POOL

INTHAN3

E3: SAME CONSTANT FOUND — YES → K4

INTHAN9

F3: CONSTANT POOL FILLED — YES → INTHAN8

F4: CONSTANT POOL NO = 25 — YES → INTHAN11

F5: MOVERRO — RECORD ERROR NO. 86

G5: SWITCH TO SYNTAX CHECKING

F3 NO → G3: LAST 56-BYTE CON. POOL REC. FILLED — NO

YES → H3: TXTTRAF — GENERATE TXT RECORD OF LAST CON. POOL RECORD

F4 NO → G4: TXTTRAF — GENERATE TXT RECORD OF LAST CON. POOL RECORD

H4: CPOLEX — ASSIGN NEW CONSTANT POOL NUMBER

INTHAN5

J3: INCREMENT CONSTANT POOL POINTERS

J4: RESET CONSTANT POOL POINTERS

INTHAN7

K3: STORE CONSTANT IN RBIN IN NEXT CON. POOL ENTRY

067 K4 — INTHAN1

K4: TRANSFER CONSTANT POOL NO. AND DISPLACEMENT → 046 G3

Chart 068: Scan III Phase - IEX30 — Microfiche IEX30-2

REALCON

REALCON

068 B1

B1: ANY MANTISSA DIGITS AT NUMBER — NO → REALCON7

REALCON7

B2: XFLOAT = 0 — NO → A4; YES → 069 B1

YES → C1: RCNT = NO. OF DGTS AT NUMBER. XFLOAT = 0. REXCORR= REXCORR - RCNT.

SHIFT DEC. POINT RIGHT. DECREASE EXPONENT IN REXCORR BY NO. OF DGTS.

D1: NO. OF DIGITS AT NUMBER > 9 — YES → REALCON1

REALCON1

D2: RCNT = RCNT-10

E2: RCNT GRTR OR EQUAL 9 — NO; YES →

F2: RCNT = RCNT - 1 REXCORR = REXCORR + 1

G2: PACK FIRST (RCNT) DIGITS OF MANTISSA AT ZPACK

H2: CONVERT ZPACK TO BINARY AND STORE AT ZFLOFIEL + 4

J2: XFLOAT = XFLOFIEL (NORMALIZED) XFLOAT = XFLOAT*10**9

K2: PACK LAST 9 DIGITS OF MANTISSA AT ZPACK → F1

D1 NO → E1: PACK MANTISSA AT NUMBER IN ZPACK WITH +SIGN.

REALCON2

F1: CONVERT ZPACK TO BINARY IN WORKR AND STORE AT ZFLOFIEL +4

G1: XFLOAT = XFLOAT+ZFLOFIEL (NORMALIZED)

H1: MANTISSA DIGITS > 7 (OR 18) — NO → A4

IS MANTISSA LONGER THAN RANGE FOR PRECISION SPECIFIED (SHORT OR LONG).

YES → J1: CERR — MOVE MANTISSA TO ZIDEX

K1: MOVERRO — RECORD ERROR NO. 84

REALCON3

A4: REXCORR = 0 — YES → 069 B1

IS EXPONENT = 0

NO → B4: ADDRESS POSITIVE POWER OF TEN TABLE ZEXTABP

C4: REXCORR < 0 — YES → C5: ADDRESS NEGATIVE POWER OF TEN TABLE ZEXTABN

IS EXPONENT NEGATIVE

REALCON6

D4: |REXCORR| GRTR OR EQUAL 64 — YES → REALCON5

REALCON5

D5: XFLOAT = XFLOAT *10**+/-64

(10**+/-64 FROM POWER OF TENTABLE ZEXTABP OR ZEXTABN)

E5: XFLOAT = 0 — YES → J5

NO → F5: REXCORR = REXCORR - 64 → D4

NO → E4: REXCORR HAS ANY DIGITS IN RANGE 1-7 — NO; YES →

X = VALUE OF REXCORR DIGITS IN RANGE 1-7 (10**+/-X) FROM POWER OF TEN TABLE ZEXTABP OR ZEXTABN.

F4: XFLOAT = XFLOAT*10**+/-X

REALCON8

G4: REXCORR HAS ANY DIGITS IN RANGE >7 — NO; YES →

Y = VALUE OF REXCORR DIGITS IN RANGE >7, WHERE Y=8,16,24,...64. (10**+/-Y) FROM POWER OF TEN TABLE ZEXTABP OR ZEXTABN.

H4: XFLOAT = XFLOAT*10**+/-Y

REALCON9

J4: XFLOAT = 0 — YES → REALERR1

EXPONENT UNDERFLOW

NO → 069 B1

REALERR1

J5: MOVERRO — RECORD ERROR NO. 82

K5: SWITCH TO SYNTAX CHECKING → 069 B1

Chart 069: Scan III Phase - IEX30 — Microfiche IEX30-2

REALHAN

REALHAN
069 B1

B1: TRANSFER TWO-BYTE REAL CONSTANT CHARACTERISTIC

C1: CONSTANT IN FOR LIST — YES → C2; NO → D1

C2: CHANGE FSTAB ENTRY TC SHOW COUNTING LOOP IMPOSSIBLE → D1

REALHAN1
D1: PRECISION — SHORT → D2; LONG → E1

D2: ROUND OFF CONSTANT IN XFLOAT AND LOAD IN RBIN → *067* A3

REALH
E1: SYNTAX CHECK MODE — YES → *067* K4; NO → F1

F1: NEXT TEXT RECORD FILLED — YES → F2; NO → G1

REALH7
F2: TXTTRAF — GENERATE TXT RECORD OF LAST CON.PCOL RECCRD → G1

G1: COMPARE CONSTANT IN XFLOAT WITH ENTRIES IN CON.POOL

H1: SAME CONSTANT FOUND — YES → C4; NO → J1

REALH11
H2: MOVERRO — RECORD ERROR NO.86 → H3; → J2

H3: SWITCH TO SYNTAX CHECKING → C4

J1: END OF CONSTANT POOL REACHED — YES → J2; NO → B4

REALH10
J2: CONSTANT POOL NO=255 — YES → H2; NO → K2

K2: TXTTRAF — GENERATE TEXT RECORD OF LCAD CON.PCOL RECORD → K3

K3: CPOLEX — ASSIGN NEW CCNSTANT POOL NC. → C4

B4: STORE CONSTANT IN XFLOAT IN NEXT CONSTANT POOL ENTRY

C4: TRANSFER CONSTANT POOL NO.AND DISPLACEMENT. → *046* G3

Chart 070: Scan III Phase - IEX30 — Microfiche IEX30-2

INCOROP

LETTER
SWILA
DIGITO
DIGIT19
DECPOIN
SCAFACT

INCOROP
070 B1

B1: SWITCH TO SYNTAX CHECKING

C1: SEARCH TO END OF OPERAND (CONSTANT OR ICENTIFIER)

D1: OPERAND FOLLOWED BY — <ANY OPERATOR> → D3; <CHARACTER STRING> → F1

INCCROP2
D3: MOVERRC — RECORD ERRCR SPECIFIED

LETTER1
E3: TRANSFER ALL-PURPOSE INTERNAL NAME → *046* G3

INCOROP3
F1: TRANSFER SIX DECIMAL POINTS TO REPL.STRING INTERN.NAME → C1

Chart 071: **Diagnostic Output - IEX31** **Microfiches IEX31-1 and IEX31M**

Overall Flow - Error Message Editing Routine

A1: IEX31

B1: LOAD ADDRESS OF FIELD CONT. MESSAGE POOL AND ADDR. TABLE ADDRESSES

C1: IEX60000 — ERROR MESSAGE EDITING ROUTINE

D1 (connector)

D1: XCTL TO IEX40

FOR DETAIL OF IEX60000 SEE LOAD MODULE IEX21-FLOWCHARTS 041-043

A3: IEX 60000

B3: SYSPRINT OPERATIVE — NO → B4; YES → C3

B4: ADRESS LAST ERROR POOL ENTRY

C3: ADDRESS ERROR POOL MESS.POOL AND ADDR.TABLE

D3: ERROR POOL EMPTY — YES → D4; NO → E3

D4: RETURN

E3: GET ERROR NUMBER AND FIND MESSAGE POOL ENTRY

F3: ASSEMBLE MESSAGE TEXT IN WORK AREA.

G3: SYSPRINT OPERATIVE — NO → H4; YES → H3

H3: WRITE MESSAGE ON SYSPRINT

H4: WRITE MESSAGE ON CONSOLE TYPEWRITER

J3: ALL ERROR POOL ENTRIES PROCESSED — NO → E3; YES → K3

K3: TERMINATING ERROR — YES → K2; NO → D1

K2: XCTL TO IEX51002

```
                                    ****A3*********
                                    *             *
                                    *    IEX40    *
                                    *             *
                                    ***************
                                           .
                                           X
SUTAB = SUBSCRIPT TABLE            *****B3**********
LVTAB = LEFT VARIABLE TABLE        *               *
OPTAB = OPTIMIZATION TABLE         *START-------073*
                                   *  GETMAIN FOR  *
                                   *SUTAB AND LVTAB*
                                   *               *
                                   *****************
                                           .
                                           X
                                   *****C3**********
                                   *READ      074  *
                                   *---------------*
                                   *READ SUTAB FROM*
                                   *    SYSUT3     *
                                   *               *
                                   *****************
                                           .
                                           X
          (FLAG FOR DELETION       *****D3**********
          ANY ENTRIES FOR          *SCAN SUTAB--073*
          SUBSCRIPTS IN            *  DELETE ANY   *
          NON-OPTIMIZABLE          * SUTAB ENTRIES *
          FOR STATEMENTS)          *FOR NONOPTIMI- *
                                   *-ZABLE FOR STMTS
                                   *****************
                                           .
                           SORTSU          X
                                   *****E3**********
                                   *SORT      074  *
                                   *---------------*
                                   *  SORT SUTAB   *
                                   *  ENTRIES BY   *
                                   *ASCEND.F.S.NO. *
                                   *****************
                                           .
                           SORTLE          X
                                   *****F3**********
                                   *READ      074  *
                                   *---------------*
                                   *READ LVTAB FROM*
                                   *    SYSUT3     *
                                   *               *
                                   *****************
                                           .
                                           X
                                   *****G3**********
                                   *SORT      074  *
                                   *---------------*
                                   *  SORT LVTAB   *
                                   *  ENTRIES BY   *
                                   *ASCEND.F.S.NO. *
                                   *****************
                                           .
                                           X
                                   *****H3**********
                                   *OPTAB-------075*
                                   *   CONSTRUCT   *
                                   * OPTIMIZATION  *
                                   *     TABLE     *
                                   *               *
                                   *****************
                                           .
                                           X
IEX400C1=CONTROL SECTION           *****J3**********
FOR INITIALIZATION OF              *TERMIN------076*
COMPILATION PHASE                  * CLOSE SYSUT3  *
(SEE CHART 078)                    *  AND RELEASE  *
                                   * MAIN STORAGE  *
                                   *               *
                                   *****************
                                           .
                                           X
                                    ****K3*********
                                    *  BRANCH TO  *
                                    *  IEX40001   *
                                    *             *
                                    ***************
```

Chart 073: Subscript Handling Phase - IEX40 Microfiche IEX40-1

Initialization and Search SUTAB

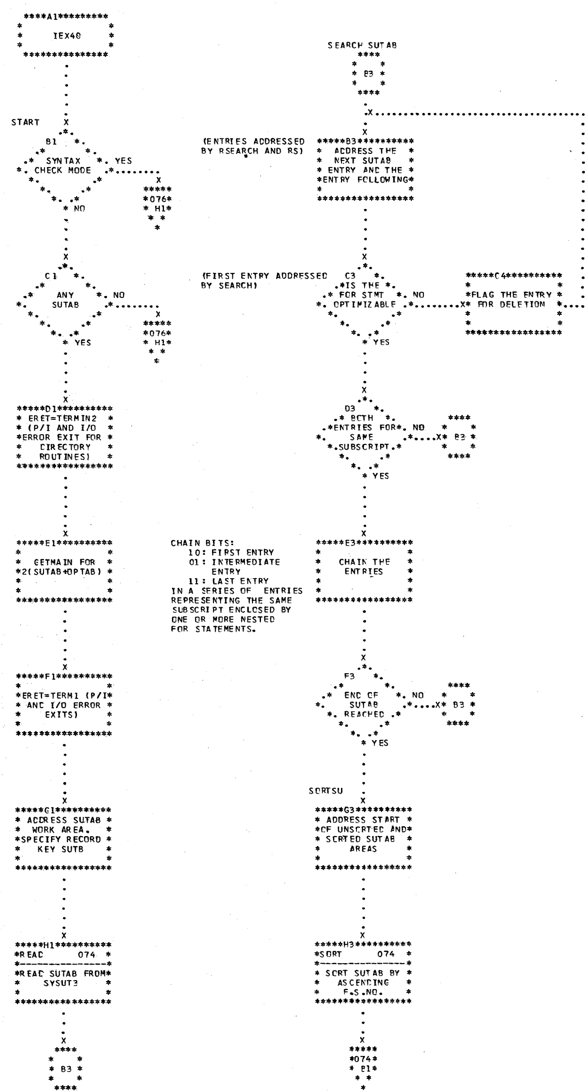


Chart 074: Subscript Handling Phase - IEX40 Microfiche IEX40-1

SORTLE, READ and SORT

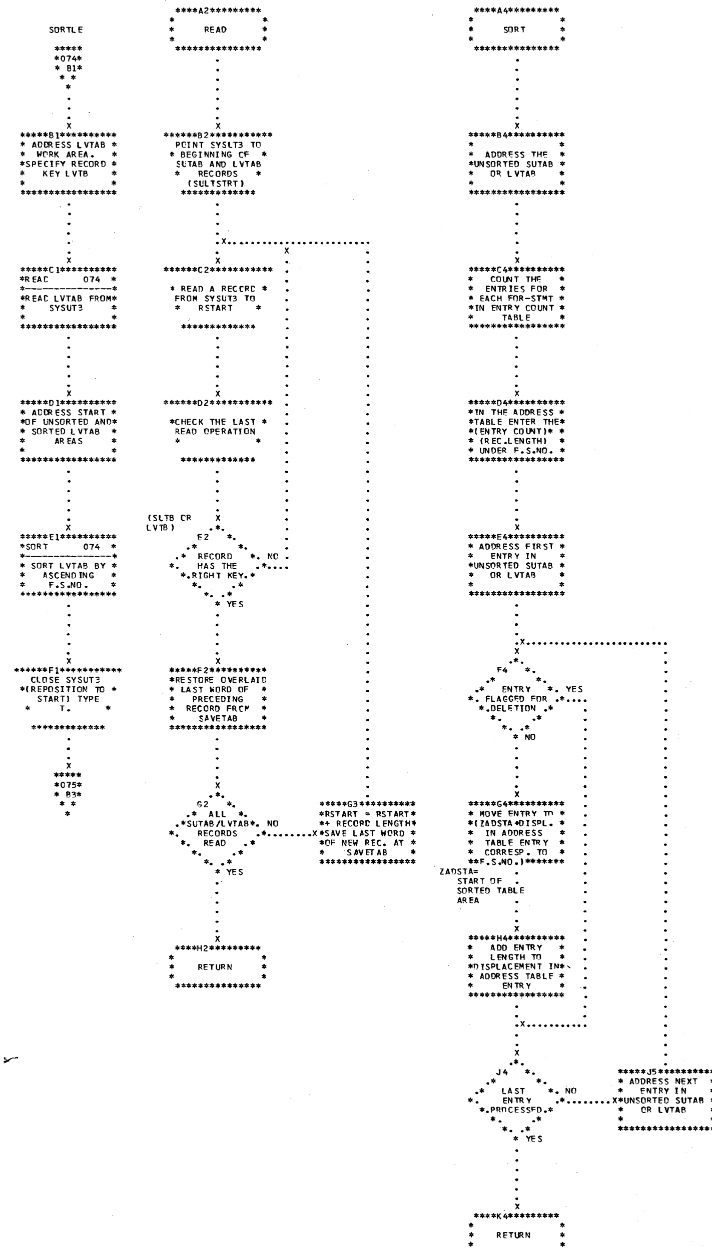


Chart 075: Subscript Handling Phase - IEX40 Microfiche IEX40-1

OPTAB

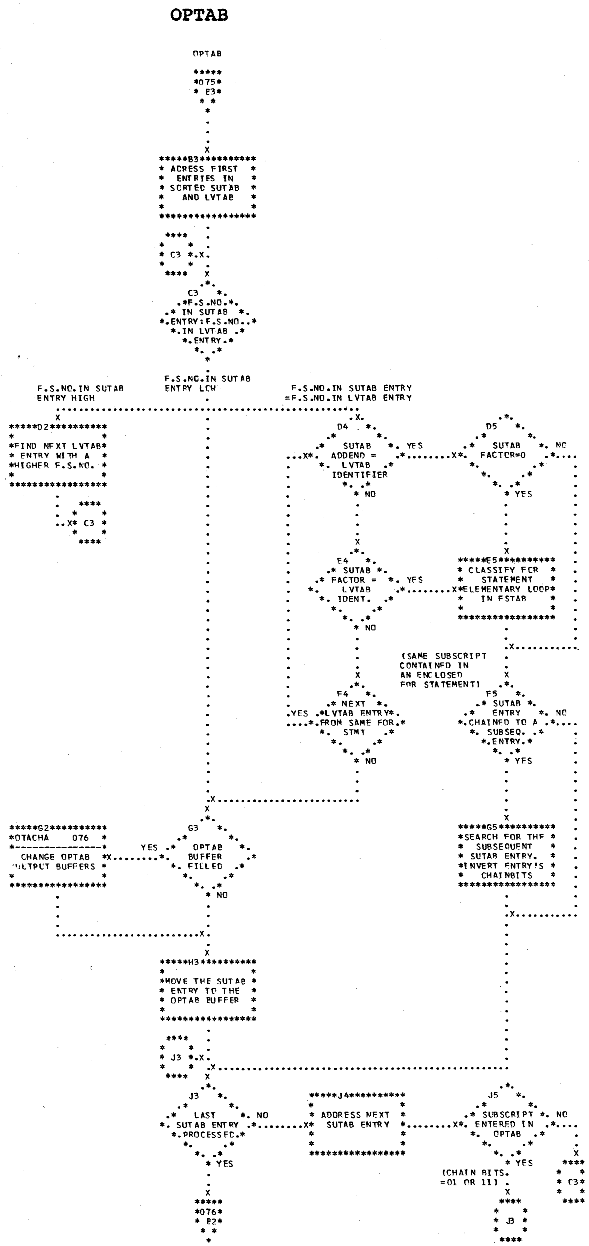


Chart 076: Subscript Handling Phase - IEX40 Microfiche IEX40-1

TERMIN and OTACHA

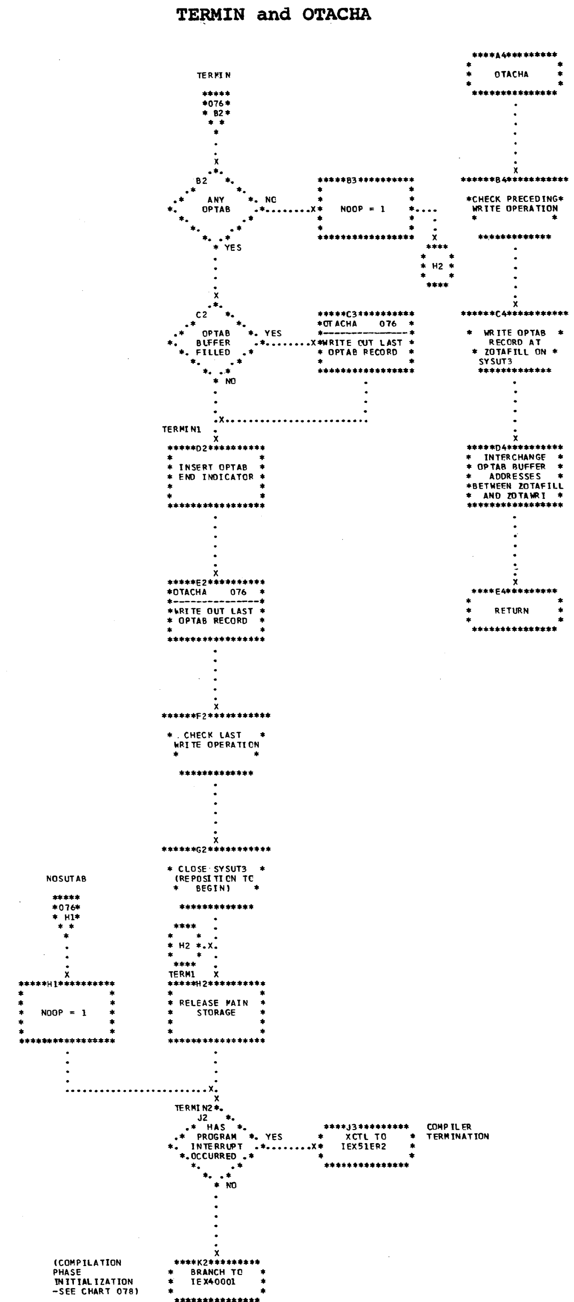


Chart 077: Compilation Phase - IEX50 Overall Flow

Microfiches IEX50-1 to IEX50-7

```
                                                                  *****A4**********
                       ****A2*********        ****A3*********     *               *
IEX40001               *             *        *             *     *CP0---------080*
IS A CONTROL           *   IEX40001  *        *    IEX50    *  ...X*   COMPILER    *....
SECTION OF             *             *        *             *  .  * PROGRAM NO. 0 *   .
LOAD MODULE IEX40.     ***************        ***************  .  *               *   .
INITIALIZATION OF             .                      .         .  *****************   X
THE COMPILATION               .                      .         .                     ****
PHASE DESCRIBED               .                      .         .                    *    *
IN CHAPTER 8.                 .                      .         .                    * D3 *
                              .                      .         .                    *    *
                              .                      .         .                     ****
                              X                      X         .
                      *****B2**********      *****B3**********  .  *****B4**********
                      *------------078*      *   LOAD BASE   *  .  *               *
                      *INITIALIZATION *      * REGISTERS TO  *  .  *CP1---------080*
                      *OF COMPILATION *      * ADDRESS SNOT  * ...X*   COMPILER    *....
                      *     PHASE     *      *AND PROG. CON. *  .  *  PROGRAM NO1  *   .
                      *               *      *    MATRIX     *  .  *               *   .
                      *****************      *****************  .  *****************   X
                              .                      .          .                     ****
                              .                      .          .                    *    *
                              .                      .          .                    * D3 *
                              .                      .          .                    *    *
                              .                      .          .                     ****
                              X                      X          .
                       ****C2*********       *****C3**********  .  *****C4**********
                       *             *       *SPECIFY OPERR1,*  .  *               *         ****C5*********
                       * XCTL TO IEX50 *     *  JB3, NX4 AS  *  .  *CP3---------080*         *             *
                       *             *       *EXITS FOR DIR. * ...X*   COMPILER    *........X* XCTL TO IEX51 *
                       ***************       *ROUTINES PIROUT*  .  * PROGRAM NO. 3 *         *             *
                                             *EODUT2 & EODUT3*  .  *               *         ***************
                                             *****************  .  *****************
                                                     .          .
                                              ****   .          .
                                             *    *  .          .
                                             * D3 *.X.          .
                                             *    *  .          .
                                              ****   .          .
                                                     X          .
                                             *****D3**********  .  *****D4**********
                                             *               *  .  *               *
                                             *SNOT--------079*  .  *CP4---------C81*
                                             * SCAN TO NEXT  * ...X*   COMPILER    *....
                                             *   OPERATOR    *  .  * PROGRAM NO. 4 *   .
                                             *               *  .  *               *   .
                                             *****************  .  *****************   X
                                                     .          .                     ****
                                                     .          .          0         *    *
                                                     .          .          0         * D3 *
                                                     .          .          0         *    *
                                                     .          .          0          ****
                                                     X          .          0
                                         COMP       .*.         .          0
                                         (079)   E3    *.       .          0
                                               .*BRANCH *.      .          0   (SEE DECISION MATRICES IN
                                             .*ACCORD. TO *.    .          0    APPENDIX V-A, V-B AND V-C
                                            *.SOURCE/STACK .*.....         0    FOR COMPILER PROGRAMS ENTERED
                                              *.OPERATORS.*     .          0    FOR EACH PAIR OF SOURCE AND
                                                *.     .*       .          0    STACK OPERATORS.
                                                  *. .*         .          0    SEE ALSO FIGURE 52.)
                                                    *           .          0
                                                                .          0
                                                                .          0
                                                                .          0
                                                                .          0
                                                                .          0
                                                                .
                                                                .  *****F4**********
                                                                .  *               *
                                                                .  *CP87--------119*
                                                               ...X*   COMPILER    *....
                                                                   *PROGRAM NR. 87 *   .
                                                                   *               *   .
                                                                   *****************   X
                                                                                      ****
                                                                                     *    *
                                                                                     * D3 *
                                                                                     *    *
                                                                                      ****
```

Chart 078: Compilation Phase - IEX50 Initialization (in IEX40)

Microfiches IEX50-1 to IEX50-7 (Initialization: IEX40-1)

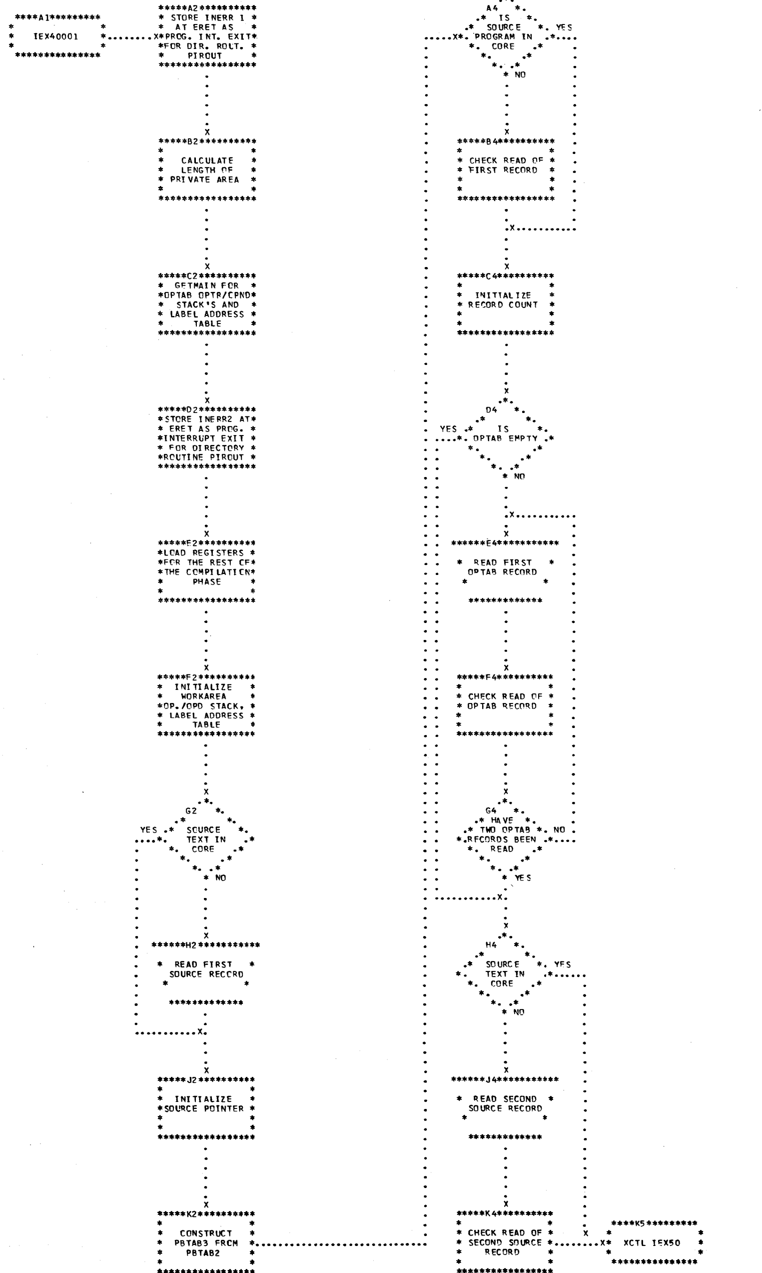


Chart 079: Compilation Phase - IEX50 SNOT, SNOTSP and COMP

Microfiche IEX50-1

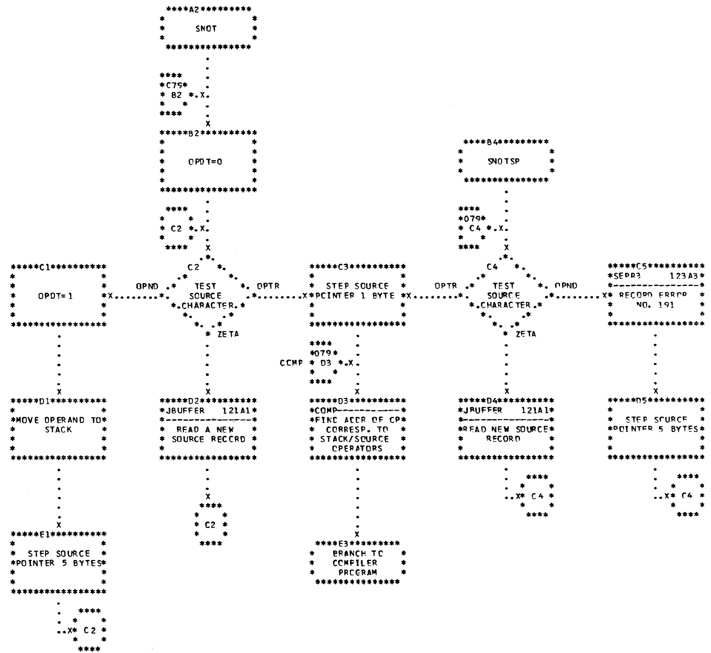


Chart 080: Compilation Phase - IEX50 CP0, CP1, CP3 and CPERR1

Microfiche IEX50-3 (CPERR1: IEX50-1)

- A1: CP0
- B1: SOURCE OPERATOR = BETA (NO / YES)
- C1: CLEARRG — GEN. CODE TO SAVE REGISTERS
- D1: GEN. CALL TO PROLOG ROUT. IN FSA
- E1: STACK SOURCE OPERATOR
- F1: SOURCE OPERATOR = BEGIN (YES → 079 B2 SNOT; NO)
- G1: PBNFDL — UPDATE PBN. ADDRESS NEW DSA (→ 079 B2 SNOT)

- A3: CP1
- B3: STORE PROGRAM POINTER IN LAT-ENTRY INDICATED IN LABEL OPERAND
- C3: CLEARRG — GEN. CODE TO SAVE REGISTERS
- D3: STACK OPERATOR BETA, PI OR PHI (NO / YES)
- E3: STACK SEMICOLON
- F3: RELEASE OPERAND
- SNOT → 079 B2

PRPOINT = DISPLACEMENT OF NEXT INSTRUCTION IN OBJECT MODULE

- A5: CP3
- B5: PROCEDURE COMPILATION (YES → CPEND; NO)
- C5: GENTXT4 122D1 — BRANCH TO TERMINATION ROUTINE IN FSA
- CPEND
- F5: XCTL IEX51000
- 080 G5 CPERR1
- G5: XCTL IEX51ER1

Chart 081: Compilation Phase - IEX50 CP4

Microfiche IEX50-3

- A1: CP4
- A2: STACK SOURCE OPERATOR (→ 081 A2)
- A3: SOURCE OPERATOR = ARRAY (YES / NO → B1)
- A4: SWITCH TO STATEMENT CONTEXT
- B1: RESERVE 2 ENTRIES IN OPERAND STACK
- B2: PROCEDURE COMPILATION FIRST BLOCK (NO / YES)
- B3: GENTXT6 122E1 — BRANCH TO DECLARATION END
- B4: SOURCE OPERATOR = SWITCH (YES → 079 B2 SNOT; NO)
- C1: PBNFDL — UPDATE PBN. ADDRESS NEW DSA
- C2: AMAA2 — GNOPDOPR 081D3 — SCAN SOURCE INPUT FOR PROCED. IDENTIFIER
- C3: SOURCE OPERATOR LEFT ((NO / YES)
- C4: AMBE1 — OPDT = 0 (→ C4)
- C5: OPERATOR = DELTA (YES → 079 C3 COMP; NO)
- D1: GNOPDOPR 081D3 — SCAN SOURCE INPUT FOR NEXT PARAMETER
- D3: GNOPDOPR
- D4: MOVE SOURCE POINTER 5 BYTES
- D5: GNOPDOPR 081D3 — SCAN SOURCE INPUT FOR NEXT OPERATOR
- E1: GENTXT2 122C1 — CHARACTERISTIC OF PARAMETER
- E2: JBUFFER 121A1 — CHANGE SOURCE INPUT BUFFER
- E3: SOURCE + 1 (ZETA / OPND / CPTR)
- E4: SAVE OPERAND
- F1: BUILD TABLE WITH INFORMATION ON PARAMETERS
- F3: MOVE SOURCE POINTER 1 BYTE
- G1: SOURCE OPERATOR = COMMA (YES / NO)
- G3: RETURN
- H1: PARAMETER CALLED BY VALUE (YES / NO)
- H2: GENTXTS 122F1 — CALL ACTUAL PARAMETER
- J1: AMBB1 — LAST PARAMETER IN TABLE (NO / YES → C4)
- J2: IS PARAMETER LABEL (YES / NO)
- J3: GENTXTS 122F1 — STORE ADR AND GCSA IN PARAMETER ENTRY
- K2: AMBC4 — GENTXTS 122F1 — CALL VALUE CALL ROUTINE IN FSA

Chart 082: Compilation Phase - IEX50 CP6, CP7 and CP8 — Microfiche IEX50-3 (CP6: IEX50-2)

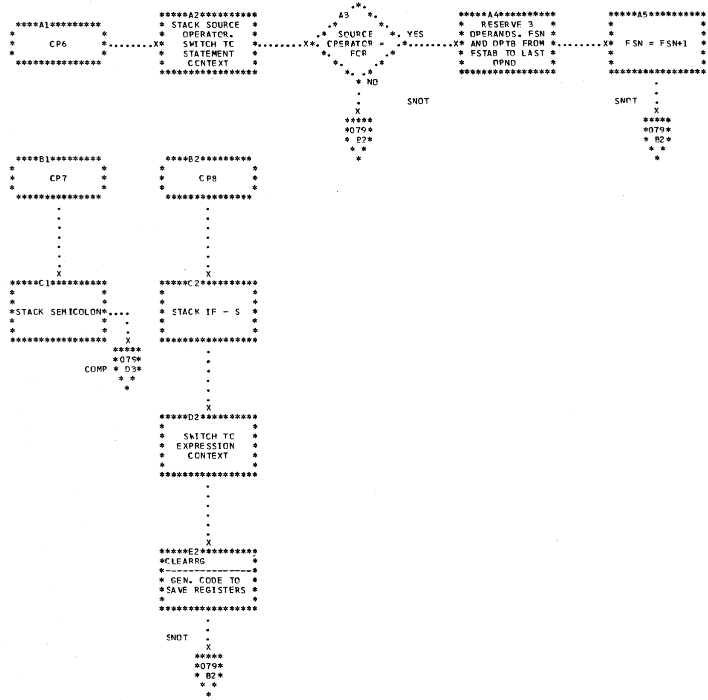


Chart 083: Compilation Phase - IEX50 CP12 and CP16 — Microfiche IEX50-4 (CP12), Microfiche IEX50-3 (CP16)

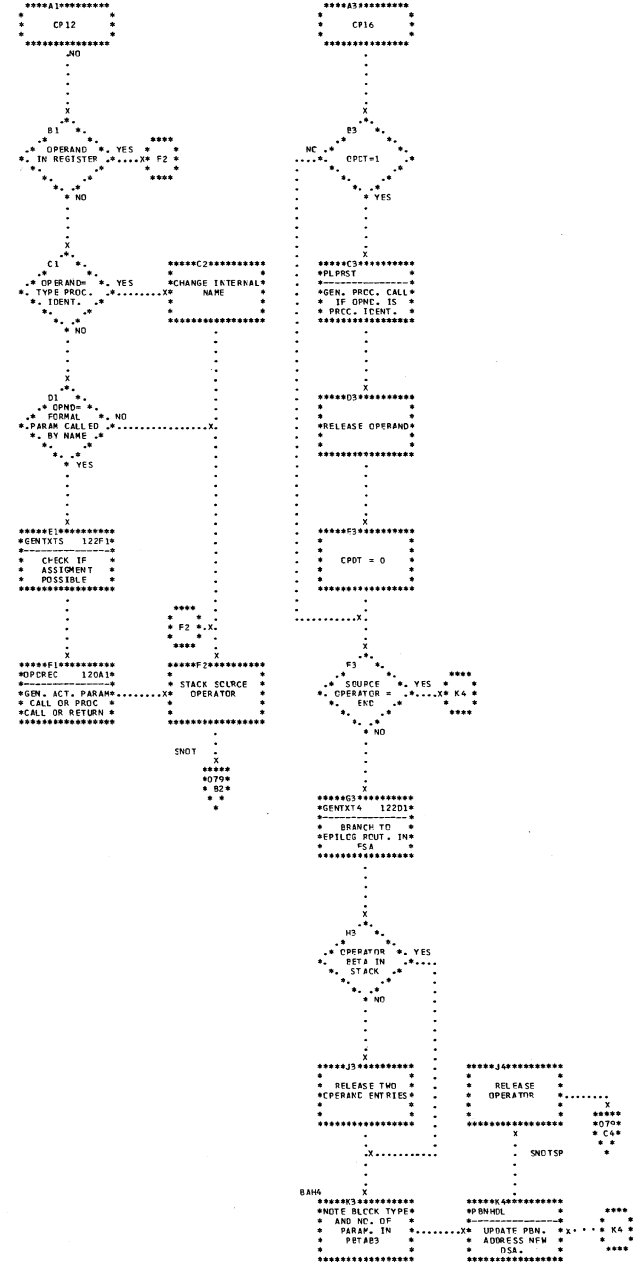


Chart 084: Compilation Phase - IEX50 CP17, CP18 and CP19

Microfiche IEX50-5 (CP19: IEX50-4)

- A1: CP17
- B1: OPDT=1 (YES → B2; NO → C1)
- B2: PLPRST — GEN. PROC. CALL IF OPND. IS PROC. IDENT.
- C1: STACK ELSE - S
- C2: RELEASE ONE OPERAND
- D1: RESERVE ONE LABEL ADDRESS TABLE ENTRY
- E1: GENTXT6 122E1 — LOAD ADDRESS FROM LAT AND BRANCH
- F1: STORE PRPOINT IN LABEL ADDRESS TABLE
- G1: SAVE LABEL NUMBER IN OPDK
- SNOT → 079 B2
- A3: CP18
- B3: OPDT=1 (YES → E3; NO → C3)
- C3: STORE PRPOINT IN LABEL ADDRESS TABLE → C3
- D3: RELEASE ONE OPERAND AND ONE OPERATOR
- COMP → 079 D3
- E3: PLPRST — GEN. PROC. CALL IF OPND. IS PROC. IDENT.
- F3: RELEASE ONE OPERAND. OPDT=0 (from C3)
- A4: CP19
- B4: SWITCH TO STATEMENT CONTEXT FROM PROGRAM CONTEXT
- COMP → 079 D3

Chart 085: Compilation Phase - IEX50 CP20 and CP21

Microfiche IEX50-4

- A1: CP20
- B1 (DEIC2): IS OPERAND IN A REGISTER (NO → B2; YES → C2; → 085 B1)
- B2: OPDREC 120A1 — GEN. ACT. PARAM CALL OR PROC CALL OR RETURN
- C1: SERR1 123A1 — RECORD ERROR NR.192
- C2 (BIE4): COMPARE TYPES (INCOMP. → C1; DIFF → D3; EQUAL → D2)
- D2: OPERAND TYPE (REAL → E1; INTEGER → E2; FOR → F2; DECLARATION BRACKET → G2; ANY OTHER → H2; ASSIGNMENT → J2; BOOLEAN → F3)
- D3: LAST OPER. INTEGER (NO → E4; YES → E3)
- E1: REALRT — REAL ROUTINE (ACHDB2)
- E2: INTEGER — ROUTINE (DDH7B1)
- E3: TRREIN — GEN. CALL TO CNVRDI IN FSA → D2
- E4: TRINRE — GEN. CALL TO CNVIRD IN FSA → D2
- F1: RELEASE ONE OPERATOR
- F2: RELEASE OPERAND CP43 → 092 B5 DJH1E43
- F3: OPER. BEFORE LAST IS AN ACC. (YES → F4; NO → G4)
- F4: ROUTINE3 — GET REG. NO AND DISPL. OF OPERAND
- G1: STACK OPERATOR
- G2: RELEASE OPERAND → CP51 099 B3 DERE2
- G4: ROUTINE1 — GET REG. NO AND DISPL. OF OPERAND
- G5: LAST OPND A CONST. (NO → B1; YES → H4)
- H2: RELEASE THE OPERANDS — COMP → 079 D3
- H4: GENTXT4 122D1 — MOVE CONSTANT TO LEFT VAR. DSA LOCATION → B1
- J2: MOVE LAST OPND DOWN 5 BYTES
- K2: RELEASE OPERAND → C2
- A5: CP21
- B5: CHECK TYPES OF OPERANDS
- CP20 → B1

Chart 086: Compilation Phase - IEX50
CP22, CP23, CP24 and CP25

Microfiche IEX50-4
(CP22 and CP23)

Microfiche IEX50-3
(CP24 and CP25)

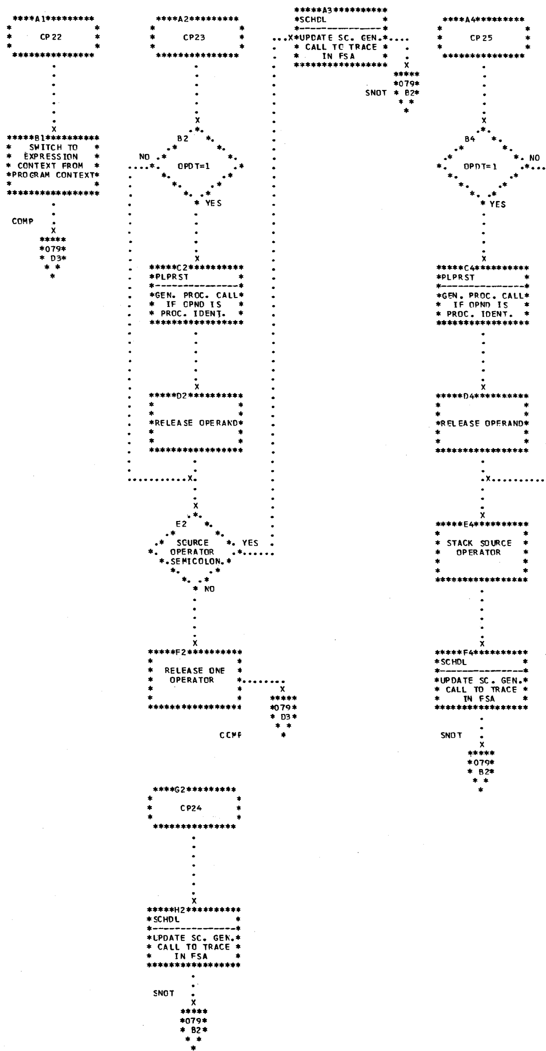


Chart 087: Compilation Phase - IEX50
CP26, CP27, CP28, CP29,
CP30, CP31 and CP33

Microfiche IEX50-5
(CP33 IEX50-4)

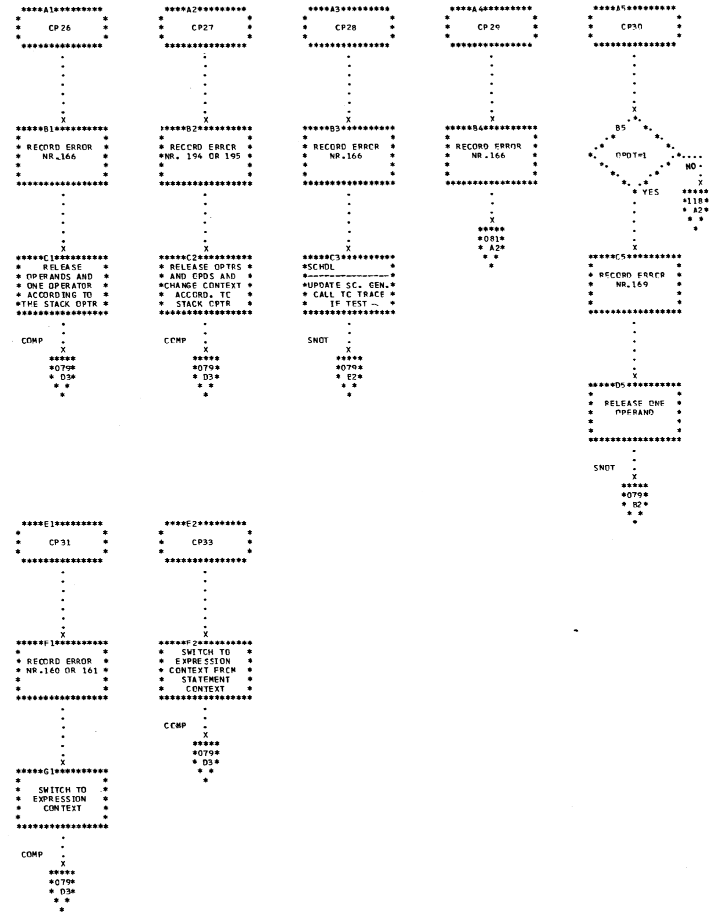


Chart 088: Compilation Phase - IEX50 CP34 and CP36 Microfiche IEX50-4 (CP34) Microfiche IEX50-3 (CP36)

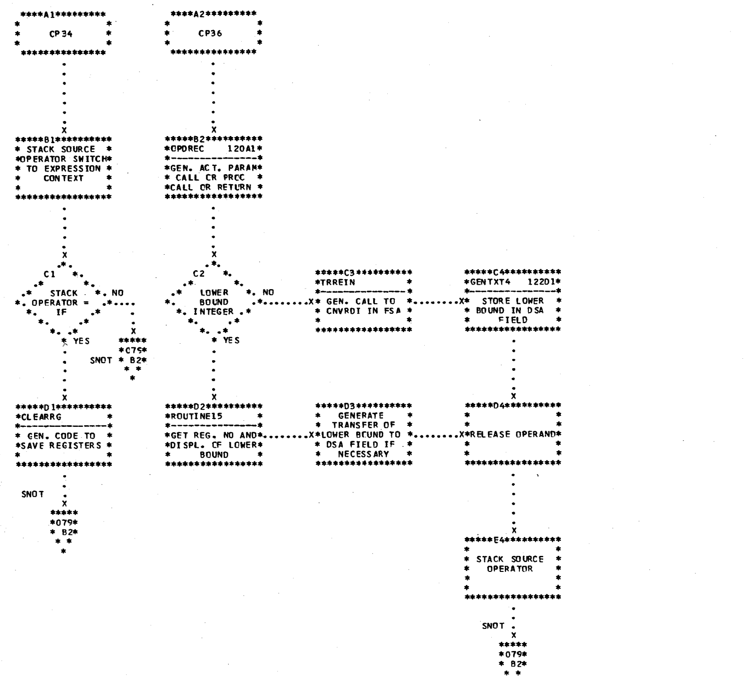


Chart 089: Compilation Phase - IEX50 CP38 Microfiche IEX50-3

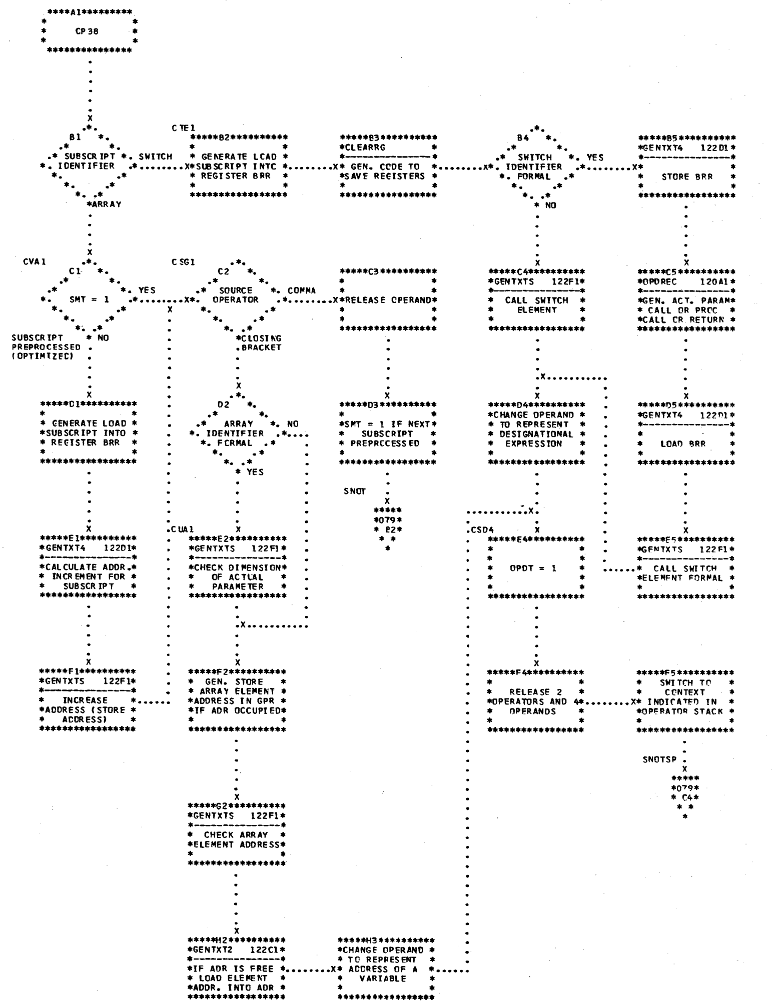


Chart 090: Compilation Phase - IEX50 Microfiche IEX50-2

CP40

Chart 091: Compilation Phase - IEX50 Microfiche IEX50-3

CP41

Chart 092: Compilation Phase - IEX50 — Microfiche IEX50-2

CP43

- A2: CP43
- DJH1F43 — FROM CP20 — 092 B5
- B2: COUNTING LOOP AND NO OPTIMIZA-TION — YES → B3; NO → C2
- B3: LAST OPERAND REPLACES PREVIOUS OPERAND
- B4: RELEASE ONE OPERAND
- B5: STACK FOR.=
- C2 (DJG3): REPLACE STACK OPTR BY ASSIGN — CP20 DBIC2 → 085 B1
- C4: TEST SOURCE OPERATOR
- D1 (DLB2, XCOMMA): SET ON BIT 7 IN OPTBYTE, GENUINE FOR LIST
- D2: RELEASE TWO LABELS IF POSSIBLE — CP43T → 090 H1
- D3 (DMB2, XDO): ONLY ONE ELEMENT IN LIST — YES → D2; NO → E3
- D4 (DKB2, STEP): NORMAL LOOP — YES → E4; NO → CP43T
- D5 (DKB4, WHILE): RESERVE ONE ENTRY IN LAT. STORE LN IN OPERAND STACK
- E1: GENTXT4 122D1 — FETCH ADDRESS OF ITERATED STATEMENT
- E3: RESERVE ONE ENTRY IN OPERATOR STACK
- E4: GENTXTS 122F1 — FIND RETURN ADDRESS AND STORE IT
- E5: GENTXT6 122E1 — MOVE RETURN ADDRESS FROM LAT TO STACK
- F1: COUNTING LOOP — YES → G2; NO → G1
- F3: COUNTING LOOP — YES → F4; NO → G3
- F4: GENERATE SET STEP TO ZERO AND LOOP COUNTER TO 1 IF NECESSARY — 094 C5 — CP47 DWITERS BEGINNING OF ITERATED STATEMENT
- F5: GENERATE SET STEP TO ZERO IF NECESSARY — CP43T → 090 H1
- G1: GENERATE SET STEP TO ZERO IF NECESSARY
- G2: GENERATE SET STEP TO ZERO AND LOOP COUNTER TO ONE IF NECESSARY
- G3: GENERATE SET STEP TO ZERO IF NECESSARY
- H1: GENTXTS 122F1 — STORE RET.ADD. BRANCH TO ITER-ATED STATEMENT
- H2: GENTXT2 122C1 — BRANCH AND LINK TO ITERATED STATEMENT — CP43T → 090 H1
- H3: GENTXT6 122E1 — MOVE RETURN ADDRESS — 097 J2 — CP49 EXITERS BEGINNING OF ITERATED STATEMENT
- J1: SUBSCRIPT OPTIMIZATION — NO → CP43T 090 H1; YES → K1
- K1: GENTXT6 122E1 — INITIALIZE CONDITIONAL ENTRY ADDRESS — CP43T → 090 H1

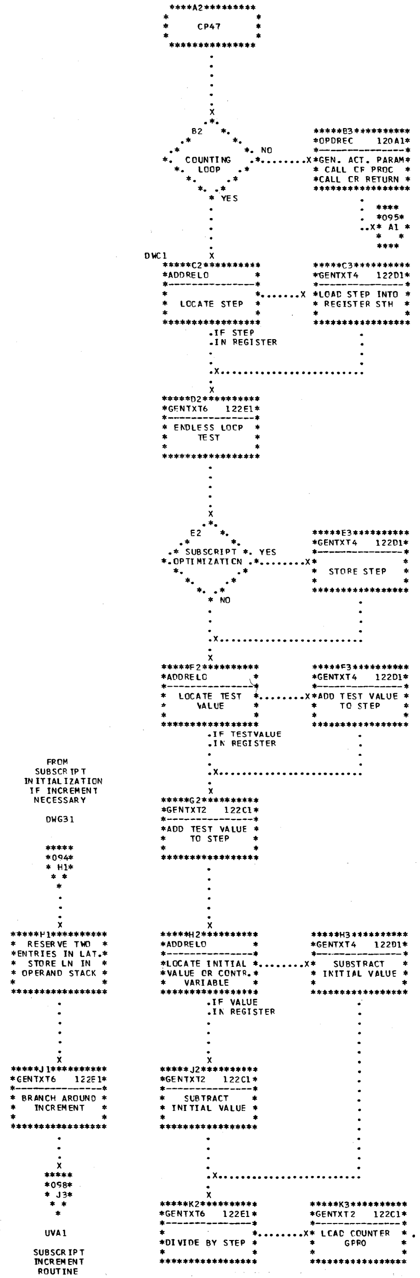
Chart 093: Compilation Phase - IEX50 — Microfiche IEX50-2

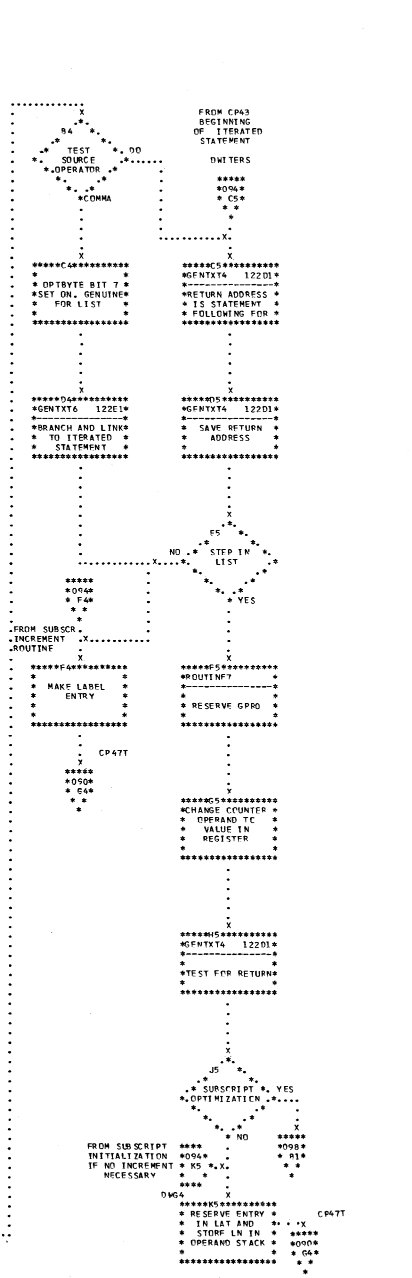
CP45

- A1: CP45
- A2: IS STEP REAL — YES → A4; NO → B2
- A4: ADDRELO — LOCATE STEP
- B1: COUNTING LOOP — YES → CP45TF 090 H5; NO → C1
- B2: ADDRELO — LOCATE STEP
- B4: IS STEP IN REGISTER — NO → B5; YES → C4
- B5: GENTXTP4 122B1 — LOAD STEP INTO FPRO
- C1: OPCREC 120A1 — GEN. ACT. PARAM CALL OR PROC CALL OR RETURN
- C2: IS STEP IN REGISTER — YES → D3; NO → D2
- C4: GENTXTP2 122A1 — LOAD AND TEST STEP
- D2: GENTXT4 122D1 — LOAD STEP INTO REG. STH
- D3: GENTXT2 122C1 — LOAD AND TEST STEP
- D4: NORMAL LOOP — NO → F5; YES → E4
- E4: RESERVE ONE ENTRY IN LAT. STORE LN IN OPERAND STACK
- E5: GENTXTS 122F1 — ENDLESS LOOP TEST. EVALUATE SIGN OF STEP
- F4: GENTXTS 122F1 — SIGN OF STEP. BR. AROUND V=V+ STEP EACH 2ND TIME
- G1: IS STEP REAL — NO → G2; YES → H1
- G2: IS CONTROL VARIABLE REAL — NO → G3; YES → H2
- G3: NORMAL LOOP — YES → G5; NO → H3
- 093 G5 — DVE2
- G5: ADDCONV — LOCATE CONTROLLED VARIABLE
- H1: IS CONTROL VARIABLE REAL — YES → J2; NO → J1
- H2: GENTXT4 122D1 — CONVERT STEP TO REAL
- H3: GENTXT4 122D1 — STORE STEP INTO STACK — CP45TF → 090 H5
- H5: GENTXTP4 122B1 — ADD CONTROL VAR TO STEP. STORE RESULT IN C. V.
- J1: GENTXT4 122D1 — CONVERT STEP TO INTEGER → G3
- J2: NORMAL LOOP — NO → J3; YES → K2
- FROM CP47 — 093 K2
- J3: GENTXTP4 122B1 — STORE STEP INTO STACK — CP45TF → 090 H5
- J5: ELEMENTARY LOOP — YES → 096 D2 (CP47 DWA51); NO → K5
- K2: ADDCONV — LOCATE CONTROLLED VARIABLE
- K3: GENTXTP4 122B1 — ADD CONTROL VAR TO STEP. STORE RESULT IN C. V.
- K5: GENTXT6 122F1 — BRANCH TO CALCULATE STEP — CP45TF → 090 H5

Chart 094: Compilation Phase - IEX50 Microfiche IEX50-2

CP47





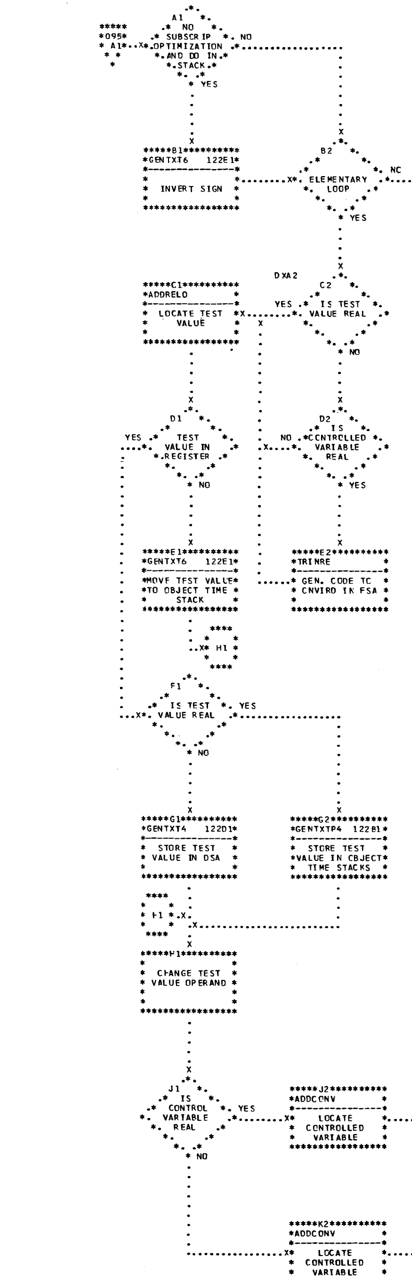


Chart 095: Compilation Phase - IEX50 Microfiche IEX50-2

CP47 (cont'd)

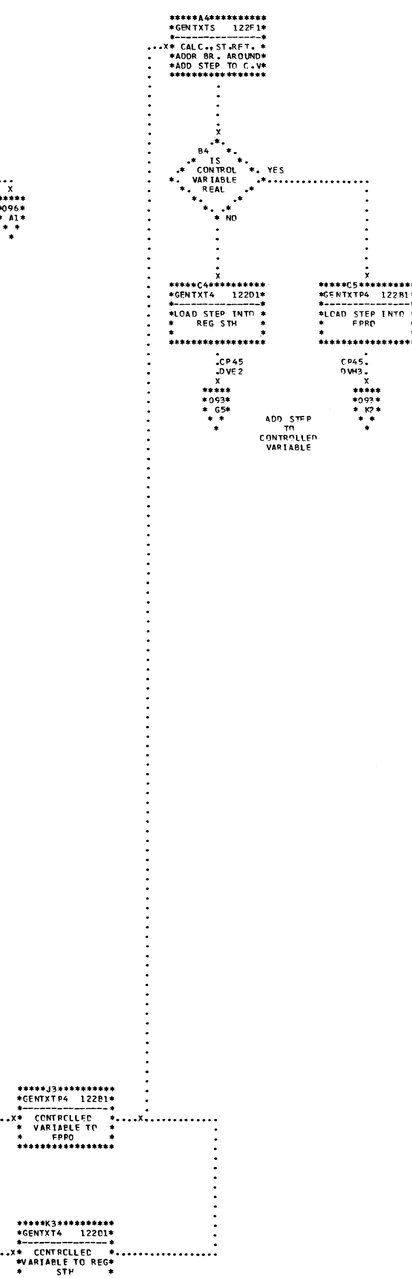


Chart 096: Compilation Phase - IEX50 CP47 (cont'd)

Microfiche IEX50-2

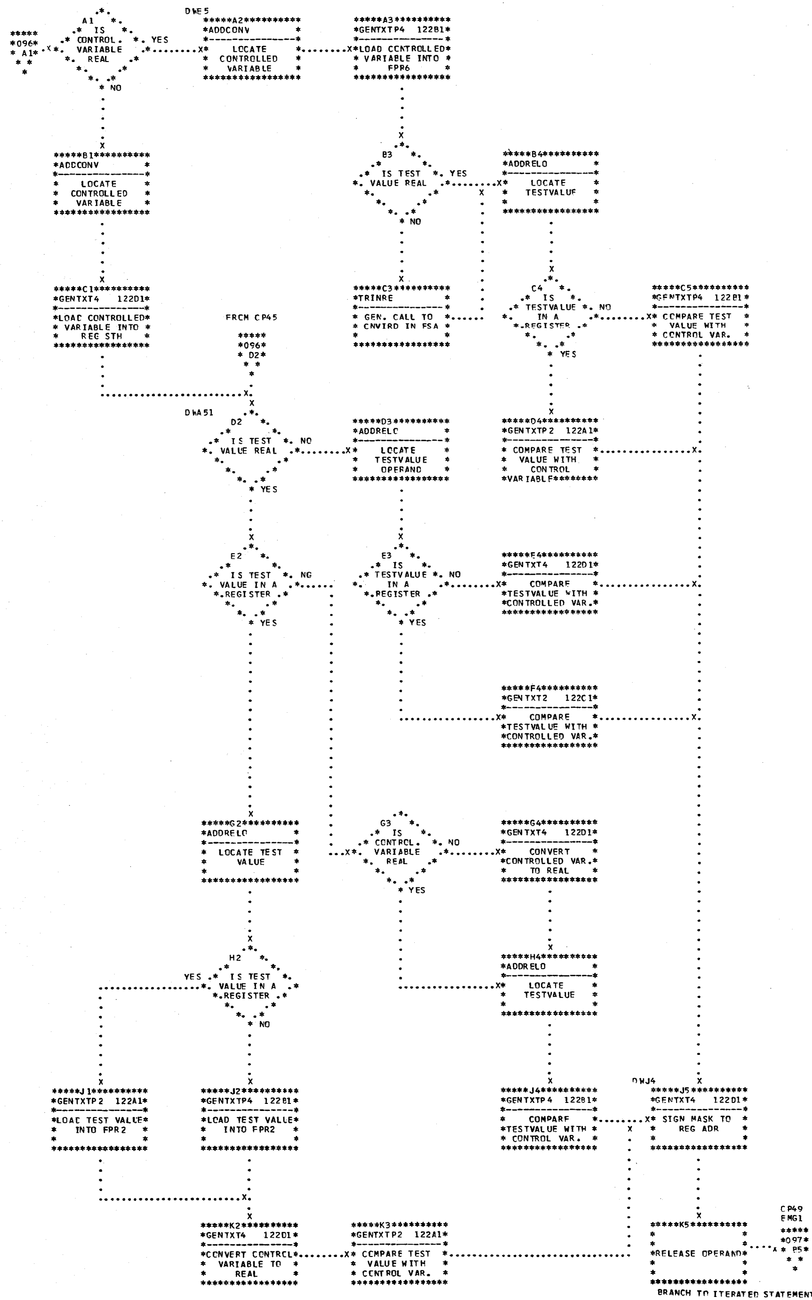


Chart 097: Compilation Phase - IEX50 CP49

Microfiche IEX50-3

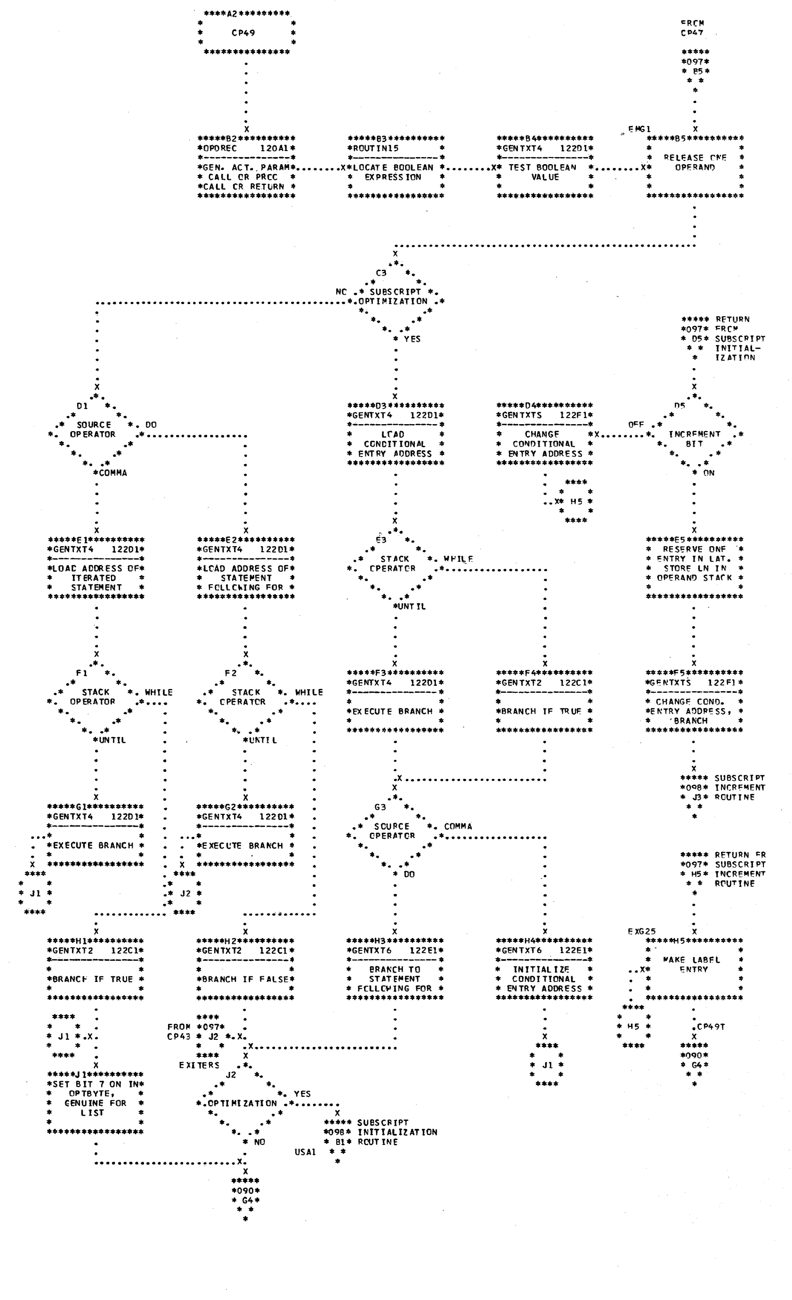


Chart 098: Compilation Phase - IEX50 CP49 (cont'd) Microfiche IEX50-3

SUBSCRIPT INITIALIZATION (098 B1)

- B1 (CWG3, USA1): CURRENT FOR STATEMENT NUMBER IN OPTAB? — NO / YES
- C1: OPENING BRACKET IN OPTAB FOUND IN SUTABC? — NO / YES
- C2 (USC2): RESERVE NEW SUTABC ENTRY OPENING BRACKET TO ENTRY
- D1: SUBSCRIPT PREPROCESSED IN THIS FOR STATEMENT? — NO / YES
- D2: INSERT FOR STATEMENT NUMBER INTO SUTABC
- E1 (USE2): SET RIGHT SUBSCRIPT POSITION BIT ON IN SUTABC
- F1: GENERATE ADD OR ADDEND SUBSCRIPT FACTOR TO NXTR
- G1: FACTOR BIT ON? — NO / YES
- G2: SET FACTOR BIT ON
- H1: GENERATE ADD OR FACTOR SUBSCRIPT FACTOR TO ADR
- H2: GENERATE LOAD OR FACTOR SUBSCRIPT FACTOR TO ADR
- J1 (USK1): NXTOPT 121A3 — FETCH NEXT OPTAB ENTRY
- B3: FIRST ENTRY FOR THIS STATEMENT? — YES / NO
- C3: FACTOR CALCULATED IN ADR? — NO / YES
- D3: GENTXTS 122F1 — PRECALCULATED SUBSCRIPT ADDR. TO REG. NEXTR
- E3: STEP ELEMENT IN FOR LIST? — NO / YES
- F3: SET INCREMENT INDICATOR ON
- G3: GENTXTS 122F1 — CALCULATE AND STORE ASSIGNMENT
- B4 (USK3): NXTR IS REGISTER 7? — YES / NO
- C4: NXTR IS SET TO NEXT REGISTER RESERVE IT (ROUTINE7)
- D4 (UTA1): RESERVE OBJECT TIME STACK AND OPERAND ENTRY
- E4: MOVE SUBSCRIPT OPERAND ADDRESS INTO SUTABC ENTRY
- F4: GENERATE LOAD BASE INTO REGISTER NXTR
- G4 (USG4): WHILE ELEMENT IN FOR LIST? — NO / YES
- B5: ROUTINE9 — STORE REGISTER NXTR
- C5: LAST OPTAB ENTRY FOR THIS F.S.? — NO / YES
- D5: COUNTING LOOP? — NO / YES (CP49 097 D5 EXG24)
- E5: SUBSCRIPT INCREMENT NECESSARY? — YES (CP47 094 H1 DWG4) / NO (CP47 094 K5 DWG31)
- G5: LAST OPTAB ENTRY FOR THIS F.S.? — NO / YES

UVA1 SUBSCRIPT INCREMENT (098 J3)

- J3: INCREMENT CALCULATED FOR SUBSCRIPT? — YES / NO
- J4: GENERATE ADD INCREMENT TO SUBSCRIPT ADDRESS
- K3: LAST OPERAND CONTAINING SUBSCRIPT? — NO / YES
- K4: COUNTING LOOP? — YES (CP47 094 F4 DWG51) / NO (CP49 097 H5 EXG25)

Chart 099: Compilation Phase - IEX50 CP51 Microfiche IEX50-3

- A1: CP51
- B1: OPDREC 120A1 — GEN. ACT. PARAM CALL OR PROC CALL OR RETURN
- C1: UPPER BOUND INTEGER? — NO / YES
- C2: TRREIN — GEN. CALL TO CNVRDI IN FSA
- D1: RESERVE OPERAND ENTRY
- D2: GENTXT4 122D1 — STORE UPPER BOUND IN STACK
- E1: LOWER BOUND UPPER BOUND SMF-ENTRY TO LAST THREE OPERANDS
- F1: OPERATORS FOR: ANC MINUS TO OPERATOR STACK
- G1: SAVE SOURCE POINTER AND ADDR. DUMMY TEXT +1 DO (TO CP65 DHZB1 110 A1)
- J1 (ETB3): RELEASE OPERAND
- J2: OPDT = 0

DERE2 — FROM CP20 AFTER ASSIGMENT OF UPPER BOUND-LOWER BOUND +1 TO SMF-ENTRY (099 B3)

- B3: RESET SOURCE POINTER TO SOURCE TEXT
- C3: GENTXT4 122D1 — CHECK IF ARRAY BOUND ERROR
- D3: SOURCE — CLOSING BRACKET / COMMA
- E3: STORE DISPL OF DSA ENTRY FOR NEXT LOWER BOUND IN LAST STACK CPND (SNOT 079 E2)
- G3: GENTXTS 122F1 — CALCULATE ADDRESS INCREMENT FACTOR
- J3: NEXT OPERATOR — OTHER OPTR / ZETA
- K3: JBUFFER 121A1 — CHANGE INPUT BUFFER
- D4 (ERE3): RELEASE OPERAND
- E4: GENTXT6 122E1 — SAVE REG. LOAD ARRAY ELEMENT LENGHT
- F4 (ESB1): FIND NEXT SMF-ENTRY AND ATTACHED LOWER BOUND IN STACK
- G4: GENTXT4 122D1 — STORE ADDRESS INCREMENT FACT. IN SMF-ENTRY
- H4: LAST SMF-ENTRY PROCESSED? — NO / YES
- J4: STEP SOURCE POINTER 1 BYTE
- D5 (ESC2): RELEASE OPERAND AND OPERATOR
- E5: GENTXTS 122F1 — CALL GETMSTO IN FSA COMPLETE SMF
- F5: ANY ARRAY IDENTIFIER LEFT IN STACK? — NO / YES
- G5: RELEASE OPERAND
- H5: GENTXT6 122E1 — COPY ORIGINAL ARRAY SMF
- J5: COMMA IN SOURCE? — NO (COPP 079 D3) / YES (SNOT 079 B2)

Chart 100: Compilation Phase - IEX50 CP52 and CP54 — Microfiche IEX50-4

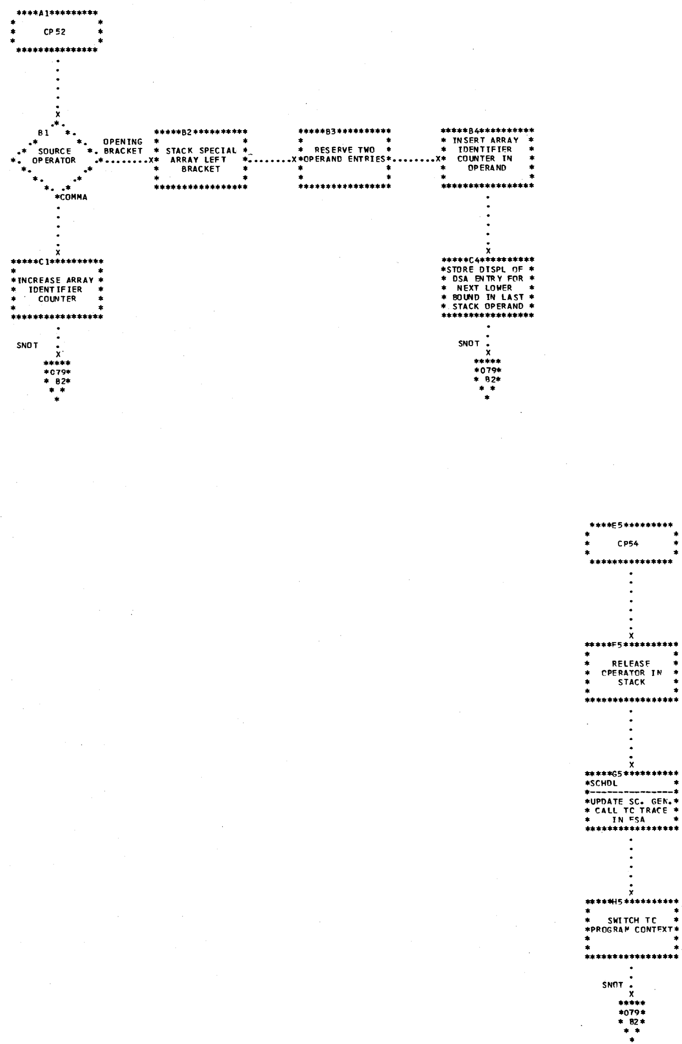


Chart 101: Compilation Phase - IEX50 CP56 and CP57 — Microfiche IEX50-4

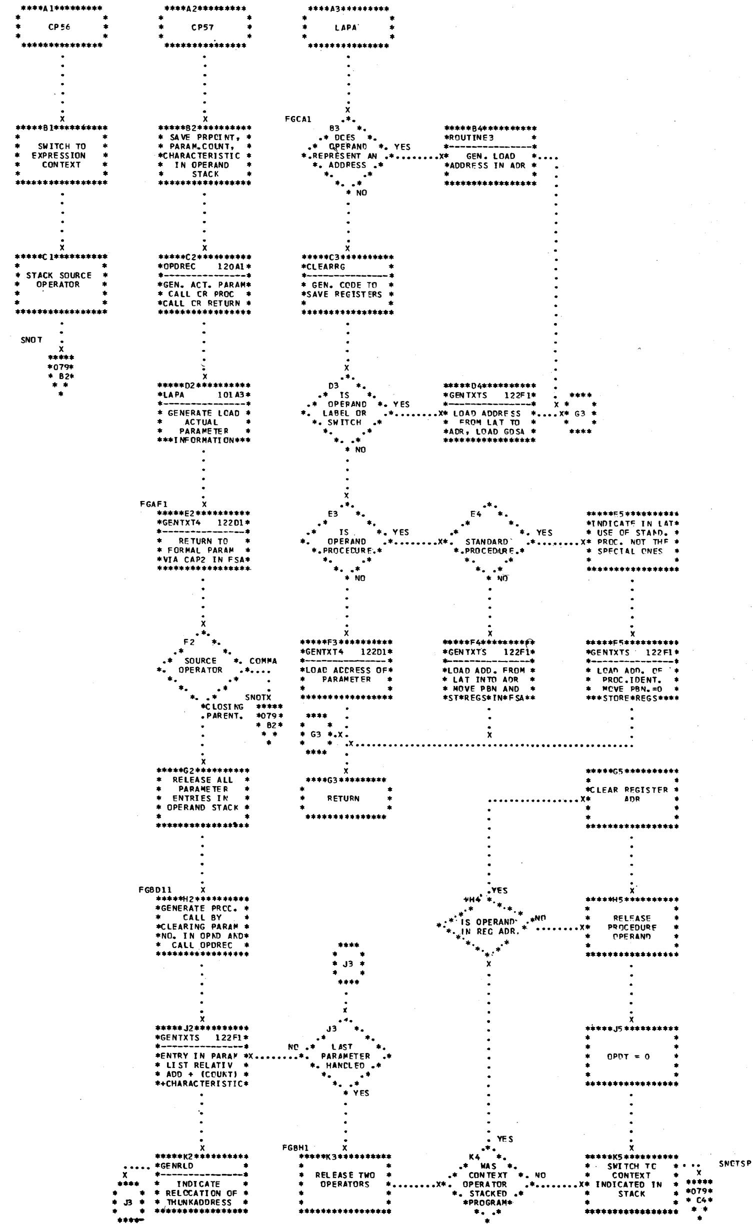


Chart 102: Compilation Phase - IEX50 CP59 — Microfiche IEX50-4

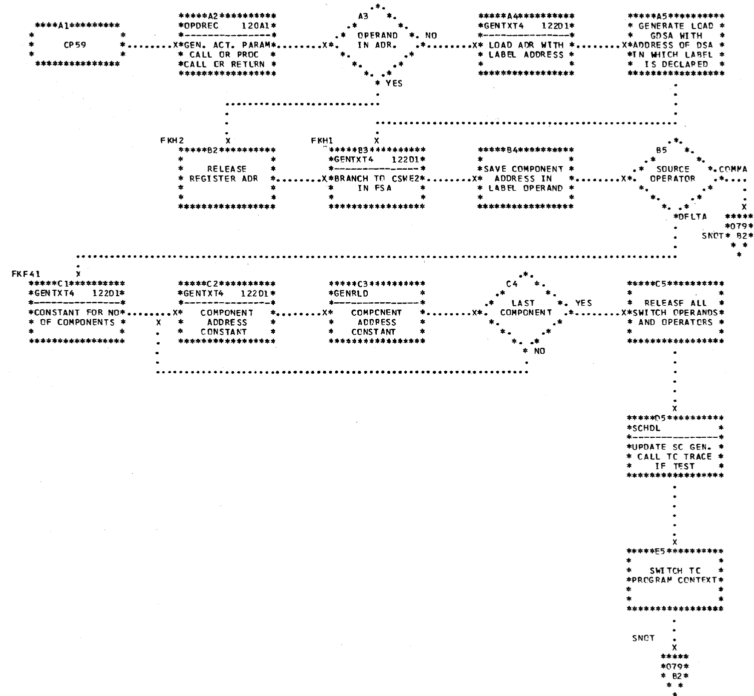


Chart 103: Compilation Phase - IEX50 CP61 — Microfiche IEX50-4

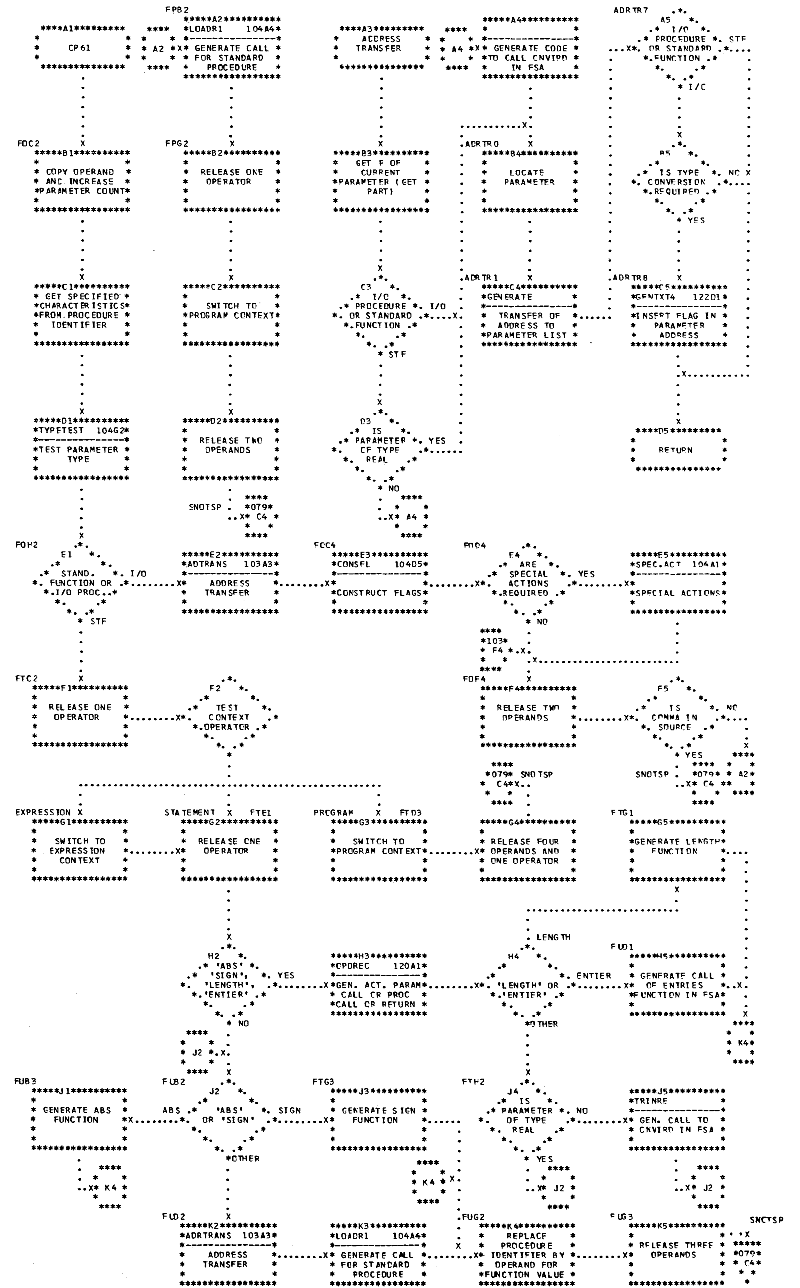


Chart 104: Compilation Phase - IEX50 Microfiche IEX50-4

SPECIAL ACTIONS and LOADR1

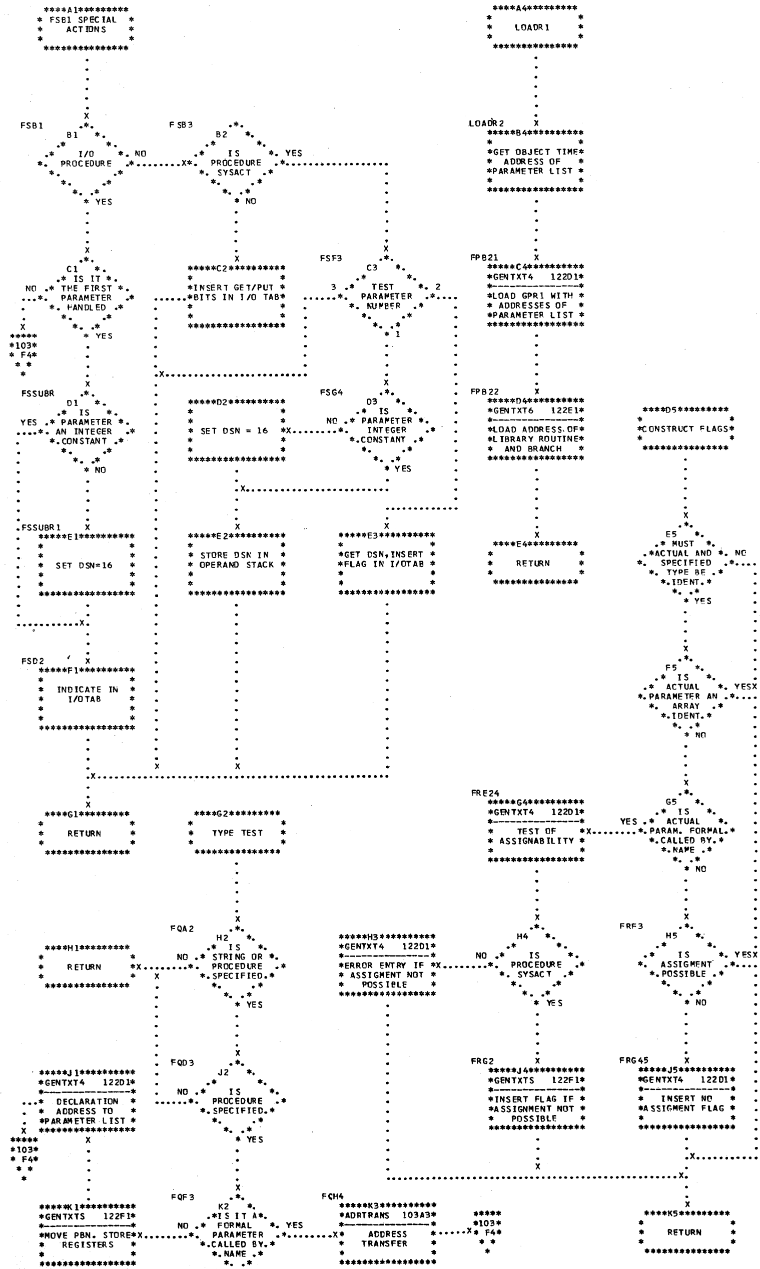


Chart 105: Compilation Phase - IEX50 Microfiche IEX50-4 (CP62)

CP62 and CP63 Microfiche IEX50-5 (CP63)

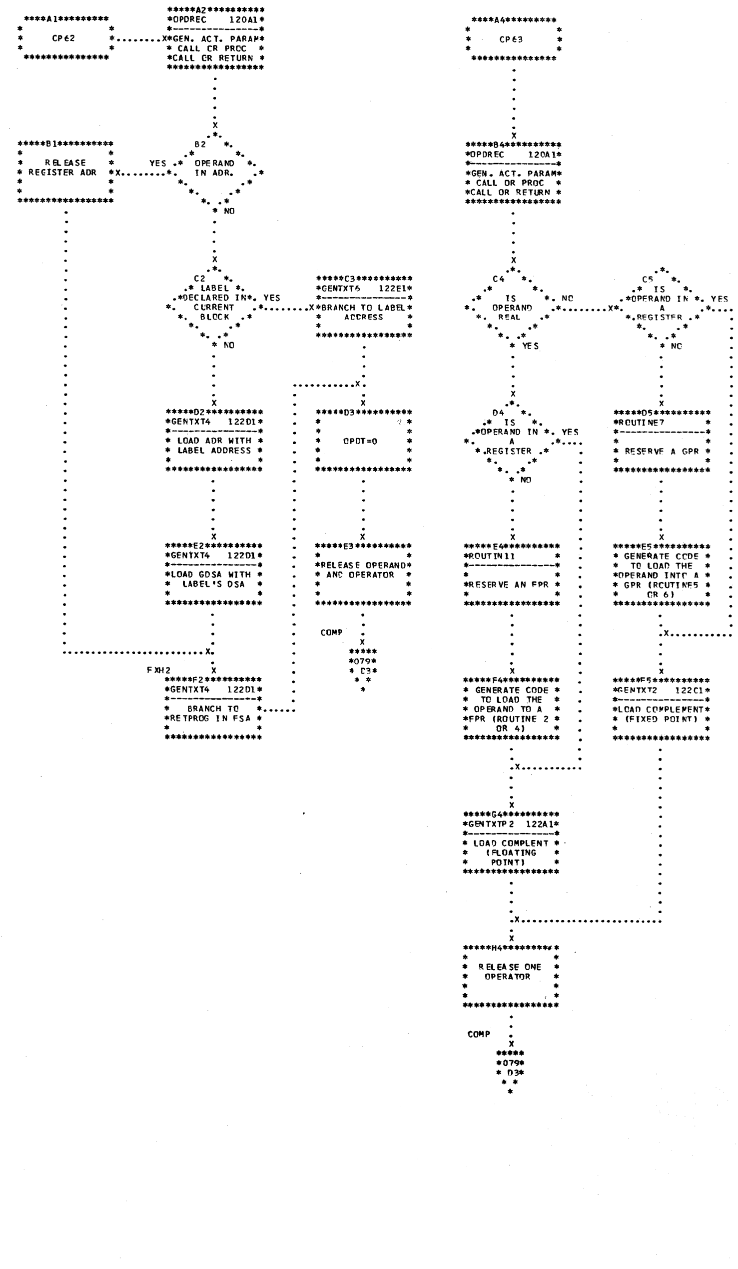


Chart 106: Compilation Phase - IEX50 — Microfiche IEX50-5

CP64, CP65, CP66, CP67 and CP68 — (CP64: IEX50-4)

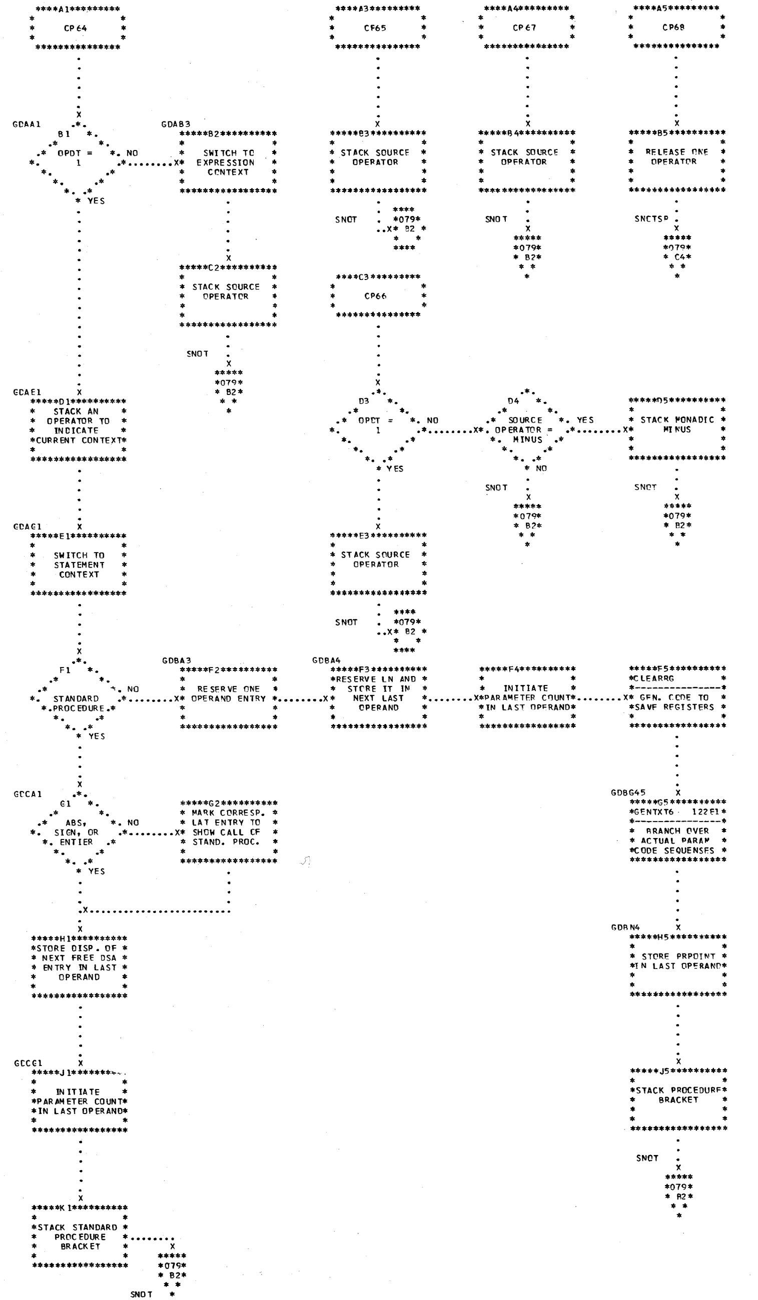


Chart 107: Compilation Phase - IEX50 — Microfiche IEX50-5

CP69

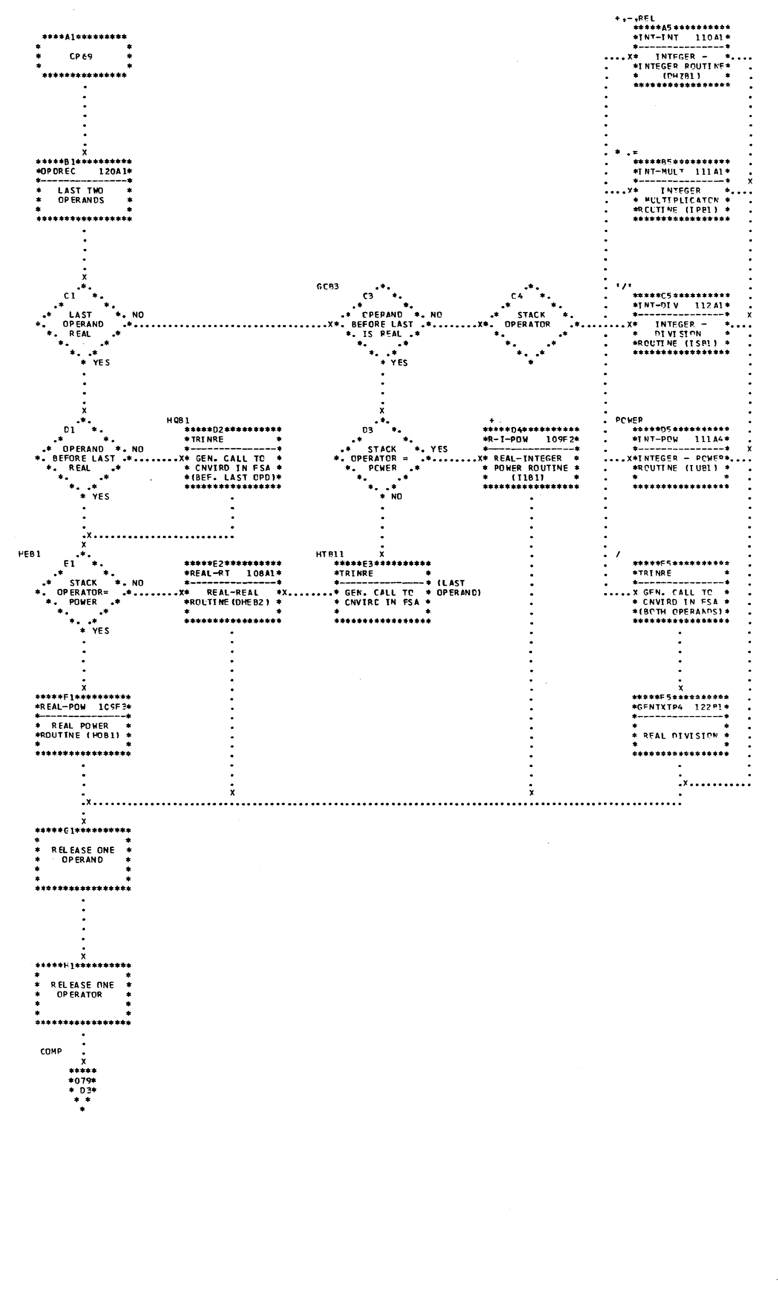


Chart 108: Compilation Phase - IEX50 Microfiches IEX50-5 and IEX50-6

REAL-REAL

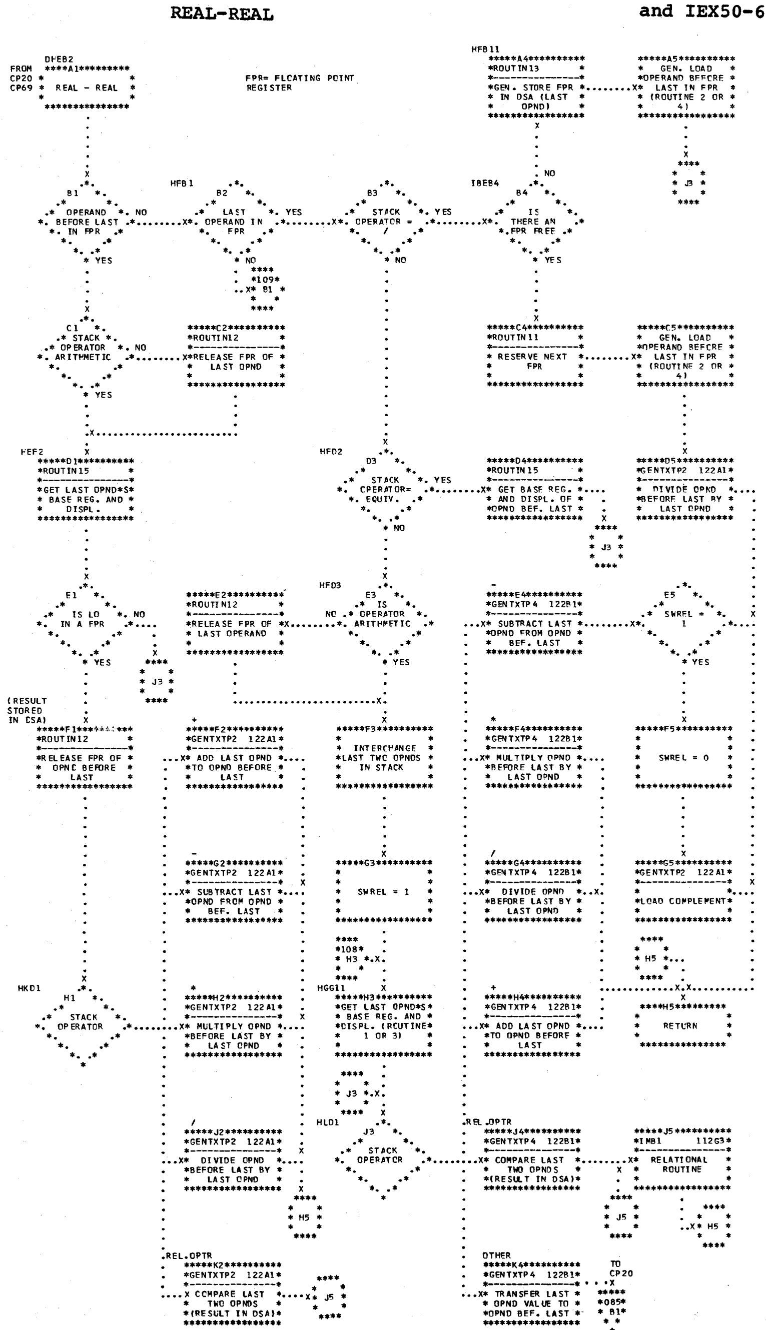


Chart 109: Compilation Phase - IEX50 Microfiches IEX50-5 and IEX50-6

REAL-REAL (cont'd), REAL-INTEGER, REAL POWER and COMMON POWER

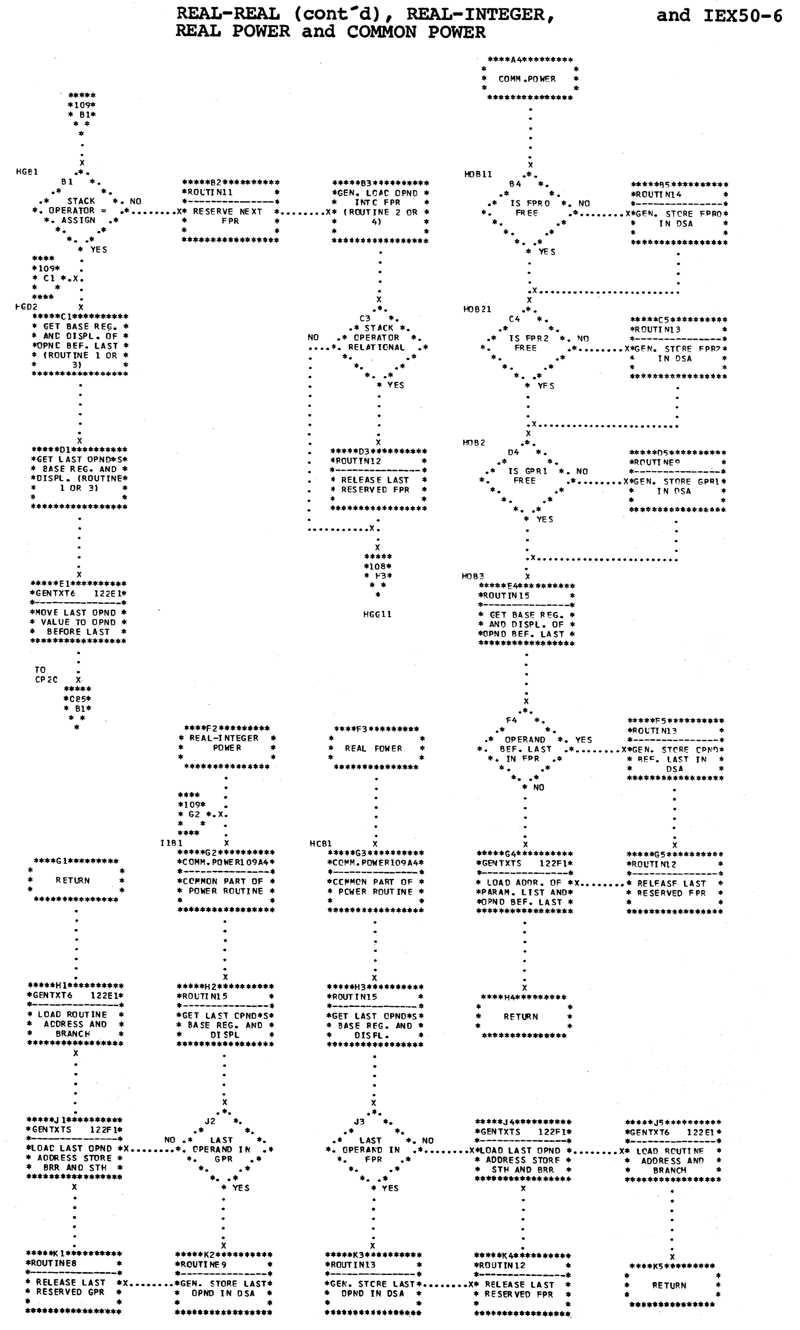


Chart 110: Compilation Phase - IEX50 — Microfiche IEX50-6

INTEGER-INTEGER

- OFZB1 — A1: INTEGER-INTEGER (entry from 110 A1)
- IAC2 — A3: STACK OPERATOR = ASSIGN — YES: to 109 C1; NO: to A4
- A4: ROUTINE7 — RESERVE NEXT GPR
- A5: GEN. LOAD OPND BEFORE LAST INTO GPR (ROUTINE 5 OR 6)
- B1: OPERAND BEF. LAST IN GPR — NO: to B3; YES: to C1
- HZC2 — B3: LAST OPERAND IN GPR — YES: to C3
- C1: STACK OPERATOR ARITHMETIC — NO: to C2; YES
- C2: ROUTINE8 — RELEASE LAST RESERVED GPR
- C3: STACK OPERATOR = ASSIGN — NO: to C4; YES: to D3
- IAE1 — C4: INTERCHANGE LAST TWO OPERANDS IN STACK
- HZF1 — D1: ROUTIN15 — GET LAST OPND'S BASE REG. AND DISPL.
- D3: GET BASE REG. AND DISPL. OF OPND BEFORE LAST (ROUTINE 1 OR 2)
- IAJ1 — D4: SWREL = 1
- E1: LAST OPERAND IN GPR — NO: to E3; YES: to F1
- IAG2 — E3: ROUTINE8 — RELEASE LAST RESERVED GPR
- E4: STACK OPERATOR ARITHMETIC — YES: to E3; NO
- F1: ROUTINE8 — RELEASE LAST RESERVED GPR
- IAF3 — F3: GET LAST OPND'S BASE REG. AND DISPLACEMENT (ROUTINE 1 OR 3)
- F4: GENTXT4 122D1 — SUBTRACT LAST OPND FROM OPND BEFORE LAST
- F5: SWREL=1 — NO; YES: to G5
- G2: GENTXT2 122C1 — ADD LAST OPND TO OPND BEFORE LAST
- IFC3 — G3: STACK OPERATOR
- G4: GENTXT4 122D1 — MOVE LAST OPND VALUE TO OPND BEFORE LAST (to 085 B1)
- G5: SWREL = 0
- IEC3 — H1: STACK OPERATOR
- H2: GENTXT2 122C1 — SUBTRACT LAST OPND FROM OPND BEFORE LAST
- H4: GENTXT4 122D1 — ADD LAST OPND TO OPND BEFORE LAST
- H5: GENTXT2 122C1 — LOAD COMPLEMENT
- REL.CPTR. — J2: GENTXT2 122C1 — COMPARE LAST TWO OPNDS (RESULT IN DSA)
- REL.OPTR. — J4: GENTXT4 122D1 — COMPARE LAST TWO OPNDS (RESULT IN DSA)
- K2: IMB1 112G3 — RELATIONAL ROUTINE
- K3: RETURN
- K4: IMB1 112G3 — RELATIONAL ROUTINE
- K5: RETURN

Chart 111: Compilation Phase - IEX50 — Microfiche IEX50-6

INTEGER MULTIPLICATION and INTEGER POWER

- A1: INTEGER MULT.
- A2: ROUTINE9 — GEN. STORE THE EVEN GPR IN DSA
- A3: ROUTINE9 — GEN. STORE THE ODD GPR IN DSA
- A4: INTEGER POWER
- IPB1 — B1: OPERAND BEF. LAST IN GPR — NO: to B2; YES: to C1
- IRB1 — B2: NEXT EVEN GPR FREE — NO: to A2; YES: to B3
- B3: FOLLOWING ODD GPR FREE — NO: to A3; YES: to C3
- IUR1 — B4: LAST OPERAND A CONSTANT — NO: to B5; YES: to C4
- IUR2 — B5: TRINRE — GEN. CALL CNVTRD IN FSA (OPD BEF. LAST) (to 109 G2)
- C1: OPERAND BEF. LAST IN ODD GPR — NO: to C2; YES: to D1
- IPD2 — C2: LAST OPND IN NEXT GPR — YES: to H1; NO: to F3
- C3: ROUTIN15 — GET BASE REG. AND DISPL. OF OPND BEF. LAST (to E1)
- C4: GPR0 FREE — NO: to C5; YES: to D4
- C5: ROUTINE9 — GEN. STORE GPR0 IN DSA
- IPH1 — D1: PRECEDING EVEN GPR FREE — NO: to D2; YES: to E1
- IPH31 — D2: LAST OPND IN THE EVEN GPR — NO: to D3; YES: to H1
- IPH4 — D3: FOLLOWING EVEN GPR FREE — YES; NO: to E3
- IUC1 — D4: GPR1 FREE — NO: to D5; YES: to E4
- D5: ROUTINE9 — GEN. STORE GPR1 IN DSA
- IPJ1 — E1: ROUTIN15 — GET BASE REG. AND DISPL. OF LAST OPND
- IPE3 — E2: GENTXT2 122C1 — LOAD OPND BEFORE LAST INTO ODD GPR
- E3: ROUTINE9 — GEN. STORE THE EVEN GPR IN DSA (to F3)
- IUC11 — E4: ROUTIN15 — GET BASE REG. AND DISPL. OF OPND BEF. LAST
- F1: LAST OPND IN GPR — NO: to F2; YES: to G1
- IQE2 — F2: GENTXT4 122D1 — MULTIPLY OPND BEFORE LAST BY LAST OPND
- IPD3 — F3: FOLLOWING ODD GPR FREE — YES; NO: to G3
- F4: OPERAND BEF. LAST IN GPR — YES: to F5; NO: to G4
- F5: ROUTINE9 — GEN. STORE OPND BEF. LAST IN DSA
- G1: ROUTINE8 — RELEASE LAST RESERVED GPR
- G3: ROUTINE9 — GEN. STORE THE ODD GPR IN DSA
- IUH21 — G4: GENTXTS 122F1 — LOAD PARAMETER LIST ADDRESS AND ADDRESS OF OPN BEF. LAST
- G5: ROUTINE8 — RELEASE LAST RESERVED GPR
- IQD11 — H1: GENTXT2 122C1 — MULTIPLY OPND BEFORE LAST BY LAST OPND
- H4: ROUTINE1 — GET BASE REG. AND DISPL. OF LAST OPND
- IQE11 — J1: GENTXT4 122D1 — CHECK FOR OVERFLOW
- J4: GENTXTS 122F1 — CALL POWER-ROUTINE
- K1: RETURN
- K4: RETURN

Chart 112: Compilation Phase - IEX50 Microfiche IEX50-6

INTEGER DIVISION and RELATIONAL

A1: INTEGER DIV.

B1 (ISB1): OPERAND BEFORE LAST IN GPR — NO → B2; YES → C1

B2 (ISC2): NEXT EVEN GPR FREE — NO → B3; YES → B4

B3 (ISD5): ROUTINE9 — GEN. STORE THE EVEN GPR IN DSA

B4: GEN. LOAD OPND BEFORE LAST INTO THE EVEN GPR (ROUTINE 5 OR 6)

C1: OPERAND BEFORE LAST IN EVEN GPR — NO → C2; YES → D1

C2 (ISG4): NEXT EVEN GPR FREE — NO → C3; YES → C4

C3: ROUTINE9 — GEN. STORE THE EVEN GPR IN DSA

C4 (ISH4): GENTXT2 122C1 — LOAD OPND. BEF. LAST INTO THE EVEN GPR

D1 (ISH11): IS FOLLOWING ODD GPR FREE — NO → D2; YES → E1

D2 (ISH2): LAST OPND IN THE ODD GPR — NO → D3; YES → E2

D3 (ISH3): ROUTINE9 — GEN. STORE THE ODD GPR IN DSA → E1

E1 (ISJ11): TRANSFER OPND BEF. LAST TO THE ODD GPR

E2: GENTXT2 122C1 — LOAD LAST OPND INTO STH

F1: ROUTIN15 — GET BASE REG. AND DISPL. OF LAST OPND

F3: IMB1 RELATIONAL

G1: IS LO IN A GPR — NO → G2; YES → H1

G2 (ITF21): GENTXTT4 — DIVIDE OPND BEFORE LAST BY LAST OPND

G3 (IMB1): GENTXT5 122F1 — MOVE TRUE TO DSA. LOAD ADDR. OF TRUE

G4: SWREL = 0

H1: ROUTINE8 — RELEASE LAST OPND'S REGISTER

H3: SWREL=1 — YES → G4; NO → J3

H4: GENTXT4 122D1 — BRANCH ACCORDING TO OPERATOR

J1: GENTXTP2 122A1 — DIVIDE OPND BEFORE LAST BY LAST OPND

J3: GENTXT4 122D1 — BRANCH ACCORDING TO OPERATOR

J4: GENTXT4 122D1 — MOVE FALSE TO DSA

J5: RETURN

K1: RETURN

Chart 113: Compilation Phase - IEX50 Microfiche IEX50-5

CP70, CP71, CP72, CP73, CP74 and CP75

A1: CP70

A2: CP71

A3: CP72

A4: CP73

A5: CP74

B1: SWITCH TO STATEMENT CONTEXT FROM EXPRESSION CONTEXT — COMP → 079 D3

B2: SWITCH TO PROGRAM CONTEXT FROM STATEMENT CONTEXT — COMP → 079 D3

B3: OPERATOR BEFORE LAST= ASSIGN — YES → CP79 116 A1; NO → CP75

B4: RECORD ERROR NO.160 OR 161

B5: RECORD ERROR NO.160 OR 161

C4: STACK (— SNOT → 079 B2

C5: STACK * — SNOT → 079 B2

E5: CP75

F5: RECORD ERROR NO.194 OR 195 — SNOT → 079 B2

Chart 114: Compilation Phase - IEX50 CP76

Microfiche IEX50-5

A1: CP76

A2: INTERCHANGE DSA FIELDS OF LAST TWO OPERANDS → D2

B1: OPDREC 120A1 — (LAST TWO OPERANDS)

B2: STACK OPERATOR =IMPL — YES / NO

B3: GENTXT6 122E1 — MOVE LAST OPND TO NEW DSA FIELD

B4: OPERAND BEFORE LAST A VARIABLE — YES / NO

B5: GET BASE REG. AND DISPL. OF OPERAND BEFORE LAST (ROUTINE 1 OR 3)

C1: OPND BEFORE LAST INTERMEDIATE VALUE IN DSA — NO / YES

C2: LAST OPND INTERMEDIATE VALUE IN DSA — YES / NO

C4: GENTXT4 122D1 — MOVE OPERAND BEFORE LAST TO NEW DSA FIELD

C5: GENTXT6 122E1 — MOVE OPERAND BEFORE LAST TO NEW DSA FIELD

D1: LAST OPND INTERMEDIATE VALUE IN DSA — NO / YES

D2 (JNF2): LAST OPERAND A VARIABLE — NO / YES → D2

D4 (JSB3): STACK OPERATOR — AND / OR / EQUIV IMPL

E2 (ISF4): GET BASE REG. AND DISPL OF LAST OPND (ROUTINE 1 OR 3)

E3 (AND): GENTXT4 122D1 — OPND BEF. LAST AND LAST OPND

E4 (OR): GENTXT4 122D1 — OPND BEF. LAST OR LAST OPND

E5 (EQUIV IMPL): GENTXT4 122D1 — INVERT OPND BEFORE LAST

F2 (JRB3): STACK OPERATOR — EQUIV IMPL / OR / AND

F4 (IRF41): RELEASE ONE OPERAND AND ONE OPERATOR → F4

F5: STACK OPERATOR — EQUIV / IMPL

G1: GENTXT4 122D1 — INVERT OPND BEFORE LAST

G2: GENTXT6 122E1 — OPND BEF. LAST OR LAST OPND

G3: GENTXT6 122E1 — OPND BEF. LAST AND LAST OPND

G5: GENTXT4 122D1 — OPND BEF. LAST EXCLUSIVE OR LAST OPND → F4

H1: OPERATOR — EQUIV. / IMPL

H3: RELEASE ONE OPERATOR

J1: GENTXT6 122E1 — OPND BEF.LAST EXCLUSIVE OR LAST OPND

J3: RELEASE ONE OPERAND

COMP → 079 C3

Chart 115: Compilation Phase - IEX50 CP77 and CP78

Microfiche IEX50-5

A1: CP77

B1: OPDREC 120A1 — GEN. ACT. PARAM CALL OR PROC CALL OR RETURN (LAST OPERAND)

C1: LAST OPND INTERMEDIATE VALUE IN DSA — NO / YES

C2: LAST OPERAND A CONSTANT — YES / NO

C3: GENTXT4 122D1 — MOVE CONSTANT TO NEW DSA FLD. AND INVERT

D2: GET BASE REG. AND DISPL. OF OPERAND (ROUTINE 1 OR 3)

E2: GENTXT6 122E1 — MOVE OPERAND TO NEW DSA FIELD

F2: GENRXT4 — INVERT OPERAND

G2: RELEASE OPERATOR

COMP → 079 D3

A4: CP78

B4: OPDREC 120A1 — GEN. ACT. PARAM CALL OR PROC CALL OR RETURN (LAST OPERAND)

C4: GET LAST OPND'S BASE REG. AND DISPLACEMENT (ROUTINE 1 OR 3)

D4: GENTXT4 122D1 — TEST IF VARIABLE IS TRUE OR FALSE

D5: STACK THEN

SNOT → 079 B2

E4: RESERVE ENTRY IN LABEL ADDRESS TABLE, SAVE LN IN OPERAND STACK

F4: STACK OPERATOR =IF-S — NO / YES

G4: STACK THEN-S

H4: SWITCH TO PROGRAM CONTEXT

SNOT → 079 B2

Chart 116: Compilation Phase - IEX50 CP79 — Microfiche IEX50-5

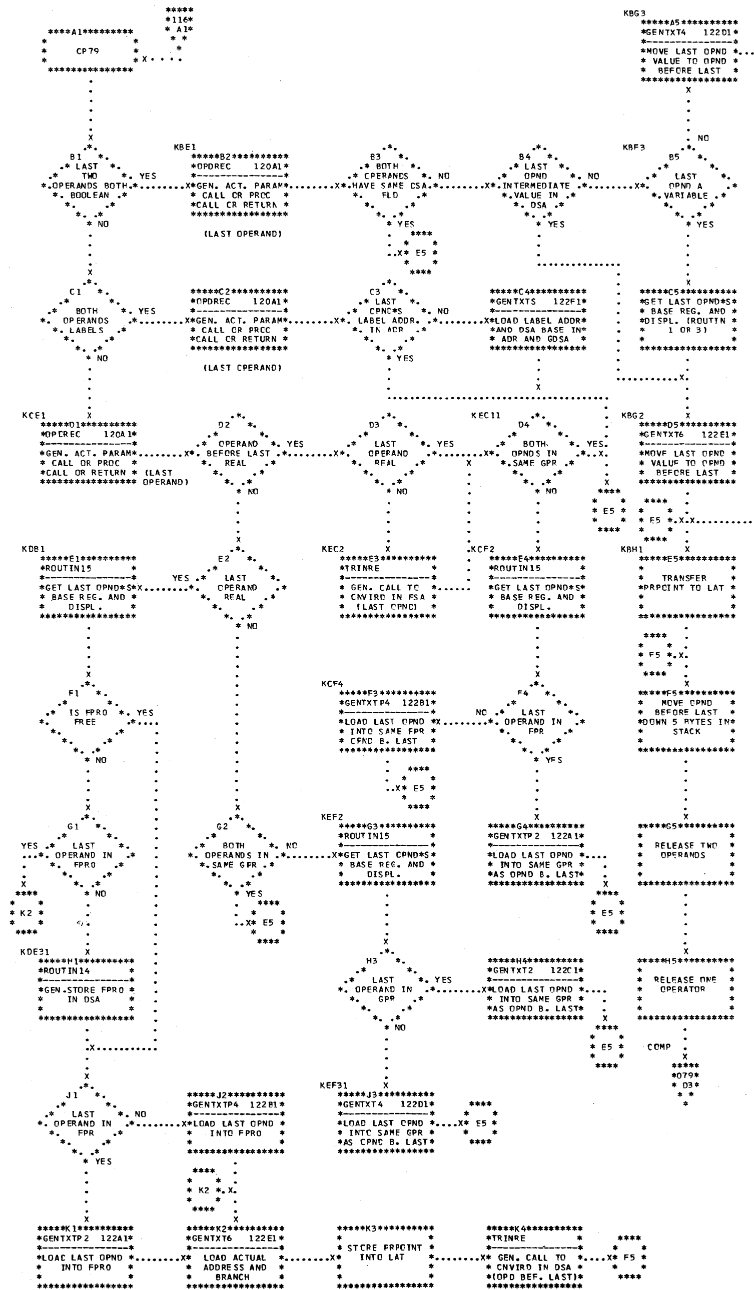


Chart 117: Compilation Phase - IEX50 CP80 and CP81 — Microfiche IEX50-5 (CP80), Microfiche IEX50-3 (CP81)

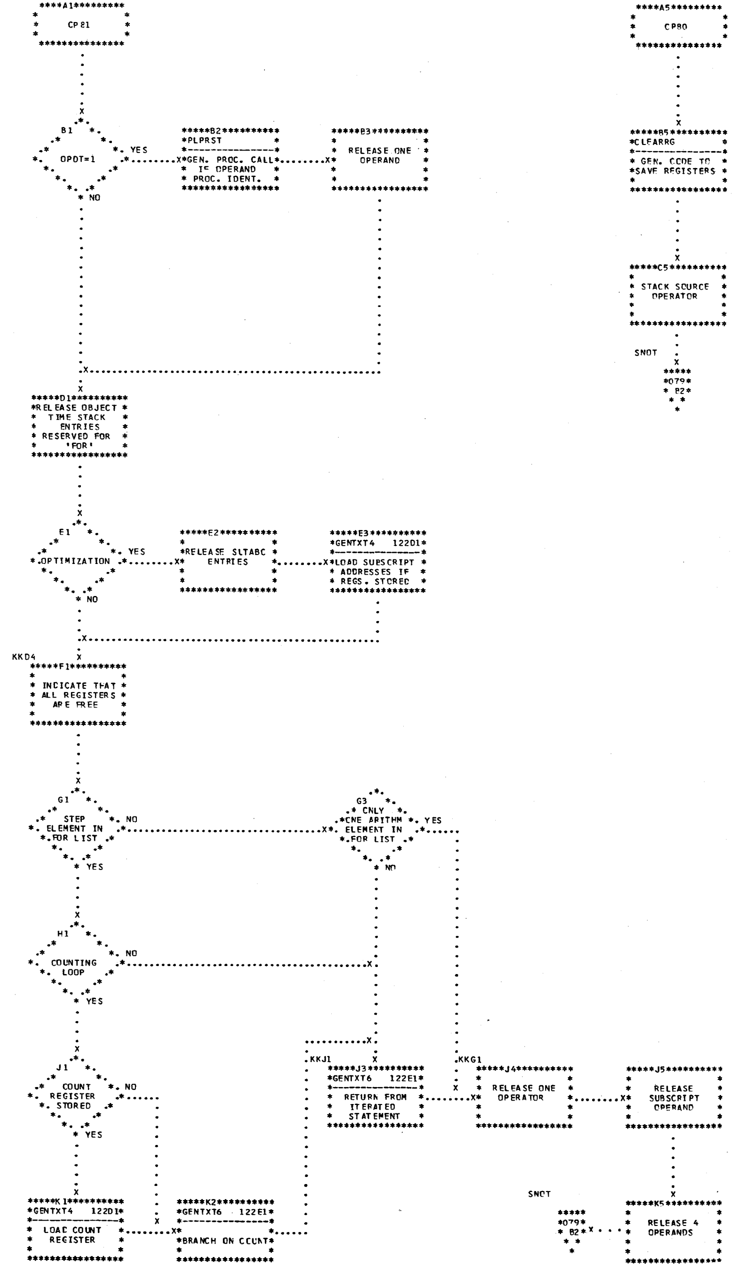


Chart 118: Compilation Phase - IEX50 — Microfiche IEX50-5

CP83, CP84, CP85 and CP86

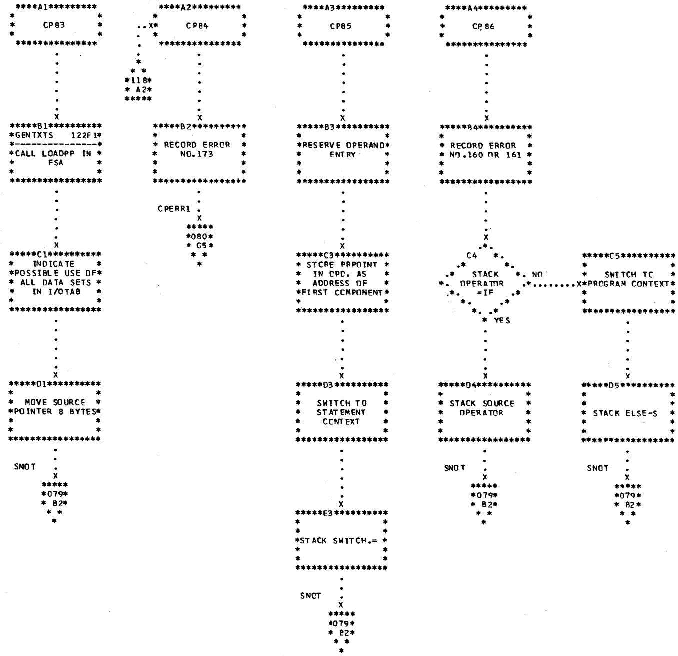


Chart 119: Compilation Phase - IEX50 — Microfiche IEX50-5

CP87

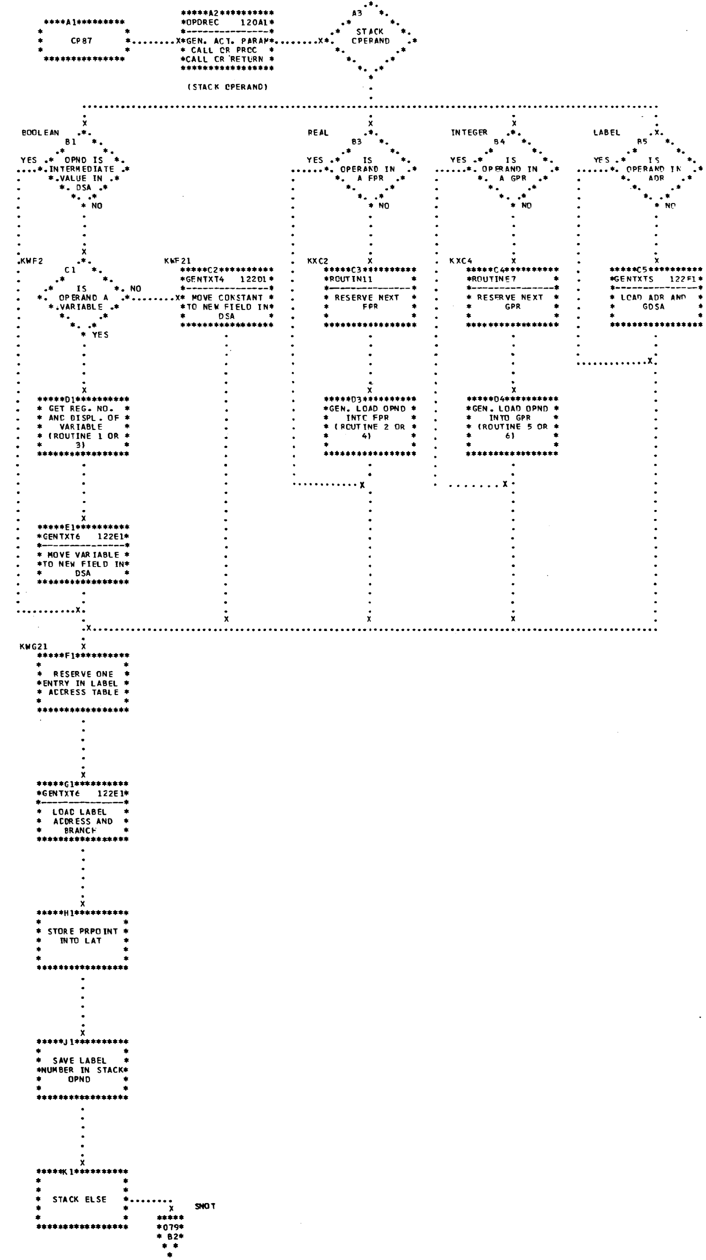


Chart 120: Compilation Phase - IEX50 Microfiche IEX50-1

OPDREC

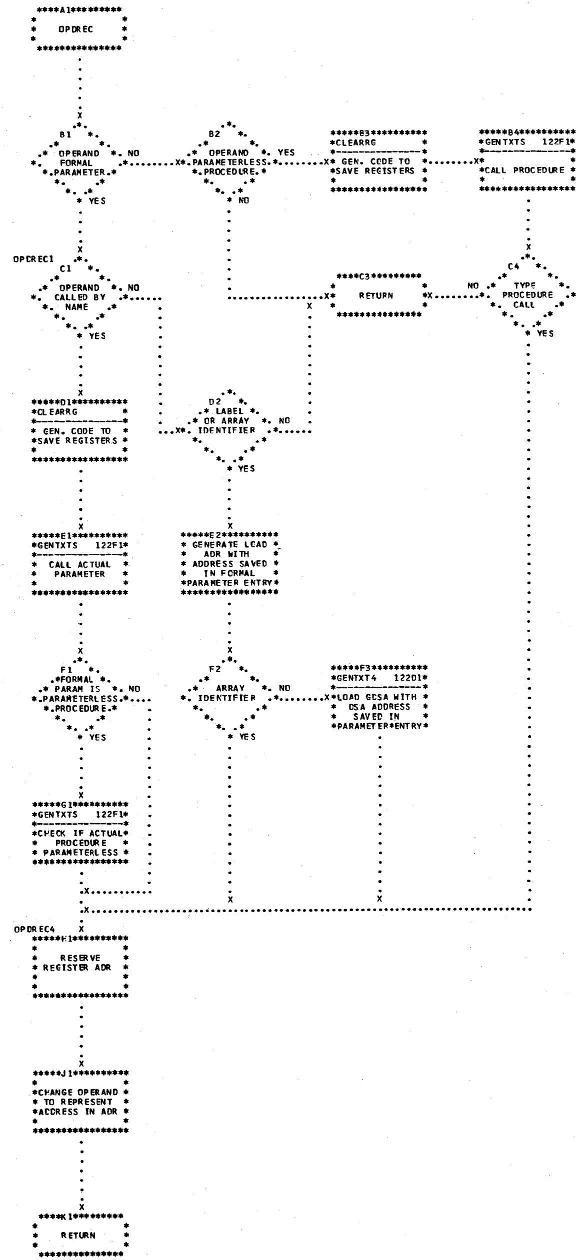


Chart 121: Compilation Phase - IEX50 Microfiche IEX50-1

JBUFFER, NXTOPT

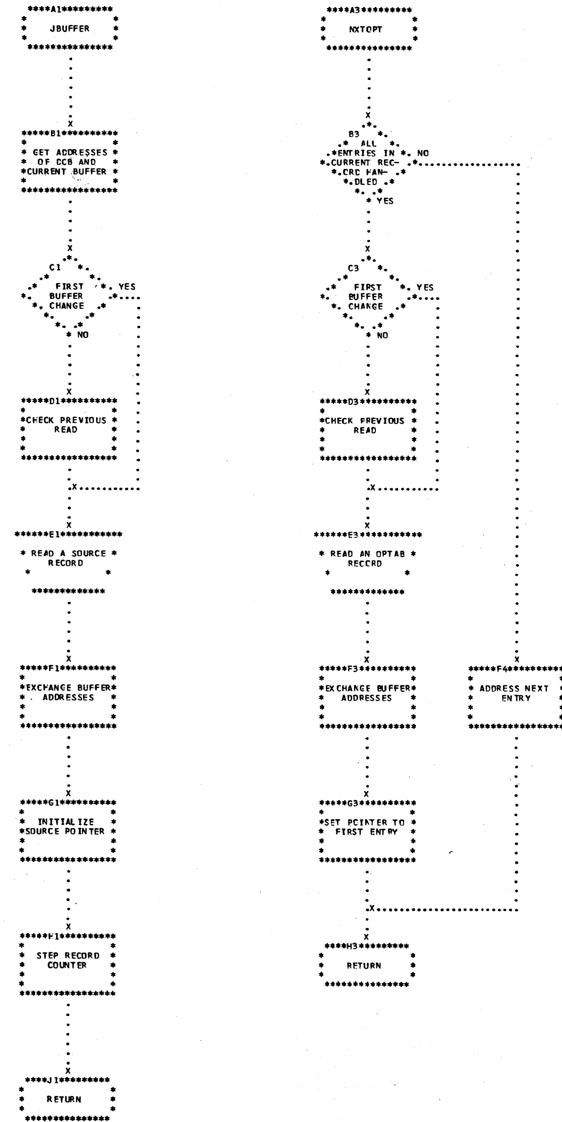


Chart 122: Compilation Phase - IEX50 Microfiche IEX50-1

GENERATE

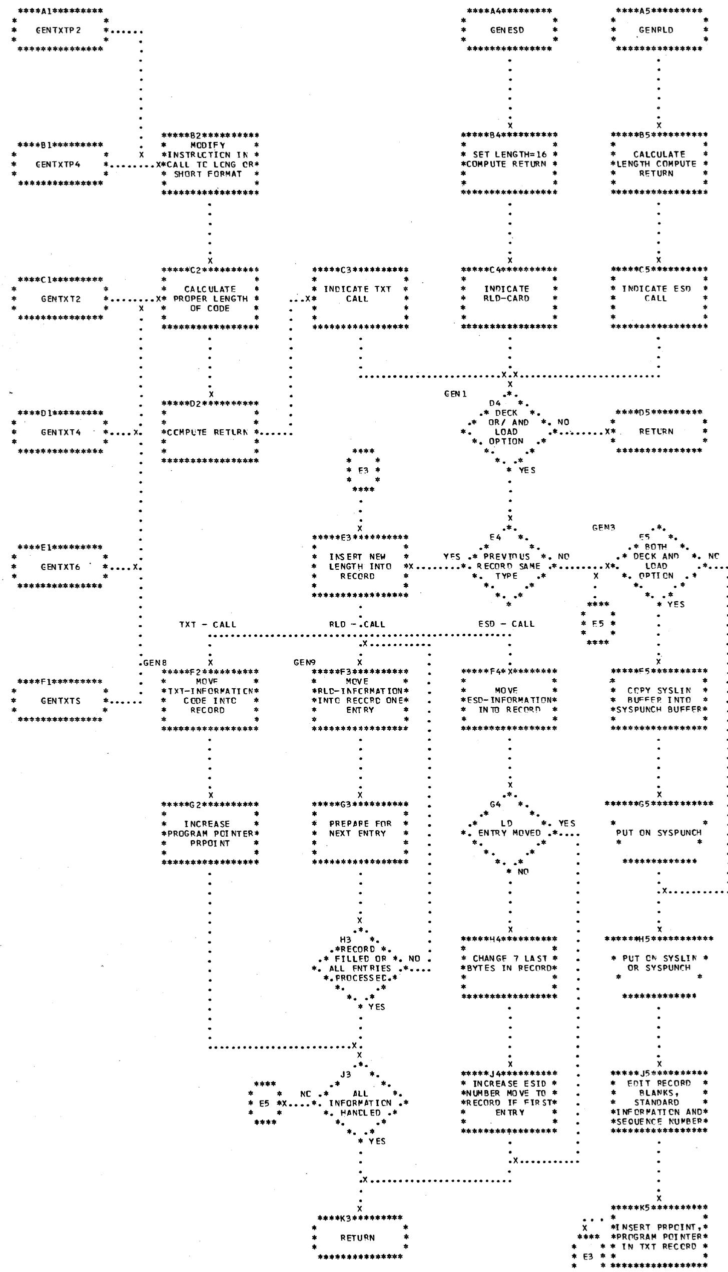


Chart 123: Compilation Phase - IEX50 Microfiche IEX50-1

Error Reading

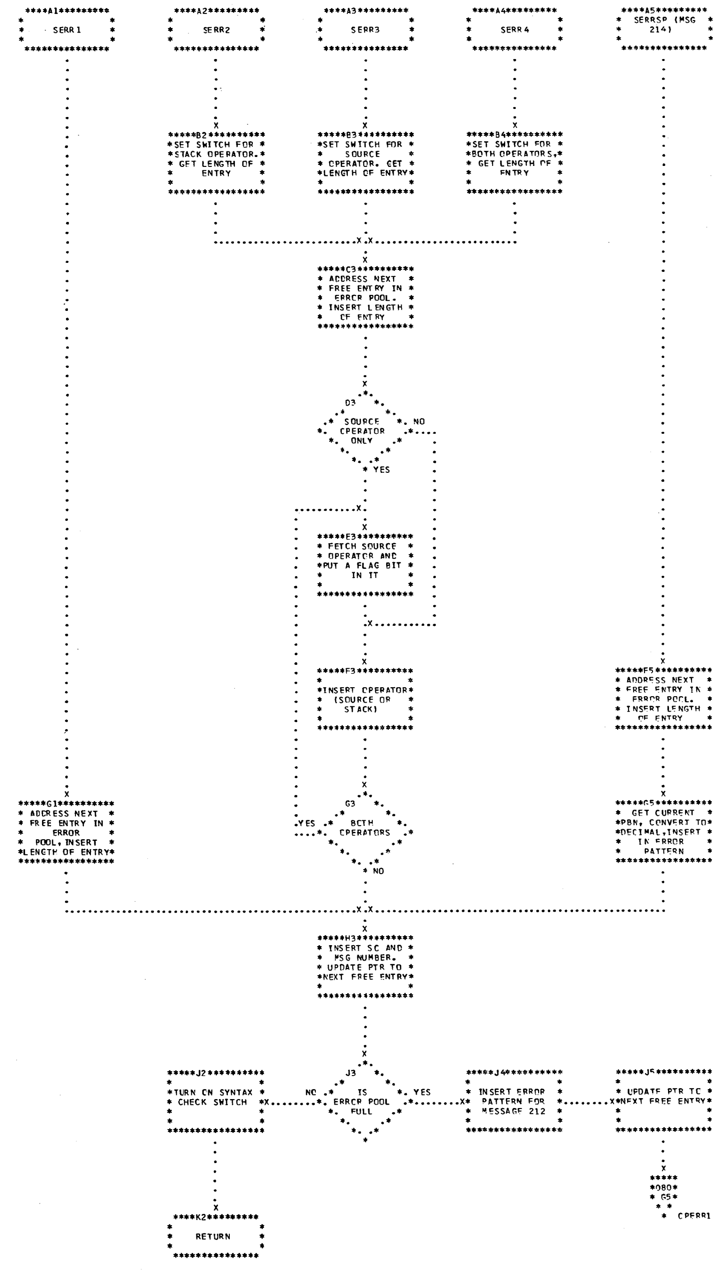


Chart 124: Termination Phase - IEX51 Overall Flow

Microfiches IEX51-1, IEX51-2 and IEX51M-1

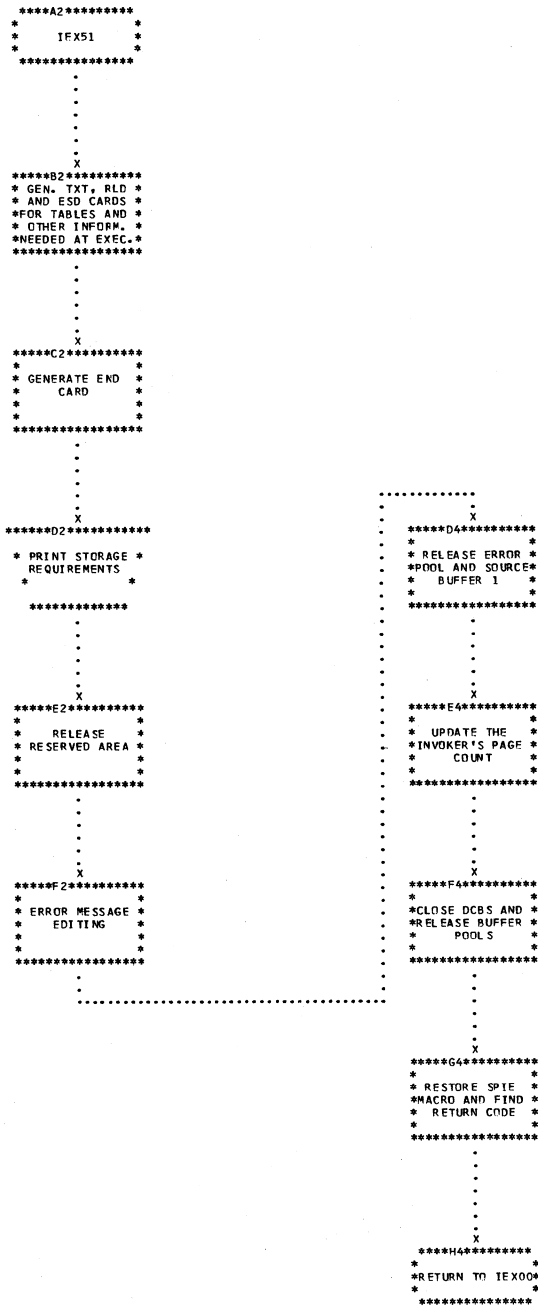


Chart 125: Termination Phase - IEX51 Object Time Table Output

Microfiche IEX51-1

Chart 126: Termination Phase - IEX51 Main Storage Release and Close of Data Sets

Microfiche IEX51-2

Chart 127: Termination Phase - IEX51 Microfiche IEX51-1

Print Storage Requirements

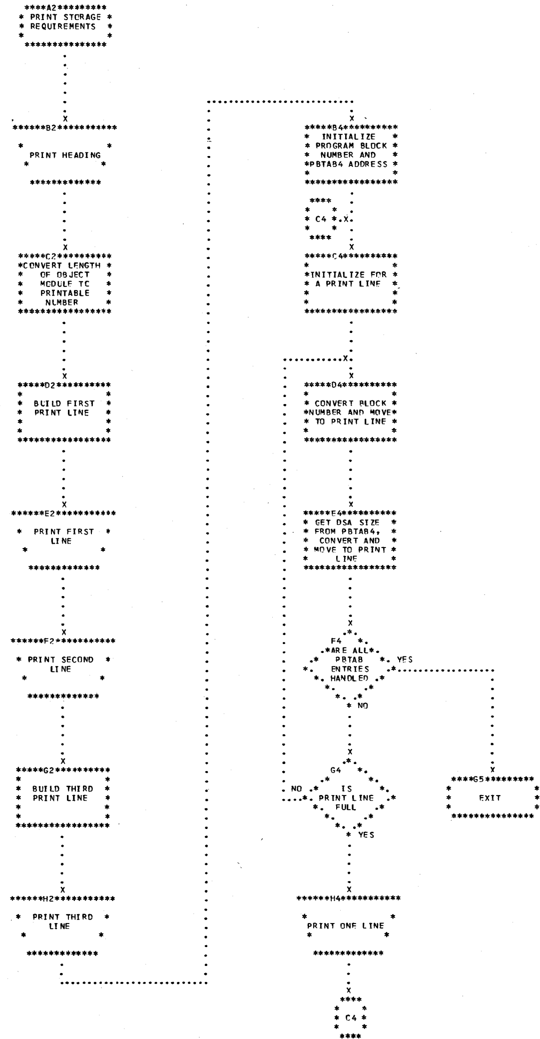


Chart 128: Termination Phase - IEX51 Microfiches IEX51-1 and IEX51M-1

Overall Flow: Error Message Editing Routine

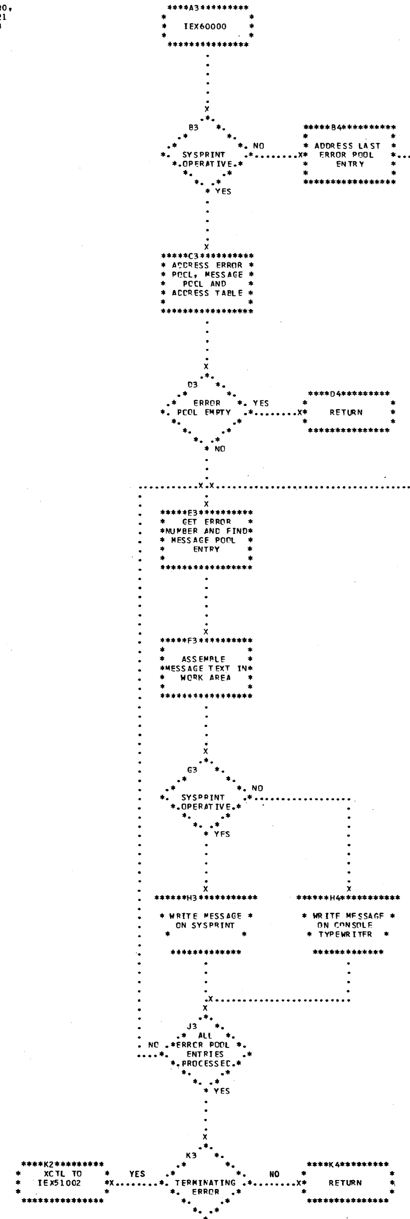


Chart 129: ALGOL Library
Overall Flow

Microfiches,
all starting with IHI

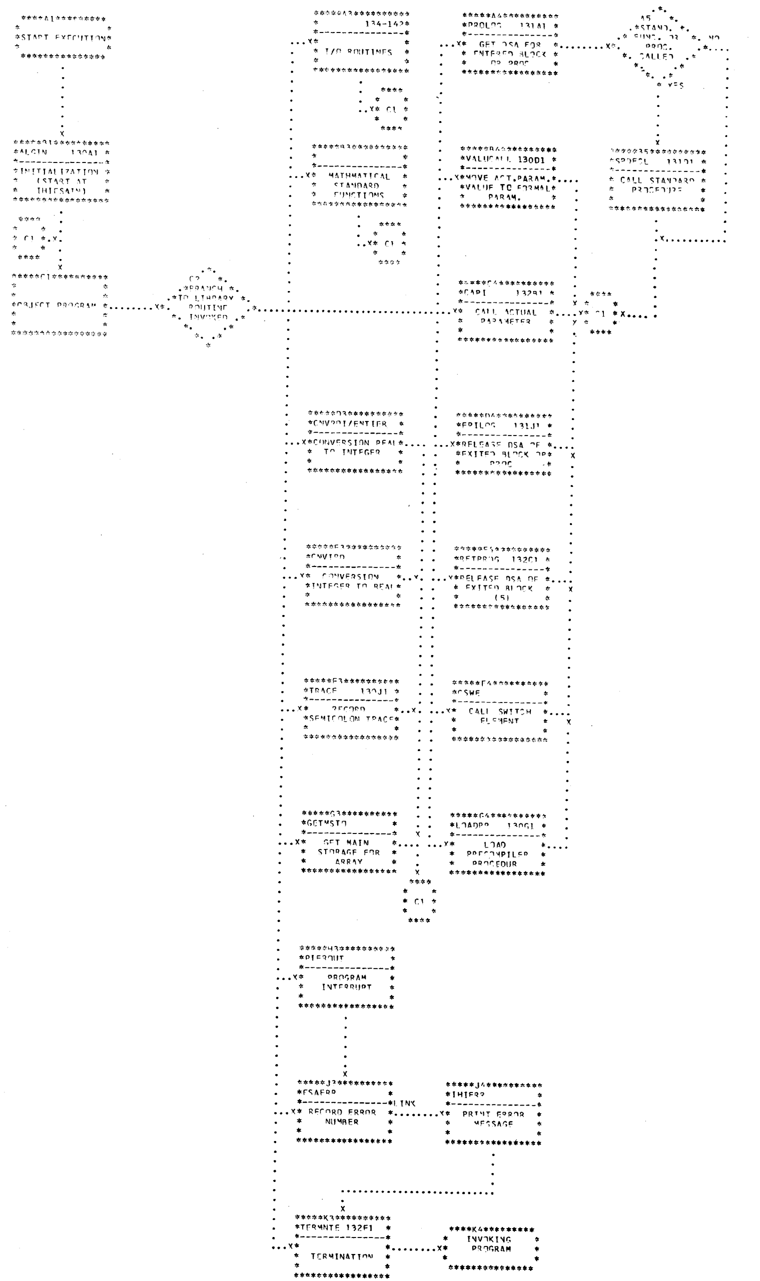


Chart 130: ALGOL Library
ALGIN, VALUCALL, LOADPP and TRACE

Microfiche IHIFSA-1
(ALGIN: IHIFSA-2)

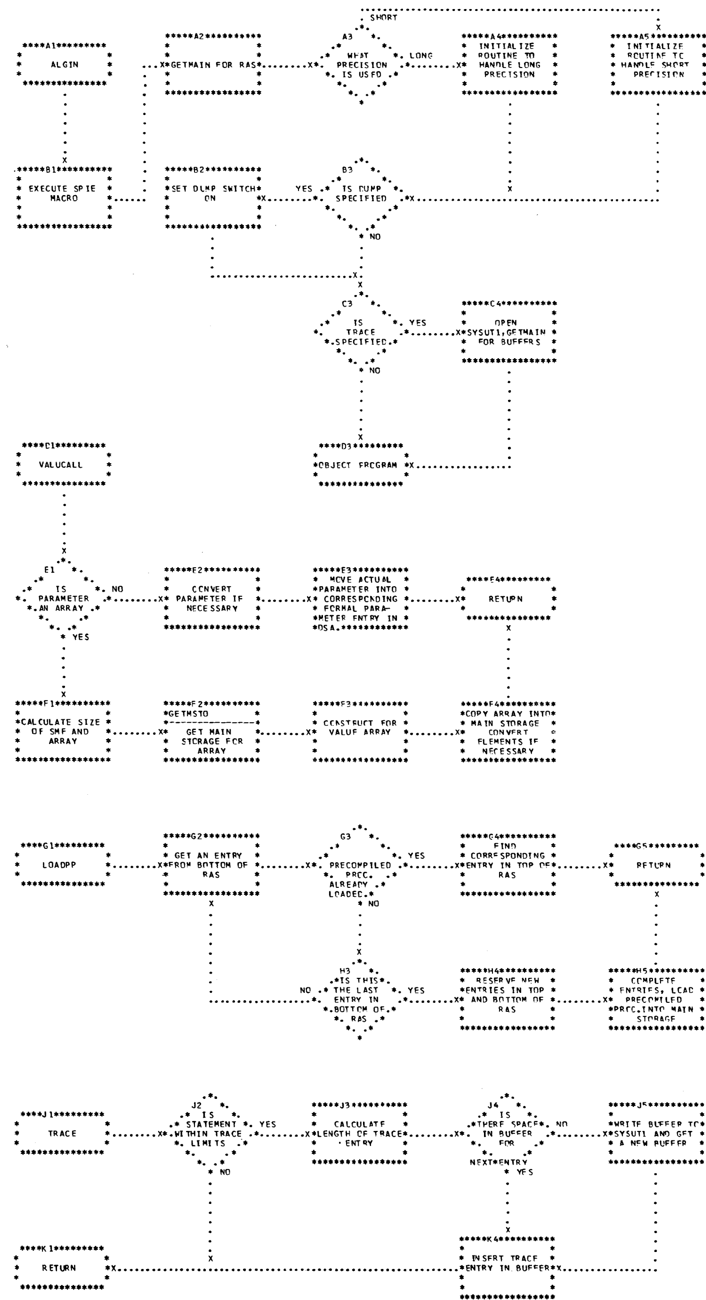


Chart 131: ALGOL Library — PROLOG, SPDECL and EPILOG

Microfiche IHIFSA-1

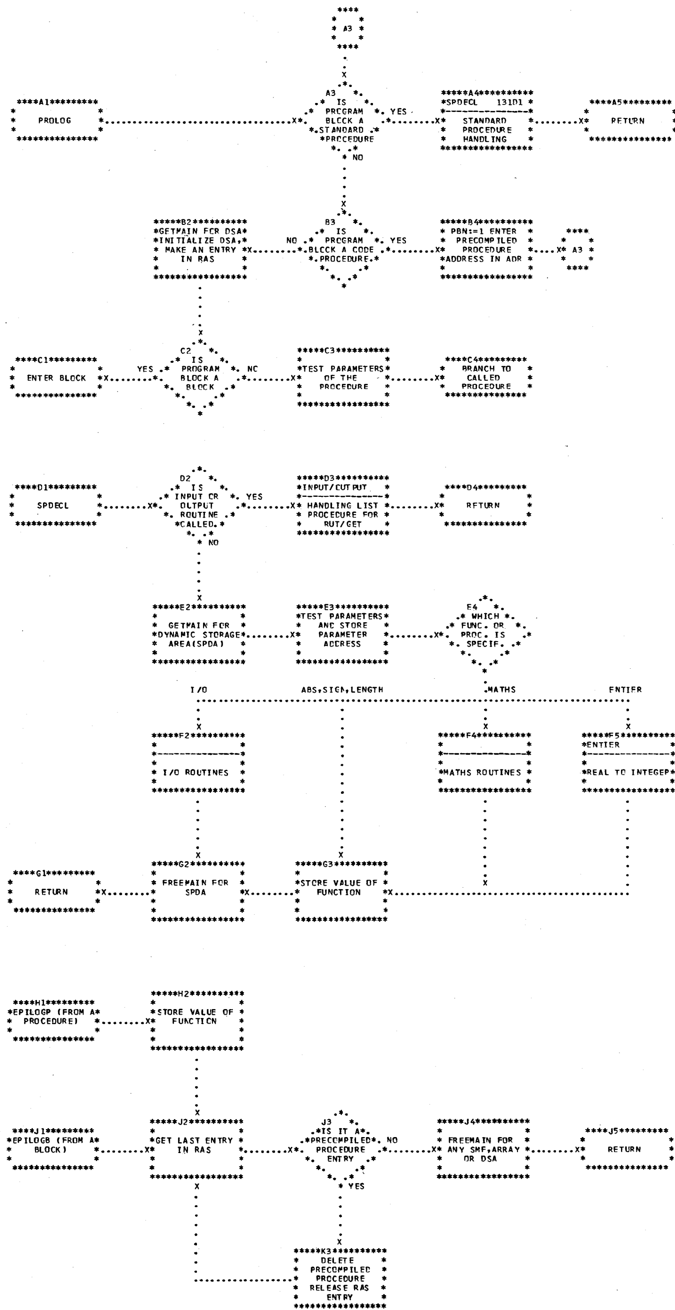


Chart 132: ALGOL Library — CAP1, CAP2 and TERMNTE

Microfiche IHIFSA-1 (TERMNTE: IHIFSA-2)

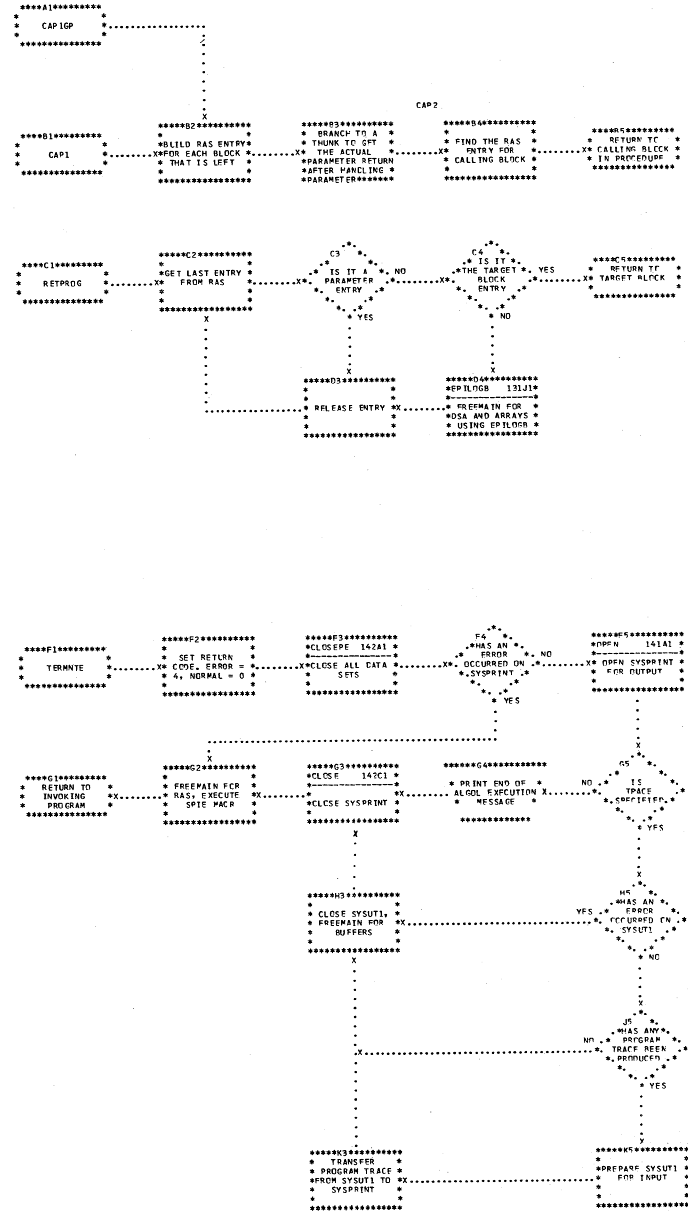
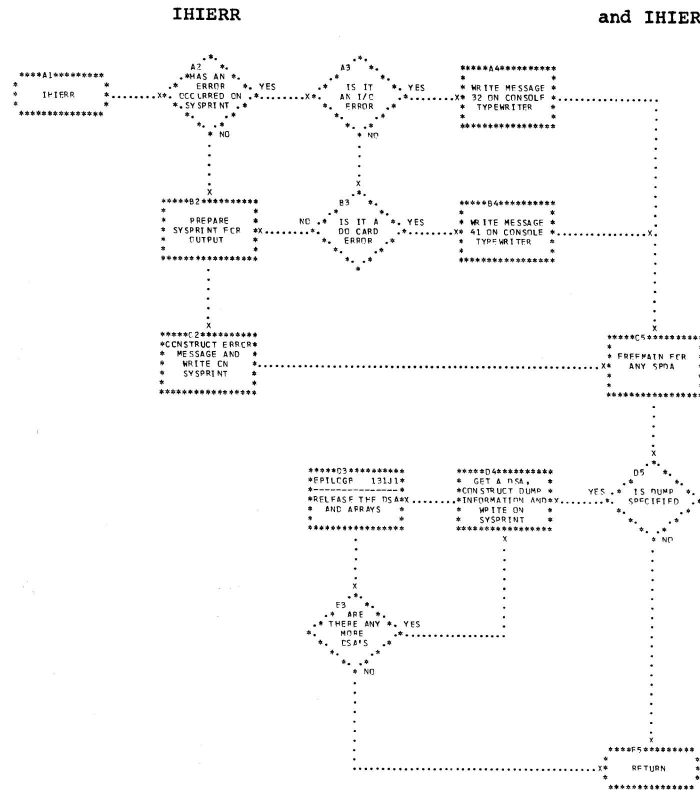


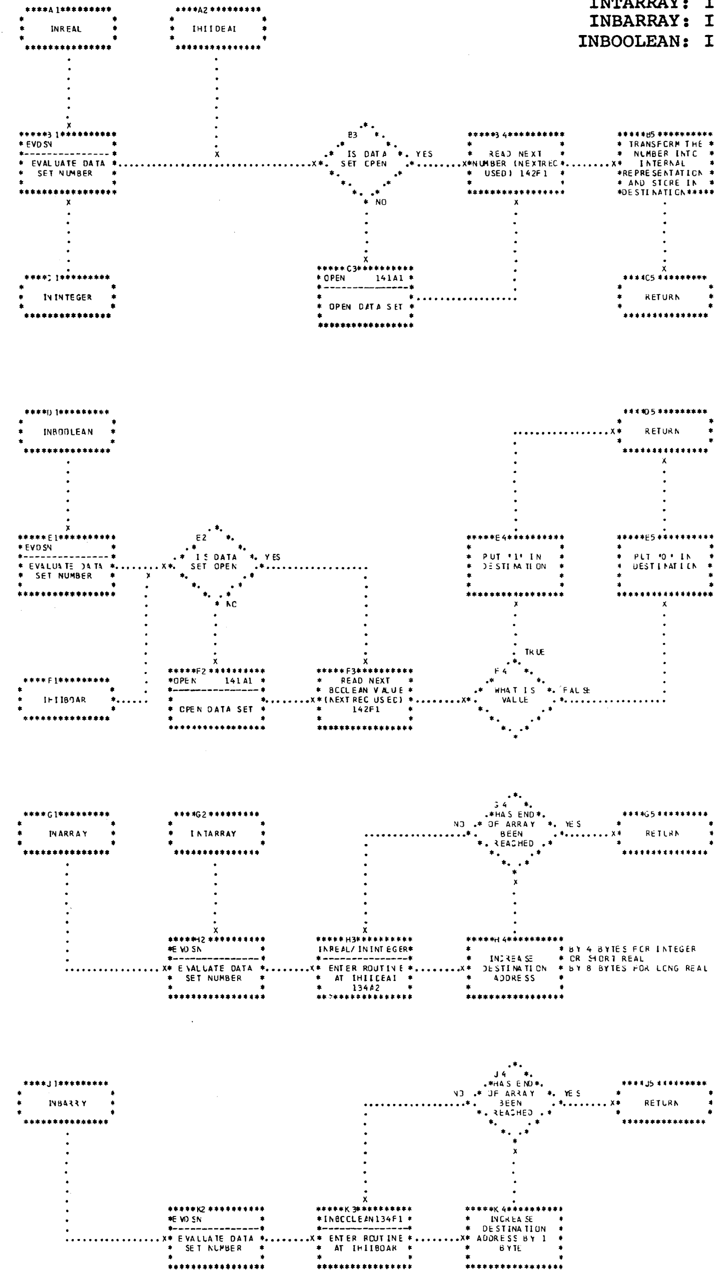
Chart 133: ALGOL Library IHIERR

Microfiches IHIERM-1 and IHIERR-1



● **Chart 134:** ALGOL Library INREAL, ININTEGER, INBOOLEAN, INARRAY, INTARRAY and INBARRAY

Microfiches (INREAL and ININTEGER: IHIIDE-1, INARRAY and INTARRAY: IHIIAR-1, INBARRAY: IHIIBA-1, INBOOLEAN: IHIIBO-1)



Form Y33-8000-0, Page Revised by TNL Y33-8003, 6/19/68

●**Chart 135**: ALGOL Library
INSYMBOL, OUTREAL, OUTINTEGER,
OUTBOOLEAN, OUTARRAY and OUTTARRAY

Microfiches
(INSYMBOL: IHIISY-1,
OUTREAL: IHILOR-1,
OUTINTEGER: IHIOIN-1,
OUTBOOLEAN: IHIOBO-1,
OUTARRAY: IHIOAR-1,
OUTTARRAY: IHIOTA-1)

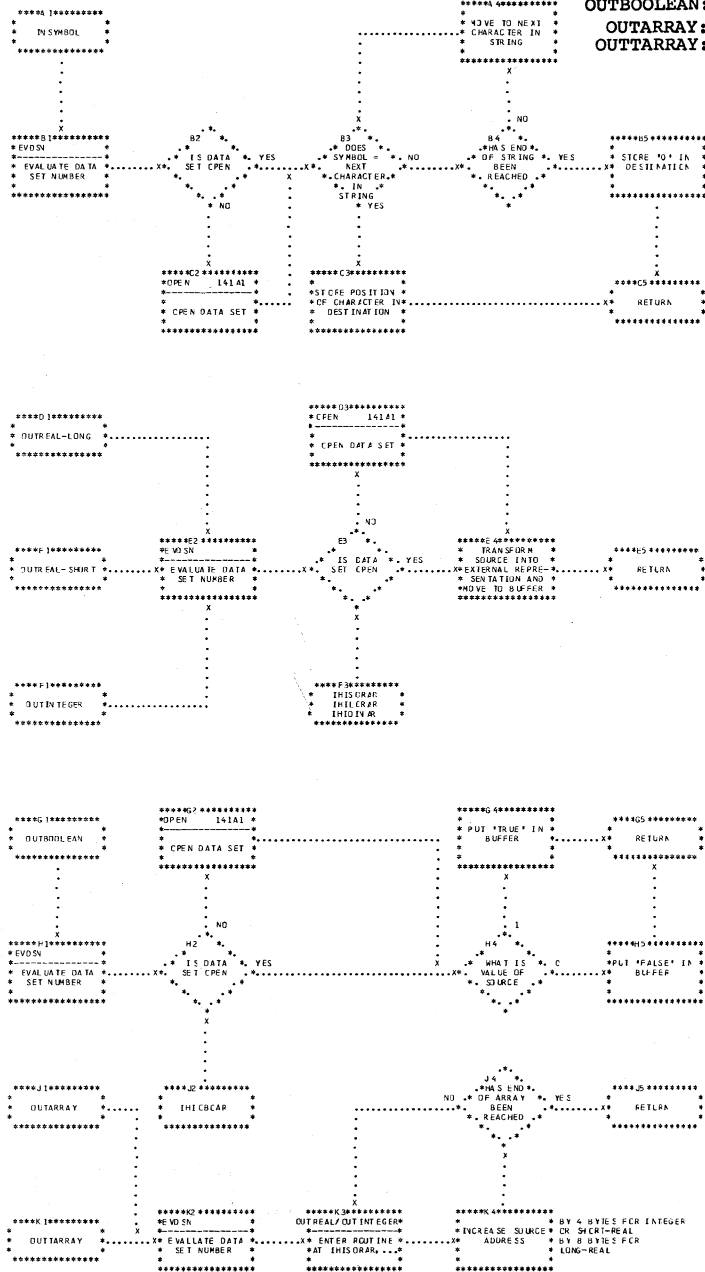
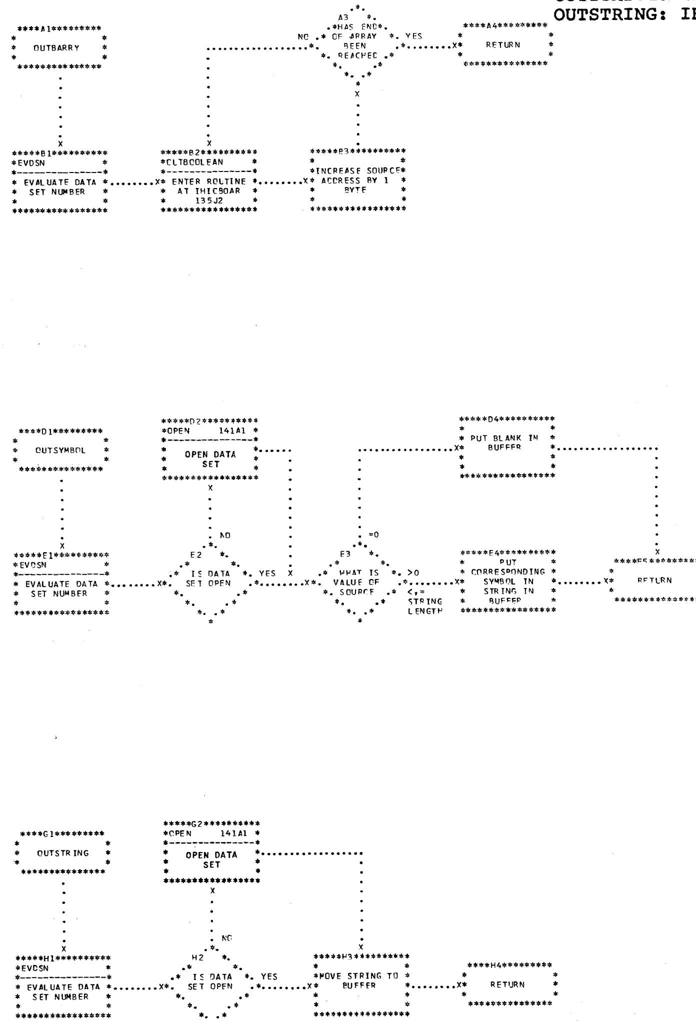


Chart 136: ALGOL Library
OUTBARRAY, OUTSYMBOL
and OUTSTRING

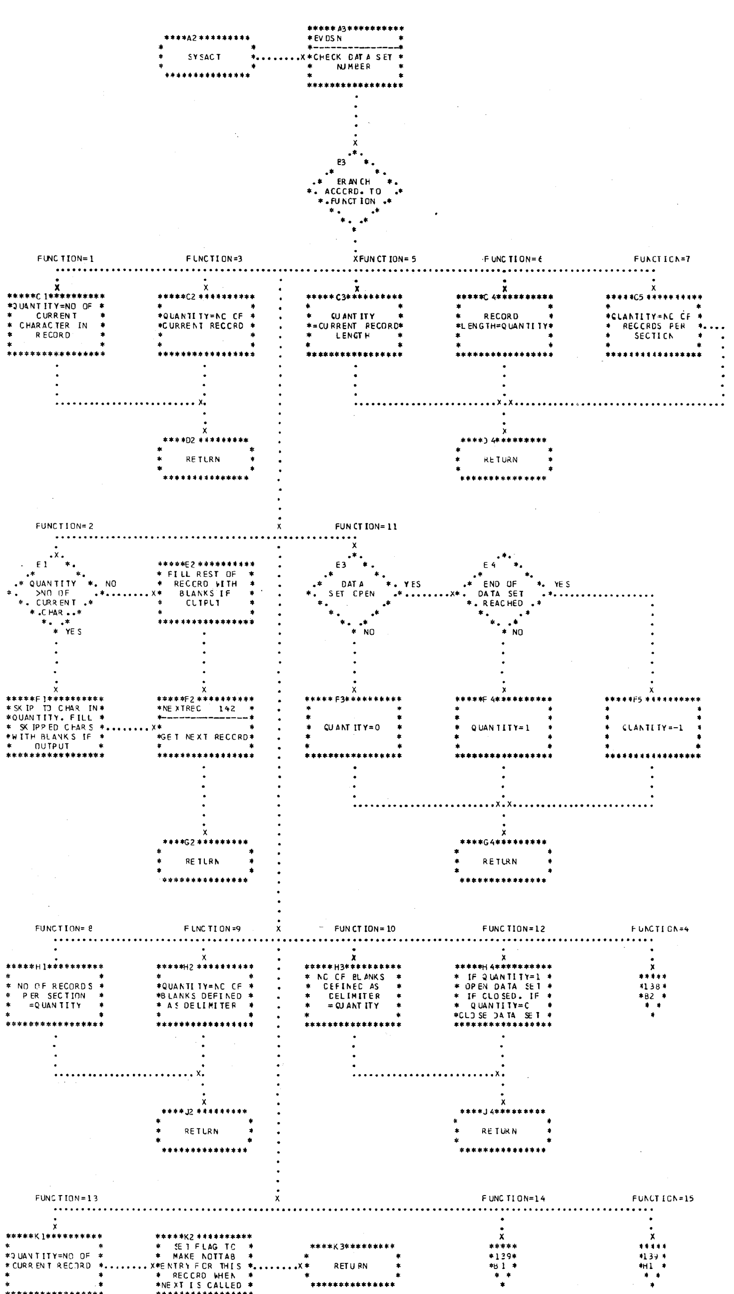
Microfiches
(OUTBARRAY: IHIOBA-1,
OUTBARRAY: IHIOBA-1,
OUTSYMBOL: IHIOSY-1,
OUTSTRING: IHIOST-1)



Form Y33-8000-0, Page Revised by TNL Y33-8003, 6/19/68

•Chart 137: ALGOL Library SYSACT

Microfiche IHISYS-1



•Chart 138: ALGOL Library SYSACT (cont'd)

Microfiche IHISYS-1

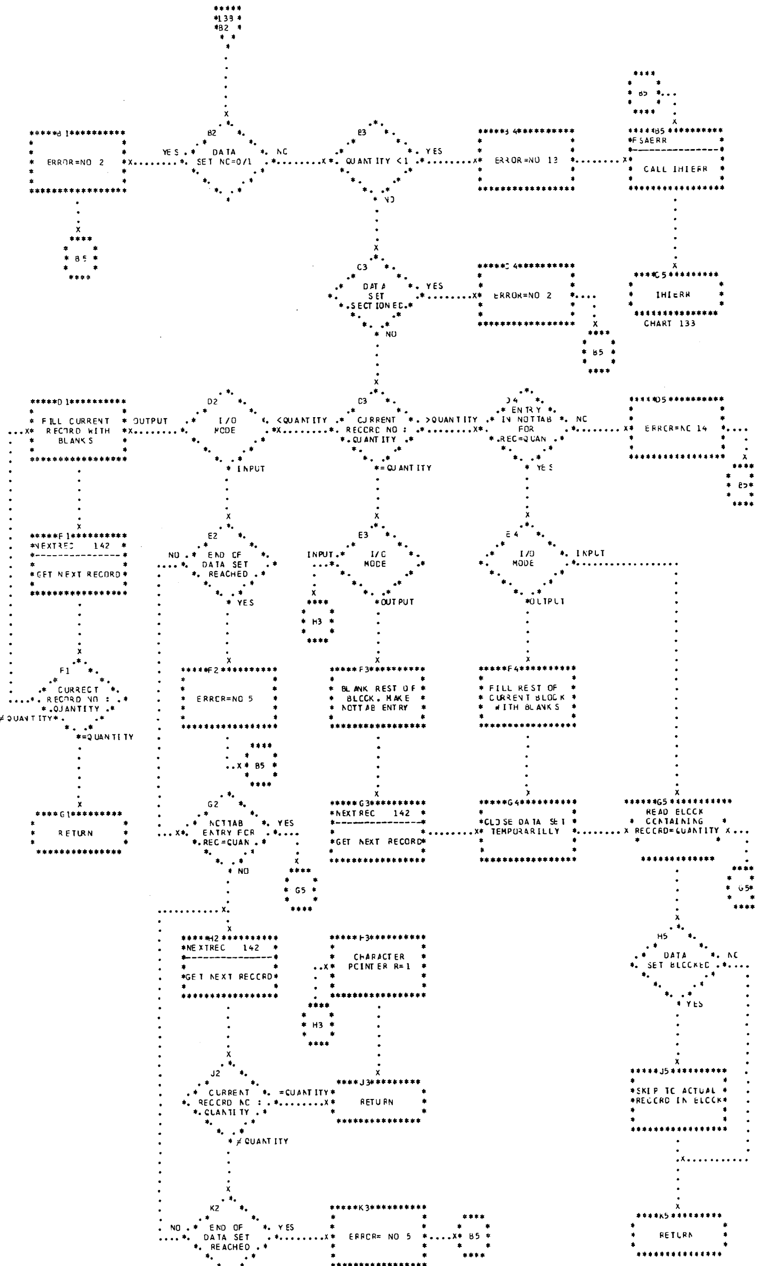


Chart 139: ALGOL Library — Microfiche IHISYS-1

SYSACT (cont'd)

- 139 B1
- B1: ADD VALUE OF CURRENT RECORD POINTER TO QUANTITY
- A5: SKIP TO RECORD SPECIFIED BY QUANTITY
- B2: WHAT IS CURRENT I/O MODE — INPUT ONLY; FIXED; INPUT
- C1: IS DATA SET SECTIONED — NO OR YES WHEN NO. OF RECORDS/SEC. >= QUANTITY; YES, WHEN NO. OF RECORDS/SECTION < QUANTITY
- C2: WHAT WAS LAST I/O OPERATION — OUTPUT
- C3
- C3: FILL REST OF CURRENT RECORD WITH BLANKS
- C4: NEXTREC 142F1 — GET NEXT RECORD
- D1: FILL REST OF CURRENT RECORD WITH BLANKS AND SKIP TO FIRST RECORD IN NEXT SECTION
- D4: WHAT IS NUMBER OF CURRENT RECORD — NOT= QUANTITY; =QUANTITY
- E4: RETURN
- 139 H1
- H1: IS DATA SET SECTIONED — NO (B1); YES
- H3: RETURN
- J1: FILL REST OF CURRENT RECORD WITH BLANKS AND SKIP TO FIRST RECORD IN NEXT SECTION
- J3: DOES QUANTITY = NO. OF CURRENT RECORD — YES; NO (C3)

Chart 140: ALGOL Library — Microfiche IHIGPR-1

PUT, GET, OUTPUT and INPUT

- B1: PUT
- A2: OPENGP 141J1 — OPEN SYSUT2
- B2: IS DATA SET OPEN — NO; YES
- B3: WHAT WAS LAST OPERATION — GET; PUT
- B4: REPOSITION DATA SET TO ACCEPT NEW RECORD
- C3: CHECK LAST I/O OPERATION
- C4: ENTRYNOTTAB — MAKE ENTRY IN NOTTAB
- C5: DELETE ANY EXISTING ENTRY IN NOTTAB WITH SAME IDENTIF. NUMBERS
- D3: RETURN
- D4: WRITE THE RECORD ON SYSUT2
- D5: PROLOG 131A1 — ENTER LIST PROCEDURE
- E1: GET
- E2: WHAT WAS LAST OPERATION — GET; PUT
- F2: CHECK LAST I/O OPERATION
- F3: ENTRYNOTTAB — MAKE ENTRY IN NOTTAB
- F4: READ THE RECORD FROM SYSUT2, AND CHECK THE OPERATION
- G4: PROLOG 131A1 — ENTER LIST PROCEDURE
- G5: RETURN
- J1: OUTPUT
- J2: CAPIGP 132A1 — CALL ACTUAL PARAMETER
- J3: TRANSFER PARAMETER TO OUTPUT BUFFER
- J4: RETURN
- K1: INPUT
- K2: CAPIGP 132A1 — CALL ACTUAL PARAMETER
- K3: TRANSFER DATA FROM INPUT BUFFER TO PARAMETER
- K4: RETURN

Chart 141: ALGOL Library — Microfiche IHIIOR-1

OPEN, DCEXIT and OPENGP

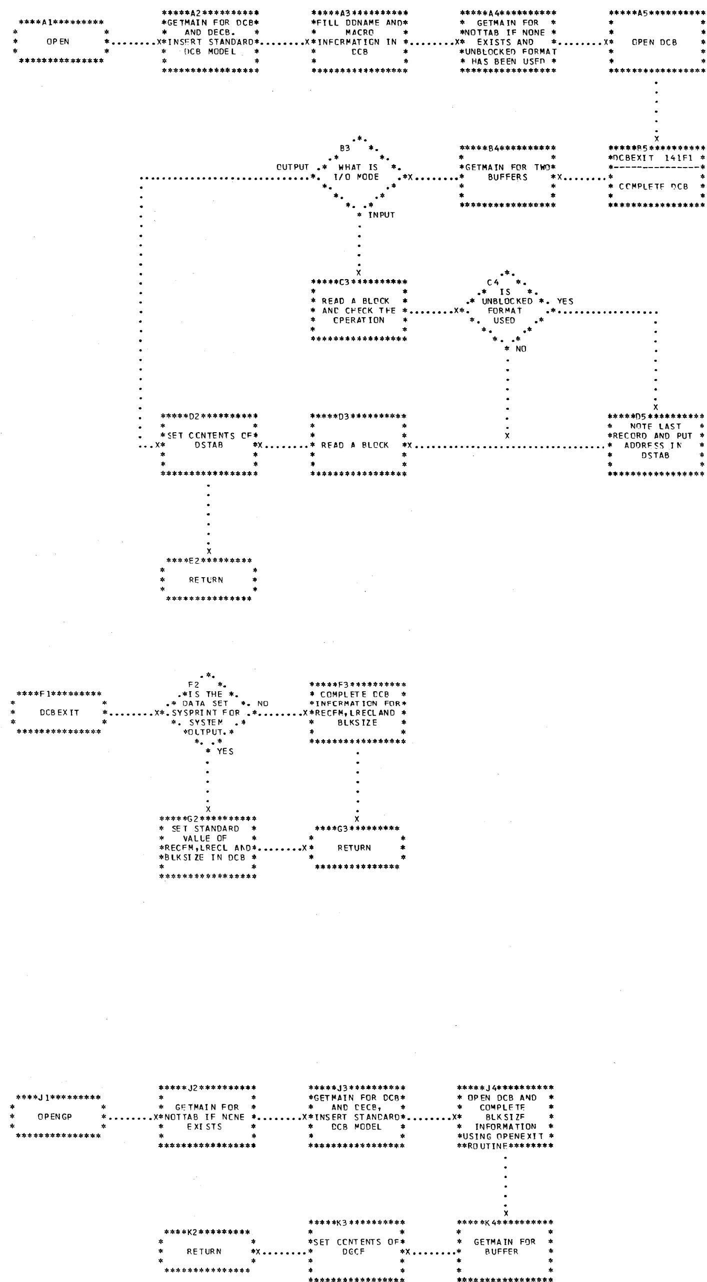
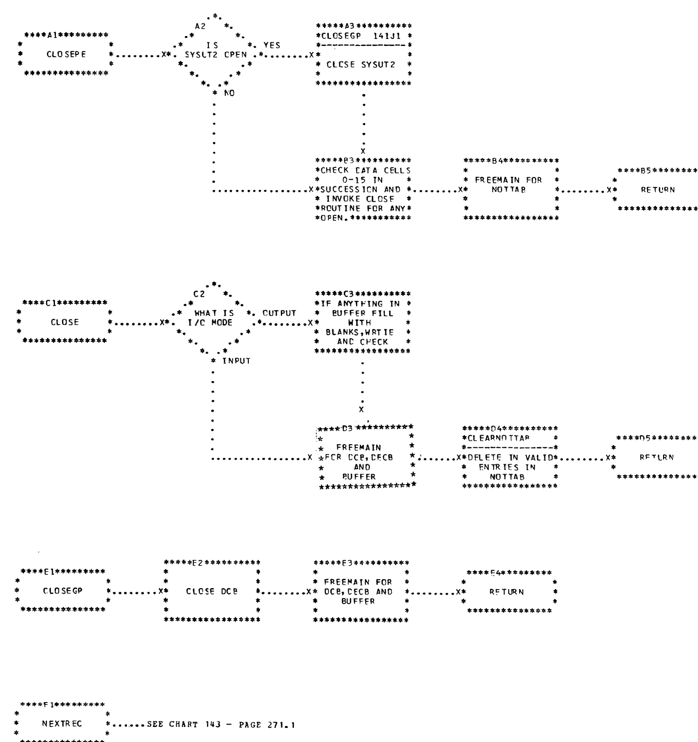


Chart 142: ALGOL Library — Microfiche IHIIOR-1

CLOSEPE, CLOSE, CLOSEGP and NEXTREC



NEXTREC ......SEE CHART 143 - PAGE 271.1

Form Y33-8000-0, Page Revised by TNL Y33-8003, 6/19/68

•Chart 143: ALGOL Library — Microfiche IHIOR-1

NEXTREC

A2: NEXTREC

B2: I/O MODE — OUTPUT / INPUT / MIXED

B3: NOTTAB ENTRY NEEDED — NO / YES

C2: POINT TO CORRECT BLOCK

C3: MAKE ENTRY IN NOTTAB

D1: CURRENT RECORD LAST IN BLOCK — NO / YES

D2: DATA SET BLOCKED — YES / NO

D3: DATA SET BLOCKED — NO / YES

E1: SET POINTERS TO NEXT RECORD IN BLOCK

E2: WRITE BLOCK CHANGE BUFFERS

E3: CURRENT RECORD LAST IN BLOCK — NO / YES

E4: CURRENT RECORD THE LAST IN SECTION — NO / YES

F1: NOTTAB ENTRY NEEDED — NO / YES

F2: HAS BACKWARD REPOSITIONING OCCURRED — NO / YES

F3: NOTE CURR REC IN NOTTAB. READ NEXT BLOCK. CHANGE BUFFERS

F4: SET POINTER S=C SET POINTER S=C INSERT CONTROL CHARACTER=C '1'

F5: INSERT CONTROL CHARACTER=C ' '

G1: WRITE BLOCK. I/O MODE MIXED

G2: CLEAR ALL NOTTAB ENTRIES FOR REC NO > CURRENT REC NO

G3: SET POINTERS TO NEXT RECORD IN BLOCK

G4: INCREASE CHARACTER POINTER BY ONE

H2: NOTTAB ENTRY FOR CURR REC NEEDED — NO / YES

H3: INCREASE RECORD POINTERS BY ONE

J2: NOTE CURR REC NO. ENTER IN NOTTAB

J3: RETURN

K2: DATA SET SECTIONED — NO (→ H3) / YES (→ E4)

APPENDICES

APPENDIX I-a - Source Module Character Set after initial translation of the source text in the Scan I/II Phase. See CIB subroutine

Second Digit (Hexadecimal)	First Digit (Hexadecimal) 0	1	2	3	4	5
0	+	=	¬ (Not)	0	A	Q
1	-	<		1	B	R
2	*	>	| (Or)	2	C	S
3	/		& (And)	3	D	T
4				4	E	U
5			, (Comma)	5	F	V
6	(		)	6	G	W
7	:			7	H	X
8				8	I	Y
9				9	J	Z
A					K	
B	;		b (Blank)		L	
C			⊗ (Invalid character)		M	
D			. (Point)		N	
E			' (Apos-trophe)		O	
F			Zeta		P	

Notes:

- The invalid character (X'2C') is assigned to any character in the source module which is not represented in the standard character set (EBCDIC or ISO) specified in the code option.
- The character Zeta (X'2F') is inserted at the end of every input record, after the initial translation to internal code.

APPENDIX I-b - Character Set of the Modification Level 1 source text output by the Scan III Phase. The character set does not include five-byte internal names representing logical constants and character strings stored in the Constant Pool. See Appendix II.

Second Digit (Hexadecimal)	First Digit (Hexadecimal) 0	1	2	3	4	5
0	+	=	Not	0	A	Q
1	-	<	Impl	1	B	R
2	*	>	Or	2	C	S
3	/	≠	And	3	D	T
4	÷	≤	Equiv	4	E	U
5	Power	≥	Comma	5	F	V
6	(	Assign	)	6	G	W
7	Colon	Goto	Label Colon	7	H	X
8	[	For	]	8	I	Y
9	Array	Step	Delta	9	J	Z
A	Switch	Until	Epsilon		K	
B	Semi-Colon	While	Eta		L	
C	Begin	Do	End	Gamma	M	
D	Beta	If	Omega	Rho	N	
E	Pi	Then	Apos-trophe	Deci-mal Point	O	
F	Phi	Else	Zeta	Scale Factor	P	

APPENDIX I-c - Character Set of the Modification Level 2 source text output by the Scan III Phase. The character set does not include five-byte internal names representing identifiers, constants, logical constants and standard procedure designators. See Appendices II and III.

Second Digit (Hexadecimal)	First Digit (Hexadecimal) 0	1	2	3	4	5
0	+	=	Not			
1	-	<	Impl			
2	*	>	Or			
3	/	≠	And			
4	÷	≤	Equiv			
5	Power	≥	Comma			
6	(	Assign	)			
7	Colon	Goto	Label Colon			
8	[	For	]			
9	Array	Step	Delta			
A	Switch	Until	Epsilon			
B	Semi-Colon	While	Eta			
C	Begin	Do	End			
D	Beta	If	Omega			
E	Pi	Then	Apos-trophe			
F	Phi	Else	Zeta			

• Character Set representing operators entered in the Operator Stack in the Compilation Phase.

Second Digit (Hexadecimal)	First Digit (Hexadecimal) 0	1	2	3	4	5
0	+	=	Not	PRC		
1	-	<	Impl			
2	*	>	Or			
3	/	≠	And	STC		
4	÷	≤	Equiv			
5	Power	≥	Alpha			
6	(	Assign	If-s			
7	Colon	Goto	Then-s			
8	[	For	Else-s			
9	Array	Step	{			
A	Switch	Until	⟨			
B	Semi-Colon	While	Monadic Minus			
C	Begin	Do	⁅			
D	Beta	If	For:=			
E	Pi	Then	Switch:=			
F	Phi	Else		EXC		

Explanation of Operators

Operator	Explanation
Operators originating in the Modification Level 2 Source Text:	
+	Addition
-	Subtraction
*	Multiplication
/	Division with real quotient
÷	Division with integer quotient
Power	Power operator
(	Left parenthesis
Colon	Colon in the bound pair list of an array declaration
[	Left bracket opening a subscript expression
Array	Array declarator
Switch	Switch declarator
Semicolon	Semicolon terminating a statement
Delta	Semicolon terminating a declaration
Begin	Delimiter opening a compound statement
Beta	Delimiter opening a block
Pi	Delimiter opening a procedure
Phi	Delimiter opening a type-qualified procedure
=	Equal to
<	Less than
>	Greater than
≠	Not equal to
≤	Not greater than
≥	Not less than
Assign	Assignment to
Goto	Delimiter specifying a branch
For	Delimiter opening a for statement
Step	Separator in a for list
Until	" " " " "
While	" " " " "
Do	Delimiter terminating a for list
If	Operator opening an if clause

Operator	Explanation
Then	Operator terminating an if clause
Else	Sequential operator in a conditional statement or expression
Not	Negation
Impl	Implication
Or	Alternativity
And	Conjunction
Equiv	Equivalence
Special operators originating in the compiler programs:	
Alpha	Designates the bottom of the Operator Stack
If-s	If opens a conditional statement
Then-s	Then belonging to a conditional statement
Else-s	Else belonging to a conditional statement
{	Parenthesis opening an actual parameter list in a procedure statement or a function designator
⟨	Parenthesis opening an actual parameter list in a standard procedure statement or function designator
Monadic Minus	Minus sign which specifies the negative quality of an operand (as opposed to the operation of subtraction)
⁅	Bracket opening a bound pair list in an array declaration
For:=	Specifies the start of a for list in a for statement
Switch:=	Specifies the start of a switch list in a switch declaration
PRC	Specifies a switch to the Program Context Matrix
STC	Specifies a switch to the Statement Context Matrix
EXC	Specifies a switch to the Expression Context Matrix

APPENDIX II: INTERNAL REPRESENTATION OF OPERANDS

Internal representation of identifiers, constants, strings, and logical values in the Modification Level 2 text. (In the Modification Level 1 text, only strings and logical values are replaced by internal names: identifiers and constants are unchanged from their external representation, except that they are translated to the internal code in Appendix I-b.)

Type	Internal Name (5 bytes)			
	<Characteristic >	<P.B. No.> or <C.P. No.>	(1/2 byte)	<Displacement > (1 1/2 bytes)
	Bytes 0 and 1	Byte 2	Bytes 3 and 4	
I. IDENTIFIERS				
Declared Identifiers	(Hexadecimal)			
REAL	CO 32	<P.B.No >		< Displacement >
INTEGER	CO 31	"		"
BOOLEAN	CO 33	"		"
ARRAY	C8 06	"	< NOS>	"
" , real	C8 06	"	< NOS>	"
" , integer	C8 05	"	< NOS>	"
" , boolean	C8 07	"	< NOS>	"
LABEL	CC 08	"		< Label No.>
SWITCH	CC 0C	"	< NOC>	"
PROCEDURE	CA C0	"	< NOP>	"
" , integer	CA C1	"	< NOP>	"
" , real	CA C2	"	< NOP>	"
" , boolean	CA C3	"	< NOP>	"
CODE PROCEDURE	CA 40	"	< NOP>	"
" " , integer	CA 41	"	< NOP>	"
" " , real	CA 42	"	< NOP>	"
" " , boolean	CA 43	"	< NOP>	"
Specified Identifiers called by name				
REAL	C2 12	"		< Displacement >
INTEGER	C2 11	"		"
BOOLEAN	C2 13	"		"
ARRAY	CA 16	"		"
" , integer	CA 15	"		"
" , real	CA 16	"		"
" , boolean	CA 17	"		"
LABEL	CA 18	"		"
SWITCH	CA 1C	"		"
PROCEDURE	CA D0	"		"
" , integer	CA D1	"		"
" , real	CA D2	"		"
" , boolean	CA D3	"		"
STRING	CB 10	"		"
Specified Identifiers called by value				
REAL	C2 22	"		"
INTEGER	C2 21	"		"
BOOLEAN	C2 23	"		"
ARRAY	CA 26	"		"
" , real	CA 26	"		"
" , integer	CA 25	"		"
" , boolean	CA 27	"		"
LABEL	CA 28	"		"
PROCEDURE, integer	C2 E1	"		"
" , real	C2 E2	"		"
" , boolean	C2 E3	"		"
II. CONSTANTS				
INTEGER CONSTANT	C8 01	<C.P. No.>		< Displacement >
REAL CONSTANT	C8 02	"		"
III. CHARACTER STRINGS				
CHARACTER STRINGS	C9 00	00		"
IV. LOGICAL VALUES				
TRUE	C8 03	00		00 07
FALSE	C8 03	00		00 00
All-purpose internal name (replaces an invalid identifier or a defective constant)	91 FF	01		00 00

Key:

<Characteristic > = (See Figure 10, Chapter 4)

<Displacement > =

(Identifiers) <Displacement in the object time Data Storage Area specified by the Program Block Number >

(Numerical and logical constants and strings) < Displacement in the Constant Pool specified by the Constant Pool Number >

<Label No.> = <Displacement in the object time Label Address Table >

<NOC> = <Number of Components - 1 >

<NOP > = <Number of Parameters >

<NOS > = <Number of Subscripts - 1>

<P.B.No.> = <Program Block Number of the block or procedure in which the identifier was declared or specified >

<C.P.No.> = <Constant Pool Number>

APPENDIX III: INTERNAL REPRESENTATION OF STANDARD PROCEDURE DESIGNATORS

- Internal representation of standard procedure designators in the Modification Level 2 text output by the Scan III Phase and input to the Compilation Phase.

Standard Procedure Designator	Internal Name (5 bytes) - Hexidecimal						
	<Characteristic>		<Character of formal parameters>			<Function>	<Displacement of entry in Label Address Table>*
			3rd param.	2nd param.	1st param.		
	Bytes 0 and 1		Byte 2		Byte 3		Byte 4
Standard Mathematical Functions							
ABS	88	82	0	0	2	0	80
ARCTAN	88	82	0	0	2	0	10
COS	88	82	0	0	2	0	0C
ENTIER	88	81	0	0	2	0	F0
EXP	88	82	0	0	2	0	18
LENGTH	88	81	0	0	0	0	E0
LN	88	82	0	0	2	0	14
SIGN	88	81	0	0	2	0	C0
SIN	88	82	0	0	2	0	08
SQRT	88	82	0	0	2	0	04
Standard I/O Procedures							
INARRAY	88	80	0	E	1	8	2E
INBARRAY	88	80	0	7	1	8	36
INBOOLEAN	8A	80	0	B	1	8	2A
ININTEGER	8A	80	0	9	1	8	26
INREAL	8A	80	0	A	1	8	22
INSYMBOL	8A	80	9	0	1	8	1F
INTARRAY	88	80	0	D	1	8	32
OUTARRAY	88	80	0	6	1	4	4A
OUTBARRAY	88	80	0	7	1	4	52
OUTBOOLEAN	88	80	0	3	1	4	46
OUTINTEGER	88	80	0	1	1	4	42
OUTREAL	88	80	0	2	1	4	3E
OUTSTRING	88	80	0	0	1	4	56
OUTSYMBOL	88	80	1	0	1	4	3B
OUTTARRAY	88	80	0	5	1	4	4E
GET	8A	80	0	4	1	1	5E
PUT	8A	80	0	4	1	1	5A
SYSACT	8A	80	9	1	1	2	03

*The last two bits of byte 4 indicate the number of parameters required by the standard I/O procedure or function.

Key: <Characteristic> =

Byte 0: X'88' <Operand, no assignment>
X'8A' <Operand, >

Byte 1: X'80' <I/O procedure>
X'81' <Standard function, integer type>
X'82' <Standard function, real type>

<Character of Formal Parameters> =

Binary 0000: <String>
" 0100: <Procedure (in PUT/GET)>
xx10: <Real>
xx01: <Integer>
xx11: <Not array>
0xxx: <Formal and actual parameters must be compatible>
1xxx: <Formal and actual parameters must be identical in type>

<Function> =

Byte 3-Bit 4=1: <Input procedure>
Bit 5=1: <Output procedure>
Bit 6=1: <SYSACT>
Bit 7=1: <PUT or GET>

Note

All standard I/O procedures and mathematical functions (except ABS, ENTIER, LENGTH and SIGN) are represented by individual load modules in the ALGOL Library. At linkage edit time, the Library modules corresponding to the I/O procedures and mathematical functions invoked in the source module, are loaded into main storage with the generated object program. The entry point addresses of the loaded modules are stored in the Label Address Table. Figure 85 shows the Label Address Table entry in which each entry point address is stored.

APPENDIX IV: COMPILE CONTROL FIELD (HCOMPMOD)

Byte No.	Bit No.	Switch Name	Bit Value	Significance
0	0	CMT	0	Compiler in normal compilation mode.
			1	Compiler in syntax check mode. An error with severity code S (serious) has been detected.
	1	SMT	0	No significance.
			1	Object code to evaluate an optimizable subscript expression in a for statement has been generated - used in IEX50 only.
	2	WERR	0	No significance.
			1	An error with severity code W (warning) has been detected. The Return Code 4, indicating a possibly defective object module, is to be transmitted to the invoking program by the terminating routine in IEX51.
	3	SERR	0	No significance.
			1	An error with severity code S (serious) has been detected. The Return Code 8, indicating an unsuccessful compilation, is to be transmitted to the invoking program by the terminating routine in IEX51.
	4	TERR	0	No significance.
			1	An error with severity code T (terminating) has been detected. The Compiler is to be terminated (after all recorded errors have been printed out), and the Return Code 16, indicating an unsuccessful compilation, is to be transmitted to the invoking program by the terminating routine in IEX51.
	5	PGR	0	The PROGRAM option has been specified. The object module is a precompiled procedure (i.e. a sub-program).
		PROC	1	The PROCEDURE option has been specified. The object module is a precompiled procedure (i.e. a sub-program to be stored in a partitioned data set).
	6	SHRT	0	The SHORT option has been specified. Short precision is to be provided for real (or floating point) operands.
		LNG	1	The LONG option has been specified. Long precision is to be provided for real (or floating point) operands.
	7	OPDT	0	The current operator in the source text was preceded by another operator - used in IEX50 only.
			1	The current operator in the source text was preceded by an operand which has been entered in the Operand Stack - used in IEX50 only.
1	0	SRCE	0	The SOURCE option has been specified. Listings of the source module and the Identifier Table are to be printed.
		NSRCE	1	The NOSOURCE option has been specified. Listings of the source module and the Identifier Table are not required.
	1	LOAD	0	The LOAD option has been specified. The object module is to be transferred to the SYSIN data set for subsequent input to the Linkage Editor.
		NLOAD	1	The NOLOAD option has been specified. The object module shall not be output on the SYSIN data set.
	2	DCK	0	The DECK option has been specified. A card deck of the object module is to be generated on the SYSPUNCH data set.
		NDCK	1	The NODECK option has been specified. A card deck of the object module is not required.
	3	EBCDIC	0	The EBCDIC option has been specified. The source module is in EBCDIC code.
		ISO	1	The ISO option has been specified. The source module is in ISO code.
	4	ERR	0	No significance.
			1	The Program Interrupt Routine PIROUT (in the Directory) was entered from the control program by reason of a program interrupt. If a second program interrupt occurs, PIROUT is to transfer control to the terminating routine (IEX51002) in IEX51.

Part 1 of 2

Byte No.	Bit No.	Switch Name	Bit Value	Significance
1	5	TERM	0 1	No significance. The terminating routine (IEX51002) in IEX51 has been entered.
	6	NOBUF	0 1	The Common Area (containing Source Buffer No. 1 and the Error Pool) has not been acquired. The Common Area has been acquired.
	7	NOGO	0 1	The Compiler has been successfully initialized (by IEX10). Compilation is impossible because both the NOLOAD and NODECK options have been specified or because a reguisite data set has not been opened. The Compiler is to be terminated at the conclusion of IEX10 by XCTL to IEX21 (for print out of recorded diagnostic messages) and thence by XCTL to the terminating routine in IEX51.
2	0	PRT	0 1	No significance An unrecoverable I/O error has occurred on the SYSPRINT data set. The Error Message Editing routine (in IEX21, IEX31 och IEX51) is to print diagnostic message No. 210 on the console typewriter, and the Compiler is to be terminated.
	1	SPIC	0 1	The modified source text (Modification Level 1 or 2) exceeds the buffer length and has been transferred to a utility data set. In IEX30, a second input buffer (in addition to the Common Area) and two output buffers are required. In IEX50, a second input buffer is required. The modified source text occupies less than a full buffer and has been transmitted (by a preceding phase) in the Common Area buffer.
	2	NOPT	0 1	The Optimization Table has been constructed by IEX40. Two Optimization Table buffers are to be acquired at initialization of the Compilation Phase. No Optimization Table was constructed by IEX40.
	3	PRTNO	0 1	The SYSPRINT data set was successfully opened. The SYSPRINT data set was not opened. The Error Message Editing routine is to print diagnostic message No. 201 on the console typewriter, and the Compiler is to be terminated.
	4	NOSC	0 1	The Semicolon Count is active, i.e. it correctly reflects the position reached by the current phase in the source text. The Semicolon Count is to be included in the error pattem for error No. 209 (recorded by the Directory routine PIROUT), in the event of a program interrupt. The Semicolon Count is inactive, i.e. it is not related to the operation being performed by the current phase. The Semicolon Count is to be omitted from diagnostic message No. 209.
	5	TEST NOTEST	0 1	The TEST option has been specified. Code is to be generated in the object module which, if the execution time TRACE option is specified, will produce a printed Semicolon Count list, after execution of the object program, indicating the logical path taken through the object program. The NOTEST option has been specified. Object code is not to be generated to implement the Trace facility at object time.
	6	SET60	0 1	The source module uses the 48 character set of the EBCDIC code. The source module uses the 60 character set of the EBCDIC code (recognized in IEX11 by detection of a semicolon in the translated source text). The same character set is to be used in printing all source text included in diagnostic messages.
	7			(Not used)

Part 2 of 2

• Program Context Matrix

The matrix is referenced by the COMP routine in the Compilation Phase (IEX 50).

Each matrix element specifies the compiler program to be entered for the given pair of operators in the source text and Operator Stack.

Stack \ Source	Beta Begin	Label Colon	Pi Phi	Array Switch	For Goto	If	Else	[	Assign	(	Arith Bool Rel.	Semi-colon	Delta	Epsilon	Eta	End	Apos-trophe	Omega	<All other op-erators>
Alpha	0	1	4	84	84	84	84	84	84	84	84	84	84	84	84	84	84	3	84
Beta, Pi Phi	0	1	4	4	7	7	75	7	7	7	75	25	24	16	84	84	83	84	75
Begin	0	1	29	29	6	8	75	41	12	64	75	25	28	84	84	16	84	84	75
Semicolon	0	1	26	26	6	8	75	41	12	64	75	23	28	23	23	23	84	84	75
Then-s	0	1	26	26	6	75	17	41	12	64	75	18	28	18	18	18	84	84	75
Else-s	0	1	26	26	6	8	75	41	12	64	75	18	28	18	18	18	**84**	84	75
Assign	27	30	26	26	27	19	20	41	21	22	22	20	28	20	20	20	84	84	75
Switch	84	84	84	84	84	84	84	84	**85**	84	84	84	**28**	84	84	84	84	84	84
Do	0	1	29	29	6	8	75	41	12	64	75	26	28	84	81	84	84	84	75
All other operators	31	30	31	31	31	31	31	31	31	31	31	31	28	31	31	31	84	84	22

APPENDIX V-b - Statement Context Matrix

The matrix is referenced by the COMP routine in the Compilation Phase (IEX 50).

Each matrix element specifies the compiler program to be entered for the given pair of operators in the source text and Operator Stack.

Stack \ Source	If	(	)	[	]	Comma	Assign	Step While	Until	<Any Arith. Optr.>	<Any Bool./ Rel.Optr>	Do	Epsilon Eta End Else Semicolon	Colon	Delta	<All other operators>
Goto	34	56	75	41	75	75	75	75	65	75	75	75	62	75	28	27
Switch :=	34	56	75	41	75	59	84	75	75	75	75	75	27	84	59	27
{	34	64	57	41	75	57	75	75	75	33	33	75	27	75	28	27
[	34	64	75	41	38	38	75	75	75	33	75	75	27	75	28	27
Array	84	84	84	52	84	52	84	84	84	84	84	84	27	84	54	84
<	34	64	61	41	75	61	75	75	75	33	33	75	27	75	28	27
For	75	75	75	41	75	75	40	75	75	75	75	27	27	75	28	27
For :=	34	64	75	41	75	43	75	43	75	33	75	43	27	75	28	27
Step	34	64	75	41	75	75	75	75	45	33	75	27	27	75	28	27
Until	34	64	75	41	75	47	75	75	75	33	75	47	27	75	28	27
While	34	64	75	41	75	49	75	75	75	33	33	49	27	75	28	27
Colon	34	64	75	41	51	51	75	75	75	33	75	75	27	75	28	27
€	34	64	75	41	75	75	84	75	75	33	75	75	27	36	28	27
<All other operators>	34	75	75	75	75	75	75	75	75	75	75	75	71	75	28	71

APPENDIC V-c - Expression Context Matrix

The matrix is referenced by the COMP routine in the Compilation Phase (IEX 50).

Each matrix element specifies the compiler program to be entered for the given pair of operators in the source text and Operator Stack.

Stack \ Source	Not	Equiv	Impl	And	Or	(	)	[	If	Then	Else	+ -	* / ÷	Power	<Relational Optr.>	<All other operators>
Not	75	77	77	77	77	64	77	41	73	77	77	66	67	67	67	77
Equiv	65	76	67	67	67	64	76	41	73	76	76	66	67	67	67	76
Impl	65	76	76	67	67	64	76	41	73	76	76	66	67	67	67	76
And	65	76	76	76	67	64	76	41	73	76	76	66	67	67	67	76
Or	65	76	76	76	76	64	76	41	73	76	76	66	67	67	67	76
(	65	67	67	67	67	64	68	41	80	27	27	66	67	67	67	27
If If-s	65	67	67	67	67	64	75	41	65	78	86	66	67	67	67	27
Then	65	67	67	67	67	64	75	41	73	75	87	66	67	67	67	27
Else	65	67	67	67	67	64	79	41	73	79	72	66	67	67	67	79
Monadic Minus	75	63	63	63	63	64	63	41	73	63	63	63	63	67	63	63
+ -	75	69	69	69	69	64	69	41	73	69	69	69	67	67	69	69
* / ÷	75	69	69	69	69	64	69	41	73	69	69	69	69	67	69	69
Power	75	69	69	69	69	64	69	41	73	69	69	69	69	69	69	69
<Relational Operator>	75	69	69	69	69	64	69	41	73	69	69	66	67	67	74	69
<All other operators>	65	67	67	67	67	64	70	41	70	75	70	66	67	67	67	76

APPENDIX VI: COMPILE TIME ERROR DETECTION

The following table lists the routines which detect syntactical errors in each of the phases of the Compiler. The corresponding diagnostic messages printed out (by the Error Message Editing routine in Load Modules IEX21, IEX31, and IEX51) are listed in OS ALGOL Programmer's Guide.

Syntactical Error Number	Severity Code	Routine(s) which detect the error*	Routine(s) which record the error (on call from the detecting routine)
Directory (IEX00)			
209	T	PIROUT	
210	T	SYNAD	
Initialization Phase (IEX10)			
200	W	Option processing (FNDSIZE,NOTFOUND,NXTPAR)	ERROR200
201	T	TSTPUNCH,TSTIN, TSTDCBRT	
202	W	Test if SYSLIN Open	
203	W	TSTPUNCH	
204	T	INEXRT	
205-A	W	LINEXRT	
-B	W	PCHEXRT	
-C	W	PRTEXRT	
206	W	ERROR200, ERROR208	FULLPOOL
207	W	DDNAMES	
208	W	Option Processing (FNDSIZE)	ERROR208
Scan I/II Phase (IEX11)			
1	W	TESTLOOP, LIST, APOSTROF NPAFTAPO	ERR1, ERR7
2	W	POINT	ERR5
3	W	LABEL, LETDEL, COLONLST	ERROR3
4	T	LETDEL	ERR4
5	S	TYPE, TYPESPEC, ARRAY, SWITCH, PROCID, IDCHECK	IERSPEC, ERR7
6	T	LABEL	ERR4
7	W	LABEL	ERR2
8	W	LABEL	ERR7
10	S	BEGPROC, END(PREND), SEMCO, STATE, FOR, CODE	ERROR10
11	S	END, SEMCO, STARTDEL, TYPESPEC	ERR8
12	W	DELIMIT	ERR6
13	W	EROUT	ERRB
14	S	EROUT	ERR2C
15	W	CODE	ERR7
16	S	TYPE, ARRAY, SWITCH, PROCID, IDCHECK	IER, ERR2, ERR2E
17	S	SEMCO	ERR7
18	W	COM	ERR7
19	W	SEMCO, WARNING	ERR7
20	T	BEGIN, FOR, PROCEDUR	ERR4
21	S	BEGI	ERROR21
22	T	BEGI, PROCEDUR	ERR4
23	S	STRING	ERR7
24	S	CODE	ERR7
25	S	SPEC	ERR7
26	S	IDCHECK(IDVALCHK)	ERR2E
27	W	IDCHECK(NOTFOUND)	ERR2E
28	S	VALUE	ERR7

29	W	VALUE(VALDLB2)	ERR7
30	W	VALUE(VALDLB2)	ERR2E
31	W	ARRAY	ERR2
32	S	SEMCLST	ERR2
33	T	SEMCLST, SLASHLST	ERR2
34	S	SLASHLST	ERR2
35	W	POINTLST	ERR5A
36	T	PROCID(PROCFIN)	ERR2B
37	S	PROCID(PROCDEL)	ERR2B
38	T	PBLCKEND	ERR4
39	S	ENDMISS	ERRDR39
41	T	FOR	ERR4
42	W	FIRSTBEG	ERR7
43	S	END, READRRUT, TED	ERR9
44	T	ENDMISS	ERR4
212	T	ERROR1	ERR0
213	T	ITABCLEAR, PROCID(TPROHEAD)	ERR4
215	T	COB	ERR4
216	S	LABEL, SWITCH, PROCEDUR	ERR7

Identifier Table Manipulation Phase (IEX20)

45	S	READBLK	E43
47	S	ALLOSTOR	E45
212	T	ERRNAME	E0
214	S	ALLOSTOR	E44

Scan III Phase (IEX30)

80	S	LETTER, DIGIT19, DIGIT , DECPOIN, SCAQL, SCAFACT, QUOTE	ERROR1, QTORLT, INCOROP
81	S	IDENT	MOVERRO
82	S	REALCON	MOVERRO
83	W	INTCON	MOVERRO
84	W	REALCON	MOVERRO
85	S	SWILA	INCOROP
86	S	INTHAN, REALHAN	MOVERRO
87	W	TABOFLO	MOVERRO
88	W	IDENT	MOVERRO
89	W	SWILA	MOVERRO

Compilation Phase (IEX50)

160	S	CP31, CP66, CP73, CP74, CP86	SERR4
161	S	CP31, CP73, CP74, CP86	SERR4
162	S	CP21, CP67, CP69, OPDTEST	SERR4
163	S	CP63, CP69, TARITHM	SERR2
164	S	CP0, CP4, CP6, CP34, CP54, CP56, CP65, CP80	SERR4
165	S	CP79	SERR1
166	S	ERR166	SERR1
168	S	CP61, CP64	SERR3
169	S	CP30	SERR2
172	S	CP21, CP32	SERR1
173	T	CP0, CP84	SERR1
174	S	CP64	SERR1
175	S	CP59, CP62	SERR2
176	S	CP12, CP57, CP64	SERR3
177	S	CP8, CP24	SERR3
178	S	CP87,CP79	SERR1
179	S	CP38	SERR1
180	S	CP38	SERR1
181	S	CP41	SERR1
182	S		
183	S	CP64, PLPRST	SERR3
184	S	CP41	SERR3
185	S	CP69	SERR3
186	T	STACKOFL	SERR1

187	S	CP57, CP61, CP64, PLPRST, ARRTEST1	SERR1
188	S	CP61	SERR1
189	S	CP61	SERR1
190	S	CP12, CP40	SERR1
191	S	CP51, SNOTSP	SERR1
192	S	CP20	SERR1
193	S	CP20	SERR1
194	S	CP27, CP69, CP75	SERR2/3
195	S	CP27, CP75	SERR3
196	S	ARRTEST1	SERR3
214	S		SERRSP
216	S	LATRES	SERR1

Object Time Table Output and Termination (IEX51)

188	S	DSTAB Construction

Diagnostic Output (IEX21, IEX31, IEX51)

211	T	WDIRET1

*The name enclosed by parentheses indicates the particular locality within the routine in which the error is detected.

The following table lists the modules in which object time errors are detected. The corresponding diagnostic messages printed out (by module IHIERR -- loaded in main storage as soon as an error is detected) are listed in the OS ALGOL Programmer's Guide.

Error number	Module(s) which detect the error (routine name inside parenthesis)	Module which prints the error message
0	IHISYS(SYSACT8), IHIIOR(EVDSN)	IHIERR
1	IHIIOR(CONVERT)	IHIERR
2	IHISYS(SYSACT4, SYSACT13), IHIIOR (OPEN, CLOSE, NEXTREC)	IHIERR
3	IHIIOR(CLOSE, NEXTREC, OPEN)	IHIERR
4	IHIIOR(ENTRYNOTTAB)	IHIERR
5	IHIIBO, IHIIDE, IHIISY, IHIIOR (ENDOFDATA)	IHIERR
6	IHIIDE	IHIERR
7	IHIIOR(OPEN)	IHIERR
8	IHIOSY	IHIERR
9	IHISYS	IHIERR
10	IHIGPR(GET), IHISYS, SYSACT1, SYSACT2, SYSACT3, SYSACT4, SYSACT5, SYSACT13, SYSACT14, SYSACT15)	IHIERR
11	IHISYS(SYSACT6, SYSACT8)	IHIERR
12	IHISYS(SYSACT1, SYSACT3, SYSACT5, SYSACT7, SYSACT9, SYSACT11, SYSACT13)	IHIERR
13	IHISYS(SYSACT2, SYSACT6, SYSACT8, SYSACT10, SYSACT12, SYSACT14, SYSACT15)	IHIERR
14	IHIGPR(GET), IHISYS(SYSACT4)	IHIERR
15	Generated object module	IHIERR
16	Generated object module	IHIERR
17	Generated object module	IHIERR
18	IHIFSA(GETMSTO)	IHIERR
19	Generated object module	IHIERR
20	IHIFSA(PROLOG), IHIGPR(INPUT, OUTPUT)	IHIERR
21	IHIFSA(PROLOG), IHIGPR(INPUT, OUTPUT)	IHIERR
22	Generated object module	IHIERR
23	IHILSQ, IHISSQ	IHIERR
24	IHILEX, IHISEX	IHIERR
25	IHILLO, IHISLO	IHIERR
26	IHISSC	IHIERR
27	IHILSC	IHIERR
28	IHIFSA(PIEROUT)	IHIERR
29	IHIFSA(PIEROUT)	IHIERR
30	IHIFSA(PIEROUT)	IHIERR
31	IHIFSA(PIEROUT)	IHIERR
32	IHIIOR(SYNAD)	IHIERR
33	IHIFSA(PIEROUT)	IHIERR
34	IHIFSA(CSWE1)	IHIERR
35	IHIFII, IHIFRI, IHIFDI, IHIFRR, IHIFDD	IHIERR
36	IHIFSA(PROLOG, CAP1, CSWE1, LOADPP) IHIGPR(CAP1GP)	IHIERR
37	IHIIOR(OPEN)	IHIERR
38	IHIGPR(INPUT, OUTPUT), IHILOR, IHIOBO, IHIOIN, IHISOR	IHIERR
39	IHIGPR(GET, PUT)	IHIERR
40	IHIFSA(CNVRDI/ENTIER)	IHIERR
41	IHIFSA(ALGIN), IHIIOR(OPEN), IHIGPR(OPENGP)	IHIERR

42	IHIFSA(ALGIN)	IHIERR
43	IHIGPR(GET, PUT, INPUT, OUTPUT)	IHIERR

APPENDIX VIII - Compile Time Work Area Sizes, as a Function of the SIZE Option

SIZE Options:	< 51,200	< 59,392	< 67,584	< 77,824	< 90,112	< 104,448	< 120,832	< 139,264	< 159,744	< 184,320	< 212,992	< 16,777,216
Work Areas and Buffers												
Identifiers Table:												
- in IEX10	8,184	11,924	17,754	21,054	23,584	23,584	32,736	32,736	32,736	32,736	32,736	32,736
- in IEX20	25,000	29,500	36,500	41,000	48,000	50,000	62,000	75,000	95,000	115,000	135,000	162,000
- in IEX30	8,800	11,500	15,000	16,000	16,000	16,000	19,000	25,000	38,000	58,000	58,000	58,000
Subscript Table:												
- in IEX30	1,400	1,400	1,400	1,400	1,400	1,400	1,400	1,400	1,400	1,400	1,400	1,400
(with track overflow)							(2,800)	(7,000)	(7,000)	(7,000)	(7,000)	(7,000)
- in IEX40	14,000	15,400	18,200	21,000	23,800	25,200	30,800	35,000	35,000	56,000	56,000	56,000
Left Variable Table:												
- in IEX30	800	800	800	800	800	800	1,600	1,600	1,600	1,600	1,600	1,600
(with track overflow)								(4,800)	(4,800)	(6,400)	(6,400)	(6,400)
- in IEX40	7,200	8,000	9,600	11,200	12,800	13,600	16,000	19,200	24,000	25,600	25,600	25,600
Optimization Table												
- in IEX40 and IEX50	224	910	1,792	1,792	1,792	1,792	1,792	1,792	1,792	1,792	1,792	1,792
Source Text Buffers												
- used in IEX11, -30, -50	1,024	1,536	2,000	2,000	2,000	2,000	2,000	2,000	2,000	2,000	2,000	2,000
(with track overflow)			(2,048)	(4,096)	(6,144)	(6,144)	(8,192)	(8,192)	(8,192)	(8,192)	(8,192)	(8,192)
Critical Identifier Table												
- in IEX30	450	450	900	900	1,350	1,350	2,250	3,600	3,600	3,600	3,600	3,600
Operator-Operand Stack												
- in IEX50	768	3,072	6,144	6,144	6,144	6,144	6,144	6,144	6,144	6,144	6,144	6,144
Error Pool	600	1,000	1,304	1,600	2,000	2,000	2,000	2,000	2,000	2,000	2,000	2,000
- used in all phases												
Maximum Block Sizes												
SYSIN	400	400	400	400	1,600	1,600	1,600	1,600	1,600	1,600	3,200	3,200
SYSPRINT	455	455	455	455	455	1,820	1,820	1,820	1,820	1,820	3,640	3,640
SYSLIN	400		400	400	400	3,200	3,200	3,200	3,200	3,200	3,200	3,200
SYSPUNCH	80	400	400	400	400	1,600	1,600	1,600	1,600	1,600	3,200	3,200

APPENDIX IX-A: STORAGE MAPS OF THE CONSTITUENT LOAD MODULES OF THE ALGOL COMPILER

The ALGOL Compiler is composed of the following ten load modules or phases:

Load Module Name	Phase Name
IEX00	Directory
IEX10	Initialization Phase
IEX11	Scan I/II Phase
IEX20	Identifier Table Manipulation Phase
IEX21	Diagnostic Output
IEX30	Scan III Phase
IEX31	Diagnostic Output
IEX40	Subscript Handling Phase
IEX50	Compilation Phase
IEX51	Termination Phase

The following tables indicate the contents of main storage in each of the phases of the Compiler, in terms of major routines, work areas and tables. The flowchart which outlines each routine is shown in the right hand column.

IEX00 - Directory

Note: The Directory (Load Module IEX00) is resident in main storage during execution of each of the other nine load modules.

Control Section	Contents	Flowchart
IEX00000	Initial Entry Routine	003
	Final Exit Routine	003
	Save Area	
	Program Interrupt Routine (PIROUT)	004
	End of Data Routines (EODAD1/-2/-3/-IN)	005
	I/O Error Routine (SYNAD, SYNPR)	005
	Print Subroutine (PRINT)	006
	Data Control Blocks for SYSPRINT, SYSLIN, SYSPUNCH, SYSUT2 and SYSUT3	
IEX00001 (Common Work Area)	Save Area	
	Communication Area, containing: DCB addresses, EOD Routine addresses, HCOMPMOD Control Field (displ. x '80'), Misc. Control Information, Semicolon Count (displ. x '9A'), Area Size Table, Printed Output Headings	
	Preliminary Error Pool	
	Data Control Block for SYSIN	
	Work Area	
	Data Control Block for SYSUT1 Work Area	

IEX10 - Initialization Phase

Control Section	Contents	Flowchart
IEX00000	(See IEX00 - Directory)	
IEX00001 (Common Work Area)	Save Area Communication Area (see IEX00) Work Area Data Control Block for SYSIN, Work Area Data Control Block for SYSUT1, Work Area	
IEX10000	Compiler Initialization Routine	007-010

Error Pool Source Text Buffer 1

Control Section	Contents	Flowchart
IEX00000	(See IEX00 - Directory)	
IEX00001 (Common Work Area)	Save Area Communication Area (see IEX00) Work Area Data Control Block for SYSIN Work Area Program Block Number Table (PBTAB1) Data Control Block for SYSUT1 Scope Table (SPTAB) Group Table (GPTAB)	
IEX11000	Initialization Routine Translation and Address Tables Main Loop Routine (TESTLOOP) Blank and Transfer Operator Routines (BLANK and TRANSOP) Point and Statement Routines (POINT and STATE) Apostrophe Routine (APOSTROF) Colon, Label and Letter Delimiter Routines (COLON, LABEL and LETDEL) Semicolon Routine (SEMCO) Error Recording Routines	013 013 014 015 015 016
IEX11001	Change Output Buffer Subroutine (COB) Change Input Buffer Subroutine (CIB) Delimiter Routine (DELIMIT) Delimiter Error Routine (EROUT) NORMAL, BOLCON, GIF, TED, and BEGIN Routines Block Begin Routine (BEG1) End Routine (END) For Statement End and Program Block End Routines (FOREND and PBLCKEND) Comment Routine (COM) For Statement Begin Routine (FOR) Type Declaration/Specification Routine (TYPE) Invalid Identifier Routine (IER) Code Routine (CODE)	017 017 018 019 020 022 021 022 032 020 023 028 020
IEX11002	Parameter Specification Routine (SPECENT or IDCHECK) Value Specification Routine (VALUE) Type-Array and Array Declaration/Specification Routines (TYPEARRY and ARRAY) Array/Switch List Routine (LIST) Switch Declaration/Specification Routine (SWITCH) String Declaration Routine (STRING) Type-Procedure and Procedure Declaration Routines (TYPPROC and PROCEDUR) End of Data Exit Routine (ENDMISS)	031 031 024 026 025 033 029-030 034
IEX11003	Termination Routine (TERMOK or EODADIN) Object Code Generate Subroutine (GENERATE)	034 122

Constant Pool Identifier Table Identifier Table Buffer Scope Handling Stack Source Text Buffer 2
Error Pool Source Text Buffer 1

IEX20 - Identifier Table Manipulation Phase

Control Section	Contents	Flowchart
IEX00000	(See IEX00 - Directory)	
IEX00001 (Common Work Area)	Save Area Communication Area (see IEX00) Program Block Table II (PBTAB2) Program Block Number Table (PBTAB1) Data Control Block for SYSUT1 Scope Table (SPTAB) Group Table (GPTAB)	
IEX20000	Initialization Routine Read and Scan Identifier Table Routine (READBLK) Storage Allocation Routine (ALLOSTOR) Write Identifier Table Routine (WRITITAB) Termination Routine	036 036 037 038 038

Address Table Identifier Table
Error Pool Source Text Buffer 1

IEX21 - Diagnostic Output

Control Section	Contents	Flowchart
IEX00000	(See IEX00 - Directory)	
IEX00001	Save Area Communication Area (see IEX00) Program Block Table II (PBTAB2) Work Area Data Control Block for SYSUT1 Scope Table (SPTAB) Group Table (GPTAB)	
IEX21000	Linking Routine	040
IEX21001	Message Pool	
IEX60000	Error Message Editing Routine	040-043

Error Pool Source Text Buffer 1

IEX30 - Scan III Phase

Control Section	Contents	Flowchart
IEX00000	(See IEX00 - Directory)	
IEX00001 (Common Work Area)	Save Area Communication Area (see IEX00) Program Block Table II (PBTAB2) For Statement Table (FSTAB) Data Control Block for SYSUT1 Scope Table (SPTAB) Group Table (GPTAB)	
IEX30000	Tables and Constants Initialization Routine (INITIATE) General Test Routine (GENTEST) Apostrophe Routine (QUOTE) Block Begin, Procedure Declaration, For Statement and Program Block End Routines (BETA, PIPHI, FOR, and EPSILON) For Statement End and For List End Routines (ETA and DO) While, Step and Divide/Power Routines (WHILE, STEP and DIPOW) Semicolon/Dela, Switch and Array Routines (SEMIDELT, SWITCH and ARRAY) Program End Routine (OMEGA) Opening Bracket, Comma and Closing Bracket Routines (OPBRACK, COMMA and CLOBRACK)	 046 046 049 047 049 047 051 050 051

IEX30001	Identifier Test, ITAB Search and Identifier Classification Routines (LETTER, IDENT, and FOLI)	057
	Non-Critical Identifier, Procedure/Parameter and Switch/Label Routines (NOCRI, PROFU, and SWILA)	058
	Critical Identifier Routine (CRITI)	059
	CRIDTAB Entry Subroutine (CRIMA)	060
	CRIDTAB Overflow, CRIDTAB Erasure, and CRIDTAB Update Subroutines (CRIFLOW, DELCRIV, and CRIFODEL)	061
	Subscript Scan Subroutine (SUCRIDEL)	055
	LVTAB Entry Subroutine (LETRAF)	062
	Subscript Test Subroutine (SUSCRITE)	052-053
	Operand Test and Multiplier-Operand Subroutines (OPERAND and SUBMULT)	054
	SUTAB Entry Subroutine (SUTABENT)	053
	Change Output Buffer and Change Input Buffer Subroutines (OUCHA and ICHA)	049
	Invalid Operand Routine (INCOROP)	070
	Error Recording Subroutine (MOVERRO)	
	Read ITAB Subroutine (ITABMOVE)	048
	Write SUTAB/LVTAB Subroutine (WRITE)	062
IEX30002	Non-zero Digit and Zero Digit Routines (DIGIT19 and DIGIT0)	064
	Decimal Point Routine (DECPOIN)	065
	Scale Factor Routine (SCAFACT)	066
	Integer Conversion Routine (INTCON)	067
	Real Conversion Routine (REALCON)	068
	Integer Handling Routine (INTHAN)	067
	Real Handling Routine (REALHAN)	069
	Generate Object Code Subroutine (GENTXT)	122

Constant Pool Identifier Table Source Text Buffers 2-4 Critical Identifier Table Subscript Table Left Variable Table
Error Pool Source Text Buffer 1

IEX31 - Diagnostic Output

Control Section	Contents	Flowchart
IEX00000	(See IEX00 Directory)	
IEX00001	Save Area Communication Area (see IEX00) Program Block Table II (PBTAB2) For Statement Table (FSTAB) Work Area	
IEX31000	Linking Routine	071
IEX31001	Message Pool	
IEX60000	Error Message Editing Routine	071

Error Pool Source Text Buffer 1

IEX40 - Subscript Handling Phase

Control Section	Contents	Flowchart
IEX00000	(See IEX00 · Directory)	
IEX00001 (Common Work Area)	Save Area Communication Area (see IEX00) Program Block Table II (PBTAB2) For Statement Table (FSTAB) Address Table (ZADSTA) Entry Count Table (ZCOSTA) Work Area	
IEX40000	Initialization Routine Read SUTAB Routine Scan SUTAB Routine Sort SUTAB Routine (SORTSU) Read and Sort LVTAB Routine (SORTLE) Construct OPTAB Routine (OPTAB) Termination Routine (TERMIN) Write OPTAB Subroutine (OTACHA) Read SUTAB/LVTAB Subroutine (READ) Sort SUTAB/LVTAB Subroutine (SORT)	073 073 073 073 074 075 076 076 074 074
IEX40001	Compilation Phase Initialization Routine	078

Optimization Table (OPTAB) Buffers 1 and 2 Subscript Table (SUTAB) and Left Variable Table (LVTAB) Work Area
Error Pool Source Text Buffer 1

Control Section	Contents	Flowchart
IEX00000	(see IEX00 - Directory)	
IEX00001 (Common Work Area)	Save Area Communication Area (see IEX00) Program Block Table II (PBTAB2) For Statement Table (FSTAB) Program Block Table III (PBTAB3) Subscript Table - C (SUTABC) Work Area	
IEX50000	Initialization Routine Scan to Next Operator and Compare Routines (SNOT and COMP) Change Input Buffer and Get Next OPTAB Entry Subroutines (JBUFFER and NXTOPT) Error Recording Subroutines Generate Call for Integer-Real and Real-Integer Conversion Subroutines (TRINRE and TRREIN) Generate Object Code Subroutine (GENERATE) Generate Store Registers Subroutine (CLEARRG) Operand Recognizer Subroutine (OPDREC) Update DSA Pointer (MAXCH) and Semicolon Handling (SCHDL) Subroutines Subroutines: ROUTINE1 - ROUTIN15 Decompose Operand (DECOMP), Stack All-purpose Operand (STACKAPI), P.B.No. Handling (PBNHDL), Parameterless Procedure (PLPRST), Generate Load Operand (LDVWPLC), Update Stack Pointers (MOVEOPTK and MOVEOPDK), Arithmetic Test (TARITHM), Reserve Label Table Entry (LATRES), Array/Procedure Test (ARRTEST1) and Operand Test (OPDTEST) Subroutines	 079 121 123 122 120
IEX50001	Decision Matrices	
IEX50002	Compiler Programs handling for statements: CP6, CP40, CP43, Cp45, CP47, CP49, and CP81	082-117

IEX50003	Compiler Programs handling blocks and compound statements, procedure declarations, array declarations, switch declarations, goto statements, subscripted variables, switcch designators, and close of source module: CP0, CP1, CP3, CP4, CP7, CP8, CP16, CP24, CP25, CP36, CP38, CP41, CP51, Cp52, CP54, CP56, CP59, CP62, and CP85	080-118
IEX50004	Compiler Programs handling assignment statements and procedure calls: CP12, CP19, CP20, CP21, CP22, CP23, CP33, CP34, CP57, CP61, CP64, CP71, CP83 and CP84	083-118
IEX50005	Compiler Programs handling conditional statements and conditional expressions, boolean expressions, and logical errors: CP17, CP18, CP26, CP27, CP28, CP29, CP31, CP63, CP65, CP66, CP67, CP68, CP70, CP72, CP73, CP74, CP75, CP76, CP77, CP78, CP79, CP80, CP86 and CP87	084-119
IEX50006	Compiler Program No. 69 (CP69)-handling of arithmetic operators	107-112

Source Text Buffer 2 Optimization Table Buffers Operator-Operand Stack Label Address Table
Error Pool Source Text Buffer 1

IEX51 - Termination Phase

Control Section	Contents	Flowchart
IEX00000	(See IEX00 - Directory)	
IEX00001 (Common Work Area)	Save Area Communication Area (see IEX00) Program Block Table III (PBTAB3) Work Area	
IEX51000	Generate Label Address Table Routine (TRM1) Generate Program Block Table Routine (TRM2) Construct Data Set Table Routine (TRM10) Generate Data Set Table Routine (TRM23) Generate Address Table Routine (TRM24) Generate END Record Routine (TRM35) Print Storage Requirements Routine (TRM51) Error 188 Routine (ERR188) Tables and Constants Generate Object Code Subroutine (GENERATE)	125 127 122
IEX51M00	Message Pool	
IEX51002	Compiler Termination Routine	126
IEX60000	Error Message Editing Routine	128

Source Text Buffer 2 Optimization Table Buffers Operator-Operand Stacks Label Address Table
Error Pool Source Text Buffer 1

Module	Contents
-	<u>Generated Code</u> Constant Pool Instructions Label Address Table Program Block Table Data Set Table Addresses
IHIFSA	<u>Fixed Storage Area</u> Common Data Area Save Area Work Area Branch List Type Conversion Subroutines (CNVIRD, ENTIER, and CNVRDI) Get Main Storage Subroutine (GETMSTO) Call Actual Parameter Routines (CAP1 and CAP2) Prologue Routine (PROLOG) Value Call Subroutine (VALUCALL) Return Routine (RETPROG) Epilogue Routine (EPILOG) Standard Procedure Declaration Routine (SPDECL) Call Switch Element Routines (CSWE1 and CSWE2) Load Precompiled Procedure Subroutine (LOADPP) Trace Subroutine (TRACE) Program Interrupt Routine (PIEROUT)
	Initial Entry Routine (IHIFSAIN) Initialization Routine (ALGIN) Termination Routine (TERMNTE)
IHIIOR	I/O Procedure Subroutine Pool
	Standard I/O Procedures and Mathematical Functions invoked by the object module
IHIERR	Message Printing Routine (IHIERR) Message Pool
	<u>Precompiled Procedure</u> Constant Pool Instructions Label Address Table Program Block Table Addresses
	Standard I/O Procedures and Mathematical Functions invoked by the precompiled procedure(s)
	Data Storage Area(s)
	Array Storage Area(s)
	Data Control Block(s)
	Input/Output Buffer(s)
	Return Address Stack (RAS)

The following table summarizes the compiler programs of the Compilation Phase (IEX50), in terms of the operator combinations which activate each program, the operands in the Operand Stack at entry and exit, the errors detected, and the subroutines called.

Operators are defined in the notes accompanying Appendix I-d. The internal names of all operands declared or specified in the source module are listed in Appendix II. Other special-purpose operands stacked by the compiler programs are explained in the following key.

Operand Name	Content
AP1	Five-byte all-purpose internal name
COUNT	Literal count of parameters, dimensions or components
DSN	Data Set Number
DISPL	Displacement of a reserved storage field in the current Data Storage Area
FLAG	Indicator whether an array element has been preprocessed
FSN	Serial For Statement Number

Operand Name	Content
LN	Displacement of a reserved full word in the Label Address Table
OPBT	For statement classification byte (copied from the For Statement Table)
PARCH	Two-byte characteristic of a parameter
PRPOINT	Saved value of the compile-time object module displacement pointer PRPOINT (register 6)
SUTABC	Last five bytes of an entry in Subscript Table-C (Figure 73)

The symbol (CWS) in the columns headed "Stack Operator" designates one of the three contest (matrix) switch operators PRC, STC or EXC.

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
0	Beta, Begin	Alpha Beta Pi Phi Begin Semicolon Then-s Else-s Do	(Source op. stacked)			PRC	PRC		SNOT	164 173	GENTXT CLEARRG, MOVEOPTK, PBNHDL, SERR
1	Label Colon	Alpha Begin Semicolon Then-s Else-s Do	(No change)	label identifier		PRC	PRC		SNOT CP48		CLEARRG, MOVEOPTK
		Beta Pi Phi	Semicolon stacked								
3	Omega	Alpha	(No change)			PRC	PRC		CPEND CP84		GENTXT
4	Pi Phi	Alpha Beta Pi Phi	(Source op. stacked)		[ ] LN procedure identifier	PRC	PRC	until Delta is found	COMP	164	GENTXT JBUFFER LATRES, PBNHDL DLCOMP, MOVEOPDK, MOVEOPTK, SERR
	Switch				[ ] LN (reserved)				SNOT		
	Array						STC				
6	For	Begin Do Semicolon	For stacked		FSN OPTP (reserved) (reserved)	PRC	STC		SNOT	164	MOVEOPDK, MOVEOPTK, SERR
	Goto		Goto stacked								
7	For Goto If [Assign (	Beta Pi Phi	Semicolon stacked			PRC	PRC		COMP		MOVEOPTK
8	If	Begin Else-s Do Semicolon	If-s stacked			PRC	EXC		SNOT	177	CLEARRG, MOVEOPTK, SERR
12	Assign	Begin Then-s Else-s Do Semicolon	Assign stacked	left variable	(No change)	PRC	PRC		COMP	176 190	GENTXT OPDREC, MOVEOPTK, STACKAPI, ROUTINE1
16	End	Beta Pi Phi Begin		procedure identifier [ ] LN procedure identifier	(No change)	PRC	PRC		SNOTSP		GENTXT PBNHDL, PLPRST, DECOMP
	Epsilon	Beta									
		Pi Phi Begin									

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
17	Else	Then-s	Else-s replaces Else	procedure identifier / LN	LN	PRC	PRC		SNOT		GENTXT PLPRST, LATRES
18	Epsilon Eta End Semicolon	**Else-s**		procedure identifier / LN		PRC	PRC		COMP		PLPRST
19	If	Assign	(No change)			PRC	STC		COMP		
20	Epsilon Eta End Else Comma	Assign		right variable / left variable		PRC	PRC		COMP	192 193	CP69 OPDREC, TRINRE, TRREIN, OPDTESF, ARRTEST, SERR, ROUTINE1, ROUTINE3, ROUTINE7
		Assign / . . . / Assign		right variable / left variable / . . / left variable							
	(entry from	For / Assign	CP43) [illegible]	right variable / left variable	left variable	STC	STC		CP43		
	(entry from	[/ Assign	CP5[illegible]) stacked						CP51		
21	Assign	Assign	Assign stacked	left variable	(No change)	PRC	PRC		CP12 SNOT	162 172	SERR
22	(Any Arithm/ Bool/ Rel Op.)	Assign	(No change)	(array, procedure, arithmetic or boolean operand or none)	(No change)	PRC	EXC		COMP		
23	Semicolon Epsilon Eta End End	Semicolon		procedure identifier		PRC	PRC		COMP		PLPRST, SCHDL
						PRC	PRC		SNOT	177	SERR, SCHDL
24	Delta	Beta Pi Phi	(No change)								
25	Semicolon	Beta Pi Phi Begin	Semicolon Stacked	procedure identifier		PRC	PRC		SNOT		PLPRST, MOVEOPTK, SCHDL
26	Array Switch Phi Pi	Semicolon		(operand)		PRC	PRC		COMP	166	ERR166
		Assign		(operand) / left variable							
		Then-s Else-s		(operand) / LN							

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
27	(See matrices - App.V)	If Goto		(operand)		EXC	EXC		COMP	194 195	SERR, STACKAPI
		Array ← Colon		(operand) (Address of lower bound in current DSA) identifier COUNT identifier		STC	STC				
		(CSW) [		(operand) (reserved) (reserved) (reserved) array/switch ident.			PRC STC EXC				
		Array		(operand) array identifier array identifier			STC				
		Assign		(operand) left variable		PRC					
		For	Do	(operand)	API	STC					
		For For:= Step									
		For For:= Until		(operand) (step value)							

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines Used
27		For For:= While		(operand)							
		Then		(operand) LN		EXC	EXC				
		(CSW) {		(operand) parameter . . LN procedure identifier		STC	PRC STC EXC				
		(CSW) <		(operand) COUNT DSN DISPL st proc identifier							
		Array ¢		(operand) DISPL array identifier COUNT array identifier . . .			STC				
		For For:=	Do	(operand)							
		Switch:=		(operand) COUNT PRPOINT switch identifier LN (reserved)							

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
28	<u>Delta</u>	<u>Array</u> / ¢ / . .		(operand) / lower bound in stack / array identifier / COUNT / array identifier		PRC	PRC		SNOT		SCHDL, ERR166
		<u>Array</u> / ¢									
		<u>Switch</u>		(operand) / IN / (reserved)							
		(See matrices - App.V)		(operand)							
29	<u>Array</u> <u>Switch</u>	<u>Begin</u> <u>Do</u>	(No change)			PRC	PRC		CP4		ERR166
30	<u>Label Colon</u>	(See matrices - App.V)	(No change)	label identifier		PRC	PRC		SNOT CP84	169	SERR
31	(See matrices - App.V)		(No change)	(operand)	(No change	PRC	EXC		COMP	160 161	SERR
33	(Arithm/ Bool/Rel Op)	(See matrices - App.V)	(No change)	(arithm/bool operand)	(No change)	STC	EXP		COMP		
34	<u>If</u>	(See matrices - App.V)	<u>If</u>			STC	EXC		SNOT	164	CLEARRG, SERR, MOVEOPTK
36	<u>Colon</u>	<u>Array</u> / ¢	<u>Colon</u> stacked	lower bound / DISPI / array identifier / COUNT / array identifier	(Address of lower bound in DSA) / array identifier / COUNT / array identifier	STC	STC		SNOT		GENTXT UPDREC, MAXCH, OPDTESF, TARITHM, TRREIN, MOVEOPTK, ROUTINE8-9, ROUTIN15

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
38	]	(CSW) —		subscript expression (reserved) (reserved) (reserved) switch identifier	designational expr	STC	PRC		SNOTSP	180	GENTXT OPDREC, MAXCH, CLEARRG, TRREIN, TARITHM, OPDTESF, SERR, ROUTINE1-3-8-9-15
				subscript expression SUTABC - last part subscript expression DISPL COUNTS FLAG array identifier	subscript variable						
	Comma		(No change)		(SUTABC - last part) subscript variable DISPL COUNTS FLAG array identifier		STC		SNOT		
40	Assign	For	For:= stacked	controlled variable FSN OPTB (reserved) (reserved)	controlled variable FSN OPTB LN IN DISPL DISPL DISPL DISPL OPTIMIZATION COUNT	STC	STC		SNOT	190	GENTXT OPDREC, MAXCH, CLEARRG, SERR, NXTOPT, LATRES, OPDTESF, TARITHM, MOVEOPTK, ROUTINE1
41		(See matrices - App.V)	(CSW) [	switch identifier	3 opds reserved switch identifier	PRC STC	STC		SNOT	181 184	GENTXT OPDREC, OPDTESF, MOVEOPTK, SERR, ROUTINE1, ROUTINE7
				array identifier	(SUTABC - last part) subscript variable DISPL COUNTS FLAG array identifier						
43	_omma	For For:=	(No change)	initial value controlled variable FSN OPTB LN IN DISPL DISPL DISPL DISPL optimization COUNT	controlled variable FSN FSN OPTB LN LN DISPL DISPL COUNT optimized elements	STC	STC		SNOT CP47 CP49		CP20 GENTXT MAXCH, OPDTESF, LATRES, TARITHM
	Step		Step stacked								
	While		While stacked								
	Do		Do				PRC				

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
45	Until	For For:= Step	For For:= Until	step value controlled variable FSN | OPTB | LN | LN | DISPL | DISPL DISPL | DISPL optimization | COUNT	(No change)	STC	STC		SNOT		GENTXT, GENTXP, OPDREC, LATRES, OPDTESTF, TARITHM, ROUTINE1, ROUTINE3, ROUTINE8, ROUTIN12, ROUTIN15
47	Comma	For or:= Until	For For:=	test value step value controlled variable FSN | OPTB | LN | LN DISPL DISPL | DISPL optimization | COUNT	controlled variable FSN DISPL DISPL DISPL | COUNT optimized element(s)	STC	STC PRC		SNOT		CP45 GENTXT, GENTXP, NXTOPT, MAXCH, OPDREC, TRINRE, LATRES, OPDTESF, TARITHM, MOVEOPDK, ROUTINE1, 3, 7, 8, 9, 12, 15
	Do		Do								
49	Comma	For For:= While	For For:=	while value controlled variable FSN | OPTB | LN | LN DISPL DISPL | DISPL optimization | COUNT	controlled variable FSN DISPL DISPL DISPL | COUNT optimized element(s)	STC	STC		SNOT		GENTXT NXTOPT, MAXCH, OPDREC, LATRES, SERR, MOVEOPDK, ARRTEST1, OPDTESF, ROUTINE1, 7, 8, 9, 15
	Do		Do								
51	Comma	Array [illegible] Colon	Array [illegible]	upper bound lower bound in stack array identifier COUNT | array identifier(s)	| DISPL array identifier COUNT | array identifier(s)	STC	STC		SNOT	191	Branch to CP69, return via CP66 CP69, CP70, CP43, CP20 GENTXT JBUFFER, MAXCH, OPDREC, TRREIN
	]		Array					one byte	SNOT COMP		SERR, OPDTESF, MOVEOPDK, MOVEOPTK, TARITHM
52	Comma	Array	(No change)	array identifier array identifier(s)	(No change)	STC	STC		SNOT		MOVEOPTK
			[illegible] stacked		| DISPL array identifier Count | array identifier(s)						

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
54	Delta	Array				STC	PRC		SNOT	164	SCHDL, SERR
56	(	Goto Switch:=	(stacked			STC	EXC		SNOT	164	MOVEOPTK, SERR
57	Comma	(CSW) [	(No change)	parameter COUNT | PRPOINT PARCH | PRPOINT . . . . | LN procedure identifier	COUNT | PRPOINT PARCH | PRPOINT | LN procedure identifier	STC	STC		SNOT	176 187	GENTXT, GENRLD, OPDREC, CLEARRG, DECOMP, SERR, STACKAPI, ROUTINE1, ROUTINE3
	)				function value		PRC STC EXC		SNOTSP		
59	Comma	Switch:=	(No change)	switch element COUNT | PRPOINT switch identifier | LN reserved	COUNT PRPOINT (No change)	STC	STC		SNOT	175	GENTXT, GENRLD, OPDREC, OPDTESF, SCHDL, STACKAPI SERR
	Delta						PRC				
61	Comma	(CSW) <	(No change)	parameter COUNT | (DSN) | DISPL st proc identifier	(No change)	STC	STC		SNOT	168 187 188 189	GENTXT, GENTXTP, OPDREC, TRINRE, LDWPLC, CLEARRG, DECOMP, OPDTESF, MOVEOPDK, STACKAPI, ARRTEST1, SERR, ROUTINE1, 7, 10, 11, 12, 14, 15
	)				function value		PRC STC EXC		SNOTSP		
62	Epsilon Eta End Else Semicolon	Goto		designational opnd		STC	STC		COMP	175	GENTXT, OPDREC, OPDTESF, SERR
63	Any operator except Not ([If Power	Monadic Minus		arithmetic operand	(No change)	EXC	EXC		COMP	163	GENTXT, GENTXTP, OPDREC, OPDTEST, ARRTEST1, SERR, ROUTINE2, 4, 5, 6, 7, 11
64	(	(Any operator)	(stacked			PRC STC	EXC		SNOT	168 174	GENTXT, MAXCH, LATRES, CLEARRG, STACKAPI
			(CSW) [	st proc identifier	COUNT DISPL st proc identifier	EXC	STC			176 183 187	MOVEOPTK, SERR

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
65	Not	(Any op. except arithm./ rel. op. or Not)	Not stacked			EXC	EXC		SNOT	164	SERR, OPTK
	If	If If-s	If stacked								
66	+	(Any op. except arithm. op.)	(Source op. stacked)	arithm operand	(No change)	EXC	EXC		SNOT	160	JBUFFER, SERR, MOVEOPTK
	-		(Source stacked)	arithm operand	(No change)						
67	(See matrices-App.V)		(Source stacked)	bool/arithm operand	(No change)	EXC	EXC		SNOT	162	MOVEOPTK
68	)	(		bool/arithm operand	(No change)	EXC	EXC		SNOTSP		OPDTEST
69	(See matrices-App.V)			arithm operand	bool/arithm operand	EXC	EXC		COMP	163 194	GENTXT, GENTXTP, OPDREC, MAXCH, TRINRE, ARRTEST2, SERR
	(entry from CP20)	Assign	(No change)			PRC	PRC		CP20		ROUTINE 1-9, 11-13, 15
70	If Else					EXC	STC		COMP		
71	Epsilon Eta End Semicolon		(No change)			STC	PRC		COMP		
72	Else	Else Assign				EXC	EXC		CP75		
		Else							CP79		
73	If	(See matrices - App.V)	(If stacked	operand		EXC	EXC		SNOT	160 161	SERR, MOVEOPTK
74	(Any rel. op.)	(Any rel. op.)	* stacked	operand	(No change)	EXC	EXC		SNOT	160 161	SERR, MOVEOPTK, STACKAPI
					API						

CP	Source Operator	Stack Operator at entry	Stack Operator at exit	Stack Operand at entry	Stack Operand at exit	Context entry	Context exit	Source pointer stepped	Exit	Errors	Subroutines used
75	(See matrices-App.V)		(No change)	(operand)		PRC STC EXC	PRC STC EXC		SNOT	194 195	SERR
76	(See matrices-App.V)			boolean operand boolean operand	boolean operand	EXC	EXC		COMP	162	GENTXT OPDREC, MAXCH, OPDTEST, ARRTEST2, SERR ROUTINE1, 3
77	(See matr-ices-App.V)	Not		boolean operand	(No change)	EXC	EXC		COMP	182	GENTXT, OPDREC, MAXCH, OPDTEST, ARRTEST1, SERR, ROUTINE1, 3
78	Then	If If-s	Then Then-s	boolean operand	LN	EXC	EXC PRC		SNOT	182	GENTXT, OPDREC, LATRES, OPDTEST, ARRTEST1, SERR, ROUTINE1, 3
79	(See matrices - App.V)	Else		arithm/bool/design arithm/bool/design, IN	arithm/bool/design	EXC	EXC		COMP	165 178	GENTXT, GENTXTP, OPDREC, TRINRE, OPDTEST, ARRTEST1, SERR, ROUTINE1, 3, 15
80	If	(	If stacked			EXC	EXC		SNOT	164	CLEARRG, MOVEOPTK, SERR
81	Eta	Do		proc identifier controlled variable FSN OPTB LN IN DISPL DISPL DISPL optimization COUNT optimized element(s)		PRC	PRC		SNOT		GENTXT, PLPRST
83	Gamma	Pi Phi Beta	(No change)	IN proc identifier	(No change)	PRC	PRC	8 bytes	SNOT		GENTXT, DECOMP
84	(See matrices-App.V) also entered from CP1, CP3, CP30					PRC STC EXC			CPERR1	173	SERR
85	Assign	Switch	Switch:=	switch identifier IN (reserved)	COUNT PRPOINT (No change)	PRC	STC		SNOT		MOVEOPTK
86	Else	If If-s	Then Then-s	(operand)	API unchanged/reserved	EXC	EXC PRC		SNOT	160	SERR, STACKAPI
87	Else	Then	Else	arithm/bool/design IN	(No change)	EXC	EXC		SNOT	178	GENTXT, OPDREC, MAXCH, LATRES, OPDTEST, ARRTEST1, SERR, ROUTINE 1-8, 10-12

Directory - IEX00	Description	Flowchart	-- Page
EODAD1	22	5	200
EODAD2	22	5	200
EODAD3	22	5	200
EODADIN	22	5	200
Final Exit	21	3	199
Initial Entry	21	3	199
PIROUT	21	4	199
SYNAD	22	5	200
SYNPR	22	5	200
PRINT	23	6	200
PRINTP	23	6	200

Initialization - IEX10	Description	Flowchart	-- Page
DDNAMES	28	9	202
FNDARSIZ	28	9	202
HEADINFO	28	9	202
Opening Data Sets	28	9 and 10	202
Option Processing	27	8	201
SPIE Macro	26	8	201

Scan I/II - IEX11	Description	Flowchart	-- Page
APOSTROF	56	15	205
ARRAY	65	24	209
ASSIGN	56	14	204
BEG1	63	22	208
BEGIN	62	20	207
BLANK	55	13	204
BLKAPOS	56	15	205
BOLCON	62	20	207
CIB	59	17	206
COB	59	17	206
COBSPEC	59	17	206
CLOSE2		34	214
CODE	65	20	207
COLON	57	15	205
COLONLST	66	26	210
COM	64	32	213
COMERR	64	32	213
COMPDEND	63	21	208
COMPFIN	48	34	214
COMMEND	64	32	213
COMMALST	66	26	210
COMPEND2		21	208
COMSPEC	62	18	206
DECPOINT	56	14	204
DELIMIT	60	18	206
END	63	21	208
ENDMISS	48	34	214
EODADIN	67	34	214
EROUT	60	19	207
Error Recording Routines	57	57	-
FIRSTBEG	63	63	-
FOR	64	20	207
FOREND	63	22	208
GENERATE	165	122	261
GIF	63	20	207
IDCHECK	65	31	213
IDCHECK1	59		

IER	64	28	211
IERSPEC	64	28	211
Initialization	52	13	204
LABEL	57	15	205
LEFTPARL	66	26	210
LETDEL	57	15	205
LIST	66	26	210
NORMAL	62	20	207
NPAFTAPO	57	15	205
PBLCKEND	64	22	208
POINT	55	14	204
PONTLST	66	27	211
PROCEDUR	67	29	212
PROCID	67	30	212
QUOTE	(see STRING)	(see STRING)	
READROUT	48	34	214
RIGHTPAR	55	13	204
RIGTPARL	66	26	210
SCALE	56	15	205
SEMCO	57	16	205
SEMC60	57	16	205
SEMCLST	66	26	210
SLASHLST	67	27	211
SPEC	65	31	213
SPECENT	65	31	213
STARTDEL	62	19	207
STATE	56	14	204
STRING	62	33	214
SWITCH	67	25	210
TED	63	20	207
TERMNT	(see EODADIN)	(see EODADIN)	
TESTLOOP	54	13	204
TRANSOP	55	13	204
TYPE	64	23	209
TYPEARRY	65	24	209
TYPESPEC	62	18	206
TYPPROC		29	212
VALUE		31	213
ZETAAPO	57	15	205

Identifier Table Manipulation - IEX20	Description	Flowchart	-- Page
ALLOSTOR	72	37	216
CLOSE	73	38	217
ITABPRNT	73	39	217
Initialization	71	36	216
READBLK	72	36	216
WRITITAB	73	38	217

Diagnostic Output - IEX21	Description	Flowchart	-- Page
Error Message Editing Routine	171	40 to 43	218-219

Scan III - IEX30	Description	Flowchart	-- Page
ARRAY	99	51	224
BETA	95	47	222
CHECK	100		
CLOBRACK	96	51	224
COMMA	96	51	224
CPOLEX	94		
CRIFLOW	90	61	229
CRIFODEL	91	61	229
CRIMA	90	60	228
CRITI	89	59	228
DECPOIN	92	65	231
DECPOIN1		65	231
DELCRIV	91	61	229

DIGIT0	92	64	230
DIGIT19	91	64	230
DIPOW	99	47	222
DO	96	49	223
EPSILON	95	47	222
ERROR1	88		
ETA	95	49	223
FOLI	88	57	221
FOR	95	47	222
GAMMA	98	49	223
GENTEST	88	46	221
GENTXT	165	122	261
ICHA	98	49	223
IDENT	88	57	227
INCOROP	99	70	233
INITIATE	86	46	221
INTCON	93	67	232
INTHAN	94	67	232
ITABMOVE	95	48	222
ITABMOP	95	48	222
LETRAF	91	62	229
LETTER	88	57	227
MOVE	99		
MOVERRO	99		
NOCRI	89	58	227
OMEGA	98	50	223
OPBRACK	96	51	224
OPERAND	97	54	225
OTHOP	98	50	223
OUCHA	99	49	223
PIPHI	95	47	222
PROFU	89	58	227
QUOTE	95	49	223
REALCON	94	68	232
REALHAN	94	69	233
RHO	99	51	224
SCAFACT	93	66	231
SEMIDELT	96	51	224
STEP	99	47	222
SUBMULT	97	54	225
SUCRIDEL	96	55	226
SUSCRITE	96	52 and 53	224-225
SUTABENT	98	53	225
SWILA	89	58	227
SWITCH	99	51	224
TXTTRAF	95		
WHILE	96	47	222
WRITE	100	62	229
ZETA	98		

Diagnostic Output - IEX31	Description	Flowchart --	Page
Error Message Editing Routine	171	71	234

Subscript Handling - IEX40	Description	Flowchart --	Page
Initialization	103	73	236
OPTAB	106	75	237
OTACHA	107	76	237
READ	107	74	236
SORT	107	74	236
SORTLE	106	74	236
SORTSU	106	74	236
Scan SUTAB	105	73	236
TERMIN	107	76	237

Compilation Phase - IEX50	Description	Flowchart	-- Page
COMP	119	79	239
CP0	120	80	240
CP1	122	80	240
CP3	164	80	240
CP4	121,127,133	81	240
CP6	123,144	82	241
CP7	161	82	241
CP8	154	82	241
CP12	151	83	241
CP16	120,134	83	241
CP17	154	84	242
CP18	154	84	242
CP19	161	84	242
CP20	152	85	242
CP21	151	85	242
CP22	162	86	243
CP23	161	86	243
CP24	161	86	243
CP25	161	86	243
CP26	162	87	243
CP27	162	87	243
CP28	163	87	243
CP29	163	87	243
CP30	163	87	243
CP31	163	87	243
CP33	162	87	243
CP34	156	88	244
CP36	128	88	244
CP38	122,130	89	244
CP40	144	90	245
CP41	122,130	91	245
CP43	147	92	246
CP45	147	93	246
CP47	147	94,95,96	247-248
CP49	148	97 and 98	248-249
CP51	128	99	249
CP52	127	100	250
CP54	130	100	250
CP56	122,123	101	250
CP57	134	101	250
CP59	122	102	251
CP61	137	103-4	251-252
CP62	123	105	252
CP63	159	105	252
CP64	134,136,156,158	106	253
CP65	156,158	106	253
CP66	158	106	253
CP67	158,159	106	253
CP68	159,161	106	253
CP69	159	107-112	253-256
CP70	162	113	256
CP71	162	113	256
CP72	163	113	256
CP73	163	113	256
CP74	163	113	256
CP75	163	113	256
CP76	158	114	257
CP77	158	115	257
CP78	154,156	115	257
CP79	157	116	258
CP80	157	117	258
CP81	151	117	258
CP83	136	118	259
CP84	163	118	259
CP85	122	118	259

CP86	164	118	259
CP87	156	119	259
DHEB2	160	108	254
DHZB1	160	110	255
DWG3	148	98	249
SERR	164	123	261
GENERATE	165	122	261
HOB1	161	109	254
Initialization	118	78	239
IPB1	160	111	255
ISB1	160	112	256
IUB1	160	111	255
I1B1	160	109	254
JBUFFER	164	121	260
NXTOPT	164	121	260
OPDREC	165	120	260
SNOT	119	79	239
SNOTSP	119	79	239
USA1	148	98	249
UVA1	151	98	249

Termination Phase - IEX51	Description	Flowchart --	Page
Error Message Editing Routine	171	128	264
Main Storage Release	169	126	263
Object Time Table Output	167-169	125	263
Print Storage Requirements	169	127	264

ALGOL Library	Description	Flowchart --	Page
ALGIN	177	130	265
ALGTRMN	179	132	266
CAP1	177	132	266
CAP2	178	132	266
CLOSE	184	142	271
CLOSEGP	181	142	271
CLOSEPE	185	142	271
CSWE1	178	-	-
CSWE2	179	-	-
DCBEXIT	185	141	271
EPILOG	177	131	266
FSAERR	179	-	-
GET	181	140	270
GETMSTO	179	-	-
INARRAY	181	134	267
INBARRAY	181	134	267
INBOOLEAN	182	134	267
ININTEGER	182	134	267
INPUT	181	140	270
INREAL	182	134	267
INSYMBOL	182	135	268
INTARRAY	181	134	267
LOADPP	179	130	265
NEXTREC	185	142	271
OPEN	185	141	271
OPENGP	181	141	271
OUTARRAY	182	135	268
OUTBARRAY	182	136	268
OUTBOOLEAN	182	135	268
OUTINTEGER	182	135	268
OUTPUT	181	140	270
OUTREAL	182	135	268
OUTSTRING	182	136	268
OUTSYMBOL	183	136	268
OUTTARRAY	183	135	268
PUT	180	140	270
PIEROUT	179	-	-
PROLOG	177	131	266

RETPROG	178	132	266
SpDECL	179	131	266
SYSACT	183	137,138,139	269-270
TRACE	179	130	265
VALUCALL	178	130	265

Indexes to program logic manuals are consolidated in the publication OS Master Index to Logic Manuals, Order No. GY28-6717. For additional information about any subject listed below, refer to other publications listed for the same subject in the Master Index.

Actual parameter 131-132,165
Address Table 169,187
ALGOL Compiler
  Summary 13-16
  Flowchart (overall flow) 197
ALGOL Library 176-186
All-purpose internal name 274
Area Size Table 24
Arithmetic expression 158-161
Arithmetic operators 44,158-161
Array 124-131
Array (actual parameter) 135
Array (formal parameter - call by name) 132
Array (formal parameter - call by value) 131,178
Array declaration
  in Identifier Table 36-37,65-66
  in intermediate source text 44
  allocation for Storage Mapping Function 73
  internal name 274
  object code 125-126,127-130
Assignment statement 151-152

Blank 55-56
Block 33-34,41-42,75,120
Boolean constant
  in Scan I/II Phase 45,62-63
  in Scan III Phase 95
  internal name 274

Character codes 272-273
Character string
  in Scan I/II Phase 30,45,62
  in Scan III Phase 95
  internal name 274
Characteristic 36
Code procedure 65,98,135-136
Code tables 272-273
Column Vector 116
Comment 62,64
Common Area 19,28
Common Data Area 176
Common Work Area 23-25
Compilation Phase (IEX50)
  Description 108-167
  Flowcharts 238-261
Compiler area requirements 285
Compiler Control Field (HCOMPMOD) 24,27,276
Compiler entry 21
Compiler exit 21
Compiler interface 21
Compiler invocation 26
Compiler modules
  see Compiler phases
Compiler options 27
Compiler organization 13
Compiler output 14
Compiler phases 13
  Directory (IEX00) 21-25
  Initialization (IEX10) 26-29
  Scan I/II (IEX11) 30-68
  Identifier Table Manipulation (IEX20) 69-73
  Diagnostic Output (IEX21) 171-175
  Scan III (IEX30) 74-100
  Diagnostic Output (IEX31) 171-175
  Subscript Handling (IEX40) 101-107
  Compilation (IEX50) 108-167
  Termination (IEX51) 167-169
Component (switch) 67,120-122
Compound statement 33-34,41-42,120
Conditional expression 154-157
Conditional statement 153-154
Constant
  in Scan I/II Phase 45
  in Scan III Phase 77,91-94
  internal name 274
Constant Pool 30,52,62,74,94-95,187-188
Context switching 161-162
Counting Loop 80,138
Conventions 20
Critical identifier 82,89-90
Critical Identifier Table 82-83
Cyclical address increment 143-144

Data Control Blocks 23,25,29
Data Control Block addresses 24,29
Data Set Table 168,190
Data sets 14,17,28-29,35,70,77,102,110
Data Storage Area 191-192
DDnames 28
Decision matrices 116,278-279
Declarations
  entry in Identifier Table 31,34,35-38,64-67
  representation in intermediate source text 44-45
  allocation of object time storage fields 72-73
  internal names of declared identifiers 274
  replacement of identifiers by internal names 75,88
  see also
    Array declaration
    Code procedure declaration
    Label
    Procedure declaration
    Switch declaration
Declarator 44,64-67
Delimiter
  initial treatment 43-45
  internal representation 61,272
Delimiter Table 60,61
Diagnostic output (IEX21, IEX31, IEX51)
  Description 170-175
  Flowcharts:
    IEX21 218
    IEX31 234
    IEX51 264
Directory (IEX00)
  Description 21-25
  Flowcharts 198-200

Elementary Loop 80,139
End of Data 22,24,48,52,71,119
END record 169,187
ERET 22,24,52,71,86,103,118
Error detection 170-171,280-284
Error Pool 19,28,172
ESD record 68,136,168,187
Expression Context Matrix 116,279

Fixed Storage Area 176
Flowcharts 196-271
For statement
  identification in intermediate source text 41-42
  classification 79-81
  object code 138-151
For statement closing entry 39
For statement heading entry 39
For Statement Table 81,139
Formal parameter
  see Specifications
  internal name 274
  storage field 193
  object time treatment 131-135
Function designator
  see Type procedure

Goto statement 123
Group Table 45

HCOMPMOD Control Field
  see Compiler Control Field

Identifier
  entry in Identifier Table 35-38
  object time storage field 73
  internal name 274
  replacement by internal name 75
  see also
    Declarations
    Specifications
    Operands
Identifier entry 36
Identifier Group Number 41-42
Identifier Table 30,35-41,69-70,78-79
Identifier Table Manipulation Phase (IEX20)
  Description 69-73
  Flowcharts 215-217
Initialization Phase (IEX10)
  Description 26-29
  Flowcharts 201-202
Input/Output activity 16,35,70,77,102,110
Input/Output procedure
  see Standard input/output procedures
Intermediate storage procedure
  see Standard input/output procedures
Internal name
  table of internal names 274
  processing
    identifiers (see Declarations and Specifications)
    character strings 62
    boolean constants 62-63
    constants 11,91-94
  treatment in generation of object code 110-112
Invalid branch 89
Invalid character 57
Invalid identifier 64
I/O error
  see Unrecoverable I/O error

I/O Table 138

Label 44-45,57,122
Label Address Table
  compile time 118-119,167-168
  object time 189
Left Variable Table 82

Library
  see ALGOL Library
Load module 188-195
Logical error 162-164
Logical operator 157-158
Loop count 138

Machine system 13
Main storage (compile time use) 18,286-294
Main storage (object time use) 188,295
Mathematical function
  see Standard mathematical functions
Message Pool 172
Modification Level 1 text 43-45
Modification Level 2 text 84-85

Normal Loop 80,140
Note Table 191
Numbers
  see Constants

Object module 187
Object stack
  (= extension of current Data Storage Area) Area)
Operand
  internal names 274
  treatment in generation of object code 110-112
Operand Stack 110-113
Operator Stack 110-113
Operators (ALGOL-defined)
  see Delimiters
Operators (internal)
  in Scope Handling Stack (Scan I/II Phase) 20
  in Modification Level 1 text 43-45,272
  in Modification Level 2 text 84,272
  in Operator Stack (Compilation Phase) 273
Optimizable subscript expression 81,101
  see also Subscript optimization
Optimization Table 103
Option
  see Compiler options

Parameter delimiter 45,57
Precompiled procedure 46,68,187-188
  see also Code procedure
Preliminary Error Pool 24
Procedure 33-34,41-42,75,131-136
Procedure call 132-133
Procedure declaration
  in Identifier Table 37-38,67
  in intermediate source text 41-42,44
  internal name 274
  object code 131-132,133-134
Program 46
Program block heading entry 38
Program Block Number 41-42
Program Block Number Table 46
Program Block Table (object time) 188-189

Program Block Table II 70-71
Program Block Table III 119,120
Program Block Table IV 167
Program Context Matrix 116,278
Program interrupt 21,52,71,86,103,118,179

Register save area 23
Registers (compile time)
register use in Compilation Phase 116
Registers (object time)
register use 194-195
register control 113-115
Relation 158-161
Relational operators 44,160
RLD record 136,168,187
Return Address Stack 193-194
Routine Index 307
Row Vector 116

Scan I/II Phase (IEX11)
Description 30-67
Flowcharts 203-214
Scan III Phase (IEX30)
Description 75-100
Flowcharts 220-233
Scope Handling Stack 42-43
Scope Table 45
Semicolon Table 38-39,63,67
Serious error 170-171
Source module 13,30,46
Source text 17-19,43-46,84-85
Special Use Bits 36,80,84,88-89
Specification
entry in Identifier Table 34,35-38,65
representation in intermediate source text 44
allocation of object time storage field 72-73
internal names of specified identifiers 274
replacement of identifier by internal name 75,79,88
see also Formal parameter
Specificator 44,65
SPIE macro 26,21,22,86
Stack operator
see Scope Handling Stack
Operator Stack
Standard input/output procedures 180-185
entries in Identifier Table 86
internal names 275
replacement of identifiers by internal names 75,88
call (object code) 136-137
ESD records 168
INARRAY 181
INBARRAY 181
INBOOLEAN 182
ININTEGER 182
INREAL 182
INSYMBOL 182
INTARRAY 181
OUTARRAY 182
OUTBARRAY 182
OUTBOOLEAN 182
OUTINTEGER 182
OUTREAL 182
OUTSTRING 182
OUTSYMBOL 183
OUTTARRAY 183
GET 181
PUT 180
SYSACT 183
Standard mathematical functions 187
entries in Identifier Table 86
internal names 275
replacement of identifiers by internal names 75,88
call (object code) 136-137
ESD records 168
ABS 137
ARCTAN 186
COS 186
ENTIER 179
EXP 186
LENGTH 137
LN 186
SIGN 137
SIN 186
SQRT 186
Statement Context Matrix 116,279
Storage Mapping Function 125-126,128-129 193
String
see Character String
Subscript expression 77,81,96-98,101-107
Subscript incrementation 144,151
Subscript initialization 144,148-151
Subscript optimization 143-144,148-151
Subscript Handling Phase
Description 101-107
Flowcharts 235-237
Subscript Table 82-83
Subscripted Variable 126-127,130-131
Switch (ALGOL-defined) 120-122
Switch declaration
in Identifier Table 37-38,67
in intermediate source text 44
internal name 274
object code 120-122
Switch designator 121-122
Syntax Check Mode 170-171

Terminating error 171
Termination Phase (IEX51)
Description 167-169
Flowcharts 262-264
Thunk
(= actual parameter code sequence)
Type declaration 36-37,44,64
Type procedure 37,131-132

Undeclared identifier 75,88
Unrecoverable I/O error 22,52,71,86,103, 118,185

Value call 131-134,178

Warning error 170

Zero-base address 124-125

Y33-8000-0

IBM System/360 Printed in U.S.A. Y33-8000-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

IBM Technical Newsletter

File Number S360-26

Re: Form No. Y33-8000-0

This Newsletter No. Y33-8001

Date December 15, 1967

Previous Newsletter Nos. None

IBM SYSTEM/360 OPERATING SYSTEM
ALGOL Program Logic Manual

This Technical Newsletter amends the publication IBM System /360 Operating System: ALGOL Program Logic Manual, Form Y33-8000-0. In replacement pages, each change or addition to the original text is indicated by a vertical bar in the left margin. Revised figures are marked by the symbol • to the left of the caption.

Pages to be Inserted	Pages to be Removed
Front Cover and Inside Front Cover	Front Cover and Inside Front Cover
13-14	13-14
65-66	65-66
137-140	137-140
149-150	149-150
179-192	179-192
195-196	195-196
203-204	203-204
207-208	207-208
211-214	211-214
217-218	217-218
221-222	221-222
227-230	227-230
233-234	233-234
243-244	243-244
249-250	249-250
259-260	259-260
263-268	263-268
271-280	271-280
285-286	285-286
315 & Back Cover	315 & Back Cover
Reader's Comment Request	Reader's Comment Request

The following additional changes should be made

Page	Amendment
77	The constants 0.0095'86 and 0.95×10^{84} should be changed to 0.0095'54 and 0.95×10^{52}, respectively

IBM Nordic Laboratory, Technical Communications, Box 962, Lidingö 9, Sweden

PRINTED IN U.S.A.

Page	Amendment
82	In the left column on lines 12 and 13, insert the following: "and the for statement is an Elementary Loop".
83	In Figure 44, replace the explanation opposite <X> Bits 2 and 3, as follows: "(Binary 00. Chain Bits used in Subscript Handling Phase)".
112	In figure 54, change the last two object code instructions as follows: A REGX, <DISP-B> (CDSA) ST REGX, <DISP-C> (CDSA)
165	In the right column on line 3, add the following: "Load CDSA with Data Storage Area base address". In the same column, lines 9-11, delete the text: "Load CDSA with Data Storage Area base address".

Summary of Amendments

This Newsletter corrects various minor errors in the publication, which relates to release 11 of the IBM System/360 Operating System.

Note: Please file this cover letter at the back of the publication.

IBM Technical Newsletter

File Number	S360-26
Re: Form No.	Y33-8000-0
This Newsletter No.	Y33-8003
Date	June 19, 1968
Previous Newsletter Nos.	Y33-8001

IBM SYSTEM/360 OPERATING SYSTEM
ALGOL (F) Program Logic Manual

This Technical Newsletter amends the publication IBM System/360 Operating System: ALGOL (F) Program Logic Manual, Form Y33-8000-0. Changes in the text are indicated by a vertical bar in the left margin.

Pages to be Inserted	Pages to be Removed
191-192	191-192
267-268	267-268
269-270	269-270
271-271.1	271-272
271.2-272	

Summary of Changes

This Newsletter reflects changes made in the logic of certain ALGOL Library routines, which take effect with Release 16 of the Operating System.

Note: Please file this cover letter at the back of the publication. Cover letters provide a useful record of changes made in a publication.

IBM Nordic Laboratory, Technical Communications, Box 962, Lidingö 9, Sweden

Technical Newsletter

File No. S360-26 (OS)

Base Publ. No. GY33-8000-0

This Newsletter No. GN33-8129

Date January 15, 1972

Previous Newsletter Nos. Y33-8001,
Y33-8003

OS ALGOL (F) Compiler Logic

©IBM Corp. 1967

This Technical Newsletter, a part of release 21 of the IBM System/360 Operating System, provides replacement pages for the subject manual. These replacement pages remain in effect for subsequent versions and modifications unless specifically altered. Pages to be inserted and/or removed are listed below.

Front Cover, Preface	173,174
11 (added)	185,186
13,14	279,280
73,74	283,284
169,170	313,314

A change to the text or to an illustration is indicated by a vertical line to the left of the change.

Summary of Amendments

References to OS publications have been updated to reflect current titles.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Nordic Laboratory, Publications Development, Box 962, S-181 09 Lidingö 9, Sweden

PRINTED IN U.S.A.