File No. S360-25 (OS)
Form Y28-6638-1

Program Logic

IBM System/360 Operating System

FORTRAN IV (G) Compiler

Program Logic Manual

Program Number 360S-FO-520

This publication describes the internal logic of the FORTRAN IV (G) compiler.

Program Logic Manuals are intended for use by IBM customer engineers involved in program maintenance, and by systems programmers involved in altering the program design. Program logic information is not necessary for program operation and use; therefore, distribution of this manual is limited to persons with program maintenance or modification responsibilities.

The FORTRAN IV (G) compiler is a processing program of the IBM System/360 Operating System. It translates one or more source modules written in the FORTRAN language into an object module that can be processed into an executable load module by the linkage editor.

Restricted Distribution

PREFACE

This publication provides customer engineers and other technical personnel with information describing the internal organization and operation of the FORTRAN IV (G) compiler. It is part of an integrated library of IBM System/360 Operating System Program Logic Manuals. Other publications required for an understanding of the FORTRAN IV (G) compiler are:

IBM System/360 Operating System:

Principles of Operation, Form A22-6821

FORTRAN IV Language, Form C28-6515

Introduction to Control Program Logic, Program Logic Manual, Form Y28-6605

FORTRAN IV (G and H) Programmer's Guide, Form C28-6817

Any reference to a Programmer's Guide in this publication applies to FORTRAN IV (G and H) Programmer's Guide, Form C28-6817. The FORTRAN IV (G) Programmer's Guide, Form C28-6639, (to which references may exist in this publication) has been replaced by the combined G and H Programmer's Guide.

Although not required, the following publications are related to this publication and should be consulted:

IBM System/360 Operation System:

Sequential Access Methods, Program Logic Manual, Form Y28-6604

Concepts and Facilities, Form C28-6535

Supervisor and Data Management Macro-Instructions, Form C28-6647

Linkage Editor, Program Logic Manual, Form Y28-6610

System Generation, Form C28-6554

This publication consists of two sections:

Section 1 is an introduction that describes the FORTRAN IV (G) compiler as a whole, including its relationship to the operating system. The major components of the compiler and relationships among them are also described in this section.

Section 2 consists of a discussion of compiler operation. Each component of the compiler is described in sufficient detail to enable the reader to understand its operation, and to provide a frame of reference for the comments and coding supplied in the program listing. Common data such as tables, blocks, and work areas is discussed only to the extent required to understand the logic of each component. Flowcharts are included at the end of this section.

Following Section 2, are appendixes that contain reference material.

If more detailed information is required, the reader should see the comments, remarks, and coding in the FORTRAN IV (G) program listing.

Second Edition (May 1968)

This is a major revision of, and makes obsolete, the previous edition, Form Y28-6638-0, and Technical Newsletters Y28-6384, Y28-6386, Y28-6388, and Y28-6820. Changes to the text, and small changes to illustrations, are indicated by a vertical line to the left of the change; changed or added illustrations are denoted by the symbol • to the left of the caption.

The specifications contained in this publication, as amended by TNL Y28-6826, dated November 15, 1968, correspond to Release 17 of the IBM System/360 Operating System.

Changes are periodically made to the specifications herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

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

Address comments concerning the contents of this publication to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, N. Y. 10020.

SECTION 1: INTRODUCTION TO THE COMPILER 9
Purpose of the Compiler 9
Machine Configuration 9
Compiler and System/360 Operating System 9
Compiler Design 9
Limitations of the Compiler 9
Compiler Implementation 10
POP Language 10
Compiler Organization 10
Control Phase: Invocation (IEYFORT) 12
Phase 1: Parse (IEYPAR) 12
Phase 2: Allocate (IEYALL) 12
Phase 3: Unify (IEYUNF) 12
Phase 4: Gen (IEYGEN) 12
Phase 5: Exit (IEYEXT) 13
Roll (IEYROL) 13
Compiler Storage Configuration 15
Compiler Output 15
Object Module 17
Components of the Object Module . . 17
Object Module General Register Usage 20
Source Module Listing 20
Object Module Listing 20
Storage Maps 21
Error Messages 21
Common Error Messages 21
Compiler Data Structures 21
Rolls and Roll Controls 21
ROLL ADR Table 22
BASE, BOTTOM, and TOP Tables 23
Special Rolls 24
Central Items, Groups, and Group Stats 24
Other Variables 26
Answer Box 26
Multiple Precision Arithmetic . . . 26
Scan Control 26
Flags 27
Quotes 27
Messages 27
Compiler Arrangement and General Register Usage 28
Pointers 29
Drivers 30
Operation Drivers 30
Control Drivers 31

SECTION 2: COMPILER OPERATION 33
Invocation Phase (IEYFORT) 33
IEYFORT, Chart 00 33
IEYPRNT, Chart 00A4 33
PRNTHEAD, Chart 01A2 34
IEYREAD, Chart 01A4 34
IEYPCH, Chart 02A3 34
PRNTMSG, Chart 03A1 34
IEYMOR, Chart 01D1 34
IEYNOCR 34
IEYRETN, Chart 03A2 35
OPTSCAN, Chart AA 35
DDNAMES, Chart AB 35
HEADOPT, Chart AC 35
TIMEDAT, Chart AD 35
Output from IEYFORT 35
Phase 1 of the Compiler: Parse (IEYPAR) 36
Flow of Phase 1, Chart 04 37
PRINT and READ SOURCE, Chart BA . . 37
STA INIT, Chart BB 38
LBL FIELD XLATE, Chart BC 38
STA XLATE, Chart BD 38
STA FINAL, Chart BE 39
ACTIVE END STA XLATE, Chart BF . . . 39
PROCESS POLISH, Chart BG 39
Output from Phase 1 39
Polish Notation 39
Source Listing 42
Phase 2 of the Compiler: Allocate (IEYALL) 44
Flow of Phase 2, Chart 05 45
ALPHA LBL AND L SPROGS, Chart CA . . 45
ALPHA SCALAR ARRAY AND SPROG, Chart CA 45
PREP EQUIV AND PRINT ERRORS, Chart CB 45
BLOCK DATA PROG ALLOCATION, Chart CC 46
PREP DMY DIM AND PRINT ERRORS, Chart CD 46
PROCESS DO LOOPS, Chart CE 46
PROCESS LBL AND LOCAL SPROGS, Chart CF 46
BUILD PROGRAM ESD, Chart CG 46
ENTRY NAME ALLOCATION, Chart CH . . 46
COMMON ALLOCATION AND OUTPUT, Chart CI 47
EQUIV ALLOCATION PRINT ERRORS, Chart CK 47
BASE AND BRANCH TABLE ALLOC, Chart CL 47
SCALAR ALLOCATE, Chart CM 47
ARRAY ALLOCATE, Chart CN 47
PASS 1 GLOBAL SPROG ALLOCATE, Chart CO 48
SPROG ARG ALLOCATION, Chart CP . . . 48
PREP NAMELIST, Chart CQ 48
LITERAL CONST ALLOCATION, Chart CR . 48
FORMAT ALLOCATION, Chart CS 48
EQUIV MAP, Chart CT 48
GLOBAL SPROG ALLOCATE, Chart CU . . 48
BUILD NAMELIST TABLE, Chart CV . . . 48
BUILD ADDITIONAL BASES, Chart CW . . 49
DEBUG ALLOCATE, Chart CX 49
Output From Phase 2 49
Error Messages Produced by Allocate 49
Unclosed DO Loops 49
Storage Maps Produced by Allocate . 50
Subprogram List 51
Cards Produced by Allocate 51
Phase 3 of the Compiler: Unify (IEYUNF) 51
Flow of Phase 3, Chart 07 52
ARRAY REF ROLL ALLOTMENT, Chart DA . 52
CONVERT TO ADR CONST, Chart DB . . . 52
CONVERT TO INST FORMAT, Chart DC . . 52

DO NEST UNIFY, Chart DD 53
IEYROL Module 53
Phase 4 of the Compiler: Gen (IEYGEN) 53
Flow of Phase 4, Chart 08 53
ENTRY CODE GEN, Chart EA 54
PROLOGUE GEN, Chart EB 54
EPILOGUE GEN, Chart EC 54
GET POLISH, Chart ED 54
LBL PROCESS, Chart EF 54
STA GEN, Chart EG 54
STA GEN FINISH, Chart EH 55
Phase 5 of the Compiler: Exit (IEYEXT) . 55
Flow of Phase 5, Chart 09 55
PUNCH TEMP AND CONST ROLL, Chart FA 55
PUNCH ADR CONST ROLL, Chart FB . . . 56
PUNCH CODE ROLL, Chart FC 56
PUNCH BASE ROLL, Chart FD 56
PUNCH BRANCH ROLL, Chart FE 56
PUNCH SPROG ARG ROLL, Chart FF . . . 56
PUNCH GLOBAL SPROG ROLL, Chart FG . 57
PUNCH USED LIBRARY ROLL, Chart FH . 57
PUNCH ADCON ROLL, Chart FI 57
ORDER AND PUNCH RLD ROLL, Chart FJ . 57
PUNCH END CARD, Chart FK 57
PUNCH NAMELIST MPY DATA, Chart FL . 57
Output From Phase 5 57

APPENDIX A: THE POP LANGUAGE 127
POP Instructions 127
Transmissive Instructions 127
Arithmetic and Logical Instructions .130
Decision Making Instructions 131
Jump Instructions 133
Roll Control Instructions 133
Code Producing Instructions 134
Address Computation Instructions . . .134
Indirect Addressing Instruction . . .135
Labels 135
Global Labels 135
Local Labels 136
Assembly and Operation 136
POP Interpreter 136
Assembler Language References to POP Subroutines 137
Global Jump Instructions 137
Local Jump Instructions 138

APPENDIX B: ROLLS USED IN THE COMPILER .140
Roll 0: LIB Roll 140
Roll 1: SOURCE Roll 140
Roll 2: IND VAR Roll 141
Roll 2: NONSTD SCRIPT Roll 141
Roll 3: NEST SCRIPT Roll 141
Roll 4: POLISH Roll 141
Roll 4: LOOP SCRIPT Roll 142
Roll 5: LITERAL CONST Roll 142
Roll 7: GLOBAL SPROG Roll 142
Roll 8: FX CONST Roll 143
Roll 9: FL CONST Roll 143
Roll 10: DP CONST Roll 143
Roll 11: COMPLEX CONST Roll 143
Roll 12: DP COMPLEX CONST Roll 143
Roll 13: TEMP NAME Roll 143
Roll 13: STD SCRIPT Roll 144
Roll 14: TEMP Roll 144
Roll 15: DO LOOPS OPEN Roll 144
Roll 15: LOOPS OPEN Roll 144
Roll 16: ERROR MESSAGE Roll 144
Roll 16: TEMP AND CONST Roll 144
Roll 17: ERROR CHAR Roll 145
Roll 17: ADCON Roll 145
Roll 18: INIT Roll 145
Roll 18: DATA SAVE Roll 145
Roll 19: EQUIVALENCE TEMP (EQUIV TEMP) Roll 145
Roll 20: EQUIVALENCE HOLD (EQUIV HOLD) Roll 145
Roll 20: REG Roll 146
Roll 21: BASE TABLE Roll 146
Roll 22: ARRAY Roll 146
Roll 23: DMY DIMENSION Roll 147
Roll 23: SPROG ARG Roll 147
Roll 24: ENTRY NAMES Roll 147
Roll 25: GLOBAL DMY Roll 148
Roll 26: ERROR Roll 148
Roll 26: ERROR LBL Roll 148
Roll 27: LOCAL DMY Roll 148
Roll 28: LOCAL SPROG Roll 149
Roll 29: EXPLICIT Roll 149
Roll 30: CALL LBL Roll 149
Roll 30: ERROR SYMBOL Roll 149
Roll 31: NAMELIST NAMES Roll 149
Roll 32: NAMELIST ITEMS Roll 150
Roll 33: ARRAY DIMENSION Roll 150
Roll 34: BRANCH TABLE Roll 150
Roll 35: TEMP DATA NAME Roll 150
Roll 36: TEMP POLISH Roll 151
Roll 36: FX AC Roll 151
Roll 37: EQUIVALENCE Roll 151
Roll 37: BYTE SCALAR Roll 151
Roll 38: USED LIB FUNCTION Roll 152
Roll 39: COMMON DATA Roll 152
Roll 39: HALF WORD SCALAR Roll 152
Roll 40: COMMON NAME Roll 152
Roll 40: TEMP PNTR Roll 153
Roll 41: IMPLICIT Roll 153
Roll 42: EQUIVALENCE OFFSET Roll 153
Roll 42: FL AC Roll 153
Roll 43: LBL Roll 153
Roll 44: SCALAR Roll 154
Roll 44: HEX CONST Roll 154
Roll 45: DATA VAR Roll 154
Roll 46: LITERAL TEMP (TEMP LITERAL) Roll 155
Roll 47: COMMON DATA TEMP Roll 155
Roll 47: FULL WORD SCALAR Roll 155
Roll 48: COMMON AREA Roll 155
Roll 48: NAMELIST ALLOCATION Roll . . .155
Roll 49: COMMON NAME TEMP Roll 156
Roll 50: EQUIV ALLOCATION Roll 156
Roll 51: RLD Roll 156
Roll 52: COMMON ALLOCATION Roll 156
Roll 52: LOOP CONTROL Roll 156
Roll 53: FORMAT Roll 157
Roll 54: SCRIPT Roll 157
Roll 55: LOOP DATA Roll 157
Roll 56: PROGRAM SCRIPT Roll 158
Roll 56: ARRAY PLEX Roll 158
Roll 57: ARRAY REF Roll 159
Roll 58: ADR CONST Roll 159
Roll 59: AT Roll 159
Roll 60: SUBCHK Roll 160
Roll 60: NAMELIST MPY DATA Roll 160
Roll 62: GENERAL ALLOCATION Roll 160
Roll 62: CODE Roll 160

Roll 63: AFTER POLISH Roll 161
Work and Exit Rolls 161
WORK Roll 161
EXIT Roll 161

APPENDIX C: POLISH NOTATION FORMATS . .163
General Form 163
Labeled Statements 163
Array References 163
ENTRY Statement 164
ASSIGN Statement 164
Assigned GO TO Statement 164
Logical IF Statement 164
RETURN Statement 164
Arithmetic and Logical Assignment Statement 164
Unconditional GO TO Statement 165
Computed GO TO Statement 165
Arithmetic IF Statement 165
DO Statement 165
CONTINUE Statement 166
PAUSE and STOP Statements 166
END Statement 166
BLOCK DATA Statement 166
DATA Statement and DATA in Explicit Specification Statements 166
I/O List 167
Input Statements 167
FORMATTED READ 167
NAMELIST READ 168
UNFORMATTED READ 168
READ Standard Unit 168
Output Statements 168
FORMATTED WRITE 168
NAMELIST WRITE 169
UNFORMATTED WRITE 169
PRINT 169
PUNCH 169
Direct Access Statements 169
READ, Direct Access 169
WRITE, Direct Access 170
FIND 170
DEFINE FILE 170
END FILE Statement 170
REWIND Statement 171
BACKSPACE Statement 171
Statement Function 171
FUNCTION Statement 171
Function (Statement or Subprogram) Reference 171
SUBROUTINE Statement 171
CALL Statement 172
Debug Facility Statements 172
AT 172
TRACE ON 172
TRACE OFF 172
DISPLAY 173

APPENDIX D: OBJECT CODE PRODUCED BY THE COMPILER 175
Branches 175
Computed GO TO Statement 175
DO Statement 175
Statement Functions 176
Subroutine and Function Subprograms . .176
Input/Output Operations 177
Formatted Read and Write Statements .177
Second List Item, Formatted 177
Second List Array, Formatted 178
Final List Entry, Formatted 178
Unformatted Read and Write Statements 178
Second List Item, Unformatted 178
Second List Array, Unformatted 178
Final List Entry, Unformatted 178
Backspace, Rewind, and Write Tape Mark 178
STOP and PAUSE Statements 179
NAMELIST READ and WRITE 179
DEFINE FILE Statement 179
FIND Statement 179
Direct Access READ and WRITE Statements 179
FORMAT Statements 180
FORMAT Beginning and Ending Parentheses 180
Slashes 180
Internal Parentheses 180
Repetition of Individual FORMAT Specifications 180
I,F,E, and D FORMAT Codes 180
A FORMAT Code 180
Literal Data 180
X FORMAT Code 181
T FORMAT Code 181
Scale Factor-P 181
G FORMAT Code 181
L FORMAT Code 181
Z FORMAT Code 181
Debug Facility 181
DEBUG Statement 181
Beginning of Input/Output 181
End of Input/Output 181
UNIT Option 181
TRACE Option 182
SUBTRACE Option 182
INIT Option 182
SUBCHK Option 183
AT Statement 183
TRACE ON Statement 183
TRACE OFF Statement 183
DISPLAY Statement 183

APPENDIX E: MISCELLANEOUS REFERENCE DATA 185
Parse Label List 185
Supplementary Parse Label List 185
Allocate Label List 193
Supplementary Allocate Label List . . .193
Unify Label List 196
Supplementary Unify Label List 196
Gen Label List 198
Supplementary Gen Label List 198
Exit Label List 208
Supplementary Exit Label List 208

APPENDIX F: OBJECT-TIME LIBRARY SUBPROGRAMS 212
IHCFCOMH 213
READ/WRITE Routines 214
READ/WRITE Statements Not Using NAMELIST 214
Examples of IHCFCOMH READ/WRITE Statement Processing 218
READ/WRITE Statement Using NAMELIST 221
Input/Output Device Manipulation Routines 221

Write-to-Operator Routines 222
Utility Routines 222
Conversion Routines (IHCFCVTH) 223
IHCFIOSH 224
Blocks and Table Used 224
Unit Blocks 224
Unit Assignment Table 225
Buffering 226
Communication with the Control
Program 227
Operation 227
Initialization 227
Read 228
Write 228
Device Manipulation 229
Closing 229
IHCDIOSE 229
Blocks and Table Used 230
Unit Blocks 230
Unit Assignment Table 232
Buffering 232
Communication With the Control
Program 233
Operation 233
File Definition Section 233
File Initialization Section 233
Read Section 234
Write Section 235
Error Processing 235
Termination Section 236
IHCIBERH 236
IHCDBUG 236
Items and Buffer 236
Operation 236
Subroutines 237
IHCTRCH 238
IHCFINTH 238
IHCERRM 239
Alter Option Table Routine
(IHCFOPT) 240

GLOSSARY 259

INDEX 263

FIGURES

Figure 1. Overall Operation of the Compiler 11
Figure 2. Compiler Organization Chart 14
Figure 3. Compiler Storage Configuration 15
Figure 4. Compiler Output 16
Figure 5. Object Module Configuration 17
Figure 6. Example of Use of Save Area 18
Figure 7. Roll Containing K Bytes of Information 23
Figure 8. Roll Containing L Bytes of Reserved Information and K Bytes of New Information 24
Figure 9. Roll With a Group Size of Twelve 25
Figure 10. Roll with Variable Group Size 25
Figure 11. First Group Stats Table 26
Figure 12. Second Group Stats Table 26
Figure 13. Scan Control Variables 27
Figure 14. Quotes Used in the Compiler 27
Figure 15. Compiler Arrangement with Registers 28
Figure 16. Relationship Between IHCFCOMH and Input/Output Data Management Interfaces 213
Figure 17. Format of a Unit Block for a Sequential Access Data Set .224
Figure 18. Unit Assignment Table Format 226
Figure 19. CTLBLK Format 227
Figure 20. Format of a Unit Block for a Direct Access Data Set . . .230
Figure 21. Unit Assignment Table Entry for a Direct Access Data Set 232

TABLES

Table 1. Internal Configuration of Operation Drivers 31
Table 2. Internal Configuration of Control Drivers . . 32
Table 3. Rolls Used by Parse . . . 36
Table 4. Rolls Used by Allocate . 44
Table 5. Rolls Used by Unify . . . 52
Table 6. Rolls Used by Gen 53
Table 7. Rolls Used by Exit . . . 55
Table 8. POP Instruction Cross-Reference List 139
Table 9. IHCFCOMH FORMAT Code Processing 215
Table 10. IHCFCOMH Processing for a READ Requiring a Format 219
Table 11. IHCFCOMH Processing for a WRITE Requiring a Format 219
Table 12. IHCFCOMH Processing for a READ Not Requiring a Format . . .220
Table 13. IHCFCOMH Processing for a WRITE Not Requiring a Format . . .220
Table 14. Description of Option Table Entry 241
Table 15. Description of Option Table Preface 242
Table 16. IHCFCOMH Subroutine Directory 246
Table 17. IHCFCVTH Subroutine Directory 246
Table 18. IHCFIOSH Routine Directory 251
Table 19. IHCDIOSE Routine Directory 251

CHARTS

Chart 00. IEYFORT (Part 1 of 4) . . 59
Chart 01. IEYFORT (Part 2 of 4) . . 60
Chart 02. IEYFORT (Part 3 of 4) . . 61
Chart 03. IEYFORT (Part 4 of 4) . . 62
Chart AA. OPTSCAN 63
Chart AB. DDNAMES 64
Chart AC. HEADOPT 65
Chart AD. TIMEDAT 66
Chart 04. PHASE 1 - PARSE 67
Chart BA. WRITE LISTING AND READ SOURCE 68
Chart BB. INITIALIZE FOR PROCESSING STATEMENT 69
Chart BC. PROCESS LABEL FIELD . . . 70
Chart BD. PROCESS STATEMENT 71
Chart BE. COMPLETE STATEMENT AND MOVE POLISH 72
Chart BF. PROCESS END STATEMENT . . 73
Chart BG. PROCESS POLISH 74
Chart 05. PHASE 2 - ALLOCATE (Part 1 of 2) 75
Chart 06. PHASE 2 - ALLOCATE (Part 2 of 2) 76
Chart CA. MOVE BLD NAMES TO BCD ROLL 77
Chart CB. PREPARE EQUIVALENCE DATA 78
Chart CC. ALLOCATE BLOCK DATA . . . 79
Chart CD. PREPROCESS DUMMY DIMENSIONS 80
Chart CE. CHECK FOR UNCLOSED DO LOOPS 81
Chart CF. CONSTRUCT BRANCH TABLE ROLL 82
Chart CG. ALLOCATE HEADING AND PUNCH ESD CARDS 83
Chart CH. CHECK ASSIGNMENT OF FUNCTION VALUE 84
Chart CI. COMMON ALLOCATION 85
Chart CK. EQUIVALENCE DATA ALLOCATION 86
Chart CL. SAVE AREA, BASE AND BRANCH TABLE ALLOCATION 87
Chart CM. ALLOCATE SCALARS 88
Chart CN. ALLOCATE ARRAYS 89
Chart CO. ADD BASES FOR SUBPROGRAM ADDRESSES 90
Chart CP. ALLOCATE SUBPROGRAM ARGUMENT LISTS 91
Chart CQ. PREPARE NAMELIST TABLES . 92
Chart CR. ALLOCATE LITERAL CONSTANTS 93
Chart CS. ALLOCATE FORMATS 94
Chart CT. MAP EQUIVALENCE 95
Chart CU. ALLOCATE SUBPROGRAM ADDRESSES 96
Chart CV. BUILD AND PUNCH NAMELIST TABLES 97
Chart CW. BUILD BASES 98
Chart CX. DEBUG ALLOCATE 99
Chart 07. PHASE 3 - UNIFY 100
Chart DA. BUILD ARRAY REF ROLL . . 101
Chart DB. MAKE ADDRESS CONSTANTS . 102
Chart DC. CONSTRUCT INSTRUCTIONS . 103
Chart DD. PROCESS NESTED LOOPS . . 104
Chart 08. PHASE 4 - GEN 105
Chart EA. GENERATE ENTRY CODE . . . 106
Chart EB. PROLOGUE CODE GENERATION 107
Chart EC. EPILOGUE CODE GENERATION 108
Chart ED. MOVE POLISH NOTATION . . 109
Chart EF. PROCESS LABELS 110
Chart EG. GENERATE STMT CODE . . . 111
Chart EH. COMPLETE OBJECT CODE . . 112
Chart 09. PHASE 5 - IEYEXT 113
Chart FA. PUNCH CONSTANTS AND TEMP STORAGE 114
Chart FB. PUNCH ADR CONST ROLL . . 115
Chart FC. PUNCH OBJECT CODE 116
Chart FD. PUNCH BASE TABLE 117
Chart FE. PUNCH BRANCH TABLE . . . 118
Chart FF. PUNCH SUBPROGRAM ARGUMENT LISTS 119
Chart FG. PUNCH SUBPROGRAM ADDRESSES 120
Chart FH. COMPLETE ADDRESSES FROM LIBRARY 121
Chart FI. PUNCH ADDRESS CONSTANTS . 122
Chart FJ. PUNCH RLD CARDS 123
Chart FK. PUNCH END CARDS 124
Chart FL. PUNCH NAMELIST TABLE POINTERS 125
Chart G0. IHCFCOMH OVERALL LOGIC AND UTILITY ROUTINES 243
Chart G1. IMPLEMENTATION OF READ/WRITE/FIND 244
Chart G2. DEVICE MANIPULATION AND WRITE-TO-OPERATOR ROUTINES 245
Chart G3. IHCFIOSH OVERALL LOGIC . 247
Chart G4. EXECUTION-TIME INPUT/OUTPUT RECOVERY PROCEDURE . . 248
Chart G5. IHCDIOSE OVERALL LOGIC - FILE DEFINITION SECTION . . 249
Chart G6. IHCDIOSE OVERALL LOGIC - FILE INITIALIZATION, READ, WRITE, AND TERMINATION SECTIONS . . 250
Chart G7. IHCIBERH OVERALL LOGIC . 252
Charts G8. ERROR MONITOR OVERALL LOGIC (Part 1 of 2) 253
Chart G9. ERROR MONITOR OVERALL LOGIC (Part 2 of 2) 254
Chart G10. ALTER OPTION TABLE ROUTINE OVERALL LOGIC (Part 1 of 3) 255
Chart G11. ALTER OPTION TABLE ROUTINE OVERALL LOGIC (Part 2 of 3) 256
Chart G12. ALTER OPTION TABLE ROUTINE OVERALL LOGIC (Part 3 of 3) 257

This section contains general information describing the purpose of the FORTRAN IV (G) compiler, the minimum machine configuration required, the relationship of the compiler to the operating system, compiler design and implementation, and compiler output. The various rolls,[1] variables, registers, pointers, and drivers used by the compiler are also discussed.

PURPOSE OF THE COMPILER

The IBM System/360 Operating System FORTRAN IV (G) compiler is designed to accept programs written in the FORTRAN IV language as defined in the publication IBM System/360: FORTRAN IV Language, Form C28-6515.

The compiler produces error messages for invalid statements, and, optionally, a listing of the source module, storage maps, and an object module acceptable to the System/360 Operating System linkage editor.

MACHINE CONFIGURATION

The minimum system configuration required for the use of the IBM System/360 Operating System with the FORTRAN IV (G) compiler is as follows:

- An IBM System/360 Model 40 computer with a storage capacity of 128K bytes and a standard and floating-point instruction set.
- A device for operator communication, such as an IBM 1052 Keyboard Printer.
- At least one direct-access device provided for system residence.

COMPILER AND SYSTEM/360 OPERATING SYSTEM

The FORTRAN IV (G) compiler is a processing program of the IBM System/360 Operating System. As a processing program, the compiler communicates with the control program for input/output and other services. A general description of the control program is given in the publication IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual.

A compilation, or a batch of compilations, is requested using the job statement (JOB), the execute statement (EXEC), and data definition statements (DD). Alternatively, cataloged procedures may be used. A discussion of FORTRAN IV compilation and the available cataloged procedures is given in the publication IBM System/360 Operating System: FORTRAN IV (G) Programmer's Guide.

The compiler receives control initially from the calling program (e.g., job scheduler or another program that CALLs, LINKs to, or ATTACHes the compiler). Once the compiler receives control, it uses the QSAM access method for all of its input/output operations. After compilation is completed, control is returned to the calling program.

COMPILER DESIGN

The compiler will operate within a total of 80K bytes of main storage. This figure includes space for the compiler code, data management access routines, and sufficient working space to meet other storage requirements stated throughout this publication.

Any additional storage available is used as additional roll storage.

LIMITATIONS OF THE COMPILER

The System/360 Operating System FORTRAN IV (G) compiler and the object module it produces can be executed on all System/360 models from Model 40 and above, under control of the operating system control program. All input information must be written in either BCD or EBCDIC representation. The compiler is designed to process all properly written programs so that the object code produced by the compiler is compatible with the existing mathematical library subroutines.

---

[1]Most of the tables used by the compiler are called rolls. (Further explanation of rolls is given in "Rolls and Roll Controls.")

If ten source read errors occur during the compilation, or if it is not possible to use SYSPRINT, the operation of the compiler is terminated. The operation of the compiler is also limited by the availability of main storage space. The compilation is terminated if:

- The roll storage area is exceeded
- Any single roll exceeds 64K bytes, thereby making it unaddressable
- The WORK or EXIT roll exceeds its allocated storage

Note: If any of these conditions occur during the first phase of the compilation, the statement currently being processed may be discarded; in this case, the compilation continues with the next statement.

COMPILER IMPLEMENTATION

The primary control and processing routines (hereafter referred to as "POP routines" or "compiler routines") of the compiler are primarily written in machine-independent pseudo instructions called POP instructions.

Interpretation of the pseudo instructions is accomplished by routines written in the System/360 Operating System assembler language. These routines (hereafter referred to as "POP subroutines") are an integral part of the compiler and perform the operations specified by the POP instructions, e.g., saving of backup information, maintaining data indicators, and general housekeeping.

Control of the compiler operation is greatly affected by source language syntax rules during the first phase of the compiler, Parse. During this phase, identifiers and explicit declarations encountered in parsing are placed in tables and a Polish notation form of the program is produced. (For further information on Polish notation, see Appendix C, "Polish Notation Formats.")

The compiler quite frequently uses the method of recursion in parsing, analysis, and optimization. All optimizing and code generating routines, which appear in later phases, operate directly on the tables and Polish notation produced by Parse.

The compiler is also designed so that reloading of the compiler is unnecessary in order to accomplish multiple compilations.

POP LANGUAGE

The FORTRAN IV (G) compiler is written in a combination of two languages: the System/360 Operating System assembler language, which is used where it is most efficient, and the POP language.

The POP language is a mnemonic macro programming language whose instructions include functions that are frequently performed by a compiler. POP instructions are written for assembly by the System/360 Operating System assembler, with the POP instructions defined as macros. Each POP instruction is assembled as a pair of address constants which together indicate an instruction code and an operand. A statement or instruction written in the POP language is called a POP. The POP instructions are described in Appendix A.

COMPILER ORGANIZATION

The System/360 Operating System FORTRAN IV (G) compiler is composed of a control phase, Invocation, and five processing phases (see Figure 1): Parse, Allocate, Unify, Gen, and Exit. The operating system names for these phases are, respectively, IEYFORT, IEYPAR, IEYALL, IEYUNF, IEYGEN, and IEYEXT. (The first level control and second level processing compiler routines used in each phase are shown in Figure 2.) In addition, Move is a pre-assembled work area, IEYROL.

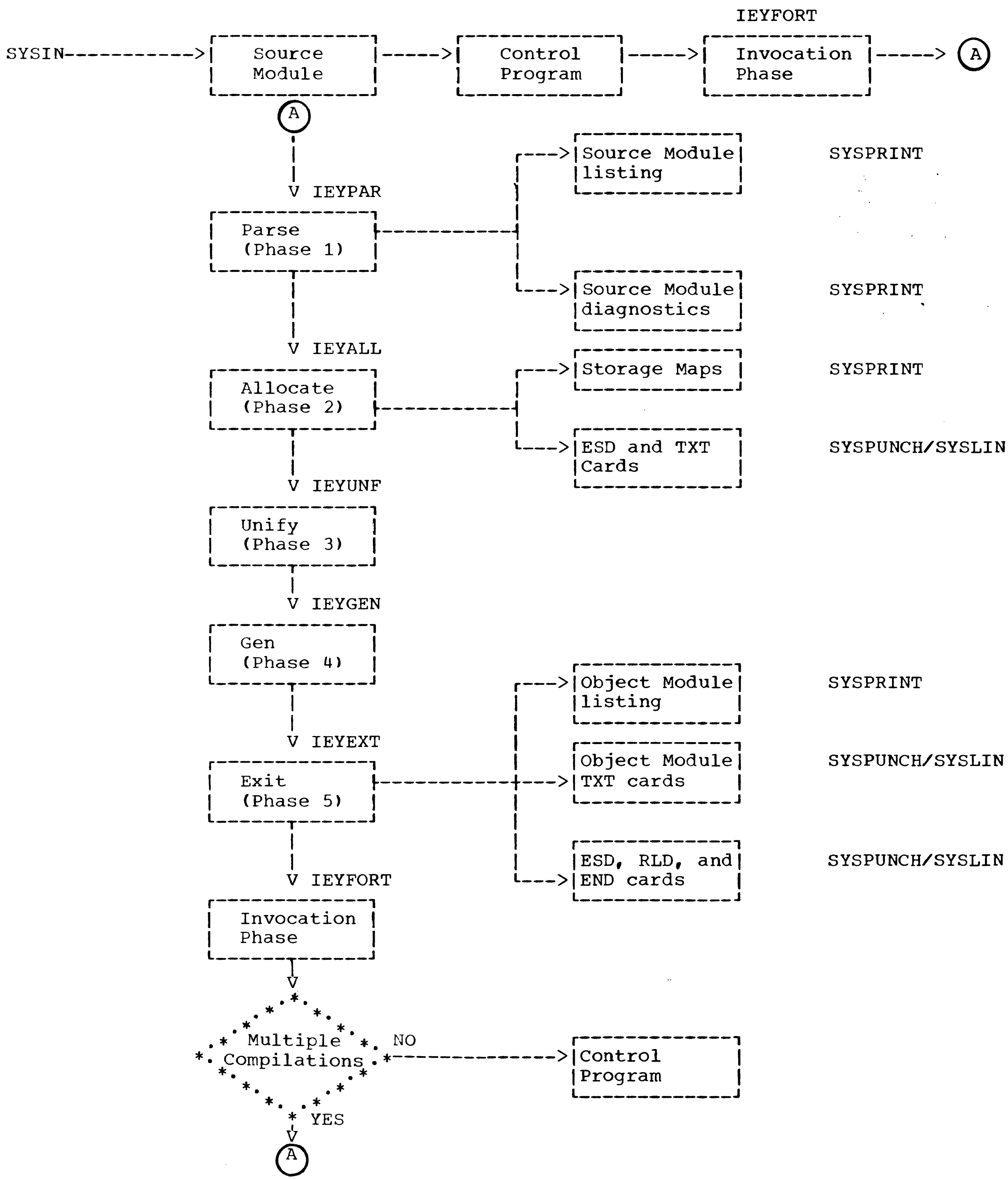


Figure 1. Overall Operation of the Compiler

Control Phase: Invocation (IEYFORT)

The Invocation phase (IEYFORT) is loaded upon invocation of the compiler and remains in core storage throughout compilation. It is entered initially from the calling program, from each module at the end of its processing, and from Exit after compilation is complete.

At the initial entry, the Invocation phase initializes bits in IEYFORT1 from the options specified by the programmer for the compilation, opens data sets, and fetches the modules IEYPAR, IEYALL, IEYUNF, IEYGEN, and IEYEXT via a series of LOAD macro instructions. These modules remain in core storage for a series of main program and subprogram compilations unless it is determined that additional space required for tables is not available. When this occurs, modules that precede the active one are deleted, and compilation is resumed. If more space is required, modules that follow the currently active one are deleted.

When a module completes processing, it returns to IEYFORT, which ensures the presence of the next module and transfers to it. During initialization for a subprogram, IEYFORT ensures that all modules are loaded.

The last entry is made from the Exit phase at the completion of a compilation. When the entry is made from Exit, the Invocation phase checks for multiple compilations. If another compilation is required, the compiler is reinitialized and the main storage space allocated for the expansion of rolls is assigned to the next compilation; otherwise, control is returned to the calling program.

Phase 1: Parse (IEYPAR)

Parse accepts FORTRAN statements in card format from SYSIN and scans these to produce error messages on the SYSPRINT data set, a source module listing (optional), and Polish notation for the program. The Polish notation is maintained on internal tables for use by subsequent phases. In addition, Parse produces the roll entries defining the symbols used in the source module.

Phase 2: Allocate (IEYALL)

Allocate, which operates immediately after Parse, uses the roll entries produced by Parse to perform the storage allocation for the variables defined in the source module. The addressing information thus produced is then left in main storage to be used by the next phase.

The ESD cards for the object module itself, COMMON blocks and subprograms, and TXT cards for NAMELIST tables, literal constants and FORMAT statements are produced by Allocate on the SYSPUNCH and/or SYSLIN data sets. Error messages for COMMON and EQUIVALENCE statements, unclosed DO loops and undefined labels are produced on SYSPRINT; on the MAP option, maps of data storage are also produced.

Phase 3: Unify (IEYUNF)

The Unify phase optimizes the usage of general registers within DO loops by operating on roll data which describes array references. The optimization applies to references which include subscripts of the form $ax \pm b$, where a and b are positive constants and x is an active induction variable (that is, x is a DO-controlled variable and the reference occurs within the DO loop controlling it), and where the array does not have any adjustable dimensions. The addressing portion of the object instruction for each such array reference is constructed to minimize the number of registers used for the reference and the number of registers which must be changed as each induction variable changes.

Phase 4: Gen (IEYGEN)

Gen uses the Polish notation produced by Parse and the memory allocation information produced by Allocate. From this information, Gen produces the code, prologues, and epilogues required for the object module. In order to produce the object code, Gen resolves labeled statement references (i.e., a branch target label) and subprogram entry references.

The final output from Gen is a complete form of the machine language code which is internally maintained for writing by the Exit phase.

Phase 5: Exit (IEYEXT)

Exit, which is the last processing phase of the compiler, produces the TXT cards for the remaining portion of the object module, the RLD cards (which contain the relocatable information), and the END card. This output is placed optionally on the SYSLIN data set for linkage editor processing and/or SYSPUNCH if a card deck has been requested. Additionally, a listing of the generated code may be written on the SYSPRINT data set in a format similar to that produced by an assembly program.

Roll (IEYROL)

Roll contains static rolls and roll information always required for compiler operations. These are described under the heading "Rolls and Roll Controls" later in this section.

```
PARSE-----┬---START COMPILER            ┌---PRINT AND READ SOURCE
          |                             |   STA INIT
          |                             |   LBL FIELD XLATE
          |                             |   STA XLATE
          |                             |   STA FINAL
          |                             |   REGISTER IBCOM
          |                             |   PROCESS POLISH
          └---STATEMENT PROCESS---------┤   ACTIVE END STA XLATE
                                        └---STA FINAL END

                                        ┌---PREP EQUIV AND PRINT ERRORS
                                        |   BLOCK DATA PROG ALLOCATION
                                        |   PREP DMY DIN AND PRINT ERRORS
                                        |   PROCESS DO LOOPS
                                        |   PROCESS LBL AND LOCAL SPROGS
                                        |   BUILD PROGRAM ESD
                                        |   ENTRY NAME ALLOCATION
                                        |   COMMON ALLOCATION AND OUTPUT
                                        |   BASE AND BRANCH TABLE ALLOC
                                        |   EQUIV ALLOCATION PRINT ERRORS
                                        |   FORMAT ALLOCATION
                                        |   SCALAR ALLOCATE
                                        |   ARRAY ALLOCATE
ALLOCATE-----START ALLOCATE-------------┤   PASS 1 GLOBAL SPROG ALLOCATE
                                        |   SPROG ARG ALLOCATION
                                        |   PREP NAMELIST
                                        |   LITERAL CONST ALLOCATION
                                        |   EQUIV MAP
                                        |   GLOBAL SPROG ALLOCATE
                                        |   BUILD NAMELIST TABLE
                                        |   ALPHA LBL AND L SPROG
                                        |   BUILD ADDITIONAL BASES
                                        |   ALPHA SCALAR ARRAY AND SPROG
                                        |   LITERAL CONST ALLOCATION
                                        |   CALCULATE BASE AND DISP
                                        └---DEBUG ALLOCATE

                                        ┌---ARRAY REF ROLL ALLOTMENT
UNIFY--------START UNIFY----------------┤   DO NEST UNIFY
                                        |   CONVERT TO ADR CONST
                                        └---CONVERT TO INST FORMAT

                                        ┌---MOVE ZEROS TO T AND C
          ┌---START GEN                 |   ENTRY CODE GEN
          |                             |   PROLOGUE GEN
          |                             └---EPILOGUE GEN
GEN-------┤
          |                             ┌---GET POLISH
          └---GEN PROCESS---------------┤   LBL PROCESS
                                        |   STA GEN
                                        └---STA GEN FINISH
                                        ┌---PUNCH TEMP AND CONST ROLL
                                        |   PUNCH ADR CONST ROLL
                                        |   PUNCH CODE ROLL
                                        |   PUNCH BASE ROLL
                                        |   PUNCH BRANCH ROLL
                                        |   PUNCH SPROG ARG ROLL
EXIT---------EXIT PASS------------------┤   PUNCH GLOBAL SPROG ROLL
                                        |   PUNCH USED LIBRARY ROLL
                                        |   PUNCH ADCON ROLL
                                        |   ORDER AND PUNCH RLD ROLL
                                        |   PUNCH END CARD
                                        |   PRINT HEADING
                                        |   PRINT A LINE
                                        └---PRINT TOTAL PROG REQMTS MESS
```

Figure 2. Compiler Organization Chart

	Load Module Name	Components	Content or Function
Low Core	IEYFORT	IEYFORT	Invocation and control
		IEYFORT1	Option bits
		IEYFORT2	Loads and deletes other modules
		IEYROL	Roll statistics (bases, tops, bottoms)
			Group statistics (displacement group sizes)
			WORK roll
			EXIT roll
			Roll address table
		IEYINT	POP Jump Table
			POP machine language subroutines
	Roll Storage is Allocated from this Area		
	IEYPAR	IEYPAR	Parse phase
			Quotes and messages
	IEYALL	IEYALL	Allocate phase
	IEYUNF	IEYUNF	Unify phase
	IEYGEN	IEYGEN	Generate phase
High Core	IEYEXT	IEYEXT	Exit phase

Figure 3. Compiler Storage Configuration

COMPILER STORAGE CONFIGURATION

Figure 3 illustrates the relative positions, but not the relative sizes of the component parts of the FORTRAN compiler as they exist in main storage. The component parts of each phase are described in Section 2.

COMPILER OUTPUT

The source module(s) to be compiled appear as input to the compiler on the SYSIN data set. The SYSLIN, SYSPRINT, and SYSPUNCH data sets are used (depending on the options specified by the user) to contain the output of the compilation.

The output of the compiler is represented in EBCDIC form and consists of any or all of the following:

Object Module (linkage editor input)

Source Module listing

Object Module listing

Storage maps

Error messages (always produced)

Relocatable card images for punching

The overall data flow and the data sets used for compilation are illustrated in Figure 4. The type of output is determined by compile time parameters.

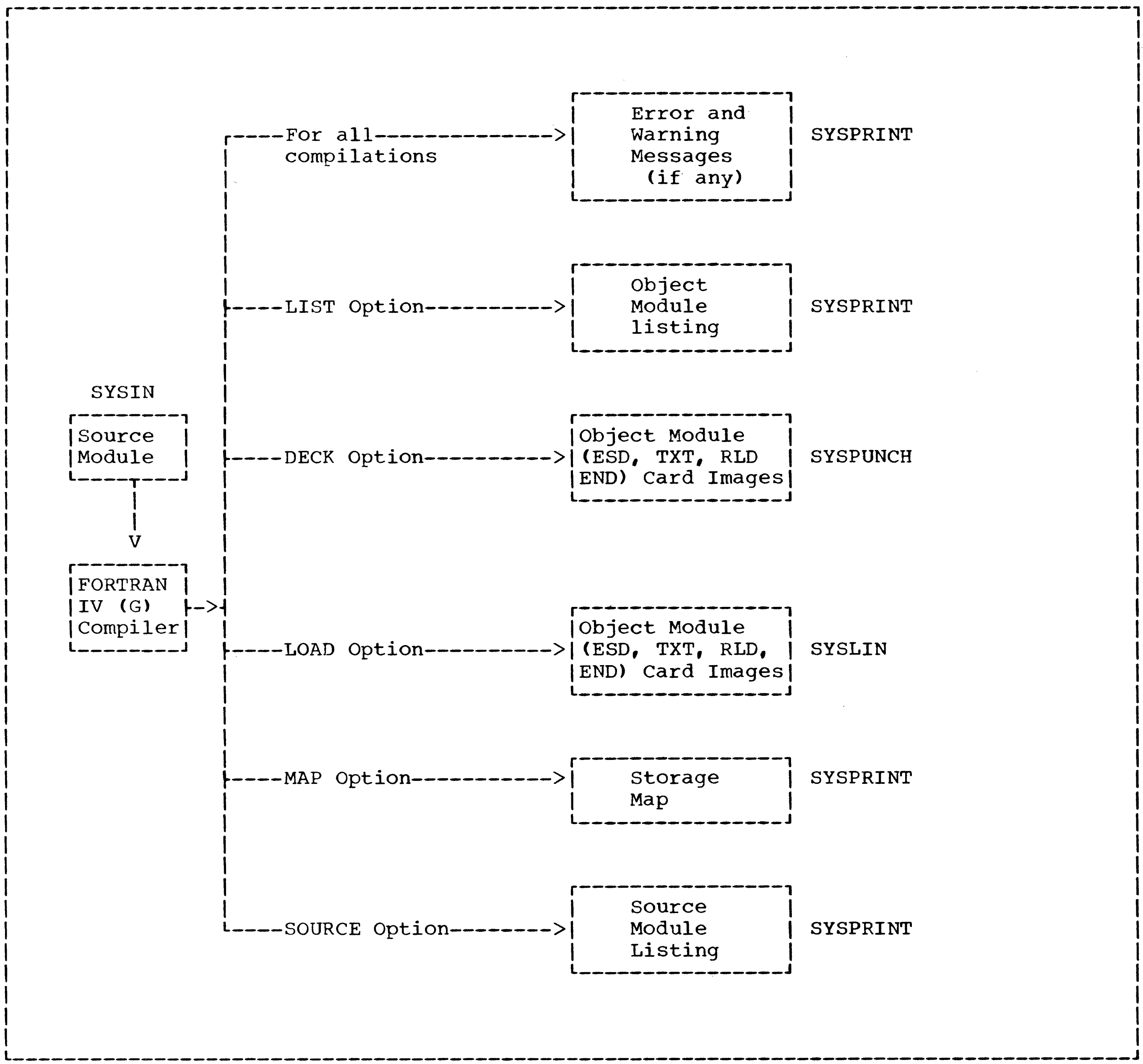


Figure 4. Compiler Output

OBJECT MODULE

The configuration of the object module produced by the FORTRAN IV (G) compiler is shown in Figure 5.

```
Entry point--->  ┌──────────────────────┐
                 │Heading               │
                 ├──────────────────────┤
                 │Save area             │
                 ├──────────────────────┤
                 │Base table            │
                 ├──────────────────────┤
                 │Branch table          │
                 ├──────────────────────┤
                 │EQUIVALENCE variables │
                 ├──────────────────────┤
                 │Scalar variables      │
                 ├──────────────────────┤
                 │Arrays                │
                 ├──────────────────────┤
                 │Subprogram argument   │
                 │lists                 │
                 ├──────────────────────┤
                 │Subprogram addresses  │
                 ├──────────────────────┤
                 │NAMELIST tables       │
                 ├──────────────────────┤
                 │Literal constants     │
                 │(except those used    │
                 │in DATA and PAUSE     │
                 │statements)           │
                 ├──────────────────────┤
                 │FORMAT statements     │
                 ├──────────────────────┤
                 │Temporary storage     │
                 │and constants         │
                 ├──────────────────────┤
                 │Program text          │
                 └──────────────────────┘
```

Figure 5. Object Module Configuration

Components of the Object Module

The following paragraphs describe the components of the object module produced by the FORTRAN IV (G) compiler.

HEADING: The object module heading includes all initializing instructions required prior to the execution of the body of the object module. Among other functions, these instructions set general register 13 (see "Object Module General Register Usage") and perform various operations, depending on whether the program is a main program or a subprogram and on whether it calls subprograms. (See "Code Produced for SUBROUTINE and FUNCTION Subprograms.")

SAVE AREA: The save area, at maximum 72 bytes long, is reserved for information saved by called subprograms. Figure 6 shows an example of the use of this area in program Y, which is called by program X, and which calls program Z.

The first byte of the fifth word in the save area (Save Area of Y + 16) is set to all ones by program Z before it returns to program Y. Before the return is made, all general registers are restored to their program Y values.

BASE TABLE: The base table is a list of addresses from which the object module loads a general register prior to accessing data; the general register is then used as a base in the data referencing instruction.

Because an interval of 4096 bytes of storage can be referenced by means of the machine instruction D field, consecutive values representing a single control section in this table differ from each other by at least 4096 bytes. Only one base table entry is constructed for an array which exceeds 4096 bytes in length; hence, there is a possibility that an interval of more than 4096 bytes exists between consecutive values for a single control section in the table.

The addresses compiled into this table are all relative, and are modified by the linkage editor prior to object module execution. Those entries constructed for references to COMMON are modified by the beginning address of the appropriate COMMON block; those entries constructed for references to variables and constants within the object module itself are modified by the beginning address of the appropriate object module.

```
<---4 bytes---->
                 ┌───────────────┐
                 |Subprogram     | <---Stored by initial entry code
Save Area of Y   |epilogue address|
                 ├───────────────┤
            +4   |Program X save | <---Stored by program Y
                 |area address   |
                 ├───────────────┤
            +8   |Program Z save | <---Stored by program Z, if it calls subroutines
                 |area address   |
                 ├───────────────┤
           +12   |Register 14    |\
                 ├───────────────┤ |
           +16   |Register 15    | |
                 ├───────────────┤ |
           +20   |Register 0     | |
                 ├───────────────┤  >  Values on leaving program Y, stored by program Z.
                 |       .       | |
                 |       .       | |
                 |       .       | |
                 ├───────────────┤ |
           +72   |Register 12    |/
                 └───────────────┘
```

Figure 6. Example of Use of Save Area

BRANCH TABLE: This table contains one fullword entry for each branch target label (a label referred to in a branch statement) and statement function in the source module. In addition, one entry occurs for each label produced by the compiler in generating the object module. These labels refer to return points in DO loops and to the statement following complete Logical IF statements, and are called made labels.

In the object module code, any branch is performed by loading general register 14 (see "Object Module General Register Usage") from this table, and using a BCR instruction. The values placed in this table by the compiler are relative addresses. Each value is modified by the base address of the object module by the linkage editor.

EQUIVALENCE VARIABLES: This area of the object module contains unsubscripted variables and arrays, listed in EQUIVALENCE sets which do not refer to COMMON.

SCALAR VARIABLES: All non-subscripted variables which are not in COMMON and are not members of EQUIVALENCE sets appear in this area of the object module.

ARRAYS: All arrays which are not in COMMON, and are not members of EQUIVALENCE sets appear in this area of the object module.

SUBPROGRAM ARGUMENT LISTS: This portion of the object module contains the addresses of the arguments for all subprograms called. In calling a subprogram, the object module uses general register 1 to transmit a location in this table. The subprogram then acquires the addresses of its arguments from that location and from as many subsequent locations as there are arguments. The sign bit of the word containing the address of the last argument for each subprogram is set to one.

SUBPROGRAM ADDRESSES: This list contains one entry for each FUNCTION or SUBROUTINE subprogram referenced by the object module. The entry will hold the address of that subprogram when it is supplied by the linkage editor. The compiler reserves the correct amount of space for the list, based on the number of subprograms referred to by the source module.

NAMELIST TABLES: For each NAMELIST name and DISPLAY statement in the source module, a NAMELIST table is constructed by the compiler and placed in this area of the object module. Each table consists of one entry for each scalar variable or array listed following the NAMELIST name or in the DISPLAY statement, and begins with four words of the following form:

Word \ Byte	1	2	3	4
1, 2	name field			
3, 4	not used			

where the name field contains the NAMELIST name, right justified. For the DISPLAY statement, the name is DBGnn#, where nn is the number of the DISPLAY statement within the source program or subprogram.

Table entries for scalar variables have the following form:

Word \ Byte	1	2	3	4
1, 2	name field			
3	address field			
4	type	mode	not used	

where:

name field
    contains the name of the scalar variable, right justified.

address field
    contains the relative address of the variable within the object module.

type field
    contains zero to indicate a scalar variable.

mode field
    contains the mode of the variable, coded as follows:

- 2 = Logical, 1 byte
- 3 = Logical, fullword
- 4 = Integer, halfword
- 5 = Integer, fullword
- 6 = Real, double precision
- 7 = Real, single precision
- 8 = Complex, double precision
- 9 = Complex, single precision
- A = Literal (not currently compiler-generated)

NAMELIST table entries for arrays have the following form:

Word \ Byte	1	2	3	4
1, 2	name field			
3	address field			
4	type	mode	no. dimens.	length
5	indicator	first dimension factor field		
6	not used	second dimension factor field		
7	not used	third dimension factor field		
. . . etc.	. . . etc.			

where:

name field
    contains the name of the array, right justified.

address field
    contains the relative address of the beginning of the array within the object module.

mode field
    contains the mode of the array elements, coded as for scalar variables, above.

no. dimens.
    contains the number of dimensions in the array; this value may be 1-7.

length field
    contains the length of the array element in bytes.

indicator field
    is set to zero if the array has been defined to have variable dimensions; otherwise, it is set to nonzero.

first dimension factor field
    contains the total size of the array in bytes.

second dimension factor field
    contains the address of the second multiplier for the array (n1*L, where n1 is the size of the first dimension in elements, and L is the number of bytes per element).

third dimension factor field
contains the address of the third multiplier for the array (n1*n2*L, where n1 is the size of the first dimension in elements, n2 is the size of the second dimension, and L is the number of bytes per element).

A final entry for each NAMELIST table is added after the last variable or array name to signify the end of that particular list. This entry is a fullword in length and contains all zeros.

LITERAL CONSTANTS: This area contains a list of the literal constants used in the source module, except for those specified in DATA and PAUSE statements.

FORMAT STATEMENTS: The FORMAT statements specified in the source module are contained in this area of the object module. The statements are in an encoded form in the order of their appearance in the source module. (See "Appendix D: Code Produced by the Compiler.") The information contains all specifications of the statement but not the word FORMAT.

TEMPORARY STORAGE AND CONSTANTS: This area always begins on a double precision boundary and contains, in no specific order, the constants required by the object module code and the space for the storage of temporary results during computations. Not all of the source module constants necessarily appear in this area, since as many constants as possible are used as immediate data in the code produced. Some constants may appear which are not present in the source module, but which have been produced by the compiler.

PROGRAM TEXT: If the object module contains statement functions, the code for these statements begins the program text and is preceded by an instruction that branches around them to the first executable statement of the program. (See "Statement Functions" in Appendix D for further explanation of this code.) Following the code for the statement functions is the code for the executable statements of the source module.

Object Module General Register Usage

The object module produced by the FORTRAN IV (G) compiler uses the System/360 general registers in the following way:

Register 0: Used as an accumulator.

Register 1: Used as an accumulator and to hold the beginning address of the argument list in branches to subprograms.

Register 2: Used as an accumulator.

Register 3: Used as an accumulator.

Registers 4 through 7: Contain index values as required for references to array variables, where the subscripts are linear functions of DO variables and the array does not have variable dimensions.

Registers 8 and 9: Contain index values as required for references to array variables, where the subscripts are of the form x±c, where x is a non DO-controlled variable and c is a constant.

Register 9: Contains index values as required for references to array variables where the subscripts are non-linear of the form I*J, where I and J are the variables.

Registers 10 through 12: Contain base addresses loaded from the base table.

Register 13: Contains the beginning address of the object module save area; this value is loaded at the beginning of program execution. Register 13 is also used for access to the base table, since the base table follows the save area in main storage.

Register 14: Contains the return address for subprograms and holds the address of branch target instructions during the execution of branch instructions.

Register 15: Contains the entry point address for subprograms as they are called by the object module.

SOURCE MODULE LISTING

The optional source module listing is a symbolic listing of the source module; it contains indications of errors encountered in the program during compilation. The error message resulting from an erroneous statement does not necessarily cause termination of compiler processing nor the discarding of the statement. Recognizable portions of declaration statements are retained, and diagnosis always proceeds until the end of the program.

OBJECT MODULE LISTING

The optional object module listing uses the standard System/360 Operating System assembler mnemonic operation codes and, where possible, refers to the symbolic variable names contained in the source module. Labels used in the source module are indicated at the appropriate places in the object code listing.

STORAGE MAPS

The optional storage map consists of six independent listings of storage information. Each listing specifies the names and locations of a particular class of variable. The listings are:

- COMMON variables
- EQUIVALENCE variables
- Scalar variables
- Array variables
- NAMELIST tables
- FORMAT statements

A list of the subprograms called is also produced.

ERROR MESSAGES

Errors are indicated by listing the statement in its original form with the erroneous phrases or characters undermarked by the dollar sign character, followed by comments indicating the type of the error. This method is described in more detail in "Phase 1 of the Compiler: Parse (IEYPAR)."

Common Error Messages

The message NO CORE AVAILABLE is produced (through IEYFORT) by all phases of the compiler when the program being compiled exhausts the main storage space available to the compiler. This message is produced only when the PRESS MEMORY routine cannot provide unused main storage space on request from the compiler.

The message ROLL SIZE EXCEEDED is produced (through the Invocation phase, IEYFORT) by all phases of the compiler when the size of any single roll or rolls is greater than permitted. The following circumstances cause this message to be produced:

- The WORK roll exceeds the fixed storage space assigned to it.
- The EXIT roll exceeds the fixed storage space assigned to it.
- Any other roll, with the exception of the AFTER POLISH roll and the CODE roll, exceeds 64K bytes of storage. In this case, the capacity of the ADDRESS field of a pointer to the roll is exceeded and, therefore, the information on the roll is unaddressable. The AFTER POLISH and CODE rolls are excepted, since pointers to these rolls are not required.

The compilation terminates following the printing of either of these messages.

COMPILER DATA STRUCTURES

The POP language is designed to manipulate certain well-defined data structures.

Rolls, which are the tables primarily used by the compiler, are automatically handled by the POP instructions; that is, when information is moved to and from rolls, controls indicating the status of the rolls are automatically updated.

Items (variables) with fixed structures are used to maintain control values for rolls, to hold input characters being processed, and to record Polish notation, etc. These item structures are also handled automatically by the POP instructions.

The arrangement of the parts of the compiler is significant because of the extensive use of relative addressing in the compiler. General registers are used to hold base addresses, to control some rolls, and to assist in the interpretation of the POP instructions.

ROLLS AND ROLL CONTROLS

Most of the tables employed by the compiler are called rolls. This term describes a table which at any point in time occupies only as much storage as is required for the maximum amount of information it has held during the present compilation (exceptions to this rule are noted later). Another distinctive feature of a roll is that it is used so that the last information placed on it is the first information retrieved -- it uses a "push up" logic.

With the exception of the WORK and EXIT rolls, the rolls of the compiler are maintained in an area called the roll storage area. The rolls in this area are both named and numbered. While the references to rolls in this document and in the compiler comments are primarily by name, the names are converted to corresponding numbers at assembly time and the rolls are arranged in storage and referred to by number.

If the roll storage area is considered to be one block of continuous storage, the rolls are placed in this area in ascending sequence by roll number; that is, roll 0 begins at the base address of the roll storage area; rolls 1, 2, 3, etc., follow roll zero in sequence, with the roll whose number is largest terminating the roll storage area.

Initially, all rolls except roll 0 are empty and occupy no space; this is accomplished by having the beginning and end of all rolls located at the same place. (Roll 0, the LIB roll, is a fixed-length roll which contains all of its data initially.) When information is to be placed on a roll and no space is available due to a conflict with the next roll, rolls greater in number than the roll in question are moved down (to higher addresses) to make the space available. This is accomplished by physically moving the information on the rolls a fixed number of storage locations and altering the controls to indicate the change. Thus, roll 0 never changes in size, location, or contents; all other rolls expand to higher addresses as required. When information is removed from a roll, the space which had been occupied by that information is left vacant; therefore, it is not necessary to move rolls for each addition of information.

With the exception of the area occupied by roll 0, the roll storage area actually consists of any number of non-contiguous blocks of 4096 bytes of storage. The space required for roll 0 is not part of one of these blocks. Additional blocks of storage are acquired by the compiler whenever current roll storage is exceeded. If the system is unable to fulfill a request for roll storage, the PRESS MEMORY routine is entered to find roll space that is no longer in use. If 32 or more bytes are found, the compilation continues. If fewer than 32 bytes are found, the compilation of the current program is terminated, the message NO CORE AVAILABLE is printed, and space is freed. If there are multiple programs, the next one is compiled.

The following paragraphs describe the controls and statistics maintained by the compiler in order to control the storage allocation for rolls and the functioning of the "push up" logic.

ROLL ADR Table

The ROLL ADR table is a 1000-byte table maintained in IEYROL. Each entry in this table holds the beginning address of a block of storage which has been assigned to the roll storage area. The first address in the table is always the beginning address of roll 0. The second address is that of the first 4K-byte block of storage and, therefore, the beginning address of roll 1. Initially, the last address recorded on the table is the beginning address of a block which holds the CODE and AFTER POLISH rolls, with the CODE roll beginning at the first location in the block.

As information is recorded on rolls during the operation of the compiler, additional storage space may eventually be required. Whenever storage is needed for a roll which precedes the CODE roll, an additional 4K block is requested from the system and its address is inserted into the ROLL ADR table immediately before the entry describing the CODE roll base. This insertion requires that any entries describing the CODE and AFTER POLISH rolls be moved down in the ROLL ADR table. The information on all rolls following (greater in number than) the roll requiring the space is then moved down a fixed number of words. The roll which immediately precedes the CODE roll moves into the new block of storage. This movement of the rolls creates the desired space for the roll requiring it. The movement of rolls does not respect roll boundaries; that is, it is entirely possible that any roll or rolls may bridge two blocks of storage.

When additional storage space is required for the AFTER POLISH roll, a block is requested from the system and its beginning address is added to the bottom of the ROLL ADR table. When the CODE roll requires more space, a new block is added in the same manner, the AFTER POLISH roll is moved down into the new block, and the vacated space is available to the CODE roll.

The CODE and AFTER POLISH rolls are handled separately because the amount of information which can be expected to reside on them makes it impractical to move them frequently in order to satisfy storage requirements for all other rolls. The CODE roll is also somewhat unique in that it is assigned a large amount of space before it is used; that is, the AFTER POLISH roll does not begin at the same location as does the CODE roll.

BASE, BOTTOM, and TOP Tables

In order to permit dynamic allocation as well as to permit the use of the "push up" logic, tables containing the variables BASE, BOTTOM, and TOP are maintained to record the current status of each of the rolls. These variables indicate addresses of rolls. Information stored on rolls is in units of fullwords; hence, these addresses are always multiples of four. The length of each of the tables is determined by the number of rolls, and the roll number is an index to the appropriate word in each table for the roll.

Each of the variables occupies a fullword and has the following configuration:

```
             1 1          1 2          3
0            1 2          9 0          1
┌────────────┬────────────┬────────────┐
│            │Entry number│            │
│            │into the    │Displacement│
│            │ROLL ADR    │ (12 bits)  │
│            │Table       │            │
└────────────┴────────────┴────────────┘
```

The entry number points to an entry in the ROLL ADR table and, hence, to the beginning address of a block of roll storage. The displacement is a byte count from the beginning ofthe indicated storage block to the location to which the variable (BASE, BOTTOM, or TOP) refers.

It is significant to note that the displacement field in these variables occupies twelve bits. If the displacement field is increased beyond its maximum value (4095), the overflow increases the entry number into the ROLL ADR table; this is the desired result, since it simply causes the variable to point to the next entry in the table and effectively indicate the next location in the roll storage area, the beginning of the next block.

The first status variable for each roll, BASE, indicates the beginning address of that roll, minus four. The second variable, BOTTOM, indicates the address of the most recently entered word on the roll.

If the roll is completely empty, its BOTTOM is equal to its BASE; otherwise, BOTTOM always exceeds BASE by a multiple of four. Figure 7 illustrates a roll which contains information.

```
                         4 bytes
BASE  (n) \         ┌──────────────┐
           >->      │              │<-----unused
TOP   (n) /         │              │
                    ├──────────────┤
                    ├──────────────┤\
                    ├──────────────┤ |
                    ├──────────────┤ |
                    │      .       │  >    K bytes
                    │      .       │ |
                    │      .       │ |
                    ├──────────────┤ |
BOTTOM(n)---->      │              │/
                    └──────────────┘
```

Figure 7. Roll Containing K Bytes of Information

When information is to be added to a roll, it is stored at the address pointed to by BOTTOM, plus four, and BOTTOM is increased by four. When a word is to be retrieved from a roll, it is read from the address specified by BOTTOM, and, under most circumstances, BOTTOM is reduced by four, thus indicating that the word is no longer occupied by the roll. This alteration of the value of BOTTOM is termed <u>pruning</u>. If the information retrieved from a roll is to remain on the roll as well as at the destination, BOTTOM is not changed. This operation is indicated by the use of the word "keep" in the POP instructions that perform it.

The current length (in bytes) of a roll is determined by subtracting its BASE from its BOTTOM. Note that this is true even though the entry number field appears in these variables, since each increase in entry number indicates 4096 bytes occupied by the roll. Thus, there is no limitation on the size of a roll from this source.

For each roll, an additional status variable, called <u>TOP</u>, is maintained. TOP enables the program to protect a portion of the roll from destruction, while allowing the use of the roll as though it were empty. Protecting a roll in this way is called reserving the roll. The contents of TOP (always greater than or equal to the contents of BASE) indicate a false BASE for the roll. The area between BASE and TOP, when TOP does not equal BASE, cannot be altered or removed from the roll. Ascending locations from TOP constitute the new, empty roll.

Like BASE, TOP points to the word immediately preceding the first word into which information can be stored. A value is automatically stored in this unused word when the roll is reserved; the value is the previous value of TOP, minus the value of BASE and is called the <u>reserve mark</u>. Storage of this value permits more than one segment of the roll to be reserved.

A single roll (roll n), then, containing K bytes of information, (where K is always a multiple of four) and having no reserved status, has the following settings for its status variables:

BOTTOM = BASE + K = TOP + K

Figure 7 also illustrates this roll. If the same roll contains L bytes reserved and K additional bytes of information, the settings of its status variables are as follows:

BOTTOM = TOP + K = BASE + L + K + 4

This roll is shown in Figure 8. Note that the relationships given above are valid because of the structure of the BASE, BOTTOM, and TOP variables.

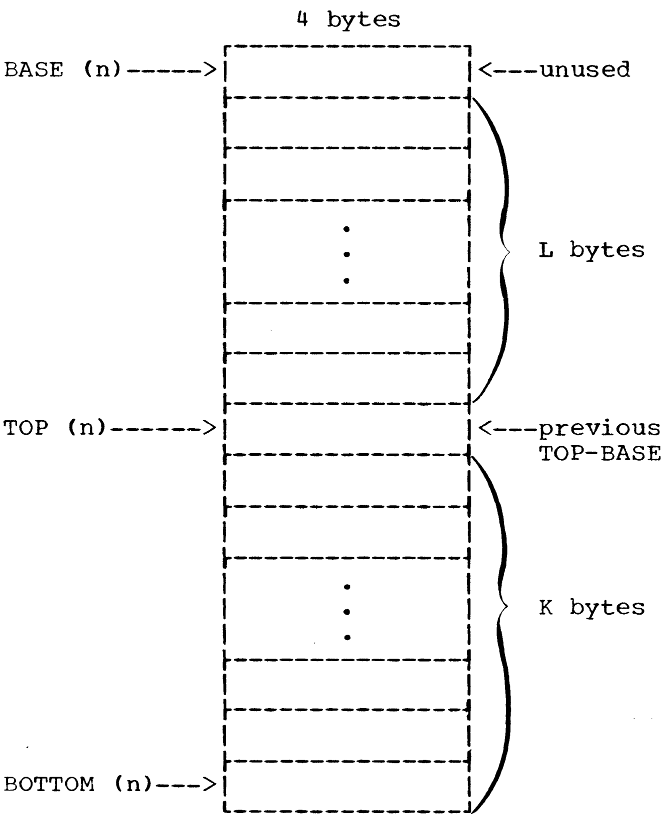


Figure 8. Roll Containing L Bytes of Reserved Information and K Bytes of New Information

Special Rolls

The WORK roll and the EXIT roll are special rolls in that they are not maintained in the roll storage area, but rather appear in IEYROL with a fixed amount of storage allocated to each. They are rolls in the sense that they employ the same push up logic which is used for the other rolls; however, they are not numbered, and their controls are, therefore, not maintained in the tables used for the other rolls.

The WORK roll is used as a temporary storage area during the operations of the compiler. Because information is moved to and from the roll frequently it is handled separately from other rolls.

The EXIT roll warrants special treatment because it is used frequently in maintaining exit and entrance addresses for compiler routines.

The bottom of the WORK roll is recorded in general register 4, WRKADR; general register 5, EXTADR, holds the address of the bottom of the EXIT roll. These values are absolute addresses rather than in the format of the BOTTOM variable recorded for other rolls.

For a more detailed explanation of the WORK and EXIT rolls, see Appendix B "Rolls Used by the Compiler."

Central Items, Groups, and Group Stats

CENTRAL ITEMS: The items SYMBOL 1, SYMBOL 2, SYMBOL 3, DATA 0, DATA 1, DATA 2, DATA 3 and DATA 4, two bytes each in length, and DATA 5, eight bytes in length, contain variable names and constants. These items are called central due to the nature and frequency of their use. They occupy storage in the order listed, with DATA 1 aligned to a doubleword boundary.

In general, SYMBOL 1, 2, and 3 hold variable names; DATA 1 and 2 are used to hold real constants, DATA 3 and 4 to hold integer constants, DATA 1, 2, 3 and 4 to hold double precision and complex constants, and DATA 1, 2, 3, 4 and 5 to hold double-precision complex constants.

GROUPS: While the basic unit of information stored on rolls is a fullword, many rolls contain logically connected information which requires more than a singleword of storage. Such a collection of information is called a group and always occupies a multiple of four bytes. A word of a group of more than one word is sometimes called a rung of the group.

Regardless of the size of the group on a given roll, the item BOTTOM for the roll always points to the last word on the roll. Figure 9 shows a roll with a group size of twelve.

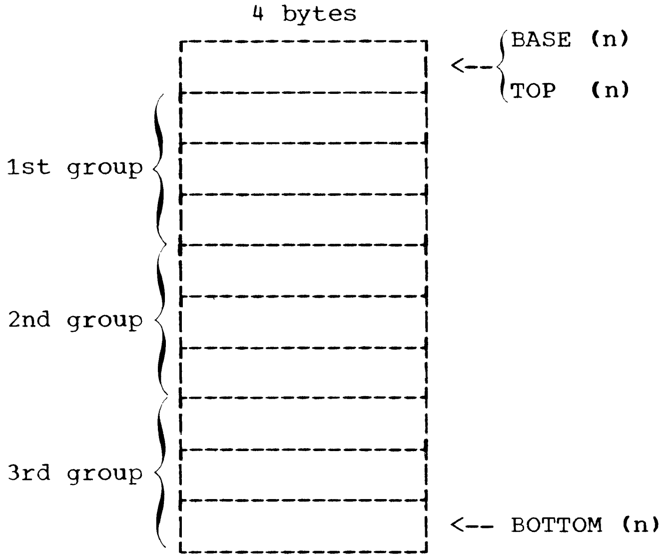


Figure 9. Roll With a Group Size of Twelve

For some rolls, the size of the group is not fixed. In these cases a construct called a "plex" is used. The first word of each plex holds the number of words in the plex, exclusive of itself; the remainder holds the information needed in the group. (See Figure 10.)

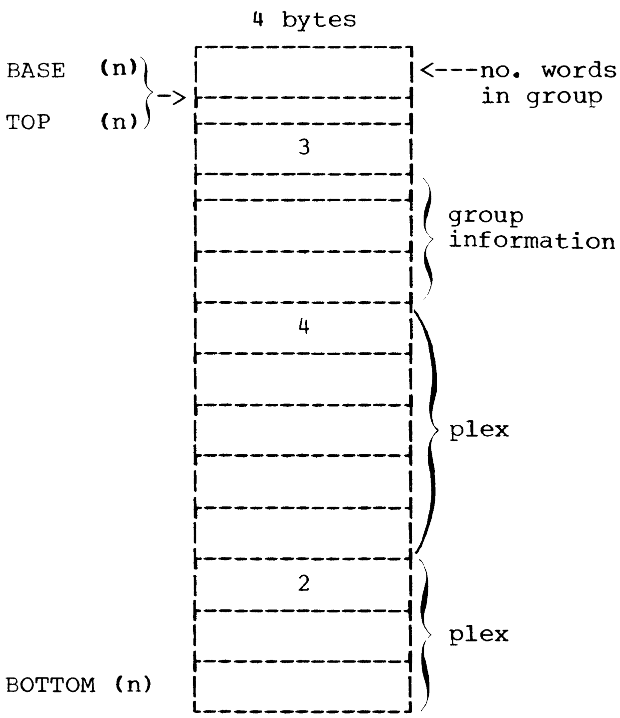


Figure 10. Roll with Variable Group Size

The assignment of roll storage does not respect group boundaries; thus, groups may be split between two blocks of roll storage.

<u>GROUP STATS</u>: Since the size of the group varies from roll to roll, this characteristic of each roll must be tabulated in order to provide proper manipulation of the roll. In addition, the groups on a roll are frequently searched against the values held in the central items (SYMBOL 1, 2, 3, etc.,). Additional characteristics of the roll must be tabulated in order to provide for this function. Four variables tabulated in the group stats tables are required to maintain this information. (See Section 2 "IEYROL Module.")

The first group stats table contains a 1-word entry for each roll. The entry is divided into two halfword values. The first of these is the displacement in bytes from SYMBOL 1 for a group search; that is, the number of bytes to the right of the beginning of SYMBOL 1 from which a comparative search with the group on the roll should begin. This value is zero for rolls which contain variable names (since these begin in SYMBOL 1), eight for rolls which contain real, double-precision, complex or double-precision complex constants (since these begin in DATA 1), and twelve for rolls which contain integer constants.

The second value in the first group stats table is also a displacement; the distance in bytes from the beginning of the group on the roll to the byte from which a comparative search with the central items should begin.

The second group stats table also holds a 1-word entry for each roll; these entries are also divided into two halfword values. The first of these is the number of consecutive bytes to be used in a comparative search, and refers to both the group on the roll and the group in the central items with which it is being compared.

The second item in the second table is the size of the group on the roll, in bytes. For rolls which hold plexes, the value of this item is four.

For example, the <u>DP CONST roll</u>, which is used to hold the double-precision constants required for the object module, has an 8-byte group. The settings of the Group Stats for this roll are 8, 0, 8, and 8, respectively. The first 8 indicates that when this roll is searched in comparison with the central items, the search should begin eight bytes to the right of SYMBOL 1 (at DATA 1). The 0 indicates that there is no displacement in the group itself; that is, no information precedes the value to be compared in the group. The second 8 is the size of the value to be searched. The final 8 is the number of bytes per group on the roll.

The group stats for the ARRAY roll (which holds the names and dimension information of arrays) are 0, 0, 6, and 20. They indicate that the search begins at SYMBOL 1, that the search begins 0 bytes to the right of the beginning of the group on the roll, that the number of bytes to be searched is 6, and that the group 6 size on the roll is 20 bytes.

Figures 11 and 12 show the two group stats tables containing the information on the DP CONST roll and the ARRAY roll discussed above. It should be noted that the information contained on these two tables is arranged according to roll numbers. In other words, the group stats for roll 5 are in the sixth entry in the tables (starting with entry number 0).

```
                              4 bytes
                  ┌────────────┬────────────┐
                  ├────────────┼────────────┤
                  ├────────────┴────────────┤
                  │            .            │
                  │            .            │
                  │            .            │
                  ├────────────┬────────────┤
DP CONST roll---->│           8│           0│
                  ├────────────┴────────────┤
                  │            .            │
                  │            .            │
                  │            .            │
                  ├────────────┬────────────┤
   ARRAY roll---->│           0│           0│
                  ├────────────┴────────────┤
                  │            .            │
                  │            .            │
                  │            .            │
                  ├────────────┬────────────┤
                  └────────────┴────────────┘
```

Figure 11. First Group Stats Table

```
                              4 bytes
                  ┌────────────┬────────────┐
                  ├────────────┼────────────┤
                  ├────────────┴────────────┤
                  │            .            │
                  │            .            │
                  │            .            │
                  ├────────────┬────────────┤
DP CONST roll---->│           8│           8│
                  ├────────────┴────────────┤
                  │            .            │
                  │            .            │
                  │            .            │
                  ├────────────┬────────────┤
   ARRAY roll---->│           6│          20│
                  ├────────────┴────────────┤
                  │            .            │
                  │            .            │
                  │            .            │
                  ├────────────┬────────────┤
                  └────────────┴────────────┘
```

Figure 12. Second Group Stats Table

OTHER VARIABLES

In addition to the central items, several other variables used in the compiler perform functions which are significant to the understanding of the POP instructions. These are described in the following paragraphs.

Answer Box

The variable ANSWER BOX, which is recorded in the first byte of the first word of each EXIT roll group, is used to hold the true or false responses from POP instructions. The value "true" is represented by a nonzero value in this variable, and "false" by zero. The value is checked by POP jump instructions.

Multiple Precision Arithmetic

Most of the arithmetic performed in the compiler is fullword arithmetic. When double-precision arithmetic is required, the variables MPAC 1 and MPAC 2, four bytes each in length, are used as a double-precision register. These variables are maintained in main storage.

Scan Control

Several variables are used in the character scanning performed by the first processing phase of the compiler, Parse. Their names, and terms associated with their values, are frequently used in describing the POP instructions.

The variable CRRNT CHAR holds the source statement character which is currently being inspected; the variable is four bytes long. The position (scan arrow) of the current character within the input statement (its column number, where a continuous column count is maintained over each statement) is held in the low-order bit positions of the fullword variable CRRNT CHAR CNT.

Non-blank characters are called "active characters," except when literal or IBM card code information is being scanned. The variable LAST CHAR CNT, which occupies one word of storage, holds the column number of the active character previous to the one in CRRNT CHAR.

Example:

```
                          1
Column number:   1234567890
                          .
                          .
                          .
                      DO 50 I = 1, 4
                      A(I) = B(I)**2
                      DO 50 J=1, 5
                   50 C(J+1) = A(I)
                          .
                          .
                          .
```

Explanation:

In the processing of the source module which contains the above statements, statement 50 is currently being parsed. The current character from the input buffer is J. The settings of the scan control variables are shown in Figure 13.

(EBCDIC)	J

CRRNT CHAR

	9

CRRNT CHAR CNT
(scan arrow)

1	8

LAST CHAR CNT

Figure 13. Scan Control Variables

Flags

Several flags are used in the compiler. These 1-word variables have two possible values: on, represented by nonzero, and off, represented by zero. The name of the flag indicates the significance of the "on" setting in all cases.

Quotes

Quotes are sequences of characters preceded by a halfword character count; they are compared with the input data to determine a statement type during the Parse phase. These constants are grouped together at the end of phase 1. The location labeled QUOTE BASE is the beginning location of the first quote; instructions which refer to quotes are assembled with address fields which are relative to this location.

Figure 14 shows some of the quotes used by the compiler and how they are arranged in storage.

4 bytes

QUOTE BASE	00	02	N	D
	00	08	I	M
	E	N	S	I
	O	N	b	b
	00	07	M	P
	L	I	C	I
	T	b	b	b
	00	07	L	O
	G	I	C	A
	L	b	b	b
		. . .		
	00	06	F	O
	R	M	A	T
		. . .		

Figure 14. Quotes Used in the Compiler

Messages

The messages used in the compiler, which are also grouped together at the end of Phase 1, are the error messages required by Parse for the source module listing. The first byte of each message holds the condition code for the error described by the message. The second byte of the message is the number of bytes in the remainder of the message. The message follows this halfword of information.

The location labeled MESSAGE BASE is the beginning location of the first message; instructions which refer to messages are assembled with address fields relative to this location.

COMPILER ARRANGEMENT AND GENERAL REGISTER USAGE

Figure 15 shows the arrangement of the compiler in main storage with the Parse phase shown in detail. General registers that hold base locations within the compiler are shown pointing to the locations they indicate. Note that the labels CBASE and PROGRAM BASE 2 appear in each phase of the compiler; the general registers CONSTR and PGB2 contain the locations of those labels in the operating phase.

General register 2, PGB2, holds the beginning address of the global jump table, a table containing the addresses of compiler routines which are the targets of jump instructions. (See Appendix A for further discussion of this table and the way in which it is used.) The global jump table appears in each phase of the compiler and is labeled PROGRAM BASE 2; thus, the value held in general register 2 is changed at the beginning of each phase of the compiler.

Register	Label	Contents	
		Invocation Phase	low storage
POPPGB--->	POP TABLE	POP Jump Table	
	POP SETUP	POP Machine Language Subroutines	
		Data for POP Subroutines	
ROLLBR--->	ROLL BASE	Roll Statistics (Bases, Tops, Bottoms)	
		Group Stats (Displacements, Group Sizes)	
		WORK Roll	
		EXIT Roll	
		ROLL ADR Table	
		Roll Storage	
~~	~~~	~~~	
		Roll Storage*	
CONSTR--->	CBASE	Parse Data Items	
		Parse Routines	
PGB2----->	PROGRAM BASE 2	Parse Global Jump Table	
		Parse Routines containing assembler language branch targets	
	QUOTE BASE	Quotes	
	MESSAGE BASE	Messages	
		PHASE 2: Allocate	high storage
		PHASE 3: Unify	
		PHASE 4: Gen	
		PHASE 5: Exit	

*Roll storage is allocated in 4K-byte blocks, beginning from the higher end of storage contiguous with Parse. Additional blocks are obtained, as needed, from preceding (lower) 4K-byte blocks of storage.

• Figure 15. Compiler Arrangement with Registers

Compiler routines which contain assembler language instructions and are either branched to by other assembler language instructions or which themselves perform internal branches, follow the global jump table. General register 2 is used as a base register for references to both the global jump table and these routines. Figure 15 shows this register in Parse.

General register 3, called POPADR in the compiler code, is used in the sequencing of the POP operations. It holds the address of the current POP, and is incremented by 2 as each POP is interpreted.

General register 4, called WRKADR, holds the address of the current bottom of the WORK roll.

General register 5, called EXTADR, holds the address of the current bottom of the EXIT roll.

General register 6, called POPXIT, holds the return location for POP subroutines. When POPs are being interpreted by POP SETUP, the return is to POP SETUP; when machine language instructions branch to the POPs, it is to the next instruction.

General register 7, called ADDR, holds the address portion of the current POP instruction (eight bits); it is also used in the decoding of the operation code portion of POP instructions.

General register 8, called POPPGB, holds the beginning address of the machine language code for the POP instructions and the POP jump table. Figure 15 shows this register, which is used as a base for references to these areas.

General register 9, called CONSTR, holds the beginning address of the data referred to by the compiler routines. This area precedes the routines themselves, and is labeled CBASE, as indicated in Figure 15. This register is, therefore, used as a base register for references to data as well as for references to the routines in the compiler; its value is changed at the beginning of each phase.

General register 10, ROLLBR, holds the beginning address of the roll area; that is, the beginning address of the base table (see Figure 15). The value in this register remains constant throughout the operation of the compiler.

General register 11, RETURN, holds return addresses for the POP subroutines.

The remaining general registers are used temporarily for various purposes in the compiler.

POINTERS

Information defining a source module variable (its name, dimensions, etc.) is recorded by the compiler when the name of the variable appears in an Explicit specification or DIMENSION statement. For variables which are not explicitly defined, this information is recorded when the first use of the variable is encountered. All constants are recorded when they are first used in the source module.

All references to a given variable or constant are indicated by a pointer to the location at which the information defining that variable or constant is stored. The use of the pointer eliminates redundancy and saves compiler space.

The pointer is a 1-word value in the following format:

1 byte	1 byte	2 bytes
TAG	OPERATOR	ADDRESS

where:

TAG
: is a 1-byte item whose value is represented in two parts: MODE, occupying the upper four bits, indicates whether the variable or constant is integer, real, complex or logical; SIZE, indicated in the lower four bits, specifies the length of the variable or constant (in bytes) minus one. (See Figure 15.1).

Value	MODE	Value	SIZE
0	Integer	0	1 byte
1	Real	1	2 bytes
2	Complex	3	4 bytes
3	Logical	7	8 bytes
4	Literal/ Hexadecimal	F	16 bytes

• Figure 15.1 TAG Field MODE and SIZE Values

OPERATOR
: is a 1-byte item which contains the roll number of the roll on which the group defining the constant or variable is stored.

ADDRESS
: is a 2-byte item which holds the relative address (in bytes) of the group which contains the information for the constant or variable; the address is relative to the TOP of the roll.

The pointer contains all the information required to determine an absolute location in the roll storage area. The roll number (from the OPERATOR field) is first used as an index into the TOP table. The ADDRESS field of the pointer is then added to the TOP, and the result is handled as follows:

1. Its entry number field (bits 12 through 19) is used as an index into the ROLL ADR table.

2. Its displacement field (bits 20 through 31) is added to the base address found in the ROLL ADR table. The result of step 2 is the address indicated by the pointer.

Example: Using a pointer whose OPERATOR field contains the value 2 and whose ADDRESS field contains the value 4, and the following tables:

TOP

0			
1 2		2	20
	. . .		

ROLL ADR

0	
1 2	1000
	. . .

the location 1024 is determined. Note that for larger values in the pointer and in TOP, the entry number field of TOP can be modified by the addition of ADDRESS. In this case the result of the addition holds 2 and 24 in the entry number and displacement fields, respectively.

Since relative addresses are recorded in pointers, it is not necessary to alter a pointer when the roll pointed to is moved. Note also that the relative address in the pointer may exceed 4096 bytes with no complication of the addressing scheme. The only limitation on the size of a roll comes about because of the size of the ADDRESS field of the pointer: 16 bits permit values less than 64K bytes to be represented.

For the purposes of object code generation, the mode and size of the constant or variable is available to influence the type of operations which can be employed, e.g., integer or floating, fullword, or doubleword.

DRIVERS

In the generation of Polish notation from the source language statements, "drivers" are also used. These "drivers" are values that are one word long and have the same format as the pointer. The two types of drivers used by the compiler are discussed in the following paragraphs.

Operation Drivers

One type of driver is the operation driver, which indicates arithmetic or logical operations to be performed. The fields of the driver are:

TAG
: is a 1-byte item whose value is represented in two parts: MODE, occupying the upper four bits, indicates the mode of the operation, e.g., integer, floating-point, complex or logical; SIZE, indicated in the lower four bits, specifies the length of the result of the operation (in bytes) minus one.

OPERATOR
: is a 1-byte item containing a value which indicates the operation to be performed, e.g., addition, subtraction, etc. The values for OPERATOR are larger than the number of any roll, and hence, also serve to distinguish a driver from a pointer.

ADDRESS
: is a 2-byte item containing a value which indicates the "forcing strength" of the operation specified by the driver; its values range from zero to ten.

The forcing strengths associated with the operation drivers are given in Table 1.

Table 1. Internal Configuration of Operation Drivers

Driver	TAG[1]	OPERATOR	ADDRESS (Forcing Strength)
Sprog[2]	00	40	00 00
Power	00	42	00 01
Unary Minus	00	43	00 02
Multiply	00	44	00 03
Divide	00	45	00 03
Add	00	46	00 04
Subtract	00	47	00 04
GT	00	48	00 05
GE	00	49	00 05
LT	00	4A	00 05
LE	00	4B	00 05
EQ	00	4C	00 05
NE	00	4D	00 05
NOT	00	4E	00 06
AND	00	4F	00 07
OR	00	50	00 08
Plus and Below Phony[3]	00	3F	00 09
EOE[4]	00	3F	00 0A

[1]The MODE and SIZE settings are placed in the driver when it is used.
[2]Indicates a function reference.
[3]Used to designate the beginning of an expression.
[4]Means "end of expression" and is used for that purpose.

Control Drivers

The other type of driver used in the generation of Polish notation is called the control driver. It is used to indicate the type of the statement for which code is to be written. The control driver may also designate some other control function such as an I/O list, an array reference, or an error linkage.

The fields of the control driver differ from those of the operation driver in that zero is contained in the TAG field, 255 in the OPERATOR field (the distinguishing mark for control drivers), and a unique value in the ADDRESS field. The value in the ADDRESS field is an entry number into a table of branches to routines that process each statement type or control function; it is used in this way during the operations of Gen. The formats of the operation drivers and control drivers are given in Appendix E.

Table 1 lists the operation drivers and the values contained in each field. The control drivers are given in Table 2. The ADDRESS field is the only field given because the TAG and OPERATOR fields are constant. All values are represented in hexadecimal.

• Table 2. Internal Configuration of Control Drivers (Part 1 of 2)

Driver	ADDRESS
AFDS	8
ARRAY	23C
ASSIGN	20
ASSIGNED GOTO	1C
ASSIGNMENT	4
AT	68
BSREF	34
CALL	2C
CGOTO	18
CONTINUE	28
DATA	3C
DEFINE FILE	44
DIRECT IO	200
DISPLAY ID	74
DO	10
DUMMY	68
END	C
END=	20C
ERROR LINK 1	54
ERROR LINK 2	58
ERROR LINK 3	5C

• Table 2. Internal Configuration of Control Drivers (Part 2 of 2)

Driver	ADDRESS
ERR=	210
EXP and ARG	480
FIND	4C
FORMAT	208
FORMAT STA	30
GOTO	14
IF	24
IOL DO CLOSE	218
IOL DO DATA	21C
IO LIST	214
LOGICAL IF	60
NAMELIST	204
PAUSE	38
READ WRITE	48
RETURN	50
STANDARD PRINT UNIT	234
STANDARD PUNCH UNIT	238
STANDARD READ UNIT	230
STOP	64
SUBPROGRAM	40
TRACE OFF	70
TRACE ON	6C

This section describes in detail the Invocation phase and the five processing phases of the compiler and their operation. The IEYROL module is also described.

INVOCATION PHASE (IEYFORT)

The Invocation phase is the compiler control phase and is the first and last phase of the compiler. (The logic of the phase is illustrated in Chart 00.) If the compiler is invoked in an EXEC statement, control is received from the operating system control program. However, control may be received from other programs through use of one of the system macro instructions: CALL, LINK, or ATTACH.

IEYFORT performs compiler initialization, expansion of roll storage assignment, input/output request processing, and compiler termination. The following paragraphs describe these operations in greater detail.

IEYFORT, CHART 00

IEYFORT is the basic control routine of the Invocation phase. Its operation is invoked by the operating system or by another program through either the CALL, LINK, or ATTACH macro instructions. The execution of IEYFORT includes scanning the specified compiler options, setting the ddnames for designated data sets, initializing heading information, and acquiring time and date information from the system.

IEYFORT sets pointers and indicators to the options, data sets, and heading information specified for use by the compiler. The options are given in 40 or fewer characters, and are preceded in storage by a binary count of the option information. This character count immediately precedes the first location which contains the option data. The options themselves are represented in EBCDIC.

On entry to IEYFORT, general register 1 contains the address of a group of three or fewer pointers. Pointer 1 of the group holds the beginning address of an area in storage that contains the execute options specified by the programmer (set in the OPTSCAN routine).

Pointer 2 contains the address of the list of DD names to be used by the compiler (set in the DDNAMES routine).

Pointer 3 contains the address of the heading information. Heading data may designate such information as the continuation of pages, and the titles of pages.

If the FORTRAN compiler is invoked by the control program (i.e., called by the system), pointers 2 and 3 are not used. However, if the compiler is invoked by some other source, all pointers may be used. The latter condition is determined through an interrogation of the high order bit of a pointer. If this bit is set, the remaining pointers are nonexistent. Nevertheless, pointers 1 and 3 may exist while pointer 2 is nonexistent; in this case, pointer 2 contains all zeros.

During the operation of IEYFORT, the SYSIN and SYSPRINT data sets are always opened through use of the OPEN macro instruction. The SYSLIN and SYSPUNCH data sets are also opened depending upon the specification of the LOAD and DECK options. The block sizes of these data sets are set to 80, 120, 80 and 80, respectively. These data sets may be blocked or unblocked (RECFM=F, FB, or FBA) depending upon the DCB specification in the DD statements. IEYFORT concludes the compiler initialization process with a branch to the first processing phase of the compiler, Parse (IEYPAR).

From this point in the operation of the compiler, each processing phase calls the next phase to be executed. However, the Invocation phase is re-entered periodically when the compiler performs such input/output operations as printing, punching, or reading. The last entry to the Invocation phase is at the completion of the compiler operation.

IEYPRNT, Chart 00A4

IEYPRNT is the routine that is called by the compiler when any request for printing is issued. The routine sets and checks the print controls such as setting the line count, advancing the line count, checking the lines used, and controlling the spacing before and after the printing of each line. These control items are set, checked, and inserted into the SYSPRINT control format, and the parameter information and print addresses are initialized for SYSPRINT.

If there is an error during the printing operation, EREXITPR sets the error code resulting from the print error. Any error occurring during an input/output operation results in a termination of compiler operation.

PRNTHEAD, Chart 01A2

PRNTHEAD is called by IEYPRNT after it has been determined that the next print operation begins on a new page. The program name and the new page number placed into the heading format and any parameter information and origin addresses are inserted into the SYSPRINT format. If an optional heading is specified by the programmer, it is inserted into the print line format. A PUT macro instruction is issued to print the designated line, and all print controls are advanced for the next print operation.

IEYREAD, Chart 01A4

IEYREAD is called by the compiler at the time that a read operation is indicated. It reads input in card format from SYSIN using the GET macro instruction. IEYREAD can handle concatenated data sets.

If an error occurs during the read operation, the routine EREXITIN is called. This routine checks the error code generated and prints the appropriate error message.

IEYPCH, Chart 02A3

When a punch output operation is requested by the compiler, control is transferred to the IEYPCH routine. The LOAD and DECK options are checked to determine what output to perform.

Any errors detected during output result in a transfer of control to the EREXITPC, for SYSPUNCH, or EREXITLN, for SYSLIN, routine. The routine sets a flag so that no further output is placed on the affected file.

PRNTMSG, Chart 03A1

PRNTMSG is called when any type of message is to be printed. The print area is initialized with blanks and the origin and displacement controls are set. The message is printed in two segments; each segment is inserted into the print area after the complete message length is determined and the length and origin of each segment has been calculated. Once the entire message has been inserted, the carriage control for printing is set and control is transferred to the system to print the message.

IEYMOR, Chart 01D1

IEYMOR is called when additional roll storage area is needed for compiler operation. This routine may be entered from any of the processing phases of the compiler. The GETMAIN macro instruction is issued by this routine and transfers control to the system for the allocation of one 4K-byte block of contiguous storage. The system returns to IEYMOR with the absolute address of the beginning of the storage block in general register 1. Once the requested storage space has been obtained, IEYMOR returns to the invoking phase. If the system is unable to allocate the requested storage, inactive modules of the compiler are deleted. Those preceding the currently active module are deleted first; then those following it are deleted, if necessary. Should additional space be needed after all inactive modules are deleted, compiler operations are terminated.

When IEYMOR returns to the invoking phase with the absolute address of the storage block in general register 1, the invoking phase then stores the contents of register 1 in the ROLL ADR table.

The ROLL ADR table is used by the compiler to record the addresses of the different blocks of storage that have been allocated for additional roll capacity. The contents of the table are later used in IEYRETN for releasing of the same storage blocks.

IEYNOCR

IEYNOCR is called by PRESS MEMORY (IEYPAR) whenever it is unable to obtain at least 32 bytes of unused storage. IEYNOCR prints the message NO CORE AVAILABLE, branches to a subroutine that checks to see if there are any source language cards to be disregarded, and then exits to IEYRETN.

IEYRETN, Chart 03A2

The compiler termination routine (IEYRETN) is invoked by Exit (IEYEXT) or by one of the input/output routines after the detection of an error.

The routine first obtains the error condition code returned by the compiler and tests this value against any previous value received during the compilation. The compiler communications area for the error code is set to the highest code received and a program name of "Main" is set in the event of multiple compilations. The routine then checks general register 1 for the address of the ROLL ADR table. Each entry of the ROLL ADR table indicates the beginning of a 4K-byte block of roll storage that must be released. A FREEMAIN macro instruction is issued for each block of storage indicated in the table until a zero entry is encountered (this denotes the end of the ROLL ADR table).

The presence of more than one source module in the input stream is checked by interrogating the end-of-file indication and the first card following this notation. If another compilation is indicated, the line, card, and page count control items are reinitialized and all save registers used by the Invocation phase are restored. The first processing phase of the compiler, Parse (IEYPAR), is called and the operation of the compiler proceeds as described in the previous paragraphs and those pertaining to the processing phases.

If another compilation is not indicated, routine IEYFINAL closes the data set files used by the compiler (by means of the CLOSE macro instruction). The terminal error condition code is obtained and set for the return to the invoking program, and all saved registers are restored before the return is made.

Routine IEYFINAL also receives control from other compiler routines when an input/output error is detected.

OPTSCAN, Chart AA

OPTSCAN determines the existence of the parameters specifying the compiler options. If options are specified, the validity of each option is checked against the parameter table and the pointer to these options is set once the options have been validated. The program name is noted depending upon the presence or absence of the NAME parameter. However, if these options are not specified, the first pointer of the group of three supplied to the compiler by the system contains zero.

DDNAMES, Chart AB

DDNAMES scans the entries made for the names of the data sets to be used by the compiler. The entries corresponding to SYSN, SYSIN, SYSPRINT, and SYSPUNCH are checked; if an alternate name has been provided, it is inserted into the DCB area.

HEADOPT, Chart AC

HEADOPT determines the existence of the optional heading information. If such information exists, its length is determined, it is centered for printing, and then inserted into the Printmsg Table, with pointer 3 being set.

TIMEDAT, Chart AD

TIMEDAT serves only to obtain the time and date information from the system and insert the data into the heading line.

OUTPUT FROM IEYFORT

The following paragraphs describe the error messages produced during the operation of the Invocation phase. These messages denote the progress of the compilation, and denote the condition which results in the termination of the compiler.

IEY028I NO CORE AVAILABLE - COMPILATION TERMINATED

The system was unable to provide a 4K-byte block of additional roll storage and PRESS MEMORY was entered. It, too, was unable to obtain space. The condition code is 16.

IEY029I DECK OUTPUT DELETED

The DECK option has been specified, and an error occurred during the process of punching the designated output. No error condition code is generated for this error.

IEY030I LINK EDIT OUTPUT DELETED

The LOAD option has been specified, and an error occurred during the process of generating the load module. The condition code is 16.

IEY031I ROLL SIZE EXCEEDED

This message is produced when: (1) The WORK or EXIT roll has exceeded the storage capacity assigned; or (2) Another roll used by the compiler has exceeded 64K bytes of storage, thus making it unaddressable. (This condition applies to all rolls except the AFTER POLISH and CODE rolls.) The condition code is 16.

IEY032I NULL PROGRAM

This message is produced when an end-of-data set is encountered on the input data set prior to any valid source statement. The condition code is 0.

IEY034I I/O ERROR [COMPILATION TERMINATED] xxx...xxx

This message is produced when an input/output error is detected during compilation. If the error occurred on SYSPUNCH, compilation is continued and the COMPILATION TERMINATED portion of the message is not printed. The condition code is 8. If the error occurred on SYSIN, SYSPRINT, or SYSLIN, compilation is terminated. The condition code is 16. xxx...xxx is the character string formatted by the SYNADAF macro instruction. For an interpretation of this information, see the publication IBM System/360 Operating System: Supervisor and Data Management Macro-Instructions, Form C28-6647.

IEY035I UNABLE TO OPEN ddname

This message is produced when the required ddname data definition card is missing or the ddname is misspelled.

PHASE 1 OF THE COMPILER: PARSE (IEYPAR)

The first processing phase of the FORTRAN IV (G) compiler, Parse, accepts FORTRAN statements in card format as input and translates them. Specification statements are translated to entries on rolls which define the symbols of the program. Active statements are translated to Polish notation. The Polish notation and roll entries produced by Parse are its primary output. In addition, Parse writes out all erroneous statements and the associated error messages. Parse produces a full source module listing when the SOURCE option is specified.

The following description of Parse consists of two parts. The first part, "Flow of Phase 1," describes the overall logic of the phase by means of both narrative and flowcharts.

The second part, "Output from Phase 1," describes the Polish notation produced by Parse. The construction of this output, from which subsequent phases produce object code, is the primary function performed by Parse. See Appendix C for the Polish format for each statement type.

The source listing format and the error messages produced by Parse are also discussed.

The rolls manipulated by Parse are listed in Table 3 and are mentioned in the following description of the phase. At the first mention of a roll, its nature is briefly described. See Appendix B for a complete description of a format of a roll.

• Table 3. Rolls Used by Parse

Roll No.	Roll Name	Roll No.	Roll Name
0	Lib	28	Local Sprog
1	Source	29	Explicit
2	Ind Var	30	Call Lbl
4	Polish	31	Namelist Names
5	Literal Const	32	Namelist Items
6	Hex Const	33	Array Dimension
7	Global	35	Temp Data Name
8	Fx Const	36	Temp Polish
9	Fl Const	37	Equivalence
10	Dp Const	38	Used Lib Function
11	Complex Const	39	Common Data
12	Dp Complex Const	40	Common Name
13	Temp Name	41	Implicit
14	Temp	42	Equivalence Offset
14	Error Temp	43	Lbl
15	Do Loops Open	44	Scalar
16	Error Message	45	Data Var
17	Error Char	46	Literal Temp
18	Init	53	Format
19	Xtend Lbl	54	Script
20	Xtend Target Lbl	55	Loop Data
22	Array	56	Program Script
24	Entry Names	59	At
25	Global Dmy	60	Subchk
26	Error	63	After Polish
27	Local Dmy		

FLOW OF PHASE 1, CHART 04

START COMPILER initializes the operation of Parse, setting flags from the user options, reading and writing out (on option) any initial comment cards in the source module, and leaving the first card of the first statement in an input area. This routine concludes with the transfer of control to STATEMENT PROCESS.

STATEMENT PROCESS (G0631) controls the operation of Parse. The first routine called by STATEMENT PROCESS is PRINT AND READ SOURCE. On return from that routine, the previous source statement and its error messages have been written out (as defined by user options), and the statement to be processed (including any comment cards) plus the first card of the next statement will be on the SOURCE roll. (This roll holds the source statements, one character per byte.) STATEMENT PROCESS then calls STA INIT to initialize for the processing of the statement and LBL FIELD XLATE to process the label field of the statement.

On return from LBL FIELD XLATE, if an error has been detected in the label field or in column 6, STATEMENT PROCESS restarts. Otherwise, STA XLATE and STA FINAL are called to complete the translation of the source statement. On return from STA FINAL, if the last statement of the source module has not been scanned, STATEMENT PROCESS restarts.

When the last card of a source module has been scanned, STATEMENT PROCESS determines whether it was an END card; if not, it writes a message. The routine then sets a flag to indicate that no further card images should be read, and calls PRINT AND READ SOURCE to write out the last statement for the source listing (depending on whether the SOURCE option was specified or was indicated as the default condition at system generation time).

When no END card appears, two tests are made: (1) If the last statement was an Arithmetic IF statement, the Polish notation must be moved to the AFTER POLISH roll; (2) If the last statement was of a type which does not continue in sequence to the next statement (e.g., GO TO, RETURN), no code is required to terminate the object module, and the Polish notation for an END statement is constructed on the POLISH roll. If the NEXT STA LBL FLAG is off, indicating that the last statement was not of this type, the Polish notation for a STOP or RETURN statement is constructed on the POLISH roll, depending on whether the source module is a main program or a subprogram.

After the Polish notation for the STOP or RETURN has been constructed on the POLISH roll, the Polish notation for the END statement is then constructed.

Parse keeps track of all inner DO loops that may possibly have an extended range. Parse tags the LABEL roll entries for those labels within the DO loops that are possible re-entry points from an extended range. These tags indicate the points at which general registers 4 through 7 must be restored. The appropriate LOOP DATA roll groups are also tagged to indicate to the Gen phase which of the inner DO loops may possibly have an extended range. Gen then produces object code to save registers 4 through 7.

After processing the last statement of the source module, a pointer to the LOOP DATA roll is placed on the SCRIPT roll, the IND VAR roll is released, and, if the source module was a main program, the routine REGISTER IBCOM (G0707) is called to record IBCOM as a required subprogram. For all source modules, the information required for Allocate is then moved to the appropriate area, and the Parse phase is terminated.

PRINT and READ SOURCE, Chart BA

PRINT AND READ SOURCE (G0837) serves three functions:

1. It writes out the previous source statement and its error messages as indicated by user options.

2. It reads the new source statement to be processed, including any comment cards, as well as the first card of the statement following the one to be processed.

3. It performs an initial classification of the statement to be processed.

The statement to be written out is found on the SOURCE roll. One line at a time is removed from this roll and placed in a 120-byte output area from which it is written out. The new statement being read into the SOURCE roll is placed in an 80-byte input area and replaces the statement being written out as space on the SOURCE roll becomes available. Any blank card images in the source module are eliminated before they reach the SOURCE roll. Comment cards are placed on the SOURCE roll exactly as they appear in the source module. The last card image placed on the SOURCE roll is the first card of the source statement following the one about to be processed; therefore, any comment cards that appear between two statements are processed with the statement which precedes them. When an END card has been read, no further reading is performed.

The initial classification of the statement that occurs during the operation of this routine determines, at most, two characteristics about the statement to be processed: (1) If it is a statement of the assignment type, i.e., either an arithmetic or logical assignment statement or a statement function, or (2) If it is a Logical IF statement, whether the statement "S" (the consequence of the Logical IF) is an assignment statement. Two flags are set to indicate the results of this classification for later routines.

At the conclusion of this routine, all of the previous source statements and their errors have been removed from the SOURCE roll and are written out. In addition, all of the statements to be processed (up to and including the first card of the statement following it) have been placed on the SOURCE roll.

STA INIT, Chart BB

STA INIT (G0632) initializes for the Parse processing of a source statement. It sets the CRRNT CHAR CNT and the LAST CHAR CNT to 1, and places the character from column 1 of the source card in the variable CRRNT CHAR.

It then determines, from a count made during input of the statement, the number of card images in the statement; multiplying this value by 80, STA INIT sets up a variable (LAST SOURCE CHAR) to indicate the character number of the last character in the statement.

The routine finally releases the TEMP NAME roll and sets several flags and variables to constant initial values before returning to STATEMENT PROCESS.

LBL FIELD XLATE, Chart BC

LBL FIELD XLATE (G0635) first saves the address of the current WORK and EXIT roll bottoms. It then inspects the first six columns of the first card of a statement. It determines whether a label appears, and records the label if it does. If any errors are detected in the label field or in column 6 of the source card, LBL FIELD XLATE records these errors for later printing and returns to STATEMENT PROCESS (through SYNTAX FAIL) with the ANSWER BOX set to false.

Pointers to all labels within DO loops are placed on the XTEND LBL roll. Labels that are jump targets (other than jumps within the DO loop) are tagged to indicate to Gen at which points to restore general registers 4 through 7.

If the statement being processed is the statement following an Arithmetic IF statement, LBL FIELD XLATE moves the Polish notation for the Arithmetic IF statement to the AFTER POLISH roll after adding a pointer to the label of the present statement to it.

STA XLATE, Chart BD

Under the control of STA XLATE (G0636) the source module statement on the SOURCE roll is processed and the Polish notation for that statement is produced on the POLISH roll, which holds Polish notation for source statements, one statement at a time. Errors occurring in the statement are recorded for writing on the source module listing.

The addresses of the bottoms of the WORK and EXIT rolls are saved. Then, if the statement is of the assignment type (the first flag set by PRINT AND READ SOURCE is on), STA XLATE ensures that a BLOCK DATA subprogram is not being compiled and falls through to ASSIGNMENT STA XLATE (G0637). If a BLOCK DATA subprogram is being compiled, STA XLATE returns after recording an invalid statement error message. If the statement is not of the assignment type, a branch is made to LITERAL TEST (G0640), which determines the nature of the statement from its first word(s), and branches to the appropriate routine for processing the statement. The names of the statement processing routines indicate their functions; for example, DO statements are translated by DO STA XLATE, while Computed GO TO statements are translated by CGOTO STA XLATE.

With the exception of LOGICAL IF STA XLATE, the statement processing routines terminate their operation through STA XLATE EXIT. LOGICAL IF STA XLATE moves the second flag set by PRINT AND READ SOURCE (which indicates whether the statement "S" is an assignment statement) into the first flag, and calls STA XLATE as a subroutine for the translation of the statement "S." When all of the Logical IF statement, including "S," has been translated, LOGICAL IF STA XLATE also terminates through STA XLATE EXIT.

STA XLATE EXIT (G0723) determines whether errors in the statement are of a severity level which warrants discarding the statement. If such errors exist, and the statement is active (as opposed to a specification statement), the Polish notation produced for the statement is removed and replaced by an invalid statement driver before a return is made to STATEMENT PROCESS. Otherwise, the Polish notation is left intact, and a return is made to STATEMENT PROCESS.

STA FINAL, Chart BE

STA FINAL (G0633) increases the statement number by one for the statement just processed. It then determines whether any Polish notation has been produced on the POLISH roll; if no Polish notation is present, STA FINAL returns to STATEMENT PROCESS.

If the statement produced Polish notation of a type which may not close a DO loop, STA FINAL bypasses the check for the close of a DO loop. Otherwise, STA FINAL determines whether the label (if there is one) of the statement corresponds to the label of the terminal statement of a DO loop. If so, the label pointer (or pointers, if the statement terminates several DO loops) is removed from the DO LOOPS OPEN roll, which holds pointers to DO loop terminal statements until the terminal statements are found.

When the statement is the target of a DO loop, extended range checking is continued. DO loops which have no transfers out of the loop are eliminated as extended range candidates. In addition, the nest level count is reduced by one and the information concerning the array references in the closed loop is moved from the SCRIPT roll to the PROGRAM SCRIPT roll.

STA FINAL then places the label pointer (if it is required) on the Polish notation for the statement, and, at STA FINAL END, adds the statement number to the Polish.

Except when the statement just processed was an Arithmetic IF statement, STA FINAL END terminates its operation by moving the Polish notation for the statement to the AFTER POLISH roll. In the case of the Arithmetic IF, the Polish notation is not moved until the label of the next statement has been processed by LBL FIELD XLATE. When the Polish notation has been moved, STA FINAL returns to STATEMENT PROCESS.

ACTIVE END STA XLATE, Chart BF

ACTIVE END STA XLATE (G0642) is invoked by STATEMENT PROCESS when the END card has been omitted and the last statement in the source module has been read. If the last statement was not a branch, the routine determines whether a subprogram or a main program is being terminated. If it is a subprogram, the Polish notation for a RETURN is constructed; if it is a main program, the Polish notation for a STOP statement is constructed. If the last statement was a branch, this routine returns without doing anything.

PROCESS POLISH, Chart BG

PROCESS POLISH (G0844) moves a count of the number of words in the Polish notation for a statement, and the Polish notation for that statement, to the AFTER POLISH roll.

OUTPUT FROM PHASE 1

The output from Parse is the Polish notation and roll entries produced for source module active statements, the roll entries produced for source module specification statements, and the source module listing (on option SOURCE) and error messages. The following paragraphs describe the Polish notation and the source and error listings. See Appendix B for descriptions of roll formats.

Polish Notation

The primary output from Phase 1 of the compiler is the Polish notation for the source module active statements. This representation of the statements is produced one statement at a time on the POLISH roll. At the end of the processing of each statement, the Polish notation is transferred to the AFTER POLISH roll, where it is held until it is required by later phases of the compiler.

The format of the Polish notation differs from one type of statement to another. The following paragraphs describe the general rules for the construction of Polish notation for expressions. The specific formats of the Polish notation produced for the various FORTRAN statements are given in Appendix C.

Polish notation is a method of writing arithmetic expressions whereby the traditional sequence of "operand_1" "operation" "operand_2" is altered to a functional notation of "operation" "operand_2" "operand_1." Use of this notation has the advantage of eliminating the need for brackets of various levels to indicate the order of operations, since any "operand" may itself be a sequence of the form "operation" "operand" "operand," to any level of nesting.

Assuming expressions which do not include any terms enclosed in parentheses, the following procedure is used to construct the Polish notation for an expression:

1. At the beginning of the expression, an artificial driver is placed on the WORK roll; this driver is the Plus and Below Phony driver, and has a lower forcing strength than any arithmetic or logical operator. (Forcing strengths are given in Table 1.)

2. As each variable name or constant in the expression is encountered, a pointer to the defining group is placed on the POLISH roll.

3. When an operator is encountered, the corresponding driver is constructed and it is compared with the last driver on the WORK roll:

   a. If the current driver has a higher forcing strength than the driver on the bottom of the WORK roll (the "previous" driver, for the purposes of this discussion), the current driver is added to the WORK roll and the analysis of the expression continues.

   b. If the current driver has a forcing strength which is lower than or equal to the forcing strength of the previous driver, then:

      (1) If the previous driver is the Plus and Below Phony driver, the current driver replaces the previous driver on the WORK roll (this situation can only occur when the current driver is an EOE driver, indicating the end of the expression) and the analysis of the expression is terminated.

      (2) If the previous driver is not the Plus and Below Phony driver, the previous driver is removed from the WORK roll and placed on the POLISH roll, and the comparison of the current driver against the previous driver is repeated (that is, using the same current driver, this procedure is repeated from 3).

The sequence of operations which occurs when the analysis of an expression is terminated removes the EOE driver from the WORK roll.

Example 1: The expression A + B produces the Polish notation

```
A
B
+
```

where:

A represents a pointer to the defining group for the variable A

\+ represents the Add driver. This notation is produced from the top down; when it is read from the bottom up, the sequence described above for Polish notation is satisfied.

Explanation: The following operations occur in the production of this Polish notation:

1. The Plus and Below Phony driver is placed on the WORK roll.

2. A pointer to A is placed on the POLISH roll.

3. An Add driver is constructed and compared with the Plus and Below Phony driver on the bottom of the WORK roll; the Add driver has a higher forcing strength and is therefore added to the WORK roll (according to rule 3a, above).

4. A pointer to B is placed on the POLISH roll.

5. An EOE (end of expression) driver is constructed and compared with the Add driver on the bottom of the WORK roll; the EOE driver has a lower forcing strength, and the Add driver is therefore removed from the WORK roll and added to the POLISH roll (rule 3b2).

6. The EOE driver is compared with the Plus and Below Phony driver on the bottom of the WORK roll; the EOE driver has a lower forcing strength, and therefore (according to rule 3b1) replaces the Plus and Below Phony driver on the WORK roll.

7. The analysis of the expression is terminated and the EOE driver is removed from the WORK roll. The Polish notation for the expression is on the POLISH roll.

Example 2: The expression A + B / C produces the Polish notation

```
A
B
C
/
+
```

which, read from the bottom up, is + / C B A.

Explanation: The following operations occur in the production of this Polish notation:

1. The Plus and Below Phony driver is placed on the WORK roll.

2. A pointer to A is placed on the POLISH roll.

3. An Add driver is constructed and compared with the Plus and Below Phony driver; the Add driver has the higher forcing strength and is placed on the WORK roll.

4. A pointer to B is placed on the POLISH roll.

5. A Divide driver is constructed and compared with the Add driver; the Divide driver has the higher forcing strength and is placed on the WORK roll.

6. A pointer to C is placed on the POLISH roll.

7. An EOE driver is constructed and compared with the Divide driver; since the EOE driver has the lower forcing strength, the Divide driver is moved to the POLISH roll.

8. The EOE driver is compared with the Add driver; since the EOE driver has the lower forcing strength, the Add driver is moved to the POLISH roll.

9. The EOE driver is compared with the Plus and Below Phony driver; since the EOE driver has the lower forcing strength, it replaces the Plus and Below Phony driver on the WORK roll, and the analysis of the expression terminates with the removal of one group from the WORK roll.

Example 3: The expression A / B - C produces the Polish notation

```
A
B
/
C
-
```

which, read from the bottom up, is - C / B A.

Explanation: The following operations occur in the production of this Polish notation:

1. The Plus and Below Phony driver is placed on the WORK roll.

2. A pointer to A is placed on the POLISH roll.

3. A Divide driver is constructed and compared with the Plus and Below Phony driver; the Divide driver has the higher forcing strength and is added to the WORK roll.

4. A pointer to B is placed on the POLISH roll.

5. A Subtract driver is constructed and compared with the Divide driver; the Subtract driver has a lower forcing strength, therefore the Divide driver is moved to the POLISH roll.

6. The Subtract driver is compared with the Plus and Below Phony driver; the Subtract driver has the higher forcing strength and is added to the WORK roll.

7. A pointer to C is placed on the POLISH roll.

8. An EOE driver is constructed and compared with the Subtract driver; since the EOE driver has a lower forcing strength, the Subtract driver is moved to the POLISH roll.

9. The EOE driver is compared with the Plus and Below Phony driver; the EOE driver replaces the Plus and Below Phony driver on the WORK roll and the analysis of the expression is terminated.

Recursion is used in the translation of an expression when a left parenthesis is found; therefore, the term enclosed in the parentheses is handled as a separate expression. The following three examples illustrate the resulting Polish notation when more complicated expressions are transformed:

Expression	Polish Notation
1. A-B*(C+D)	-*+DCBA
2. (A-B)/(C*D)	/*DC-BA
3. X/Z/(X-C)+C**X	+**XC/-CX/ZX

The following should be noted with respect to the exponentiation operation:

- Exponentiations on the same level are scanned right to left. Thus, the expression A**B**C**D is equivalent to the expression A**(B**(C**D)).

- Two groups are added to the POLISH roll to indicate each exponentiation operation. The first of these is the Power driver; the second is a pointer to the group on the global subprogram roll (GLOBAL SPROG roll) which defines the required exponentiation routine. Thus, the expression A ** B produces the following Polish notation:

Pointer to A
Pointer to B
Power driver
Pointer to exponentiation routine

The concept of Polish notation is extended in the FORTRAN IV (G) compiler to include not only the representation of arithmetic expressions, but also the representation of all parts of the active statements of the FORTRAN language. The particular notation produced for each type of statement is described in Appendix C. Once an entire source statement has been produced on the POLISH roll, phase 1 copies this roll to the AFTER POLISH roll and the processing of the next statement begins with the POLISH roll empty.

Source Listing

The secondary output from Parse is the source module listing. If a source listing is requested by the user (by means of the option SOURCE), source module cards are listed exactly as they appear on the input data set with error messages added on separate lines of the listing. If no source module listing is requested, Parse writes only erroneous statements and their error messages.

The following paragraphs describe the error recording methods used in phase 1, the format of the source listing and the error messages generated.

ERROR RECORDING: As a rule, Parse attempts to continue processing source statements in which errors are found. However, certain errors are catastrophic and cause Parse to terminate processing at the point in the statement where the error occurred.

Statements which cannot be compiled properly are replaced by a call to the FORTRAN error routine IHCIBERH.

Throughout Parse, three techniques of error recording are used. The first of these is used when the error is not catastrophic. This method records the character position in the statement at which the error was detected (by means of IEYLCE, IEYLCT, or IEYLCF instructions) and the number of the error type on the ERROR roll; after recording this information, Parse continues to scan the statement.

The second and third techniques of error recording are used when the error detected is catastrophic, at least to part of the statement being scanned. The second technique is a jump to an error recording routine, such as ALLOCATION FAIL or SUBSCRIPTS FAIL, which records the error and jumps to FAIL. The third technique is the use of one of the instructions, such as IEYCSF or IEYQSF, which automatically jump to SYNTAX FAIL if the required condition is not met. SYNTAX FAIL also exits through FAIL.

If the statement being processed is active and errors have been detected in it, FAIL removes any Polish notation which has been produced for the statement from the POLISH roll, replacing it with an error indicator. FAIL then restores WORK and EXIT roll controls to their condition at the last time they were saved and returns accordingly.

Some translation routines modify the action of the FAIL routine through the use of the IEYJPE instruction so that FAIL returns immediately to the location following the IEYJPE instruction. The translation routine can then resume the processing of the statement from that point.

FORMAT OF THE SOURCE MODULE LISTING: Error information for a source module card containing errors appears on the listing lines immediately following that card. For each error encountered, a $ sign is printed beneath the active character preceding the one which was being inspected when the error was detected. The only exception would be in the case of a SYNTAX error. In such a case, the $ sign undermarks the character being inspected when the error is detected. The listing line which follows the printed card contains only the $ sign markers.

The next line of the listing describes the marked errors. The errors are numbered within the card (counting from one for the first error marked); the number is followed by a right parenthesis, the error number, and the type of the error. Three errors are described on each line, for as many lines as are required to list all the marked errors on the source card.

The following is an illustration of the printed output from phase 1:

```
DIMENSION ARY(200), BRY(200) CRY(5,10,10)
                                       $
1)   IEY004I COMMA
        .
        .
        .
       IF (AA + BB) 15, 20, 250000
                                       $
1)   IEY010I SIZE
       ARY(J) = BRY
$                $
1)  IEY002I LABEL     2) IEY012I SUBSCRIPT
       GTO 30
        $
1)   IEY013I SYNTAX
```

ERROR TYPES: The types of errors detected and reported by Parse are described in the following paragraphs. For each error type, the entire message which appears on the source output is given; the condition code and a description of the causes of this error follows the message.

IEY001I ILLEGAL TYPE: This message is associated with the source module statement when the type of a variable is not correct for its usage. Examples of situations in which this message would be given are: (1) The variable in an Assigned GO TO statement is not an integer variable; (2) In an assignment statement, the variable on the left of the equal sign is of logical type and the expression on the right side is not. The condition code is 8.

IEY002I LABEL: This message appears with a statement which should be labeled and is not. Examples of such statements are: (1) A FORMAT statement; (2) The statement following a GO TO statement. The condition code for the error is 0.

IEY003I NAME LENGTH: The name of a variable, COMMON block, NAMELIST, or subprogram exceeds six characters in length. If two variable names appear in an expression without a separating operation symbol, this message is produced. The condition code is 4.

IEY004I COMMA: A comma is supposed to appear in a statement and it does not. The condition code is 0.

IEY005I ILLEGAL LABEL: The usage of a label is invalid for example, if an attempt is made to branch to the label of a FORMAT statement, ILLEGAL LABEL is produced. The condition code is 8.

IEY006I DUPLICATE LABEL: A label appearing in the label field of a statement is already defined (has appeared in the label field of a previous statement). The condition code is 8.

IEY007I ID CONFLICT: The name of a variable or subprogram is used improperly, in the sense that a previous statement or a previous portion of the present statement has established a type for the name, and the present usage is in conflict with that type. Examples of such situations are: (1) The name listed in a CALL statement is the name of a variable, not a subprogram; (2) A single name appears more than once in the dummy list of a statement function; (3) A name listed in an EXTERNAL statement has already been defined in another context. The condition code is 8.

IEY008I ALLOCATION: Storage assignments specified by a source module statement cannot be performed due to an inconsistency between the present usage of a variable name and some prior usage of that name, or due to an improper usage of a name when it first occurs in the source module. Examples of the situations causing the error are: (1) A name listed in a COMMON block has been listed in another COMMON block; 2) A variable listed in an EQUIVALENCE statement is followed by more than seven subscripts. The condition code is 8.

IEY009I ORDER: The statements of a source module are used in an improper sequence. This message is produced, for example, when: (1) An IMPLICIT statement appears as anything other than the first or second statement of the source module; (2) An ENTRY statement appears within a DO loop. The condition code is 8.

IEY010I SIZE: A number used in the source module does not conform to the legal values for its use. Examples are: (1) The size specification in an Explicit specification statement is not one of the acceptable values; (2) A label which is used in a statement exceeds the legal size for a statement label; (3) An integer constant is too large. The condition code is 8.

IEY011I UNDIMENSIONED: A variable name indicates an array (i.e., subscripts follow the name), and the variable has not been dimensioned. The condition code is 8.

IEY012I SUBSCRIPT: The number of subscripts used in an array reference is either too large or too small for the array. The condition code is 8.

IEY013I SYNTAX: The statement or part of a statement to which it refers does not conform to FORTRAN IV syntax. If a statement cannot be identified, this error message is used. Other cases in which it appears are: (1) A non-digit appears in the label field; (2) Fewer than three labels follow the expression in an Arithmetic IF statement. The condition code is 8.

IEY014I CONVERT: In a DATA statement or in an Explicit specification statement containing data values, the mode of the constant is different from the mode of the variable with which it is associated. The compiler converts the constant to the correct mode. Therefore, this message is simply a notification to the programmer that the conversion is performed. The condition code is 0.

IEY015I NO END CARD: The source module does not contain an END statement. The condition code is 0.

IEY016I ILLEGAL STA.: The statement to which it is attached is invalid in the context in which it has been used. Examples of situations in which this message appears are: (1) The statement S in a Logical IF statement (the result of the true condition) is a specification statement, a DO statement, etc.; 2) An ENTRY statement appears in the source module and the source module is not a subprogram. The condition code is 8.

IEY017I ILLEGAL STA. WRN : A RETURN I statement appears in any source module other than a SUBROUTINE subprogram. The condition code is 0.

IEY018I NUMBER ARG: A reference to a library subprogram appears with the incorrect number of arguments specified. The condition code is 4.

IEY027I CONTINUATION CARDS DELETED: More than 19 continuation lines were read for 1 statement. All subsequent lines are skipped until the beginning of the next statement is encountered. The condition code is 8.

IEY033I COMMENTS DELETED: More than 30 comment lines were read between the initial lines of 2 consecutive statements. The 31st comment line and all subsequent comment lines are skipped until the beginning of the next statement is encountered. (There is no restriction on the number of comment lines preceding the first statement.) The condition code is 0.

IEY036I ILLEGAL LABEL WRN: The label on this nonexecutable statement has no valid use beyond visual identification, and may produce errors in the object module if the same label is the target of a branch-type statement. (Only branches to executable statements are valid.) This message is produced, for example, when an END statement is labeled. The message is issued as a warning only. The condition code is 4.

PHASE 2 OF THE COMPILER: ALLOCATE (IEYALL)

Phase 2 of the compiler performs the assignment of storage for the variables defined in the source module. The results of the allocation operations are entered on tables which are left in storage for the next phase. In addition, Allocate writes (on option) the object module ESD cards, the TXT cards for NAMELIST tables, literal constants, and FORMAT statements, and produces error messages and storage maps (optionally) on the SYSPRINT data set.

The following paragraphs describe the operations of Allocate in two parts. The first part, "Flow of Phase 2," describes the overall logic of the phase by means of narrative and flowcharts.

The second part, "Output from Phase 2," describes the error messages and memory maps which are produced on the source module listing during the operation of the phase, as well as the ESD and TXT cards produced. It also describes the types of error detection performed during Allocate.

Rolls manipulated by Allocate are listed in Table 4, and are briefly described in context. Detailed descriptions of roll structures are given in Appendix B.

Table 4. Rolls Used by Allocate

Roll No.	Roll Name	Roll No.	Roll Name
1	Source	39	Halfword Scalar
5	Literal Const	40	Common Name
7	Global Sprog	41	Implicit
14	Temp	42	Equivalence Offset
15	Do Loops Open	43	Lbl
18	Init	44	Scalar
19	Equiv Temp	45	Data Var
20	Equiv Hold	47	Common Data Temp
21	Base Table	48	Namelist Allocation
22	Array	48	Common Area
23	Dmy Dimension	49	Common Name Temp
24	Entry Names	50	Equiv Allocation
25	Global Dmy	52	Common Allocation
26	Error Lbl	53	Format
27	Local Dmy	60	Subchk
28	Local Sprog	68	General Allocation
29	Explicit		
30	Error Symbol		
31	Namelist Names		
32	Namelist Items		
34	Branch Table		
37	Equivalence		
37	Byte Scalar		
38	Used Lib Function		
39	Common Data		

FLOW OF PHASE 2, CHART 05

START ALLOCATION (G0359) controls the operation of the Allocate phase. The primary function of this routine is to call the subordinate routines which actually perform the operations of the phase.

The operation of Allocate is divided into three parts: the first part performs initialization; the second part (called pass 1) makes an estimate of the number of base table entries required to accommodate the data in the object module; the third part actually assigns storage locations for the object module components, leaving indications of the assignment in main storage for use by subsequent phases.

The first part of Allocate's operation is performed by calling the routines ALPHA LBL AND L SPROG, PREP EQUIV AND PRINT ERRORS, BLOCK DATA PROG ALLOCATION, PREP DMY DIM AND PRINT ERRORS, PROCESS DO LOOPS, PROCESS LBL AND LOCAL SPROGS, BUILD PROGRAM ESD, ENTRY NAME ALLOCATION, COMMON ALLOCATION AND OUTPUT, and EQUIV ALLOCATION PRINT ERRORS.

After return from EQUIV ALLOCATION PRINT ERRORS, START ALLOCATION initializes for and begins pass 1. The variable PROGRAM BREAK, which is used to maintain the relative address being assigned to an object module component, is restored after being destroyed during the allocation of COMMON and EQUIVALENCE. The groups in the <u>BASE TABLE roll</u> (which becomes the object module base table) are counted, and the value ten is added to this count to provide an estimate of the size of the object module base table. The BASE TABLE roll is then reserved so that groups added to the roll can be separated from those used in the count. The value one is assigned to the variable AREA CODE, indicating that storage to be assigned is all relative to the beginning of the object module and carries its ESD number.

When these operations are complete, START ALLOCATION calls BASE AND BRANCH TABLE ALLOC, and upon return from this routine again increases the variable PROGRAM BREAK by the amount of storage allocated to EQUIVALENCE. START ALLOCATION continues its operation by calling BUILD ADDITIONAL BASES, PREP NEMELIST, SCALAR ALLOCATE, ARRAY ALLOCATE, PASS 1 GLOBAL SPROG ALLOCATE, SPROG ARG ALLOCATION, LITERAL CONST ALLOCATION and FORMAT ALLOCATION.

After the operation of FORMAT ALLOCATION, the last part of Allocate is begun. The variable PROGRAM BREAK is reinitialized to the value it was assigned prior to pass 1. The BASE TABLE roll groups are counted to determine the total size of the roll after groups have been added by pass 1; again, five extra groups (or ten words) are added to the count to provide for data values which will appear in the object module, but which are not yet defined. The PASS 1 FLAG is then turned off, and START ALLOCATION calls DEBUG ALLOCATE, ALPHA SCALAR ARRAY AND SPROG, BASE AND BRANCH TABLE ALLOC, EQUIV MAP, SCALAR ALLOCATE, ARRAY ALLOCATE, GLOBAL SPROG ALLOCATE, SPROG ARG ALLOCATION, BUILD NAMELIST TABLE, LITERAL CONST ALLOCATION, and FORMAT ALLOCATION.

At RELEASE ROLLS, START ALLOCATION concludes its operation by releasing rolls, increasing the PROGRAM BREAK to ensure that the next base begins on a doubleword boundary, and calling CALCULATE BASE AND DISP and BUILD ADDITIONAL BASES in order to guarantee that at least three bases are allotted for the <u>TEMP AND CONST roll</u>. After this calculation, Allocate prepares for and relinquishes control to Unify.

<u>ALPHA LBL AND L SPROGS, Chart CA</u>

This routine (G0543) is the first routine called by START ALLOCATION. It moves the binary labels from the LBL roll and the statement function names from the LOCAL SPROG roll to the <u>DATA VAR roll</u>. The order of the labels and statement function names on their respective rolls is maintained, and the location on the DATA VAR roll at which each begins is recorded. The names are moved because Allocate destroys them in storing allocation information, and Exit needs them for writing the object module listing.

<u>ALPHA SCALAR ARRAY AND SPROG, Chart CA</u>

This routine moves the names of scalars, arrays, and called subprograms to the DATA VAR roll from the rolls on which they are placed by Parse. The order of names is preserved and the beginning location for each type of name on the DATA VAR roll is saved.

<u>PREP EQUIV AND PRINT ERRORS, Chart CB</u>

Subscript information on the <u>EQUIVALENCE OFFSET roll</u> (which indicates the subscripts used in EQUIVALENCE statements in the source module) is used by this routine

(G0362) to calculate the relative addresses of array elements referred to in statements. (Pointers to the EQUIVALENCE OFFSET roll are found on the EQUIVALENCE roll for all subscripted references in EQUIVALENCE statements.) The addresses computed are relative to the beginning of the array. When an array reference in a source module EQUIVALENCE statement is outside the array, designates an excessive number of dimensions, or specifies too few dimensions, an error message is printed by this routine.

BLOCK DATA PROG ALLOCATION, Chart CC

This routine (G0361) controls the allocation of data specified in DATA, COMMON, DIMENSION, EQUIVALENCE, and Type statements in a BLOCK DATA subprogram. Since all data specified in EQUIVALENCE must be allocated under COMMON, this routine registers an error upon encountering on the EQUIVALENCE roll. The routine terminates with a jump to RELEASE ROLLS (G0360), which, in turn, terminates the Allocate phase.

PREP DMY DIM AND PRINT ERRORS, Chart CD

This routine (G0365) constructs the DMY DIMENSION roll, placing a pointer to the ENTRY NAMES roll group defining the ENTRY with which a dummy array is connected, and a pointer to the array for each dummy array containing a dummy dimension.

Before the roll is constructed, this routine ensures that each array having dummy dimensions is itself a dummy, and that each dummy dimension listed for the array is either in COMMON or is a global dummy variable in the same call. If any of these conditions are not satisfied, error messages are written.

PROCESS DO LOOPS, Chart CE

This routine (G0371) inspects the DO LOOPS OPEN roll for the purpose of determining whether DO loops opened by the source module have been left unclosed; that is, whether the terminal statement of a DO loop has been omitted from the source module. The DO LOOPS OPEN roll holds pointers to labels of target statements for DO loops until the loops are closed. If any information is present on this roll, loops have been left unclosed.

On encountering information on the DO LOOPS OPEN roll, this routine records the undefined labels for listing as DO loop errors, and (on option) lists them. It also sets the high order bit of the TAG field of the LBL roll group which refers to the undefined label to zero; this indicates to Gen that the loop is not closed.

PROCESS LBL AND LOCAL SPROGS, Chart CF

This routine (G0372) constructs the BRANCH TABLE roll, which is to become the object module branch table. The routine first processes the LBL roll. For each branch target label found on that roll, a new BRANCH TABLE roll group is constructed, and the label on the LBL roll is replaced with a pointer to the group constructed. Undefined labels are also detected and printed during this process.

When this operation is complete, the LOCAL SPROG roll (which lists the names of all statement functions) is inspected, and for each statement function, a group is added to the BRANCH TABLE roll, and part of the statement function name is placed with a pointer to the constructed group.

BUILD PROGRAM ESD, Chart CG

This routine (G0374) constructs and punches the ESD cards for the object module itself (the program name) and for each ENTRY to the object module. It also assigns main storage locations to the object module heading by increasing the PROGRAM BREAK by the amount of storage required.

ENTRY NAME ALLOCATION, Chart CH

This routine (G0376) does nothing if the source module is other than a FUNCTION subprogram. If, however, the source module is a FUNCTION, this routine places the names of all ENTRYs to the source module on the EQUIVALENCE roll as a single EQUIVALENCE set; it also ensures that the ENTRY name has been used as a scalar in the routine. If the variable has not been used, an appropriate error message is printed and the scalar variable is defined by this routine.

COMMON ALLOCATION AND OUTPUT, Chart CI

This routine (G0377) allocates all COMMON storage, one block at a time, generating the COMMON ALLOCATION roll (which holds the name, base pointer, and displacement for all COMMON variables) in the process. Groups are added to the BASE TABLE roll as they are required to provide for references to variables in COMMON. The ESD cards for COMMON are constructed and written out. All errors in COMMON allocation are written on the source listing and the map of COMMON storage is also written (on option).

EQUIV ALLOCATION PRINT ERRORS, Chart CK

This routine (G0381) allocates storage for EQUIVALENCE variables, creating the EQUIVALENCE ALLOCATION roll in the process. For each variable appearing in an EQUIVALENCE set, except for EQUIVALENCE variables which refer to COMMON (which have been removed from the EQUIVALENCE roll during the allocation of COMMON storage), the name of the variable and its address are recorded.

The information pertaining to EQUIVALENCE sets is stored on the EQUIV ALLOCATION roll in order of ascending addresses. Required bases are added to the BASE TABLE roll, and all remaining EQUIVALENCE errors are printed.

BASE AND BRANCH TABLE ALLOC, Chart CL

This routine (G0437) assigns main storage for the object module save area, base table, and branch table. The required base table entries are added as needed, PROGRAM BREAK is increased, and the base pointer and displacement for each of these areas is recorded in a save area for use by Gen. During pass 1 of Allocate, this assignment of storage is tentative and depends on the estimate of the size of the base table. The second time this routine is operated, the actual number of base table entries required in the object module has been determined by pass 1 and the space allocation is final.

SCALAR ALLOCATE, Chart CM

Each group on the SCALAR roll is inspected by this routine (G0397), which defines all nonsubscripted variables. It allocates storage for the variables listed on the roll, except for those which are in COMMON or members of EQUIVALENCE sets. The first time SCALAR ALLOCATE operates, it determines the number of base table entries required to accommodate references to the object module scalar variables. The information on the SCALAR roll is not altered, nor is any other roll built or modified by the routine.

At the second operation of the routine, the SCALAR roll is modified, and the actual storage locations (represented by the base pointer and displacement) to be occupied by the scalar variable are either computed and stored on the SCALAR roll or copied from the COMMON or EQUIV ALLOCATION roll to the SCALAR roll.

All "call by name" dummy variables are placed on the FULL WORD SCALAR roll; as each remaining scalar is inspected, its mode is determined. If it is of size 8 or 16 (double-precision real or single- or double-precision complex), storage is allocated immediately. If the variable does not require doubleword alignment, it is moved to one of three rolls depending on its size: FULL WORD SCALAR, HALF WORD SCALAR, or BYTE SCALAR.

When all groups on the SCALAR roll have been processed in this manner, the variables on the FULL WORD SCALAR roll, then the HALF WORD SCALAR roll, then the BYTE SCALAR roll are assigned storage. The map of scalars is produced (on option) by this routine.

ARRAY ALLOCATE, Chart CN

This routine (G0401), like SCALAR ALLOCATE, is called twice by START ALLOCATE. The first time it is called, it determines the number of base table entries required for references to the object module arrays. The second time the routine is operated, it actually assigns storage for the arrays, and records the appropriate base pointer and displacement on the ARRAY roll.

As each array name is found on the ARRAY roll, it is compared with those on the COMMON, EQUIV, and GLOBAL DMY rolls. For COMMON and EQUIVALENCEd arrays, the allocation information is copied from the appropriate roll. Since all dummy arrays are "call by name" dummies, dummy array groups are always replaced with pointers to the GLOBAL DMY roll. For each array to be assigned storage, new base table entries are constructed as required. In no case is more than one base used for a single array.

Since arrays are allocated in the order of their appearance, some unused storage space may appear between consecutive arrays due to the required alignment. The array map is produced (on option) by this routine.

PASS 1 GLOBAL SPROG ALLOCATE, Chart CO

This routine (G0402) counts the groups on the GLOBAL SPROG and USED LIB FUNCTION rolls (which hold, respectively, the non-library and library subprogram names referred to in the source module) to determine the number of base table entries required for references to the subprogram addresses region of the object module. The required BASE TABLE roll groups are added.

SPROG ARG ALLOCATION, Chart CP

This routine (G0442) adds the number of arguments to subprograms (and thus, the number of words in the argument list area of the object module) to the PROGRAM BREAK, thus allocating storage for this portion of the object module. BASE TABLE roll groups are added as required.

PREP NAMELIST, Chart CQ

This routine (G0443) determines the amount of main storage space required for each object module NAMELIST table. The NAMELIST ALLOCATION roll is produced during this routine's operation; it contains, for each NAMELIST data item, the name of the item and a pointer to the SCALAR or ARRAY roll group defining it. If any data name mentioned in a NAMELIST is not the name of a scalar or array, the appropriate error message is printed by this routine.

The NAMELIST NAMES roll is left holding the NAMELIST name and the absolute location of the beginning of the corresponding object module NAMELIST table. Required BASE TABLE roll groups are added by this routine.

LITERAL CONST ALLOCATION, Chart CR

This routine (G0444) is called twice by START ALLOCATION. Its first operation determines the number of BASE TABLE roll groups which should be added to cover the literal constants in the object module. The second operation of the routine actually assigns storage for all literal constants (except those appearing in source module DATA and PAUSE statements) and writes (on option) the TXT cards for them.

FORMAT ALLOCATION, Chart CS

This routine (G0445) is called twice by START ALLOCATION. The first time it is called is during the operation of pass 1. In pass 1, the PROGRAM BREAK is increased by the number of bytes occupied by each FORMAT.

The second time that FORMAT ALLOCATION is called, each FORMAT is written out and the FORMAT roll is rebuilt. The base and displacement information and a pointer to the label of the FORMAT statement are the contents of the rebuilt FORMAT group. The map of the FORMAT statements used in the object module is also written out (on option) by this routine.

EQUIV MAP, Chart CT

This routine (G0441) adjusts the values on the EQUIVALENCE ALLOCATION roll to the corrected (for the correct allocation of the base table, since this routine operates after the completion of pass 1) base pointer and displacement, and constructs the BASE TABLE roll groups required. The map of EQUIVALENCE variables is produced (on option) by this routine.

GLOBAL SPROG ALLOCATE, Chart CU

This routine (G0403) goes through the GLOBAL SPROG and USED LIB FUNCTION rolls, inserting the base pointer and displacement for each of the subprograms listed there; this is the allocation of storage for the subprogram addresses region of the object module. The ESD cards for the subprograms are written, the required BASE TABLE roll groups are added, and a list of the subprograms called is produced (on option).

BUILD NAMELIST TABLE, Chart CV

This routine (G0405) operates after pass 1 of Allocate. It uses the NAMELIST NAMES roll in determining the base and displacement for each NAMELIST reference in the source module. The BASE TABLE roll groups are added as required. The PROGRAM BREAK is increased as indicated, and the TXT cards are written out according to the base and displacement calculations for each entry on the NAMELIST ALLOCATION roll. A map of the NAMELIST tables is produced (on option) by this routine.

BUILD ADDITIONAL BASES, Chart CW

This routine (G0438) is called whenever it may be necessary to construct a new BASE TABLE roll group. It determines whether a new base is required and, if so, constructs it.

DEBUG ALLOCATE, Chart CX

This routine (G0545) processes the information on the INIT and SUBCHK rolls, marking the groups on the SCALAR, ARRAY, and GLOBAL DMY rolls which define the variables listed. When all the information on the SUBCHK roll has been processed, the routine returns.

OUTPUT FROM PHASE 2

The following paragraphs describe the output from Allocate: error messages, maps, and cards. Allocate also produces roll entries describing the assignment of main storage. See Appendix B for descriptions of the roll formats.

Error Messages Produced by Allocate

The source module listing, with error indications and error messages for the errors detected during initial processing of the source statements, is produced by phase 1 of the compiler. Certain program errors can occur, however, which cannot be detected until storage allocation takes place. These errors are detected and reported (if a listing has been requested), at the end of the listing by ALLOCATE; the error messages are described in the following paragraphs.

FUNCTION ERROR: When the program being compiled is a FUNCTION subprogram, a check is made to determine whether a scalar with the same name as the FUNCTION and each ENTRY is defined. If no such scalars are listed on the SCALAR roll, the message

IEY019I FUNCTION ENTRIES UNDEFINED

is written on the source module listing. The message is followed by a list of the undefined names. The condition code is 0.

COMMON ERRORS: Errors of two types can exist in the definitions of EQUIVALENCE sets which refer to the COMMON area. The first type of error exists because of a contradiction in the allocation specified, e.g., the EQUIVALENCE sets (A,B(6),C(2)) and (B(8),C(1)). The second error type is due to an attempt to extend the beginning of the COMMON area, as in COMMON A,B,C and EQUIVALENCE (A,F(10)).

An additional error in the assignment of COMMON storage occurs if the source program attempts to allocate a variable to a location which does not fall on the appropriate boundary. Since each COMMON block is assumed to begin on a double-precision boundary, this error can be produced in either (or both) the COMMON statement and an EQUIVALENCE statement which refers to COMMON.

When each block of COMMON storage has been allocated, the message

IEY020I COMMON BLOCK /   / ERRORS

is printed if any error has been detected (the block name is provided). The message is followed by a list of the variables which could not be allocated due to the errors. The condition code is 4.

Unclosed DO Loops

If DO loops are initiated in the source module, but their terminal statements do not exist, Allocate finds pointers to the labels of the nonexistent terminal statements on the DO LOOPS OPEN roll. If pointers are found on the roll, the message

IEY021I UNCLOSED DO LOOPS

is printed, followed by a list of the labels which appeared in DO statements and were not defined in the source module. The condition code is 8.

UNDEFINED LABELS: If any labels are used in the source module but are not defined, they constitute label errors. Allocate checks for this situation. At the conclusion of this check, the message

IEY022I UNDEFINED LABELS

is printed. If there are undefined labels used in the source module, they are listed on the lines following the message. The condition code is 8.

EQUIVALENCE ERRORS: Allocation errors due to the arrangement of EQUIVALENCE statements which do not refer to COMMON variables may have two causes. The first of these is the conflict between two EQUIVALENCE sets; for example, (A,B(6),C(3)) and (B(8),C(1)).

The second is due to incompatible boundary alignment in the EQUIVALENCE set. The first variable in each EQUIVALENCE set is assigned to its appropriate boundary, and a record is kept of the size of the variable. Then, as each variable in the set is processed, if any variable of a greater size requires alignment, the entire set is moved accordingly. If any variable is encountered of the size which caused the last alignment, or of lower size, and that variable is not on the appropriate boundary, this error has occurred.

If EQUIVALENCE errors of either of these types occur, the message

IEY023I EQUIVALENCE ALLOCATION ERRORS

is printed. The message is followed by a list of the variables which could not be allocated according to source module specifications. The condition code is 4.

Another class of EQUIVALENCE error is the specification, in an EQUIVALENCE set, of an array element which is outside the array. These errors are summarized under the heading

IEY024I EQUIVALENCE DEFINITION ERRORS

on the source module listing. The condition code is 4.

DUMMY DIMENSION ERRORS: If variables specified as dummy array dimensions are not in COMMON and are not global dummy variables, they constitute errors. These are summarized under the heading

IEY025I DUMMY DIMENSION ERRORS

on the source module listing. The condition code is 4.

BLOCK DATA ERRORS: If variables specified within the BLOCK DATA subprogram have not also been defined as COMMON, they constitute errors. The message

IEY026I BLOCK DATA PROGRAM ERRORS

is produced on the source module listing followed by a summarization of the variables in error. The condition code for this type of error is 4.

Storage Maps Produced by Allocate

Allocate produces the storage maps described below during its operations; these maps are printed only if the MAP option is specified by the programmer.

COMMON MAP: The map of each COMMON block is produced by Allocate. The map is headed by two title lines; the first of these is

COMMON / name / MAP SIZE n

and the second is the pair of words

SYMBOL LOCATION

printed five times across the line. The title lines are followed by a list of the variables assigned to the COMMON block and their relative addresses, five variables per line, in order of ascending relative addresses. The name contained within the slashes is the name of the COMMON block. The amount of core occupied by the block (n) is given in hexadecimal and represents the number of bytes occupied.

SCALAR MAP: The scalar map is produced by Allocate and consists of two title lines, the first of which reads

SCALAR MAP

and the second of which is identical to the second title line of the COMMON maps. The title is followed by a list of the non-COMMON scalar variables, five variables per line, and their relative addresses, in order of ascending relative addresses.

ARRAY MAP: The first title line of the array map reads

ARRAY MAP

In all other respects, the array map is identical to the scalar map.

EQUIVALENCE MAP: The first title line of the map of EQUIVALENCE sets reads

EQUIVALENCE DATA MAP

The second line for both maps is standard. The variables listed in the EQUIVALENCE map are those not defined as COMMON.

NAMELIST MAP: This map shows the locations of the NAMELIST tables. The first title line reads

NAMELIST MAP

and the second line is standard. The symbol listed is the NAMELIST name associated with each of the tables.

FORMAT MAP: This map gives the labels and locations of FORMAT statements. The first title line is

FORMAT STATEMENT MAP

and the second title is the same as the others described. The symbol listed is the label of the FORMAT statement.

Subprogram List

Allocate prints a list of the subprograms called by the source module being compiled. This list is printed only if the MAP option is specified by the programmer. The subprogram list is headed by the line

SUBPROGRAMS CALLED

and contains the names of the SUBROUTINEs and FUNCTIONs referred to in the source module.

Cards Produced by Allocate

Allocate produces both ESD and TXT cards, provided that a DECK option or a LOAD option has been specified by the programmer. All ESD cards required by the object module are produced during this phase. These include cards for the CSECT in which the object module is contained for each COMMON block and for each subprogram referred to by the object module.

The ESD cards that are produced by Allocate are given in the following order according to type:

ESD, type 0 - contains the name of the program and indicates the beginning of the object module.

ESD, type 1 - contains the entry point to a SUBROUTINE or FUNCTION subprogram, or the name specified in the NAME option, or the name MAIN. The name designated on the card indicates where control is given to begin execution of the module.

ESD, type 2 - contains the names of subprograms referred to in the source module by CALL statements, EXTERNAL statements, explicit function references, and implicit function references.

ESD, type 5 - contains information about each COMMON block.

The TXT cards produced during this phase fill the following areas of the object module:

- The NAMELIST tables
- The literal constants
- The FORMAT statements

The other TXT cards required for the object module are produced by later phases of the compiler.

PHASE 3 OF THE COMPILER: UNIFY (IEYUNF)

The third phase of the compiler optimizes the subscripting operations performed by the object module by deciding, on the basis of frequency of use, which subscript expressions within DO loops are to appear in general registers, and which are to be maintained in storage.

The following paragraphs, "Flow of Phase 3," describe the operation of Unify by means of narrative and flowcharts.

The rolls manipulated by Unify are listed in Table 5 and are mentioned in the following discussion of the phase; these rolls are briefly described in context. See Appendix B for a complete description of any roll used in the phase.

Table 5. Rolls Used by Unify

Roll Number	Roll Name
2	Nonstd Script
3	Nest Script
4	Loop Script
13	Std Script
14	Temp
20	Reg
21	Base Table
22	Array
52	Loop Control
54	Script
55	Loop Data
56	Program Script
57	Array Ref
58	Adr Const

FLOW OF PHASE 3, CHART 07

START UNIFY (G0111) controls the operation of this phase of the compiler. It initializes for the phase by setting the proper number of groups on the ARRAY REF roll to zero (this function is performed by the routine ARRAY REF ROLL ALLOTMENT) and moving the information transmitted on the PROGRAM SCRIPT roll to the SCRIPT roll. When the initialization is complete, the reserve blocks on the SCRIPT roll are in order from the outermost loop of the last source module DO nest (at the top of the roll) to the innermost loop of the first source module DO nest (at the bottom of the roll).

After initialization, START UNIFY begins the optimizing process by inspecting the last group of a reserve block on the SCRIPT roll; a value of zero in this group indicates the end of the SCRIPT roll information. When the value is nonzero, DO NEST UNIFY is called to process the information for an entire nest of DO loops. On return from this routine, the nest has been processed; the count of temporary storage locations required is updated, and START UNIFY repeats its operations for the next nest of loops.

When all loops have been processed, START UNIFY makes a complete pass on the ARRAY REF roll, setting up the instruction format for the array references from pointers which have been left on the roll (CONVERT TO INST FORMAT actually sets up the instruction fields). When all groups on the ARRAY REF roll have been processed, a jump is made to CONVERT TO ADR CONST. This routine sets up groups as required on the ADR CONST roll from data on the LOOP CONTROL roll. When the LOOP CONTROL roll has been processed, this routine terminates the Unify phase by calling Gen.

ARRAY REF ROLL ALLOTMENT, Chart DA

This routine (G0145) constructs the ARRAY REF roll. The groups on this roll are initialized with values of zero. Pointers to the roll have been placed on the SCRIPT roll and in the Polish notation by Parse, but information has not actually been put on the roll before this routine is called. The number of groups required has been transmitted from Parse.

CONVERT TO ADR CONST, Chart DB

This routine (G0113) constructs the ADR CONST roll from the base address information on the LOOP CONTROL roll.

When the third word of the LOOP CONTROL roll group contains an area code and displacement, Unify requires a base address which it does not find in the base table. Since no values can be added to the base table by Unify, the required value must be placed in the temporary storage and constant area of the object module. The ADR CONST roll holds the information required for Exit to place the value in a temporary storage and constant location and to produce the RLD card required to get the proper modification of the value in that location at load time. This routine builds that information on the ADR CONST roll by allocating the temporary storage and constant locations for the area codes and displacement values it finds on the LOOP CONTROL roll. See Appendix B for further explanation of the rolls involved.

CONVERT TO INST FORMAT, Chart DC

This routine (G0112) sets up the first word (zero rung) of each ARRAY REF roll group by testing the contents of the later words (the register rungs) of the same roll. The result is the skeleton of the instruction to be used for an array reference. When the second and third words of the group point to a general register, they are shifted into the appropriate position and inserted into the zero rung. (See Appendix B for the configuration of the ARRAY REF roll group.) At each entry to this routine, one word is processed and that word is cleared to zero before the routine exits.

DO NEST UNIFY, Chart DD

This routine (G0115) first initializes for the processing of one nest of DO loops. For each DO loop, a reserve block exists on the SCRIPT roll and one group exists on the LOOP DATA roll. These blocks and groups are ordered so that, reading from the bottom of the rolls up, a nest level of one indicates the end of a nest of loops; that is, for each nest, the bottom block represents the inner loop and the top block represents the outer loop.

DO NEST UNIFY serves a control function in this phase, arranging information to be processed by DO LOOP UNIFY and LEVEL ONE UNIFY; it is these latter routines which actually perform the optimization of subscripting by means of register assignment. The main result of the optimization is that in the initialization code for each loop, only that portion of each subscript which depends on the DO loop variable is computed.

DO LOOP UNIFY expects to find a reserved block on the bottom of the NEST SCRIPT roll describing a loop one nest level deeper than the loop described by the bottom reserved block on the SCRIPT roll. Moreover, both the block on the SCRIPT roll and the block on the NEST SCRIPT roll must already reflect the allocation of arrays by Allocate; that is, both blocks must have been processed by NOTE ARRAY ALLOCATION DATA, another routine called by DO NEST UNIFY. This arrangement is required so that DO LOOP UNIFY can pass information from the loop being processed (on the NEST SCRIPT roll) to the next outer loop (on the SCRIPT roll).

A special case is made of the reserved block describing a loop of nest level one, since there is no outer loop to which information can be passed. The routine LEVEL ONE UNIFY processes in place of DO LOOP UNIFY in this case; it expects to find the reserved block describing the level one loop on the NEST SCRIPT roll.

IEYROL MODULE

The IEYROL module is loaded into main storage by program fetch, along with the Invocation phase and the five processing phases. It contains two static rolls (the WORK roll and the EXIT roll), roll statistics, group stats, and the ROLL ADR table. Throughout the operation of the compiler, it maintains a record of the storage space allocated by the control program to the dynamic rolls.

Phase 4 of the Compiler: Gen (IEYGEN)

Gen produces object code from the Polish notation and roll information left by previous phases of the compiler. The code produced by this phase appears, one statement at a time, on the CODE roll, and is saved there until it is written out by EXIT.

The following paragraphs, "Flow of Phase 4," describe the operation of this phase by means of narrative and flowcharts.

The rolls manipulated by Gen are listed in Table 6 and are mentioned in the following description of the phase; these rolls are briefly described in context. See Appendix B for a complete description of all of the rolls used in the phase.

Table 6. Rolls Used by Gen

Roll No.	Roll Name	Roll No.	Roll Name
1	Source	24	Entry Names
4	Polish	25	Global Dmy
8	Fx Const	34	Branch Table
9	Fl Const	36	Fx Ac
10	Dp Const	40	Temp Pntr
11	Complex Const	42	Fl Ac
12	Dp Complex Const	43	Lbl
		44	Scalar
14	Temp	45	Data Var
15	Do Loops Open	52	Loop Control
15	Loops Open	55	Loop Data
16	Temp and Const	56	Array Plex
17	Adcon	57	Array Ref
18	Data Save	59	At
22	Array	62	Code
23	Dmy Dimension	63	After Polish
23	Sprog Arg		

FLOW OF PHASE 4, CHART 08

START GEN (G0491) initializes for the operation of the Gen phase. It then calls ENTRY CODE GEN to produce the object heading code and PROLOGUE GEN and EPILOGUE GEN for the required prologues and epilogues. On return from EPILOGUE GEN, START GEN falls through to GEN PROCESS.

GEN PROCESS (G0492) controls the repetitive operations of Gen. It first calls GET POLISH, which moves the Polish notation for one statement from the AFTER POLISH roll to the POLISH roll. Using the Polish notation just moved, GEN PROCESS determines whether the statement to be processed was labeled; if it was, the routine LBL PROCESS is called. If the source statement was not labeled, or when LBL PROCESS returns, GEN PROCESS calls STA GEN and STA GEN FINISH. On return from STA GEN FINISH, GEN PROCESS restarts.

The termination of the Gen phase of the compiler occurs when an END statement has been processed. END STA GEN jumps directly to TERMINATE PHASE after the object code is produced, rather than returning to GEN PROCESS. TERMINATE PHASE is described in Chart EG and in the accompanying text.

ENTRY CODE GEN, Chart EA

ENTRY CODE GEN (G0499) first determines whether the source module is a subprogram. If it is not, the heading code for a main program is placed on the CODE roll, the location counter is adjusted, and the routine returns.

If the source module is a subprogram, ENTRY CODE GEN determines the number of entries to the subprogram, generates code for the main entry and for each secondary entry and, when all required entry code has been produced, it then returns.

PROLOGUE GEN, Chart EB

PROLOGUE GEN (G0504) processes the main entry and each additional ENTRY to the source subprogram, producing the required prologues. Prologue code transfers arguments as required and is, therefore, not produced if no arguments are listed for the ENTRY. The prologue code terminates with a branch to the code for the appropriate entry point to the subprogram; in preparation for the insertion of the address of that entry point, this routine records the location of the branch instruction on the ENTRY NAMES roll. If the source module is not a subprogram, PROLOGUE GEN exits.

EPILOGUE GEN, Chart EC

EPILOGUE GEN (G0508) processes the main entry and each additional ENTRY to a subprogram, producing the required epilogues. Epilogue code returns argument values and returns to the calling program. If this routine determines that the source module is not a subprogram, main program prologue and epilogue code are produced.

GET POLISH, Chart ED

This routine (G0712) moves the Polish notation for a single statement from the AFTER POLISH roll to the POLISH roll. The Polish notation is moved from the beginning of the AFTER POLISH roll, and a pointer is maintained to indicate the position on the roll at which the next statement begins.

Note: Unlike the other rolls, data from the AFTER POLISH roll is obtained on a first-in first-out basis (i.e., the BASE rather than the BOTTOM pointer is used). This is done to maintain the sequence of the source program.

LBL PROCESS, Chart EF

LBL PROCESS (G0493) stores the label pointer left on the WORK roll by GEN PROCESS in STA LBL BOX. It then inspects the LBL roll group defining the label, and determines whether the label is a jump target. If so, the base register table is cleared to indicate that base values must be reloaded.

If the label is not the target of a jump, or when the base register table has been cleared, the AT roll is inspected. For each AT roll entry (and, therefore, AT statement) referring to the labeled statement being processed, made labels are constructed for the debug code and for the next instruction in line, pointers to these labels are recorded on the AT roll, and an unconditional branch to the debug code is placed on the CODE roll.

When all AT references to the present label have been processed, an instruction is placed on the CODE roll to inform Exit that a label was present and that a branch table entry may be required. Then, if the trace flag is on (indicating the presence of the TRACE option in the source DEBUG statement), the debug linkage for TRACE and the binary label are placed on the CODE roll. If the trace flag is off, or when the code has been completed, LBL PROCESS returns.

STA GEN, Chart EG

STA GEN (G0515) uses the control driver left on the WORK roll by GEN PROCESS to index into a jump table (STA RUN TABLE), jumping to the appropriate routine for constructing the object code for the specific type of statement being processed. This operation is called a "run" on the driver; other "runs" occur in Gen for building specific instructions or for generating data references.

The names of the code generating routines indicate the functions they perform;

for example, assignment statements are processed by ASSIGNMENT STA GEN, while GO TO statements are processed by GO TO STA GEN. These routines construct the code for the statement on the CODE roll and, when the code is complete, return to GEN PROCESS.

END STA GEN processes the END statement and provides the normal termination of the Gen phase by jumping to TERMINATE PHASE after producing the code. The code produced for the END statement is identical to that for the STOP statement if a main program is being compiled or a RETURN statement if a subprogram is being compiled. If an AT statement precedes the END, an unconditional branch instruction is constructed to return from the debug code to the main line of code.

TERMINATE PHASE (G0544) prepares for and calls the Exit phase of the compiler.

STA GEN FINISH, Chart EH

STA GEN FINISH (G0496) determines whether the present statement is the closing statement of any DO loops; if it is, this routine generates the code required for the DO loop closing and repeats the check for additional loops to be closed.

When all DO closings have been processed, STA GEN FINISH resets pointers to temporary locations, clears accumulators, and returns to GEN PROCESS.

PHASE 5 OF THE COMPILER: EXIT (IEYEXT)

Exit produces the SYSPUNCH and/or SYSLIN output requested by the programmer, except for the ESD cards and TXT card produced by the Allocate phase. It also produces the listing of the object module on SYSPRINT, if it has been requested by the programmer.

The description of this phase of the compiler is divided into two parts. The first of these, "Flow of Phase 5," describes the overall logic of the phase by means of narrative and flowcharts.

The second part of the description of the phase, "Output from Phase 5," describes the output written by the phase.

The rolls used by Exit are listed in Table 7, and are briefly described in context. For further description of rolls, see Appendix B.

Table 7. Rolls Used by Exit

Roll Number	Roll Name
7	Global Sprog
16	Temp and Const
17	ADCON
20	CSECT
23	Sprog Arg
38	Used Lib Function
45	BCD
46	Base Table
51	RLD
52	Branch Table
58	Adr Const
62	Code

FLOW OF PHASE 5, CHART 09

The routine EXIT PASS (G0381) controls the operation of this phase. After initializing, this routine calls PUNCH NAMELIST MPY DATA and PUNCH TEMP AND CONST ROLL. The routine PUNCH ADR CONST ROLL is then called and, if an object module listing was requested, the heading for that listing is written out.

After this operation, EXIT PASS calls PUNCH CODE ROLL, records the memory requirements for the code, and prints the corresponding message. PUNCH BASE ROLL, PUNCH BRANCH ROLL, PUNCH SPROG ARG ROLL, PUNCH GLOBAL SPROG ROLL, PUNCH USED LIBRARY ROLL, PUNCH ADCON ROLL, ORDER AND PUNCH RLD ROLL, and PUNCH END CARD are then called in order. On return from the last of these, EXIT PASS releases rolls and exits to the Invocation phase of the compiler.

PUNCH TEMP AND CONST ROLL, Chart FA

This routine (G0382) initializes the location counter for the temporary storage and constant area of the object module. It then initializes a pointer to the TEMP AND CONST roll and begins the processing of that roll from top to bottom. Each group on the roll is moved to the output area; when the output area is full, a TXT card is written. When the entire TEMP AND CONST roll has been processed, a jump is made to PUNCH PARTIAL TXT CARD, which writes out any partial TXT card remaining in the output area and returns to EXIT PASS.

PUNCH ADR CONST ROLL, Chart FB

The information on the ADR CONST roll is used by this routine (G0383) to produce TXT cards for temporary storage and constant area locations which contain addresses. RLD roll entries are also produced to cause correct modification of those locations by the linkage editor. The beginning address of the temporary storage and constant area is computed. Then, for each ADR CONST roll entry, the TEMP AND CONST roll pointer is added to that value to produce the address at which an address constant will be stored. This address is placed in the TXT card and on the RLD roll, the address constant from the ADR CONST roll initializes that location, and the area code from the ADR CONST roll is placed on the RLD roll. (See Appendix B for roll descriptions.)

PUNCH CODE ROLL, Chart FC

PUNCH CODE ROLL (G0384) initializes a location counter and a pointer to the CODE roll. Inspecting one group at a time, it determines the nature of the word. If it is a statement number, PUNCH CODE ROLL simply stores it and repeats the operation with the next word.

If a group is a constant, it is placed in the output area for SYSPUNCH and/or SYSLIN. This category includes literals which appear in-line and, thus, the constant to be written may occupy several groups on the roll.

Groups representing code are placed in the output area and, if an object module listing has been requested, the line entered into the output area is listed before it is punched. The contents of the DATA VAR roll are used for the listing of the operands.

If the group on the CODE roll is an indication of the definition of an address constant, the location counter is stored accordingly, and the operation of the routine continues with the next group.

PUNCH CODE ROLL also determines whether the group is an indication of the definition of a label, if it is, the routine defines the label on the BRANCH TABLE roll as required, inserts the label in the output line for the object module listing and repeats with the next group on the roll.

When all groups on the roll have been processed, a transfer to PUNCH PARTIAL TXT CARD is made; that routine writes out any incomplete TXT card which may be in the output area, and returns to EXIT PASS.

PUNCH BASE ROLL, Chart FD

PUNCH BASE ROLL (G0399) initializes a pointer to the BASE TABLE roll and initializes the location counter to the beginning address of the object module base table. It then enters each group on the BASE TABLE roll into the TXT card output area; it also records the object module ESD number and the location counter on the RLD roll for later production of the RLD cards. Whenever the output area is full, a TXT card is written. When all groups on the BASE TABLE roll have been processed, the routine makes a jump to PUNCH PARTIAL TXT CARD, which writes out any incomplete card in the output area and returns to EXIT PASS.

PUNCH BRANCH ROLL, Chart FE

This routine (G0400) first initializes a pointer to the BRANCH TABLE roll, and the location counter to the beginning location of the object module branch table. When these operations are completed, the routine inspects the BRANCH TABLE roll from top to bottom, making the requisite entries on the RLD roll and entering the addresses from the roll in the TXT card output area. TXT cards are written when the output area is full. When all BRANCH TABLE roll groups have been processed, the routine jumps to PUNCH PARTIAL TXT CARD, which writes out any incomplete card in the output area and returns to EXIT PASS.

PUNCH SPROG ARG ROLL, Chart FF

PUNCH SPROG ARG ROLL (G0402) initializes a pointer to the SPROG ARG roll and initializes the location counter to the beginning address of the subprogram arguments area of the object module.

The routine then inspects the groups on the SPROG ARG roll. If the first word of the group contains the value zero (indicating an argument whose address will be stored dynamically), the group is placed in the TXT card output area, and the card is written if the area is full. The routine then repeats with the next group on the roll.

If the SPROG ARG roll group does not contain zero, the group is then inspected to determine whether it refers to a temporary location. If it does, the correct location (address of the temporary storage and constant area plus the relative address within that area for this location) is determined. The required RLD roll entries are then made, the address is moved to the output area, and PUNCH SPROG ARG ROLL repeats this process with the next group on the roll.

If the group from the SPROG ARG roll contained neither a zero nor a temporary location, the argument referenced must have been a scalar, an array, a label or a subprogram. In any of these cases, a base table pointer and a displacement are on the pointed roll. From these, this routine computes the location of the variable or label or the subprogram address, enters it in the TXT card output area, and records the RLD information required on the RLD roll. The routine then repeats with the next group on the SPROG ARG roll.

This routine exits to EXIT PASS through PUNCH PARTIAL TXT CARD when all SPROG ARG roll groups have been processed.

PUNCH GLOBAL SPROG ROLL, Chart FG

This routine (G0403) first inverts the GLOBAL SPROG roll and moves one word from that roll to the WORK roll. If these actions indicate that there is no information on the roll, the routine exits.

Otherwise, for each group on the GLOBAL SPROG roll, this routine enters the ESD number for the subprogram and the location at which its address is to be stored on the RLD roll. The routine also writes a word containing the value zero for each subprogram listed (these words become the object module subprogram addresses region). When all groups on the GLOBAL SPROG roll have been processed, the routine exits through PUNCH PARTIAL TXT CARD, which writes out any incomplete card remaining in the output area before returning to EXIT PASS.

PUNCH USED LIBRARY ROLL, Chart FH

This routine (G0404) performs the same function for the USED LIB FUNCTION roll that the previous routine performs for the GLOBAL SPROG roll, thus completing the subprogram addresses region of the object module. The techniques used for the two rolls are identical.

PUNCH ADCON ROLL, Chart FI

This routine (G0405) returns immediately to EXIT PASS if there is no information on the ADCON roll. Otherwise, it writes out one TXT card for each group it finds on the roll, obtaining the area code, the address constant, and the address of the constant from the ADCON roll. The ESD number and the address of the constant are placed on the RLD roll for subsequent processing. A TXT card is punched containing the constant. The operation of PUNCH ADCON ROLL terminates when all groups on the roll have been processed.

ORDER AND PUNCH RLD ROLL, Chart FJ

This routine (G0416) sorts the RLD roll and processes the groups on that roll, producing the object module RLD cards. The card images are set up, and the RLD cards are actually written out as they are completed. When all information on the roll has been processed, this routine returns to EXIT PASS.

PUNCH END CARD, Chart FK

PUNCH END CARD (G0424) produces the object module END card. It moves the required information into the card image and initiates the write operation; it then returns to EXIT PASS.

PUNCH NAMELIST MPY DATA, Chart FL

This routine (G0564) is responsible for the punching of TXT and RLD cards for those words in the object module NAMELIST tables which contain pointers to array dimension multipliers. The multipliers themselves are placed on the TEMP AND CONST roll. The required information is found on the NAMELIST MPY DATA roll; when all groups have been processed, this routine returns to EXIT PASS.

OUTPUT FROM PHASE 5

Three types of output are produced by the Exit phase of the compiler: TXT cards, RLD cards, and the object module listing. The cards are produced on SYSPUNCH and or SYSLIN, according to the user's options. The listing, if requested, is produced on SYSPRINT.

The formats of the TXT and RLD cards are described in the publication IBM System/360 Operating System: Linkage Editor Program Logic Manual. The object module listing consists of the following fields:

- Location, which is the hexadecimal address, relative to the beginning of the object module control section, of the displayed instruction.
- Statement number (entitled STA NUM), which is the consecutive statement number assigned to the source module statement for which the displayed instruction is part of the code produced. This value is given in decimal.
- Label, which is the statement label, if any, applied to the statement for which the code was produced. The statement label is given in decimal.
- Operation code (entitled OP), which is the symbolic operation code generated.
- Operand, which is given in assembly format but does not contain any variable names.
- Operand (entitled BCD OPERAND), which contains the symbolic name of the variable referred to in the source module statement which resulted in the code.

Chart 00. IEYFORT (Part 1 of 4)

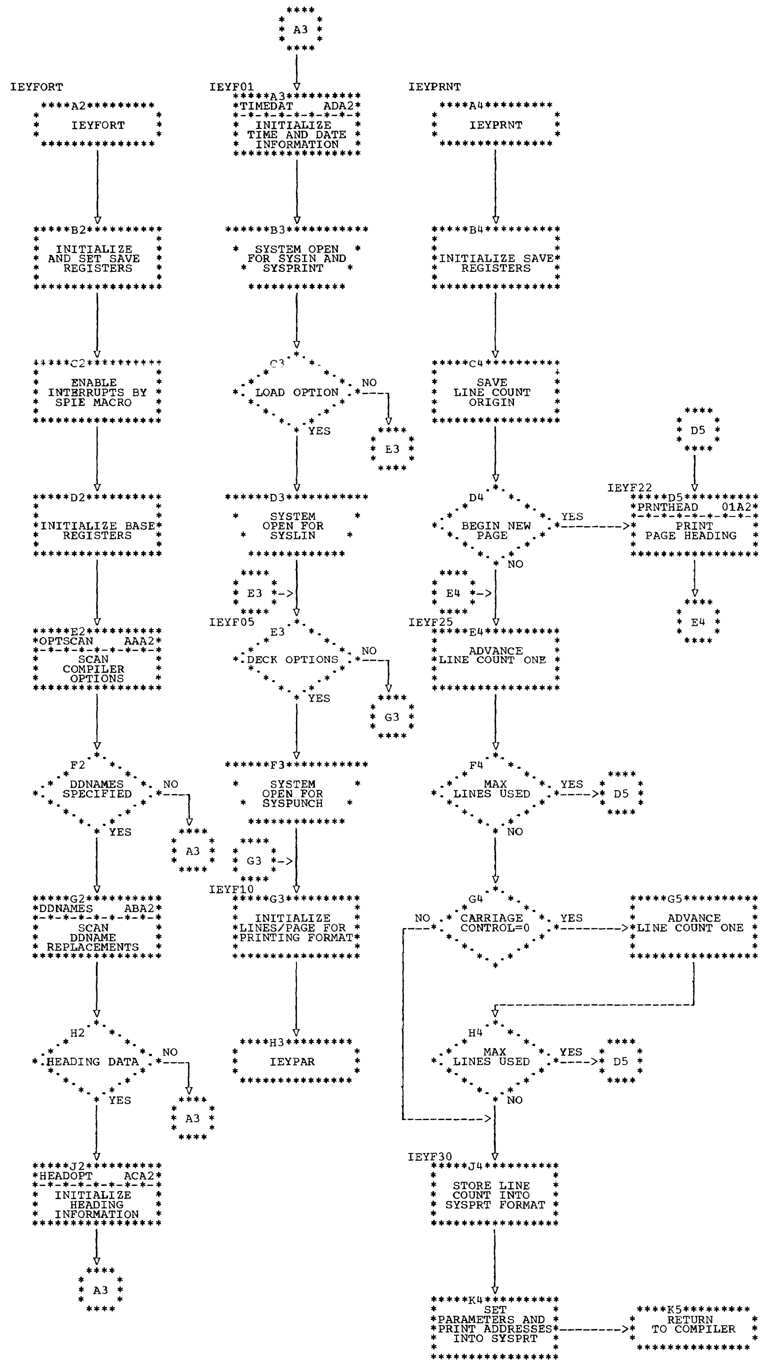


Chart 01. IEYFORT (Part 2 of 4)

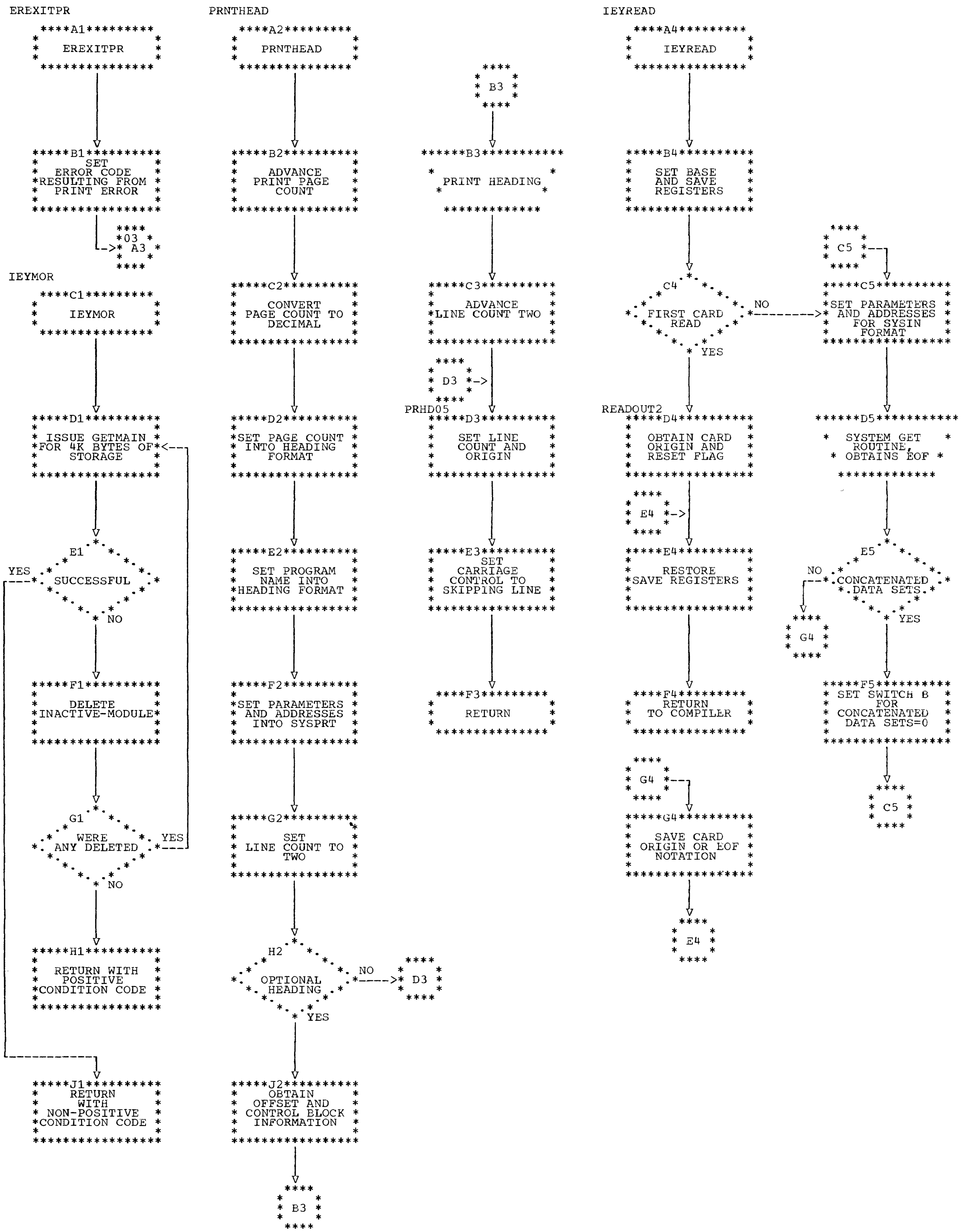


Chart 02. IEYFORT (Part 3 of 4)

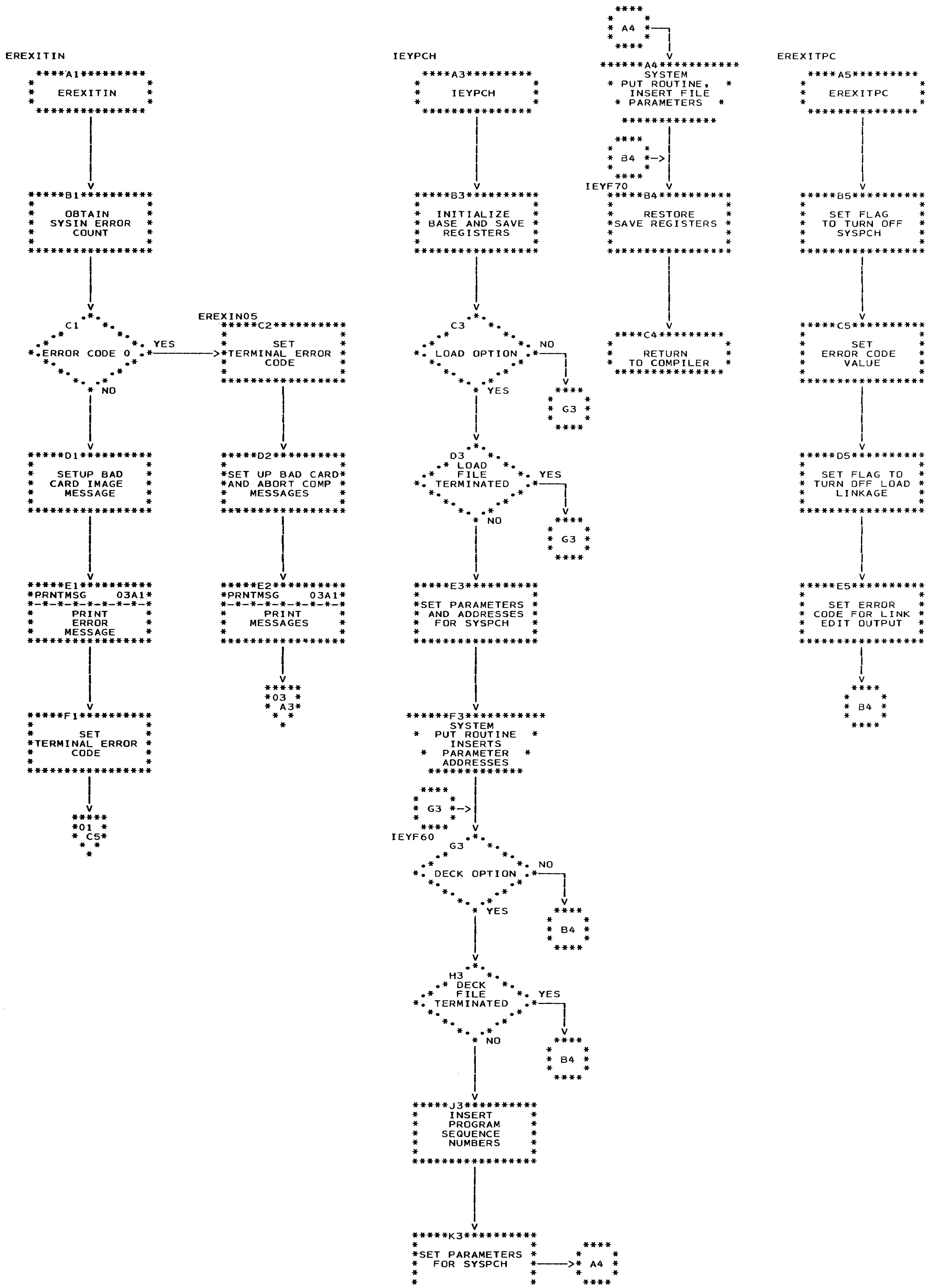


Chart 03. IEYFORT (Part 4 of 4)

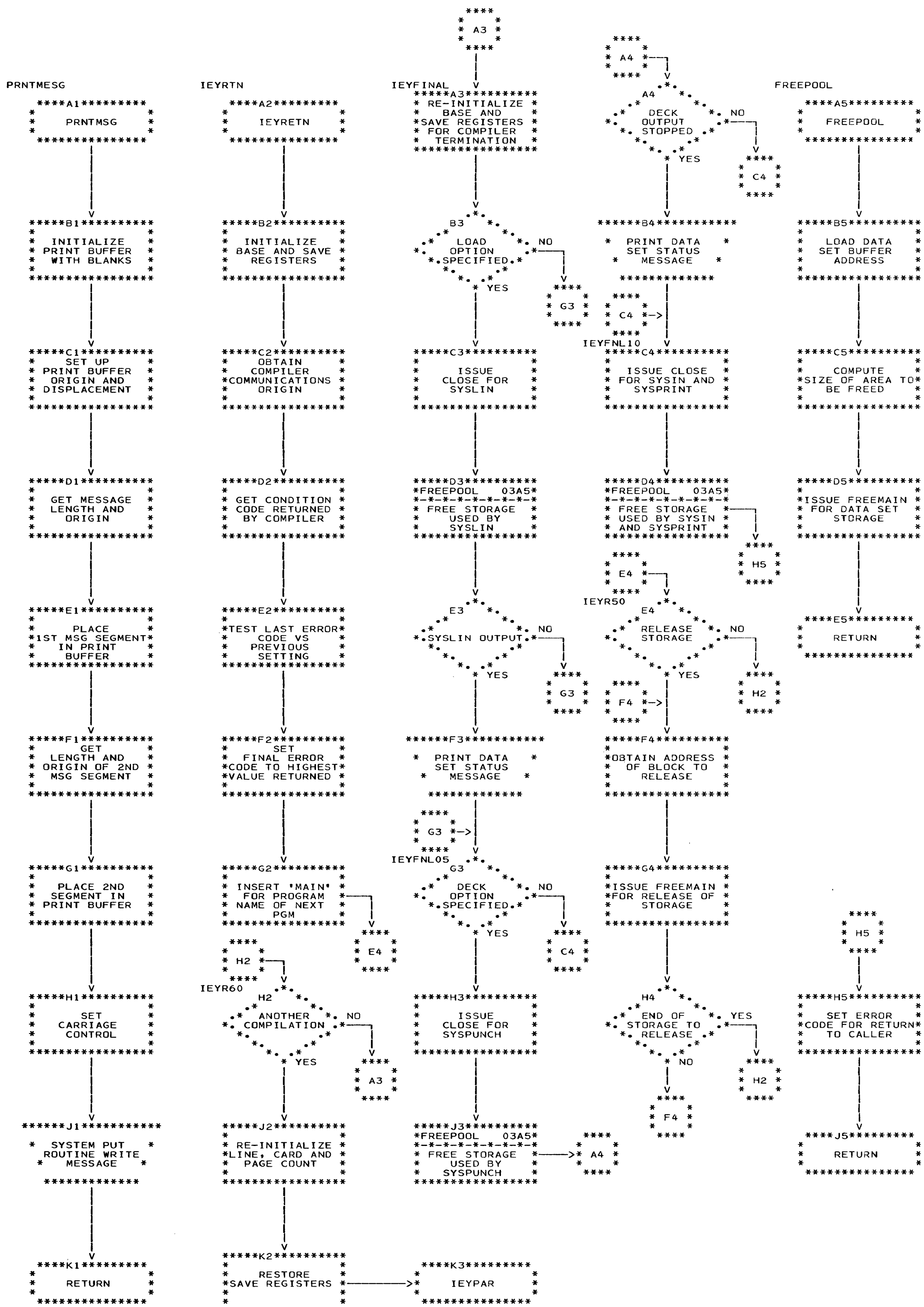


Chart AA. OPTSCAN

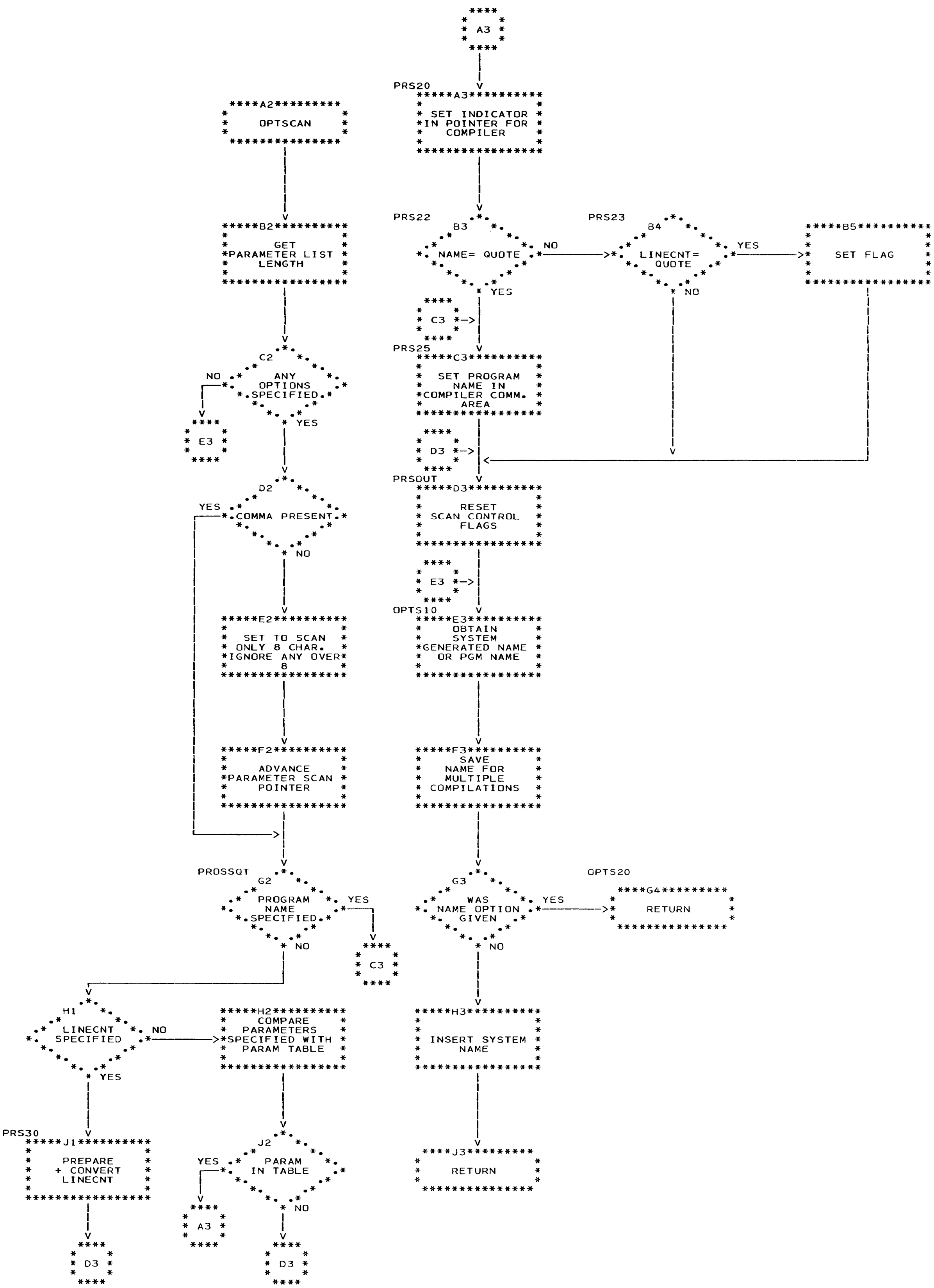


Chart AB. DDNAMES

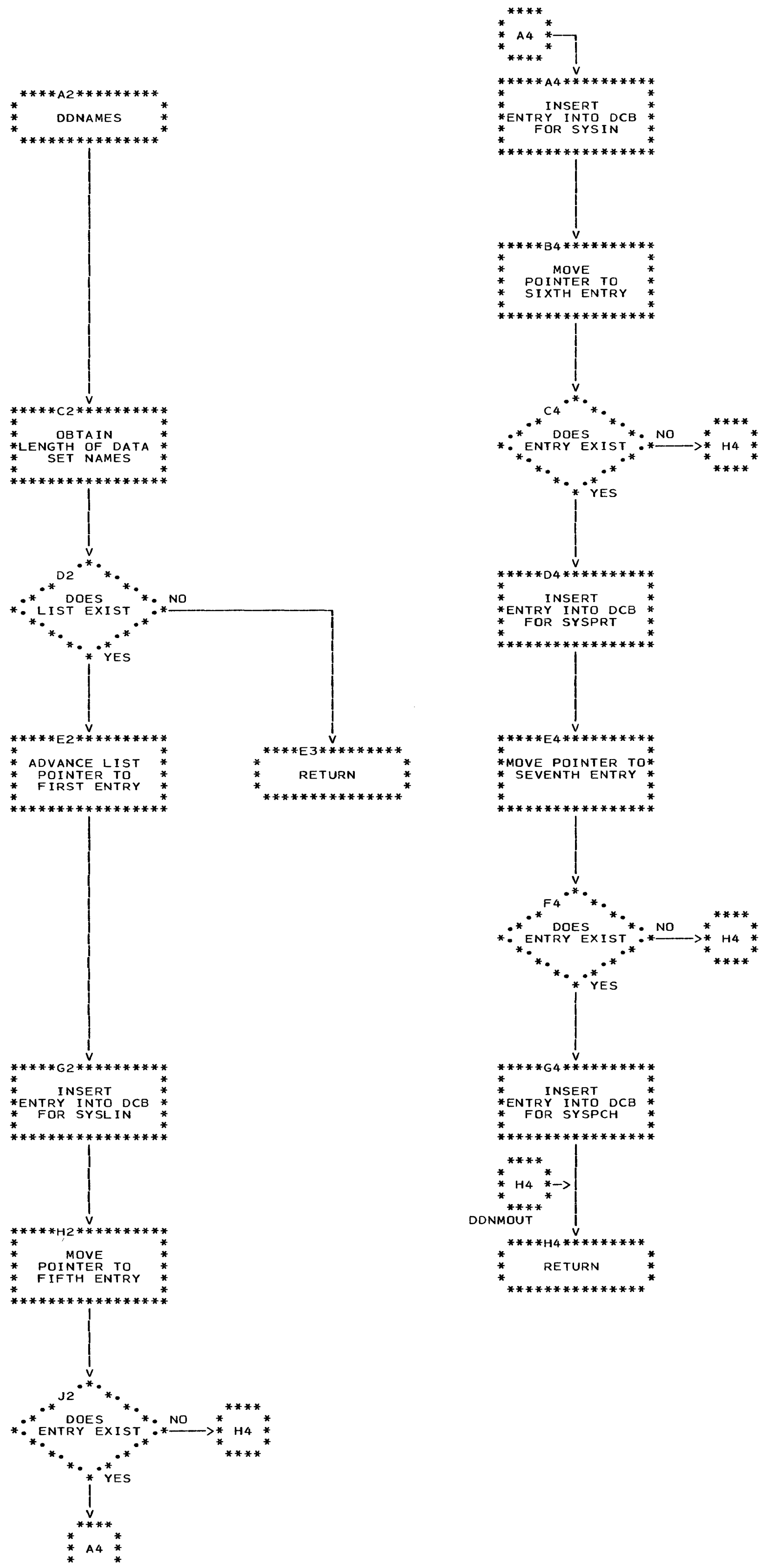


Chart AC. HEADOPT

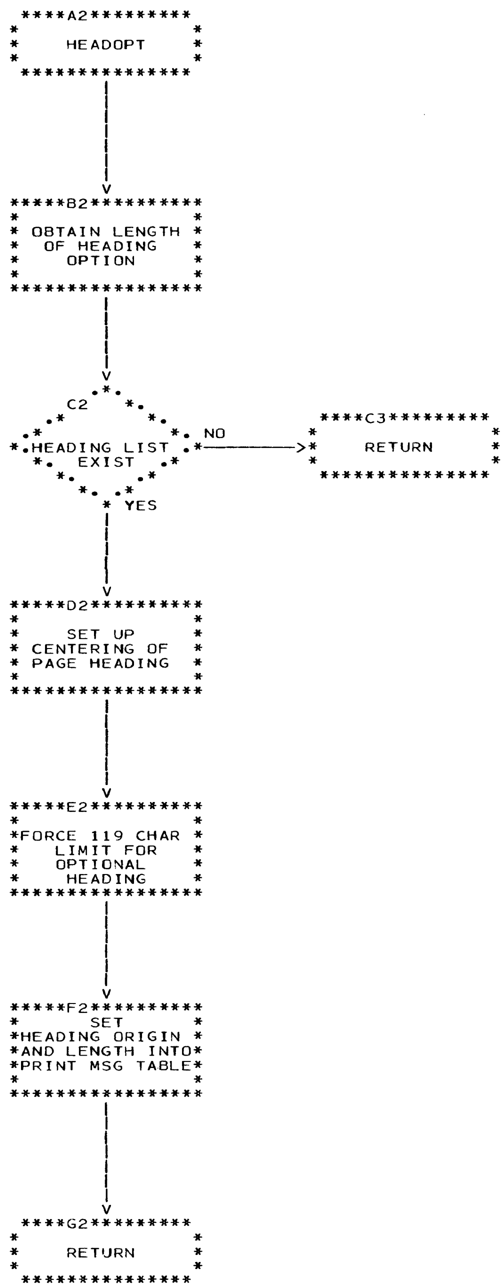
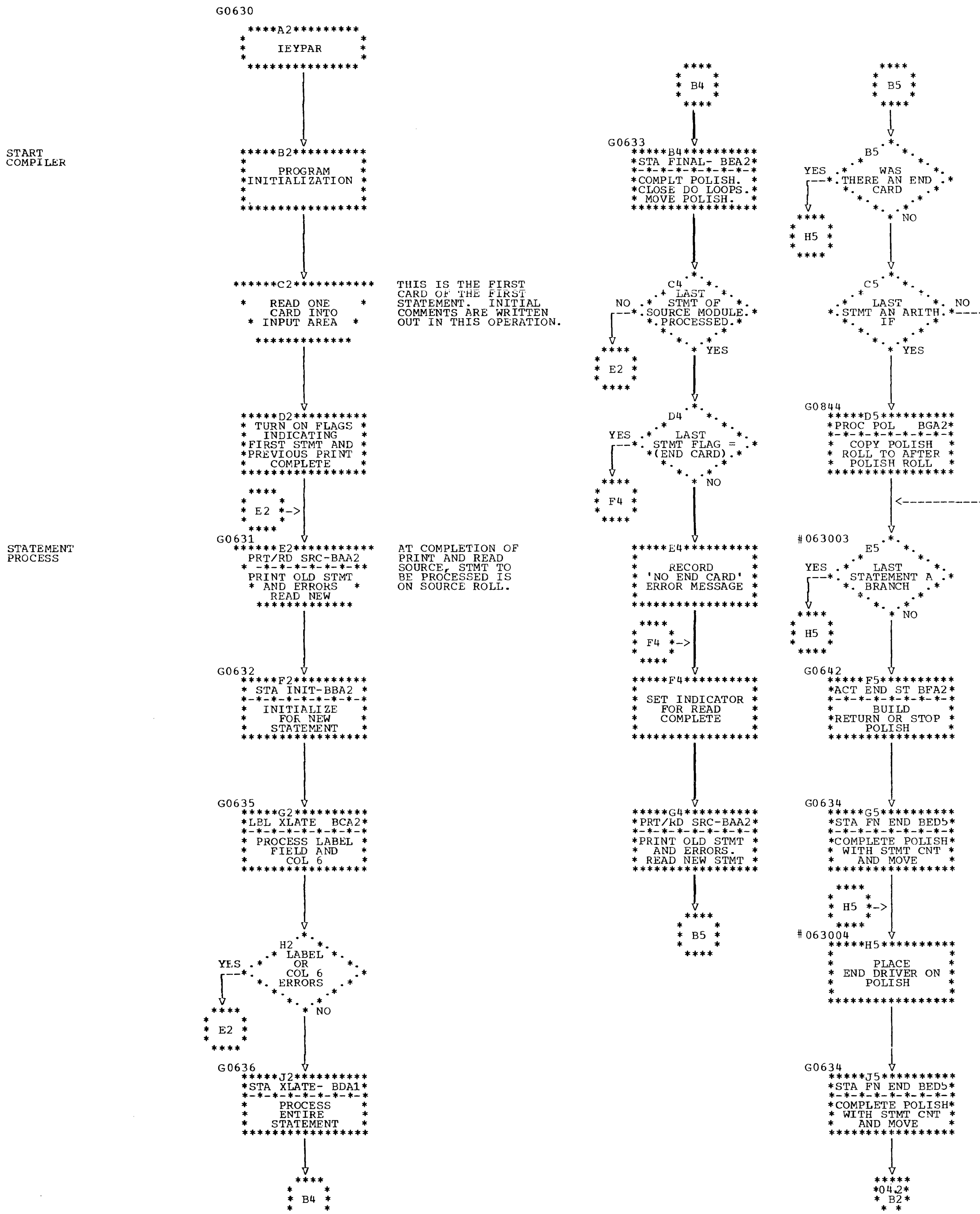


Chart AD. TIMEDAT

```
  ****A2*********
 *               *
 *    TIMEDAT    *
 *               *
  ***************
         |
         |
         |
         |
         |
         |
         V
*****B2**********
*     SET UP    *
*      UNIT     *
* SPECIFICATION *
*FOR TIME OF DAY*
*               *
*****************
         |
         |
         |
         |
         |
         V
*****C2**********
*      GET      *
* TIME AND DATE *
*  FROM SYSTEM  *
*   SUPERVISOR  *
*               *
*****************
         |
         |
         |
         |
         |
         V
*****D2**********
*               *
*     INSERT    *
*   TIME INTO   *
* HEADING LINE  *
*               *
*****************
         |
         |
         |
         |
         |
         V
*****E2**********
*               *
*     INSERT    *
*DATE INTO LINE *
*               *
*               *
*****************
         |
         |
         |
         |
         |
         V
  ****F2*********
 *               *
 *    RETURN     *
 *               *
  ***************
```

• Chart 04.1. PHASE 1 - PARSE (Part 1 of 2)



• Chart 04.2. PHASE 1 - PARSE (Part 2 of 2)

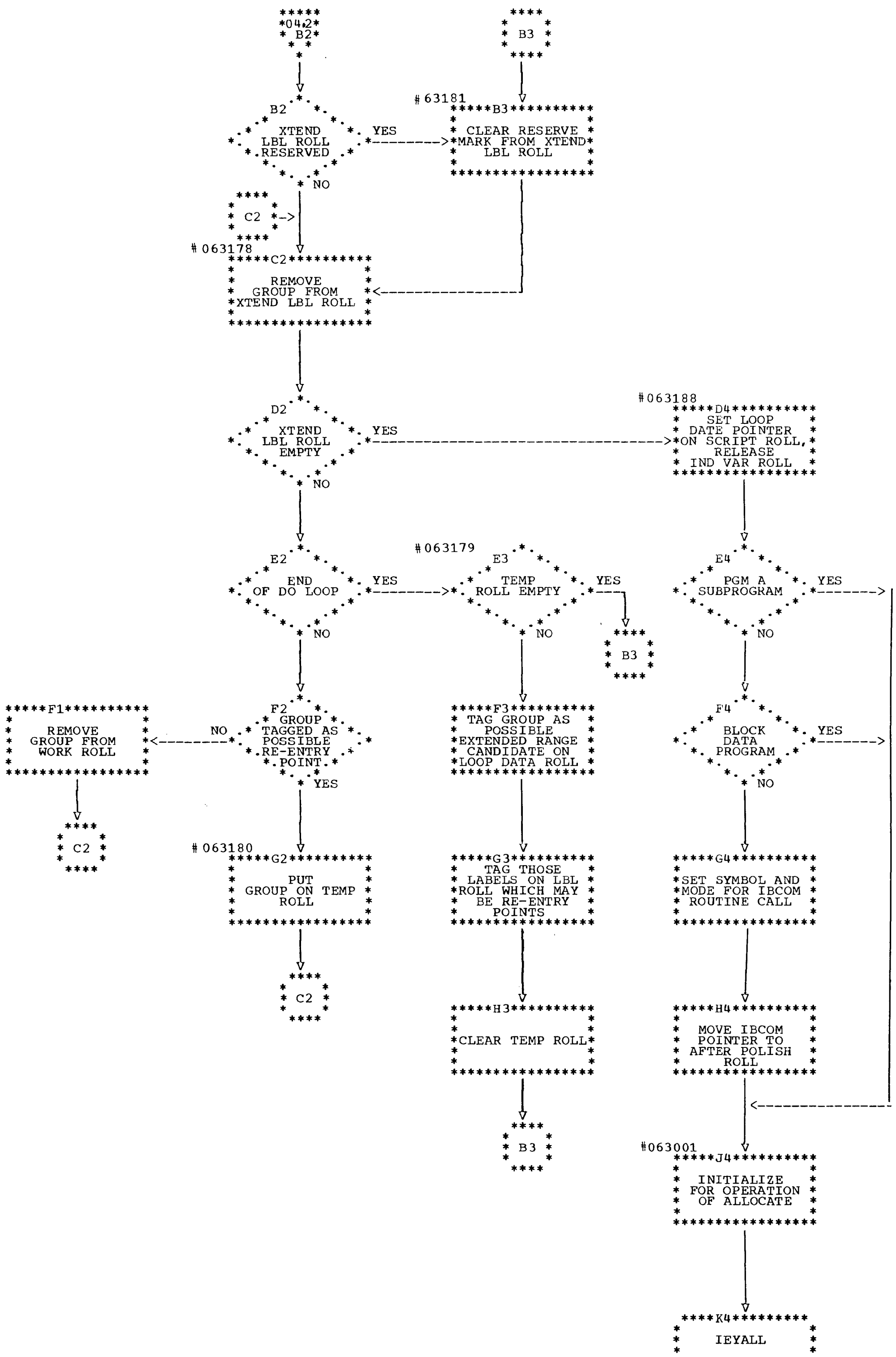


Chart BA. WRITE LISTING AND READ SOURCE

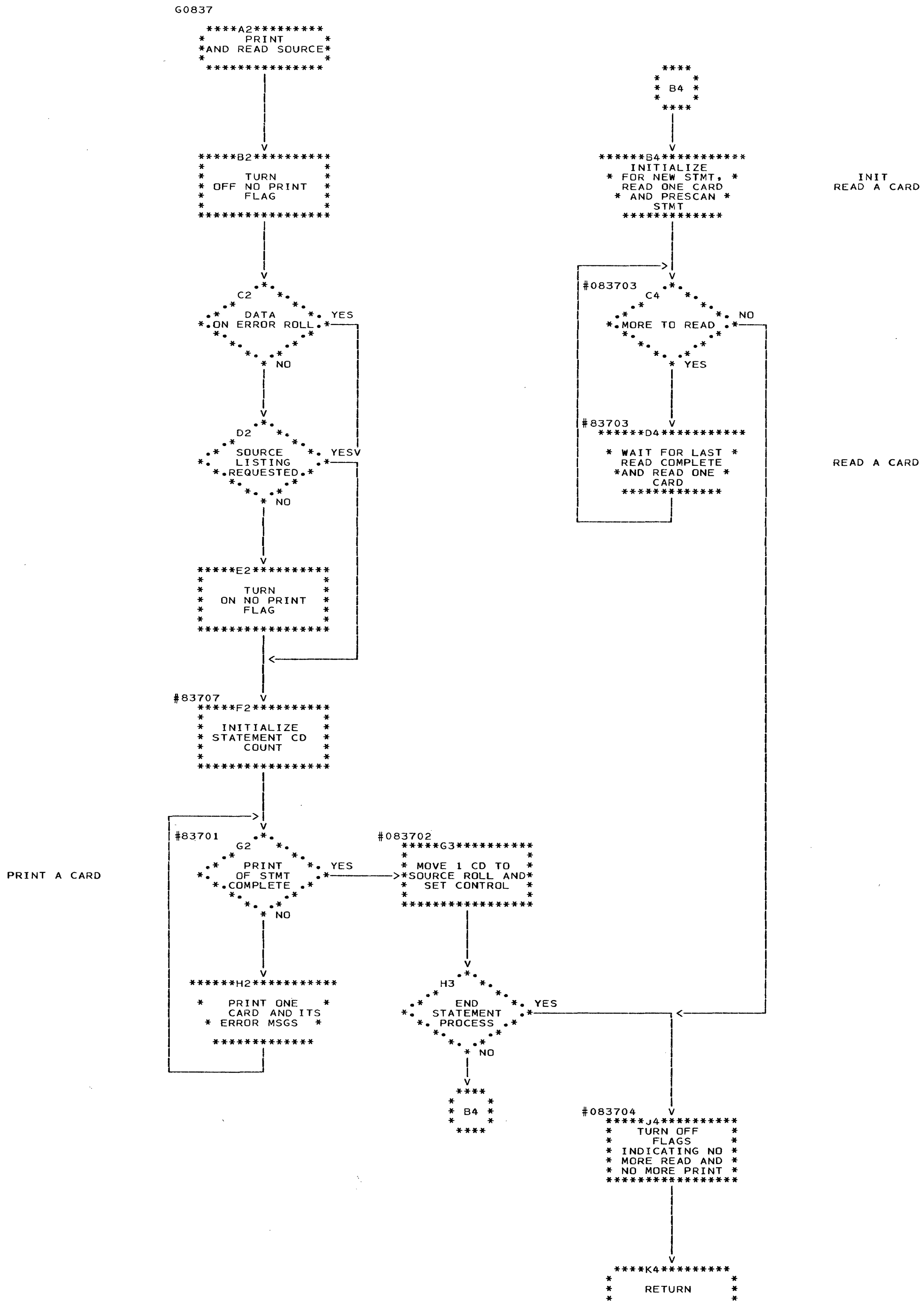
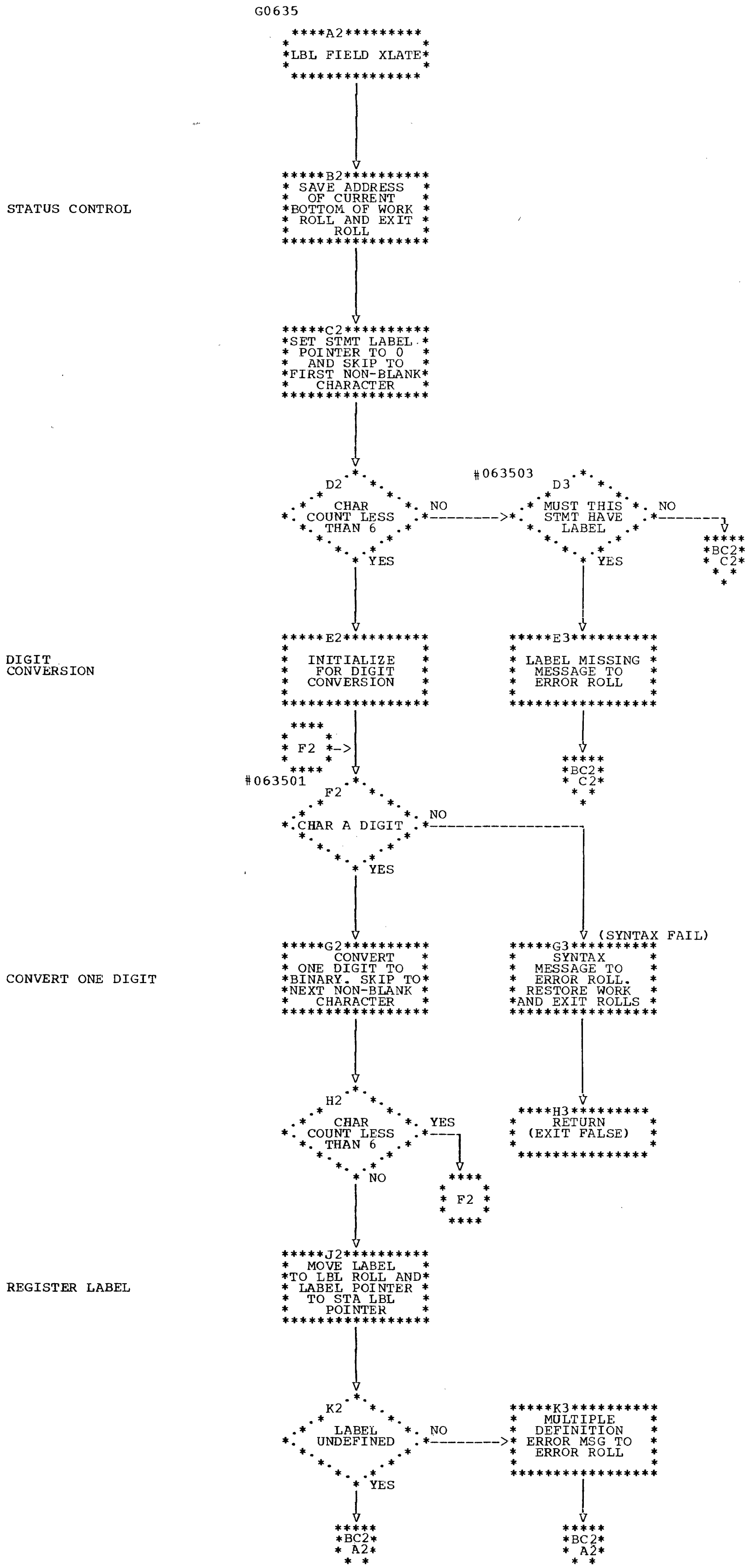


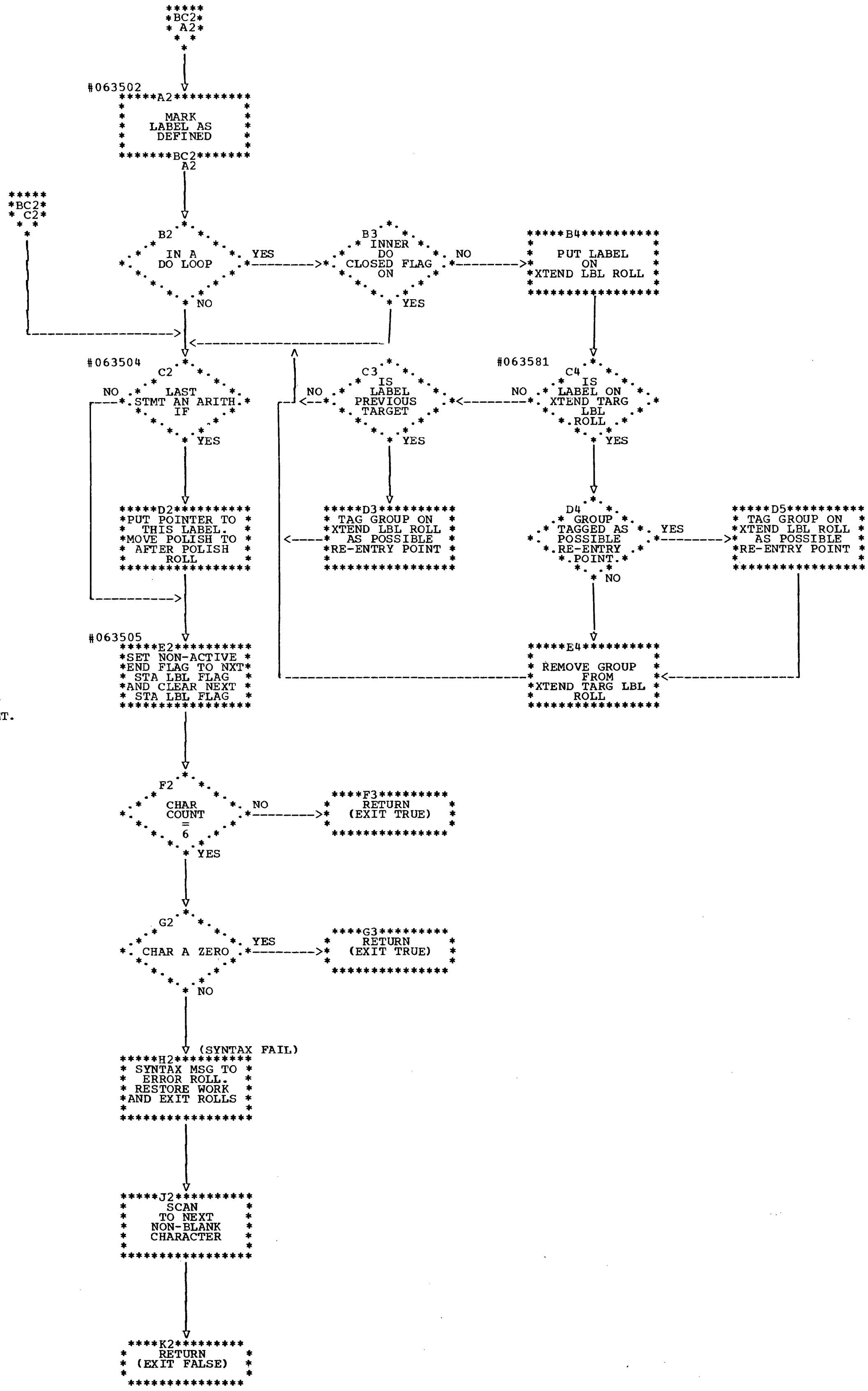
Chart BB. INITIALIZE FOR PROCESSING STATEMENT

```
        G0632
           ****A2*********
           *             *
           *   STA INIT  *
           *             *
           ***************
                  |
                  |
                  V
          *****B2**********
          *               *
          *  INITIALIZE   *
          *   CHARACTER   *
          *    COUNTS     *
          *               *
          *****************
                  |
                  |
                  V
          *****C2**********
          *               *
          *SET CRRNT CHAR *
          *TO FIRST SOURCE*
          *   CHARACTER   *
          *               *
          *****************
                  |
                  |
                  V
          *****D2**********
          *      SET      *
          *COUNT OF SOURCE*
          *STMT CHARACTERS*
          *TO NO. CARDS X *
          *      80       *
          *****************
                  |
                  |
                  V
          *****E2**********
          *               *
          *               *
          *  CLEAR FLAGS  *
          *               *
          *               *
          *****************
                  |
                  |
                  V
           ****F2*********
           *             *
           *   RETURN    *
           *             *
           ***************
```

• Chart BC1. PROCESS LABEL FIELD (Part 1 of 2)



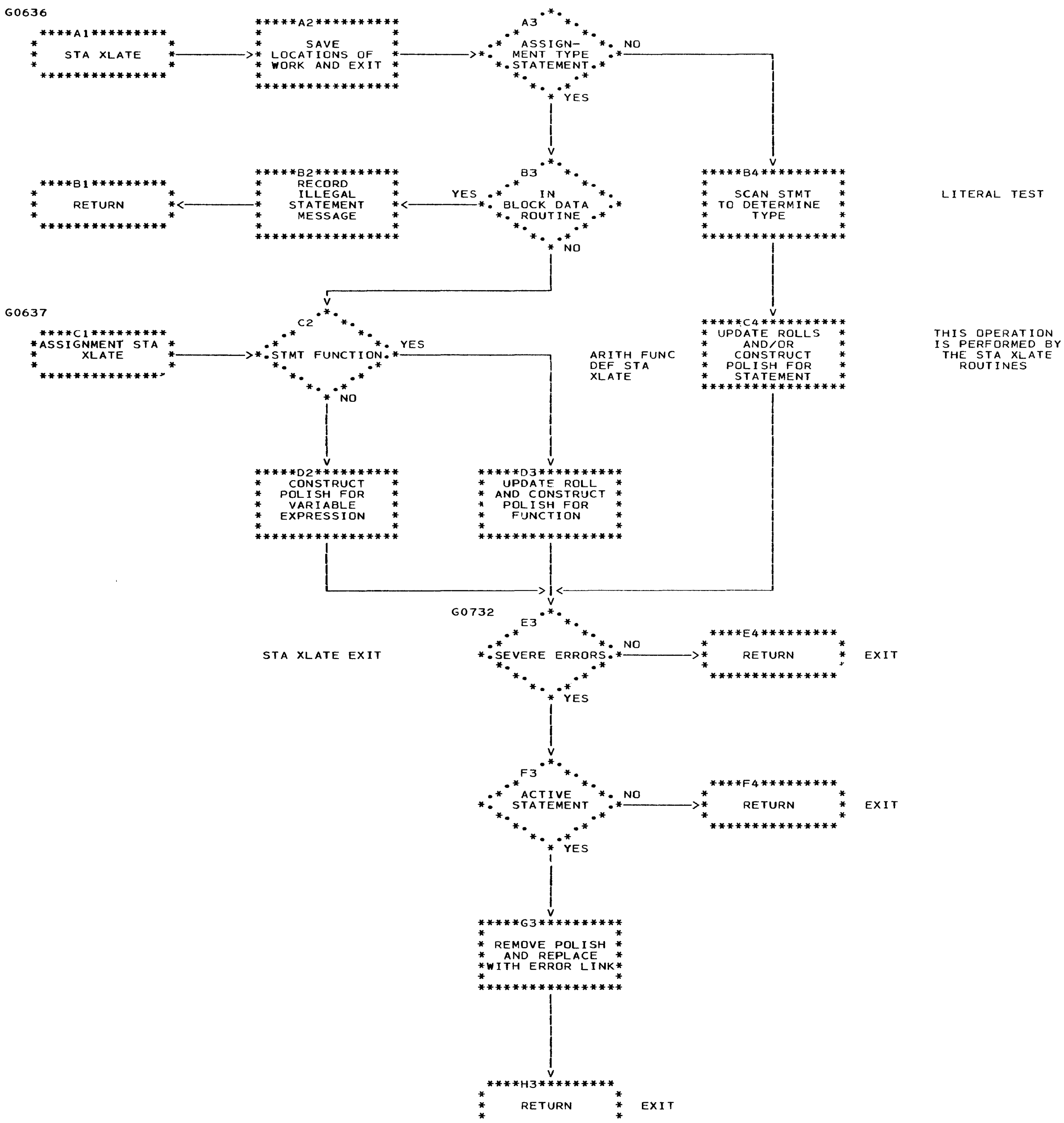
• Chart BC2. PROCESS LABEL FIELD (Part 2 of 2)



PROCESS POLISH

NON-ACTIVE END FLAG INDICATES PREVIOUS STMT ALWAYS BRANCHES WHEN IT IS ON. USED IN TEXT FOR GENERATION OF CODE FOR END STMT.

Chart BD. PROCESS STATEMENT



•Chart BE. COMPLETE STATEMENT AND MOVE POLISH

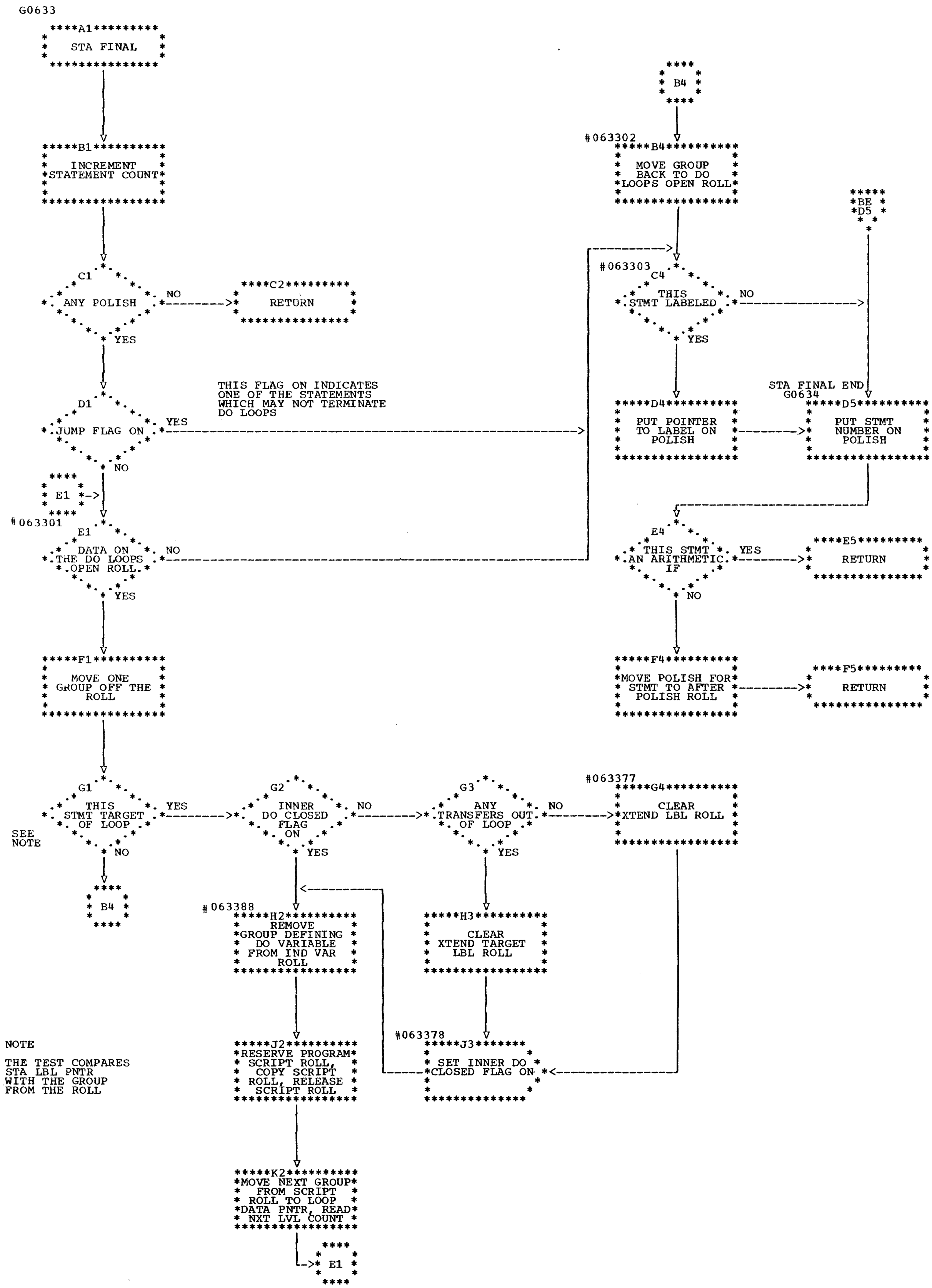


Chart BF. PROCESS END STATEMENT

G0642

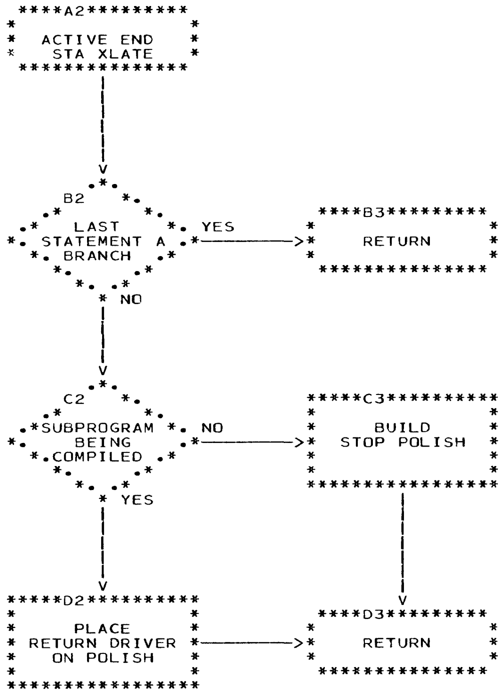
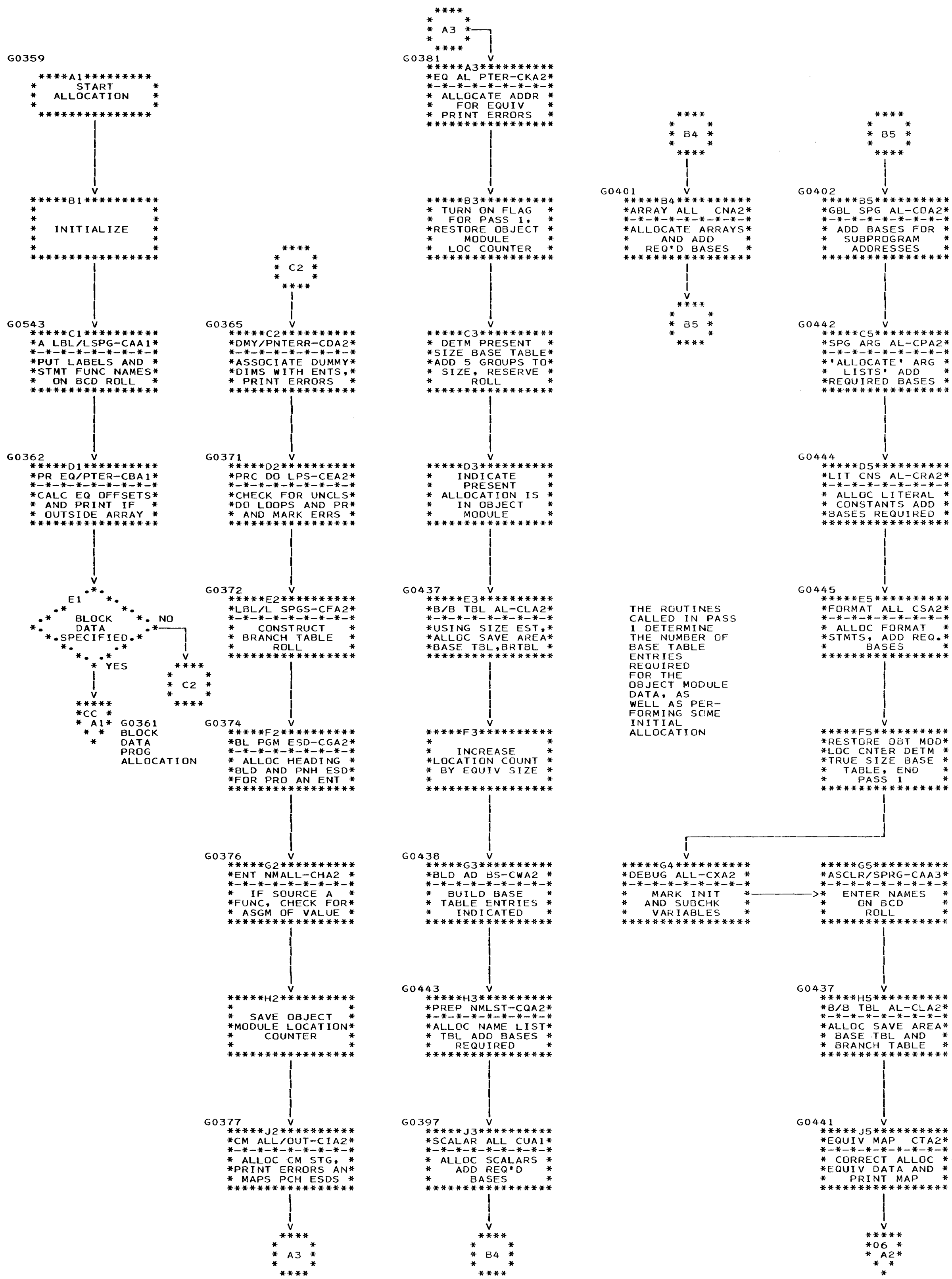


Chart BG. PROCESS POLISH

```
G0844
     ****A2*********
     *             *
     *   PROCESS   *
     *   POLISH    *
     ***************
            |
            |
            |
            |
            v
    *****B2**********
    *               *
    *  GET NUMBER   *
    *  OF WORDS ON  *
    *  POLISH ROLL  *
    *               *
    *****************
            |
            |
            |
            |
            v
    *****C2**********
    *               *
    *     PLACE     *
    *COUNT ON AFTER *
    *  POLISH ROLL  *
    *               *
    *****************
            |
            |
            |
            |
            v
    *****D2**********
    *               *
    *  COPY POLISH  *
    * ROLL TO AFTER *
    *  POLISH ROLL  *
    *               *
    *****************
            |
            |
            |
            |
            v
    *****E2**********
    *               *
    *               *
    *RELEASE POLISH *
    *     ROLL      *
    *               *
    *****************
            |
            |
            |
            |
            v
     ****F2*********
    *               *
    *    RETURN     *
    *               *
     ***************
```

Chart 05. PHASE 2 - ALLOCATE (Part 1 of 2)



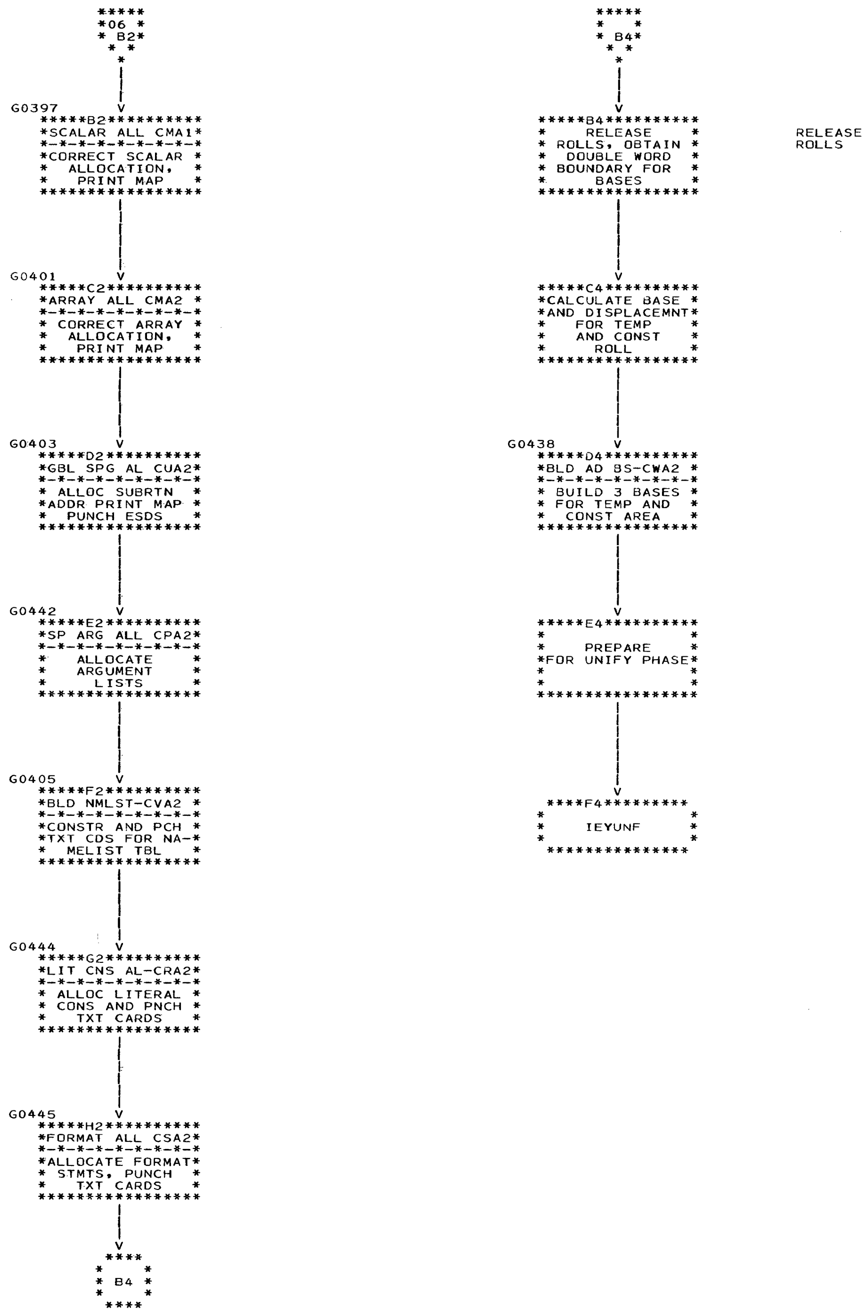

06 B2
G0397
B2
SCALAR ALL CMA1
CORRECT SCALAR ALLOCATION, PRINT MAP
G0401
C2
ARRAY ALL CMA2
CORRECT ARRAY ALLOCATION, PRINT MAP
G0403
D2
GBL SPG AL CUA2
ALLOC SUBRTN ADDR PRINT MAP PUNCH ESDS
G0442
E2
SP ARG ALL CPA2
ALLOCATE ARGUMENT LISTS
G0405
F2
BLD NMLST-CVA2
CONSTR AND PCH TXT CDS FOR NA-MELIST TBL
G0444
G2
LIT CNS AL-CRA2
ALLOC LITERAL CONS AND PNCH TXT CARDS
G0445
H2
FORMAT ALL CSA2
ALLOCATE FORMAT STMTS, PUNCH TXT CARDS
B4
B4
B4
RELEASE ROLLS, OBTAIN DOUBLE WORD BOUNDARY FOR BASES
RELEASE ROLLS
C4
CALCULATE BASE AND DISPLACEMNT FOR TEMP AND CONST ROLL
G0438
D4
BLD AD BS-CWA2
BUILD 3 BASES FOR TEMP AND CONST AREA
E4
PREPARE FOR UNIFY PHASE
F4
IEYUNF

Chart CA. MOVE BLD NAMES TO DATA VAR ROLL

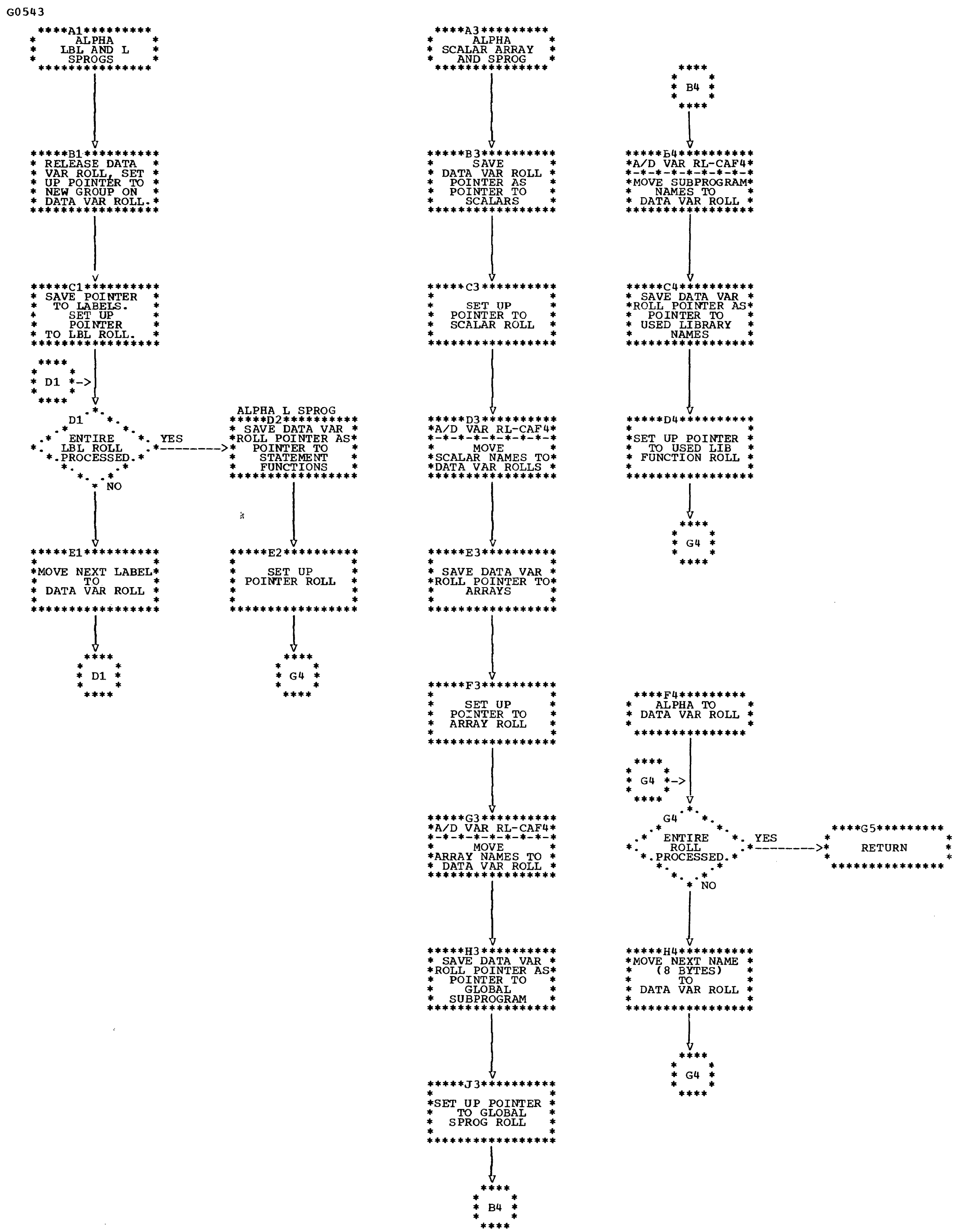


Chart CB. PREPARE EQUIVALENCE DATA

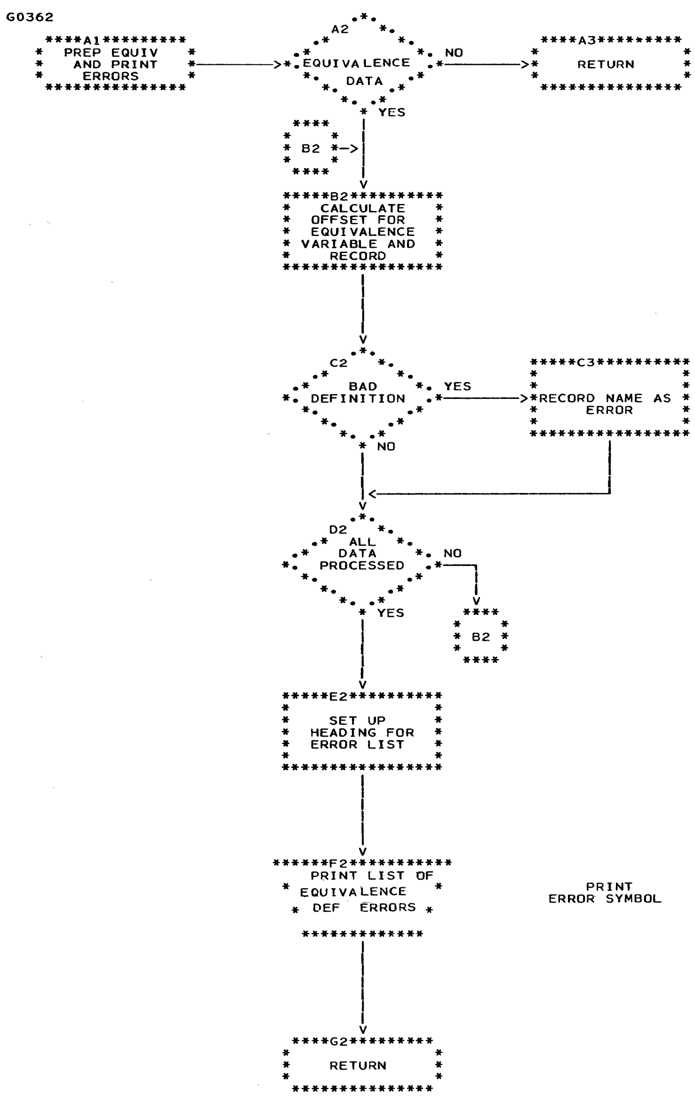


Chart CC. ALLOCATE BLOCK DATA

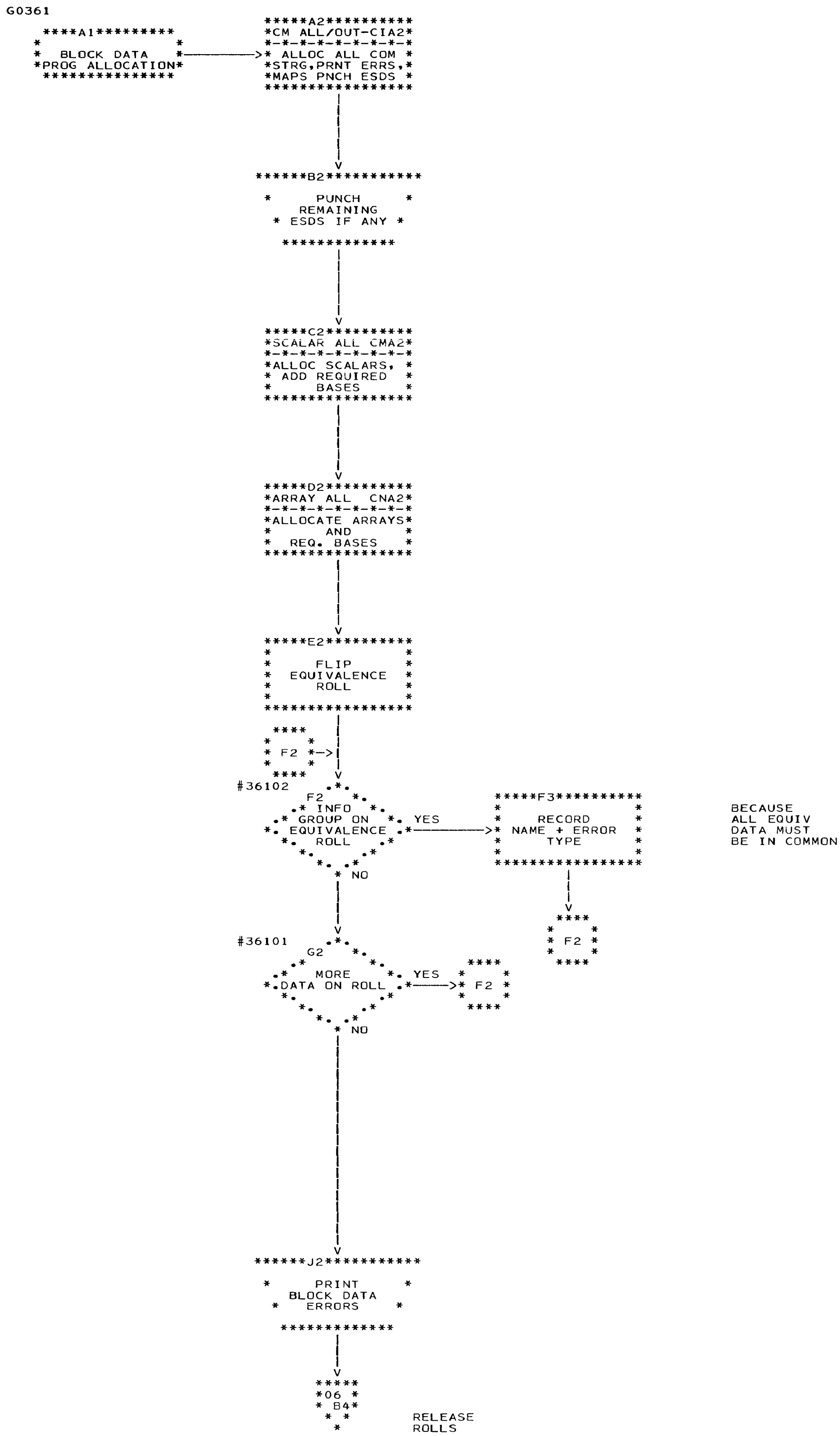


Chart CD. PREPROCESS DUMMY DIMENSIONS

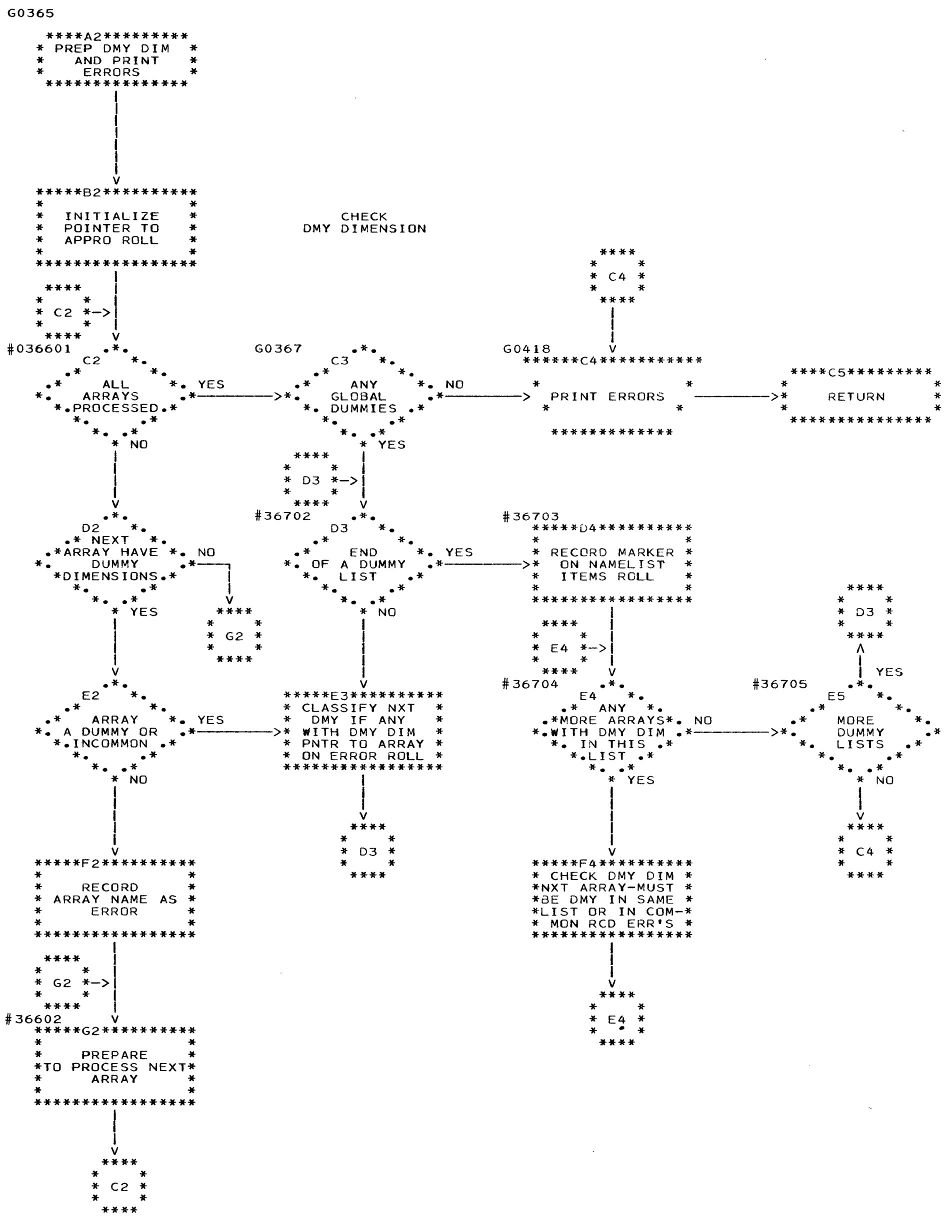


Chart CE. CHECK FOR UNCLOSED DO LOOPS

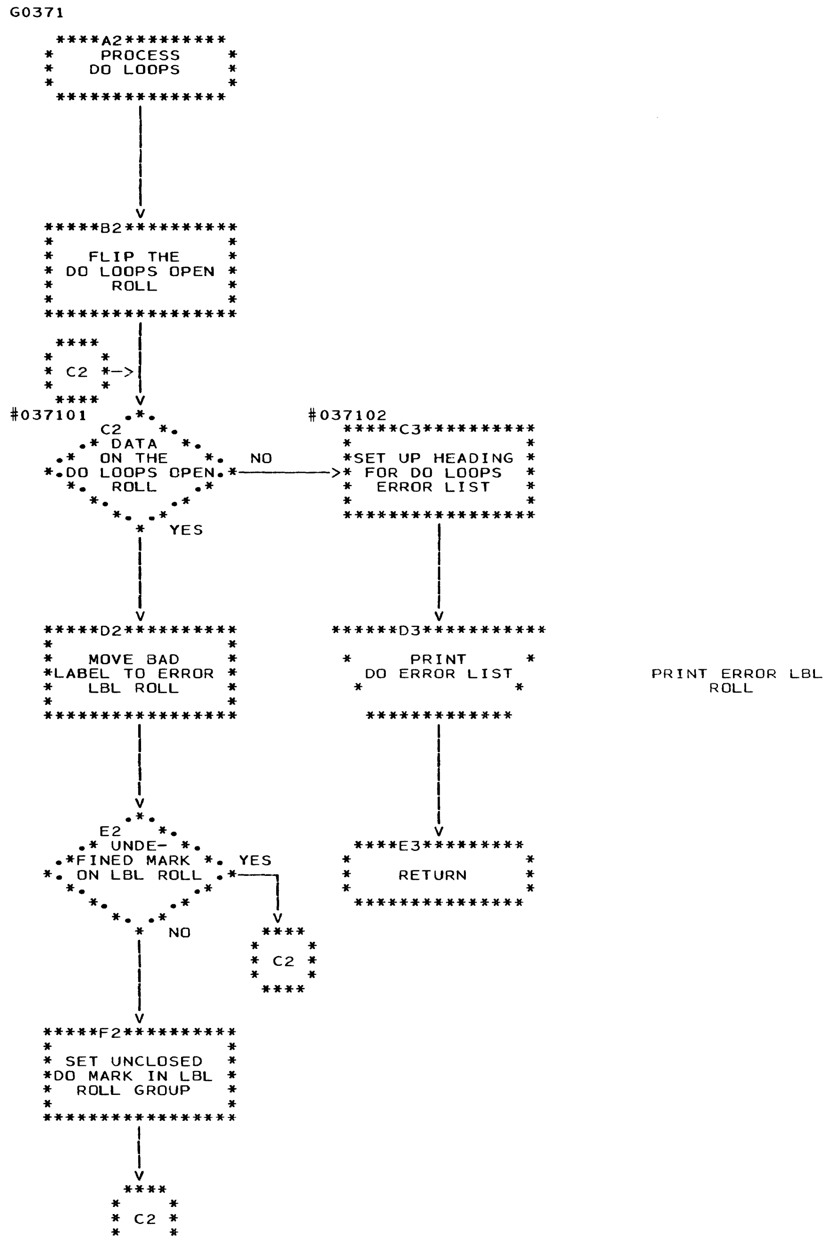


Chart CF. CONSTRUCT BRANCH TABLE ROLL

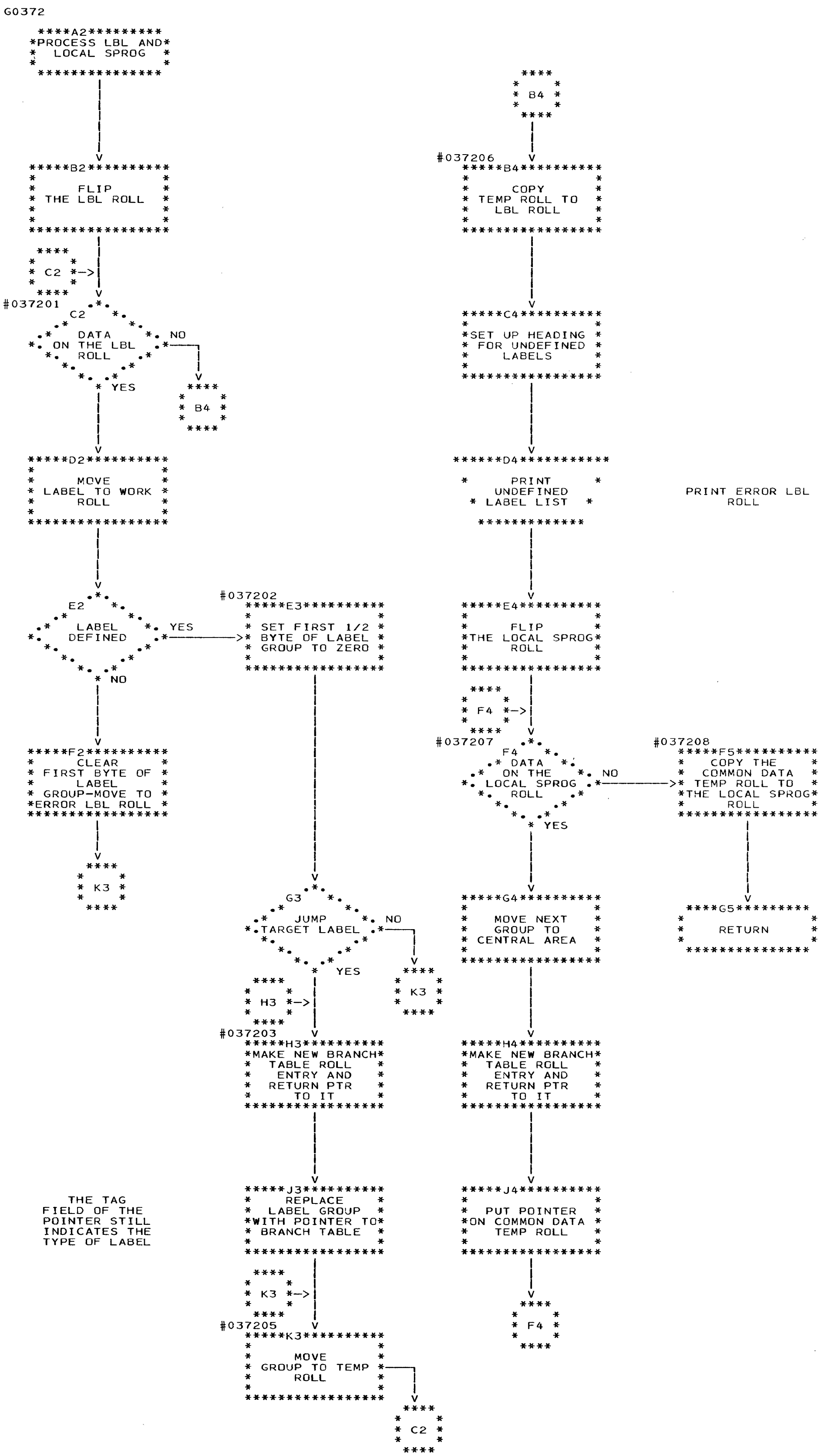


Chart CG. ALLOCATE HEADING AND PUNCH ESD CARDS

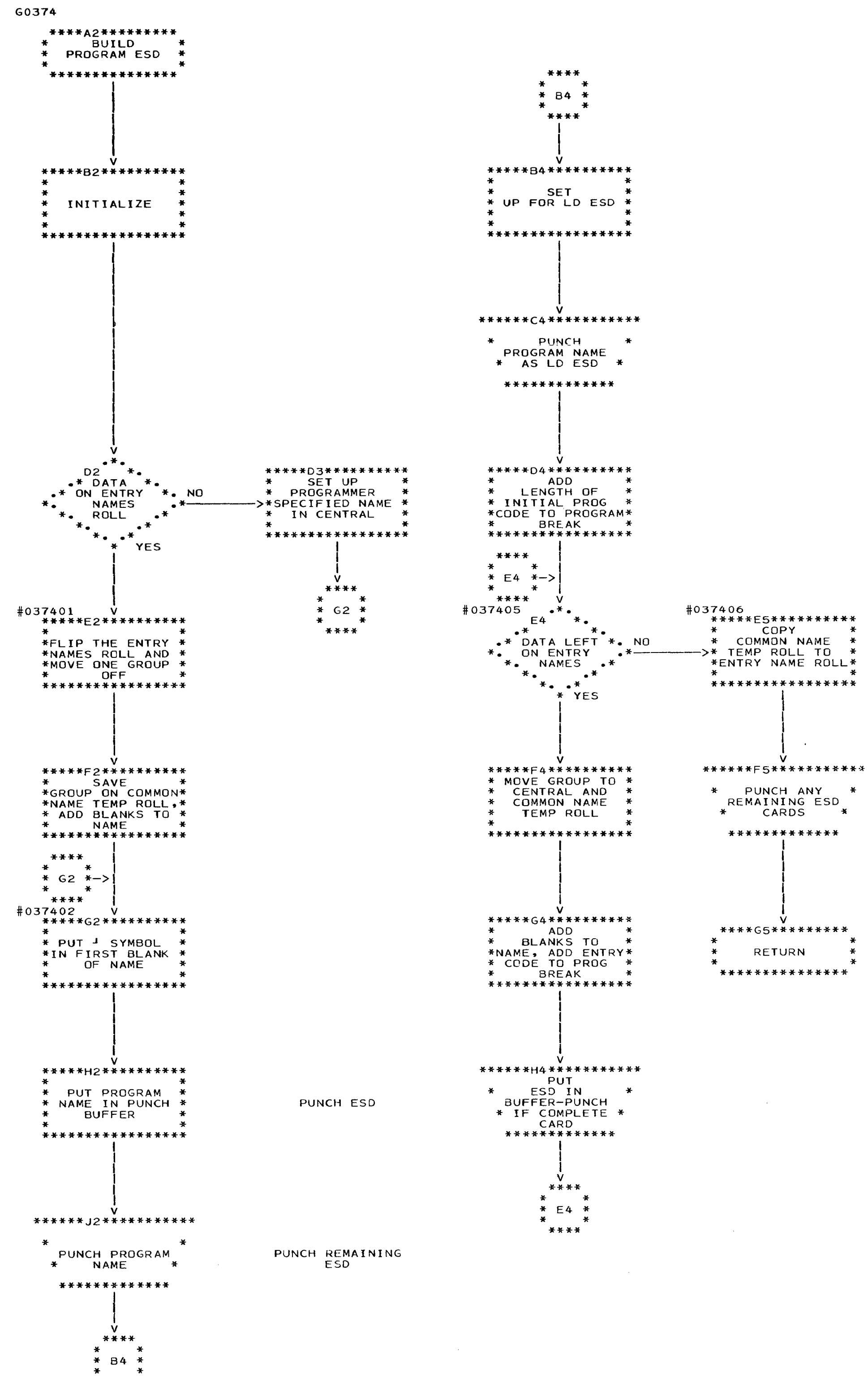


Chart CH. CHECK ASSIGNMENT OF FUNCTION VALUE

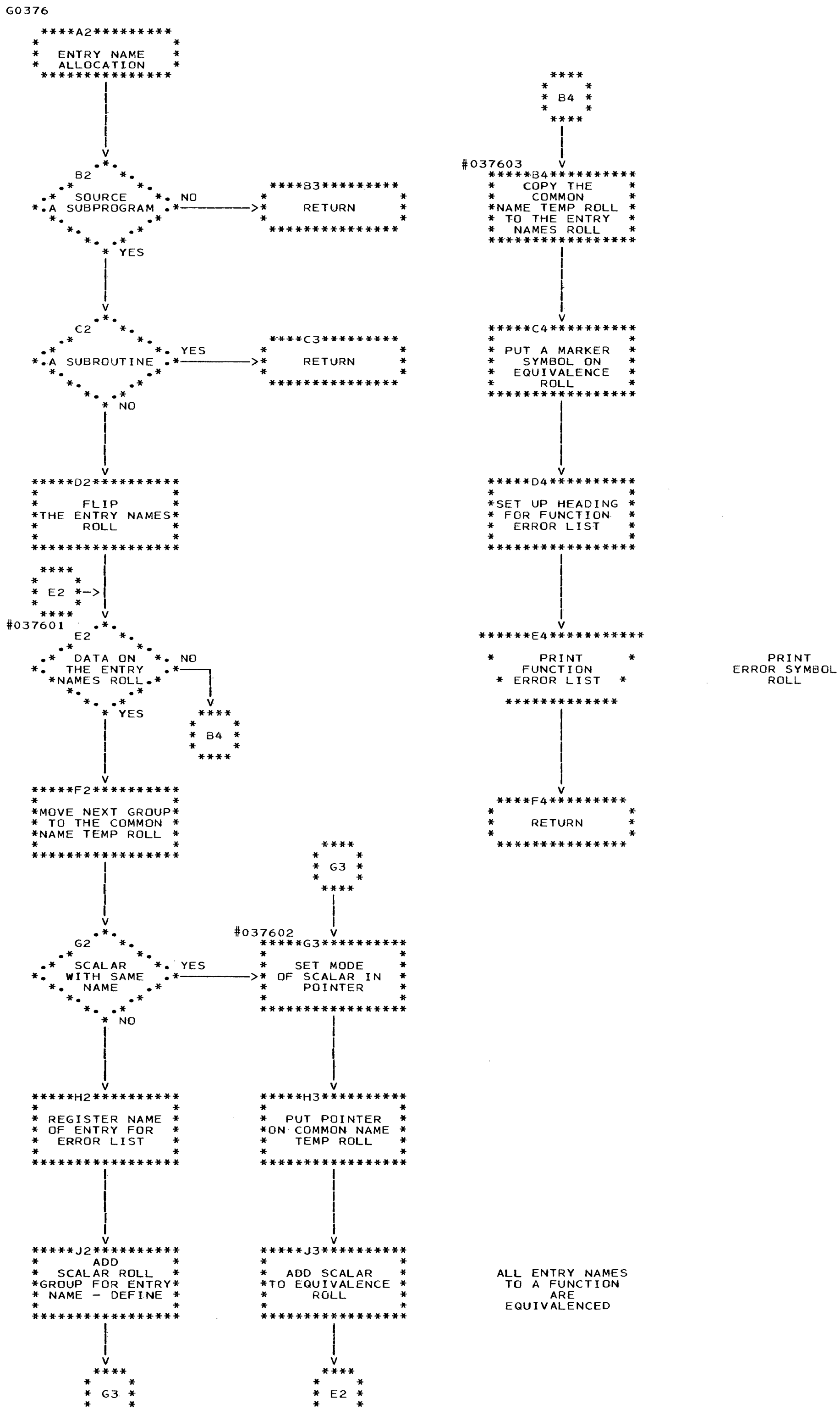


Chart CI. COMMON ALLOCATION

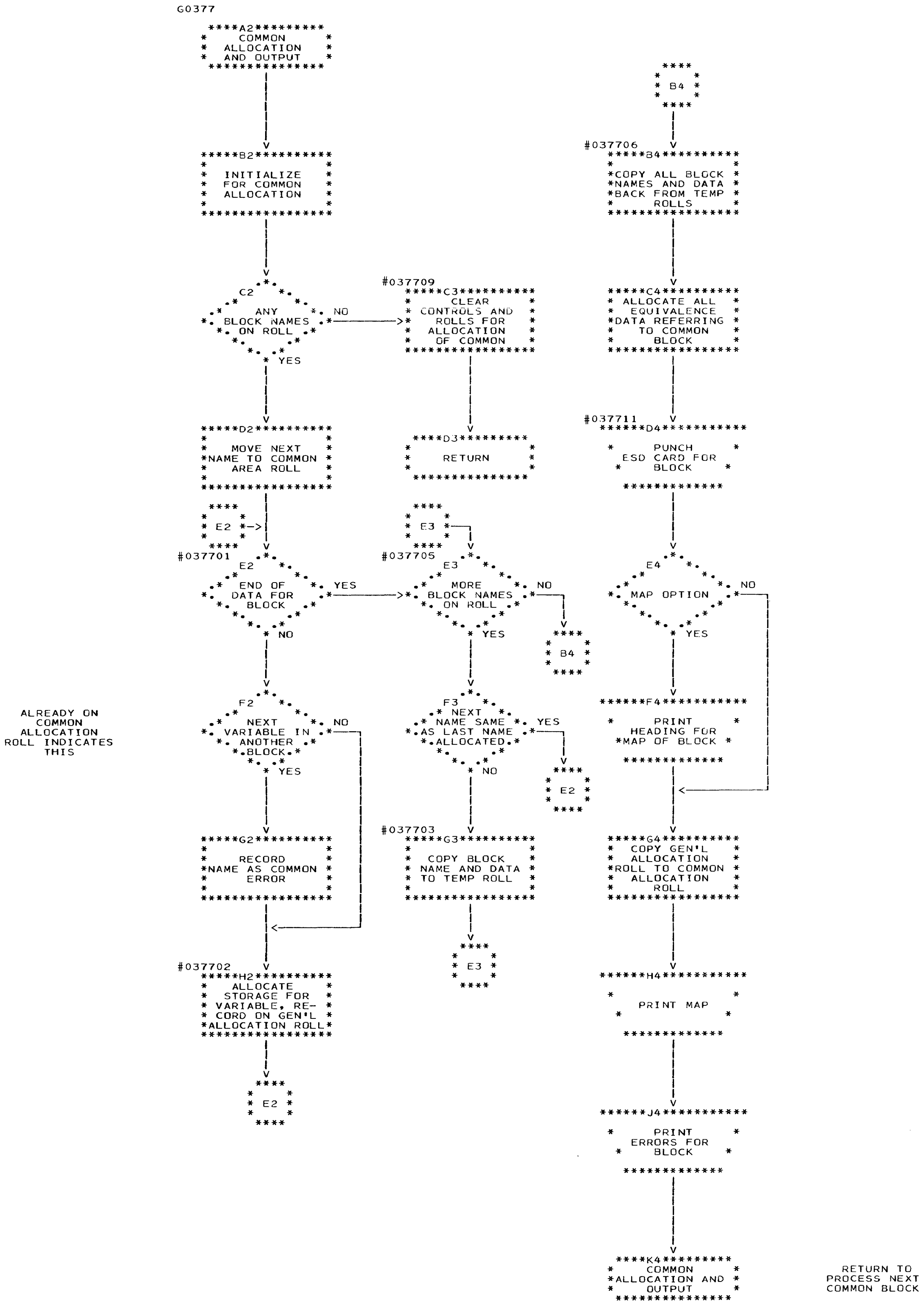


Chart CK. EQUIVALENCE DATA ALLOCATION

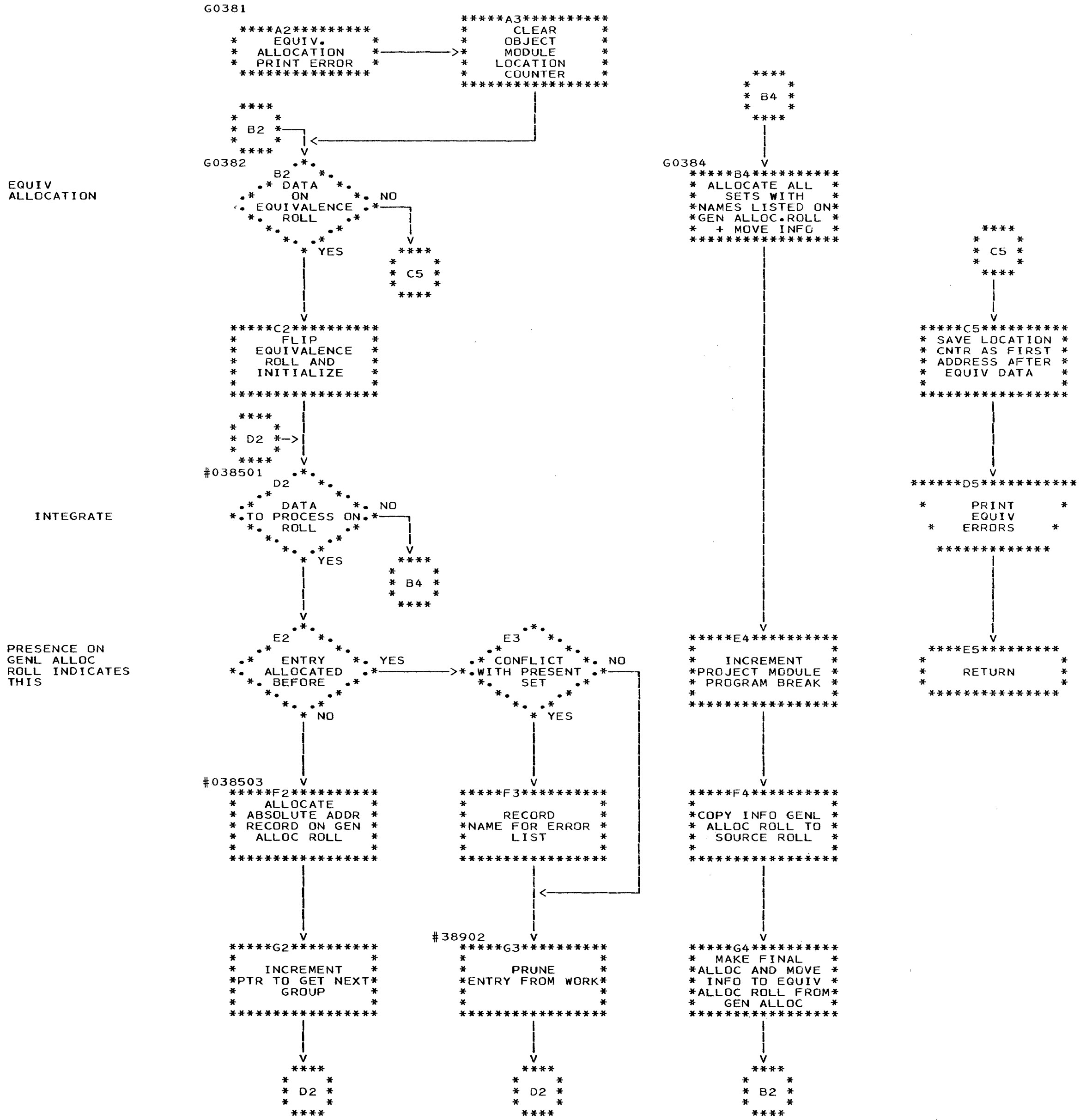


Chart CL. SAVE AREA, BASE AND BRANCH TABLE ALLOCATION

```
G0437
 ****A2*********
 *   BASE AND   *
 * BRANCH TABLE *
 *  ALLOCATION  *
 ***************
        |
        v
*****B2**********
*SAVE BASE TBL. *
*    PTR AND    *
* DISPLACEMENT  *
* FOR START OF  *
*   SAVE AREA   *
*****************
        |
        v
*****C2**********             THIS VARIABLE
*               *             IS USED
*   INCREASE    *             TO HOLD OBJECT
* PROGRAM BREAK *             MODULE ADDRESSES
* BY SAVE AREA  *             BEING ALLOC.
*     SIZE      *
*****************
        |
        v
*****D2**********
* SAVE BASE TBL *
*PTR AND DISPLA-*
*  CEMENT FOR   *
* START OF BASE *
*     TABLE     *
*****************
        |
        v
*****E2**********
*               *
*   INCREASE    *
* PROGRAM BREAK *
* BY BASE TABLE *
*     SIZE      *
*****************
        |
        v
*****F2**********
*               *              BUILD
*   CONSTRUCT   *              ADDITIONAL
* REQUIRED BASE *              BASES
* TABLE ENTRIES *
*               *
*****************
        |
        v
*****G2**********
* SAVE BASE TBL *
*PTR DISPLACEMT *
* FOR START OF  *
* BRANCH TABLE  *
*               *
*****************
        |
        v
*****H2**********
*INCREASE PROG. *
*   BREAK BY    *
*  SIZE BRANCH  *
*TABLE AND MAKE *
* LABEL ENTRIES *
*****************
        |
        v
*****J2**********
*               *              BUILD
*   CONSTRUCT   *              ADDITIONAL
* REQUIRED BASE *              BASES
* TABLE ENTRIES *
*               *
*****************
        |
        v
 ****K2*********
*               *
*    RETURN     *
*               *
 ***************
```

Chart CM. ALLOCATE SCALARS

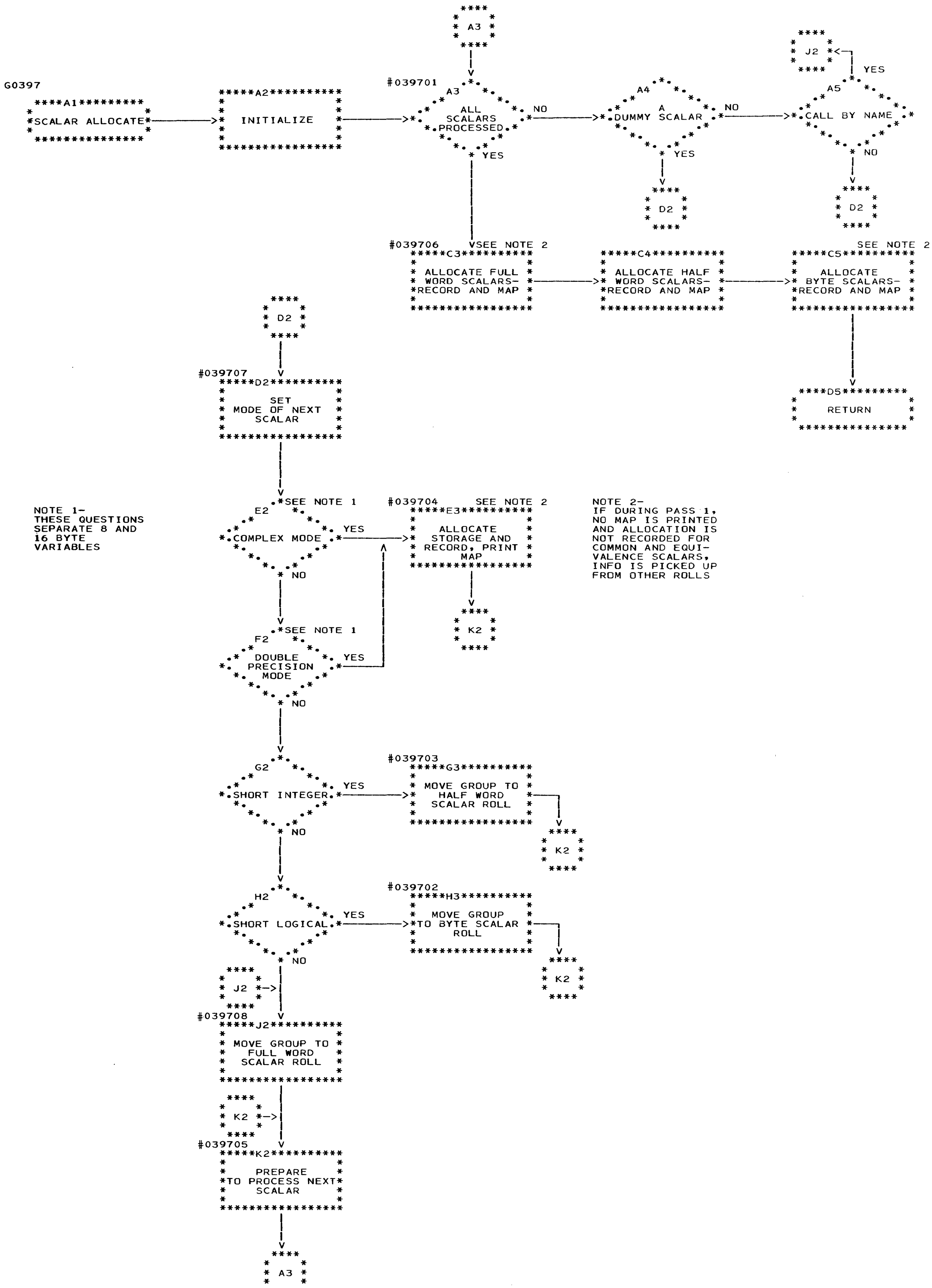


Chart CN. ALLOCATE ARRAYS

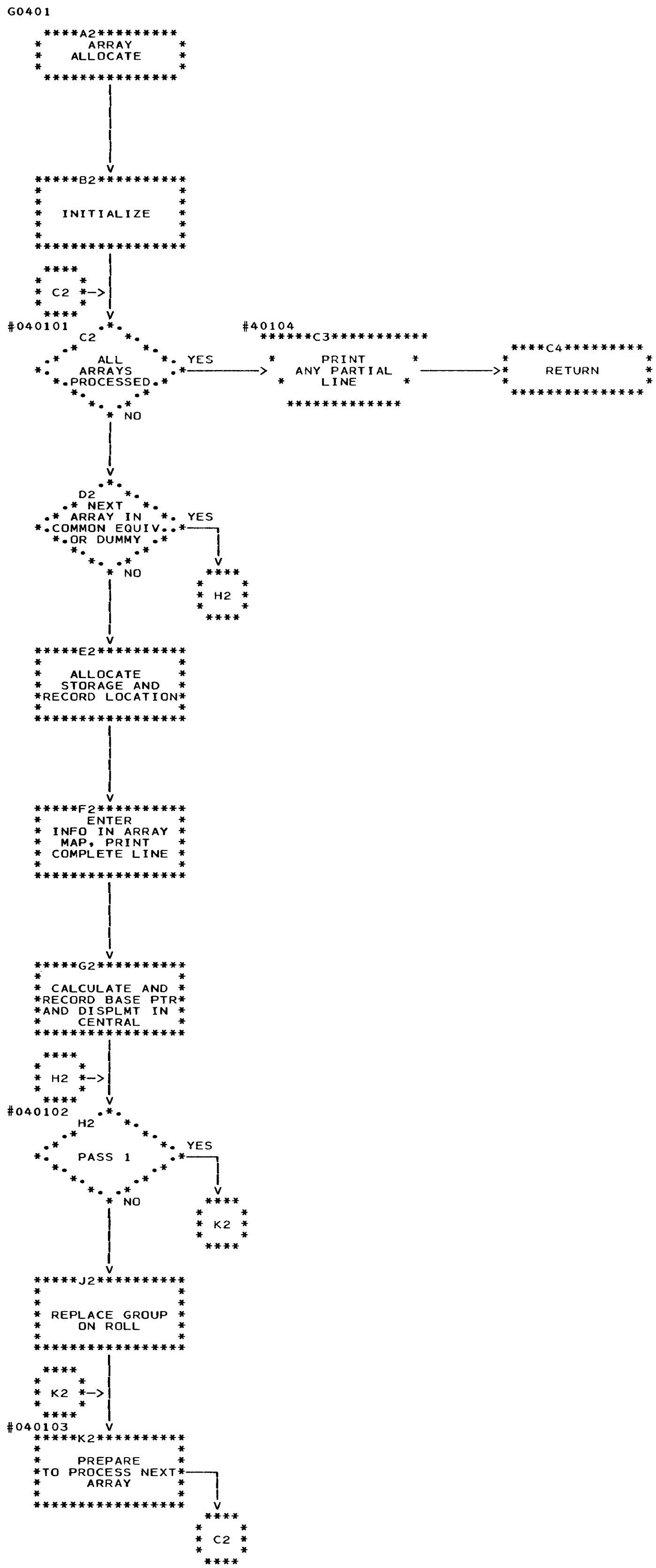


Chart CO. ADD BASES FOR SUBPROGRAM ADDRESSES

```
G0402

 ****A2*********
 * PASS 1 GLOBAL *
 *SPROG ALLOCATE *
 *               *
  ***************
         |
         |
         |
         |
         |
         V
*****B2**********
*               *
*    ALIGN TO   *
*   FULL WORD   *
*   BOUNDARY    *
*               *
*****************
         |
         |
         |
         |
         V
*****C2**********
*DETERMINE BASE *
*    PTR AND    *
* DISPLACEMENT  *
*FOR PRESENT LOC*
*               *
*****************
         |
         |
         |
         |
         V
*****D2**********
*    COMPUTE    *
*   LENGTH OF   *
* OBJECT MODULE *
*SUBPROGRAM ADR *
*               *
*****************
         |
         |
         |
         |
         V
*****E2**********
*COMPUTE LENGTH *          BUILD
*   OF OBJECT   *          ADDITIONAL
*    MODULE     *          BASES
*  SUBPROGRAM   *
*     ADDR      *
*****************
         |
         |
         |
         |
         V
 ****F2*********
*               *
*    RETURN     *
*               *
 ***************
```

Chart CP. ALLOCATE SUBPROGRAM ARGUMENT LISTS

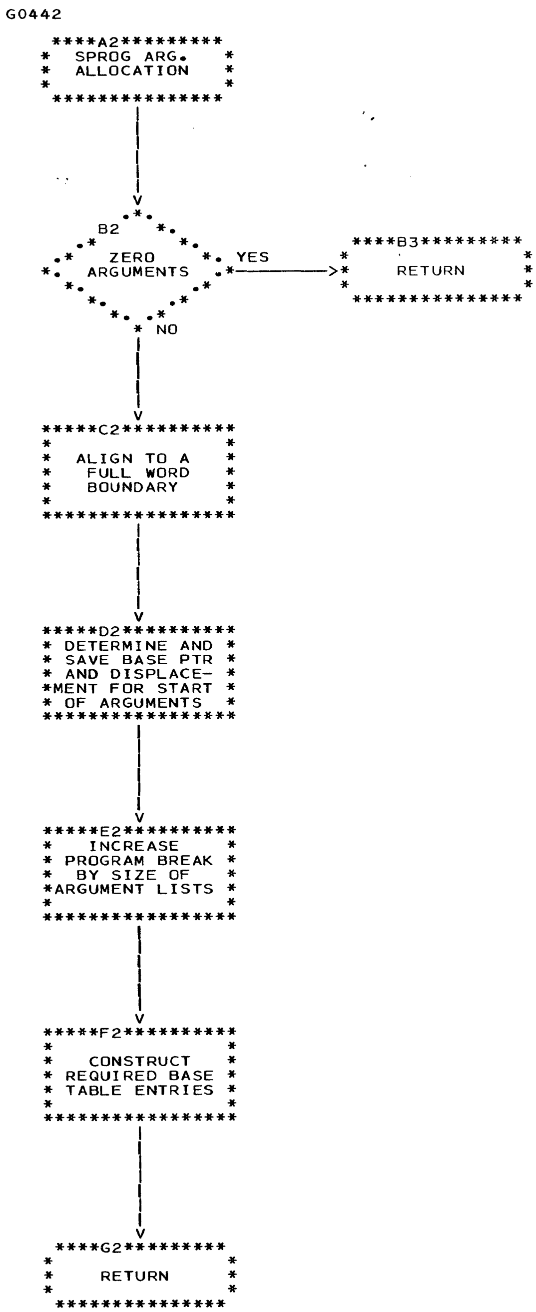


Chart CQ. PREPARE NAMELIST TABLES

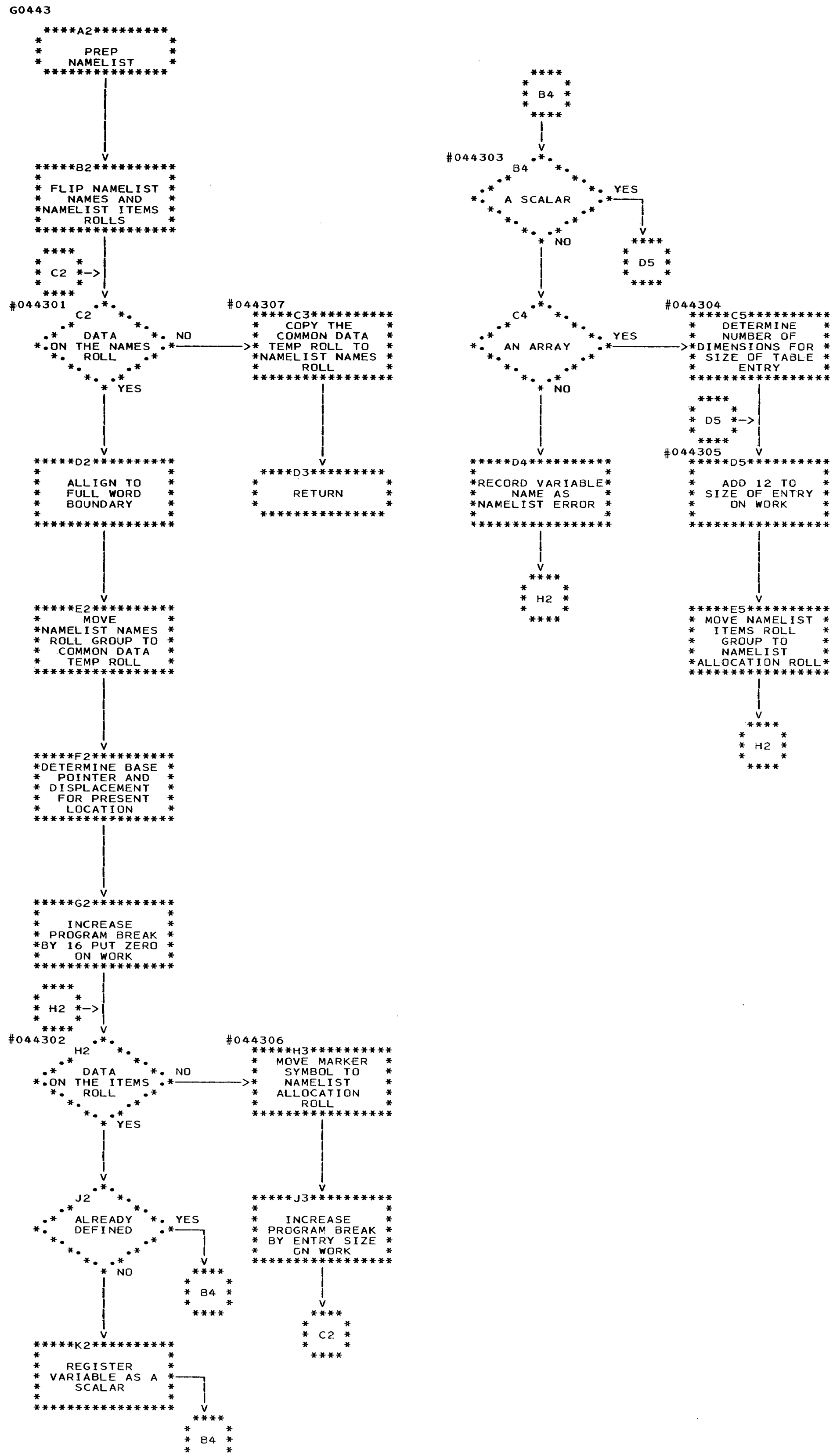


Chart CR. ALLOCATE LITERAL CONSTANTS

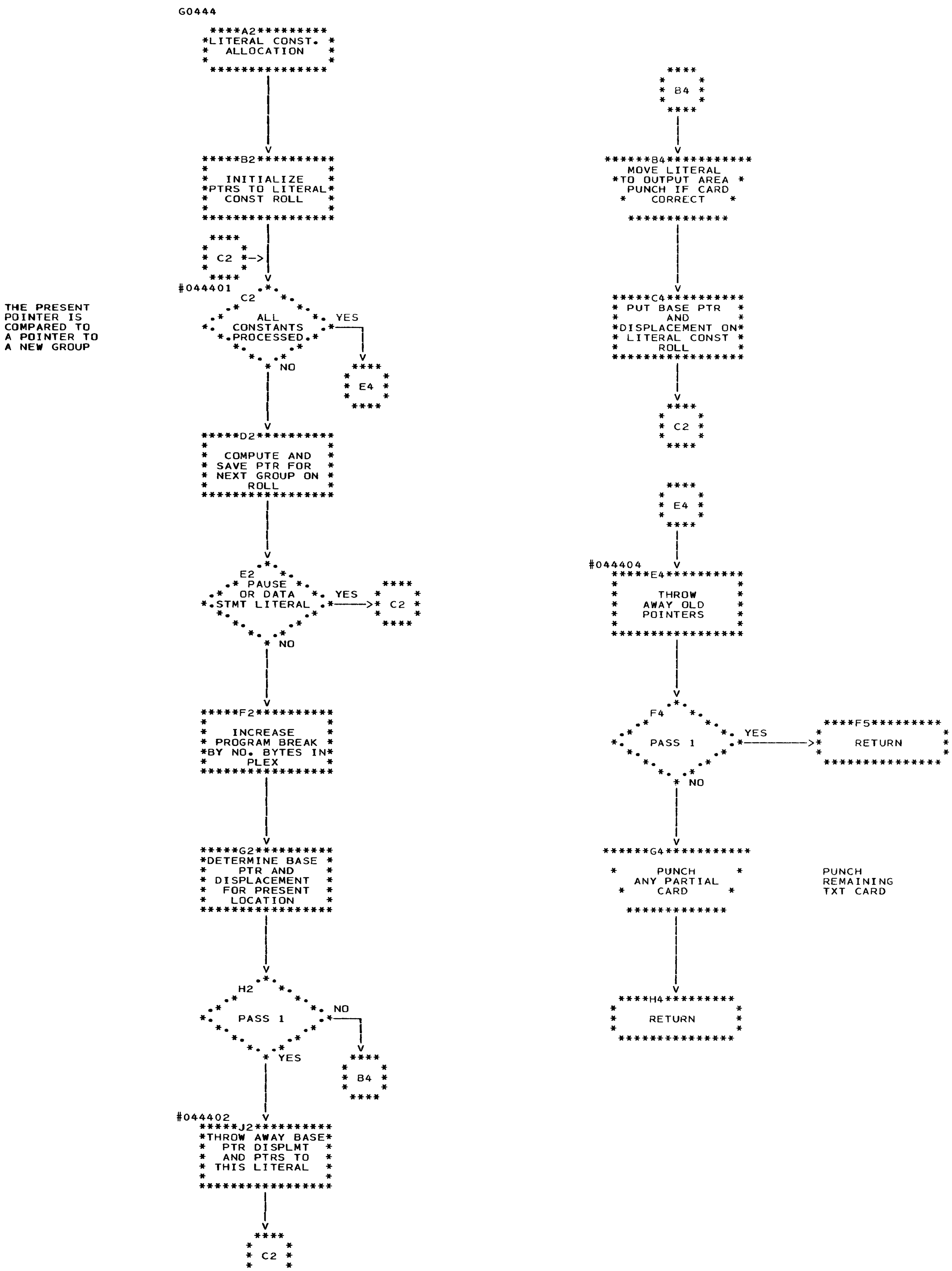


Chart CS. ALLOCATE FORMATS

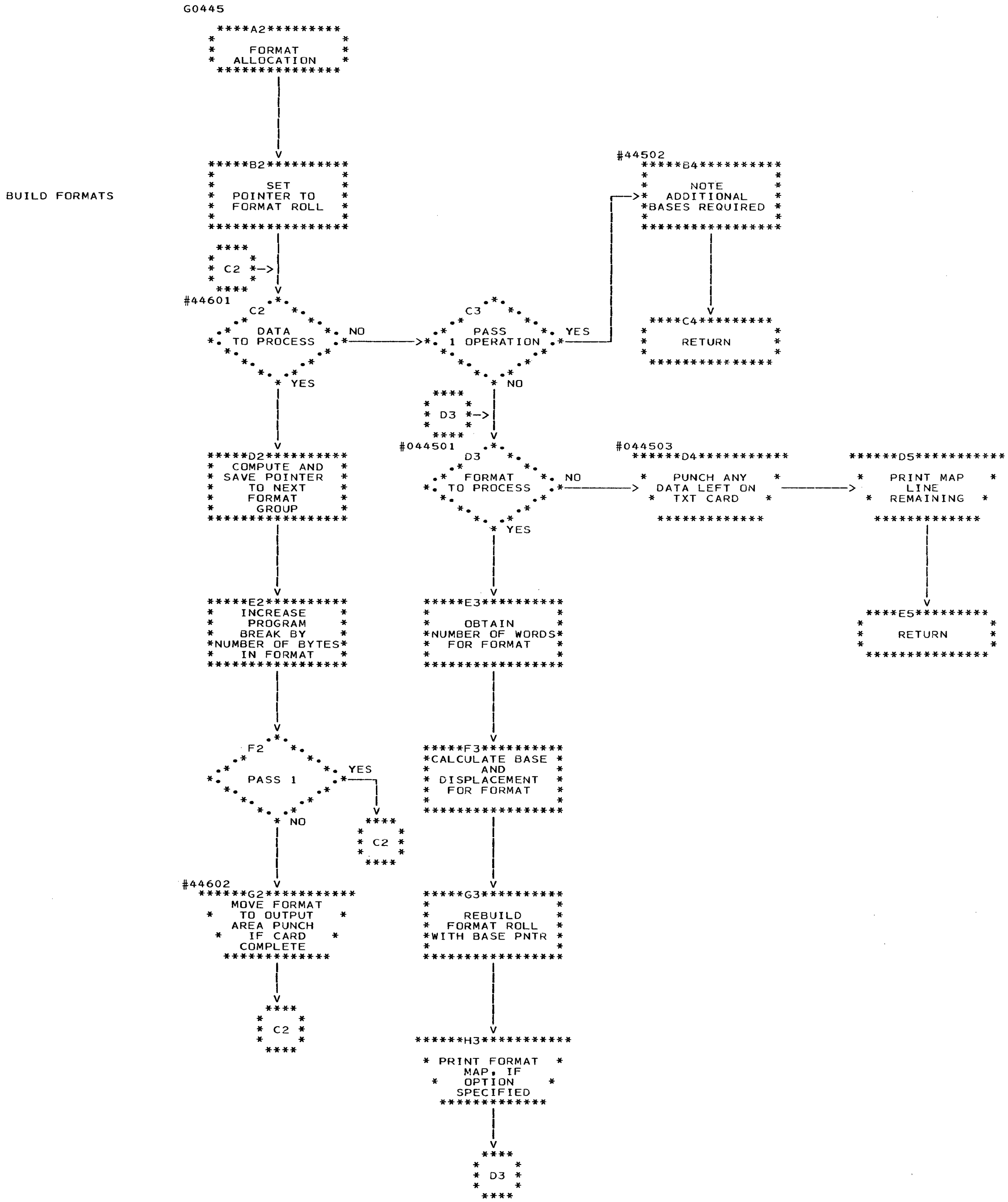


Chart CT. MAP EQUIVALENCE

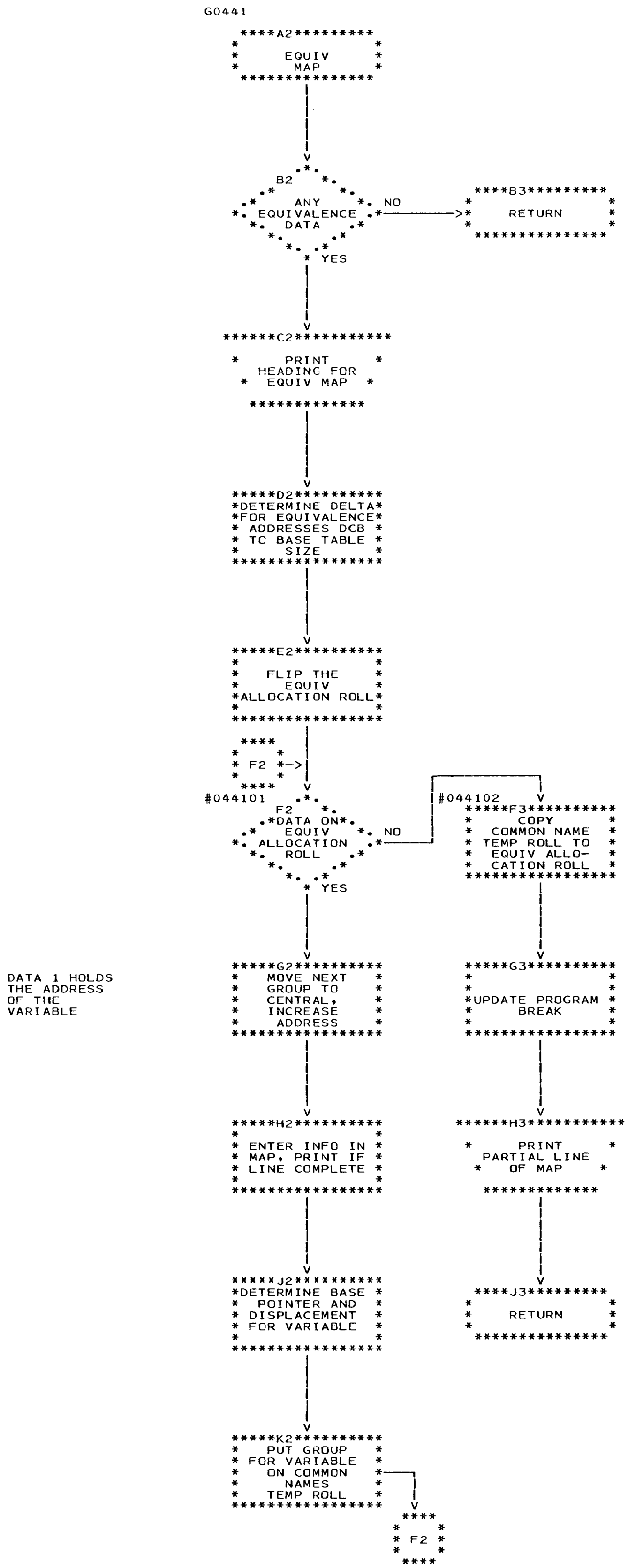


Chart CU. ALLOCATE SUBPROGRAM ADDRESSES

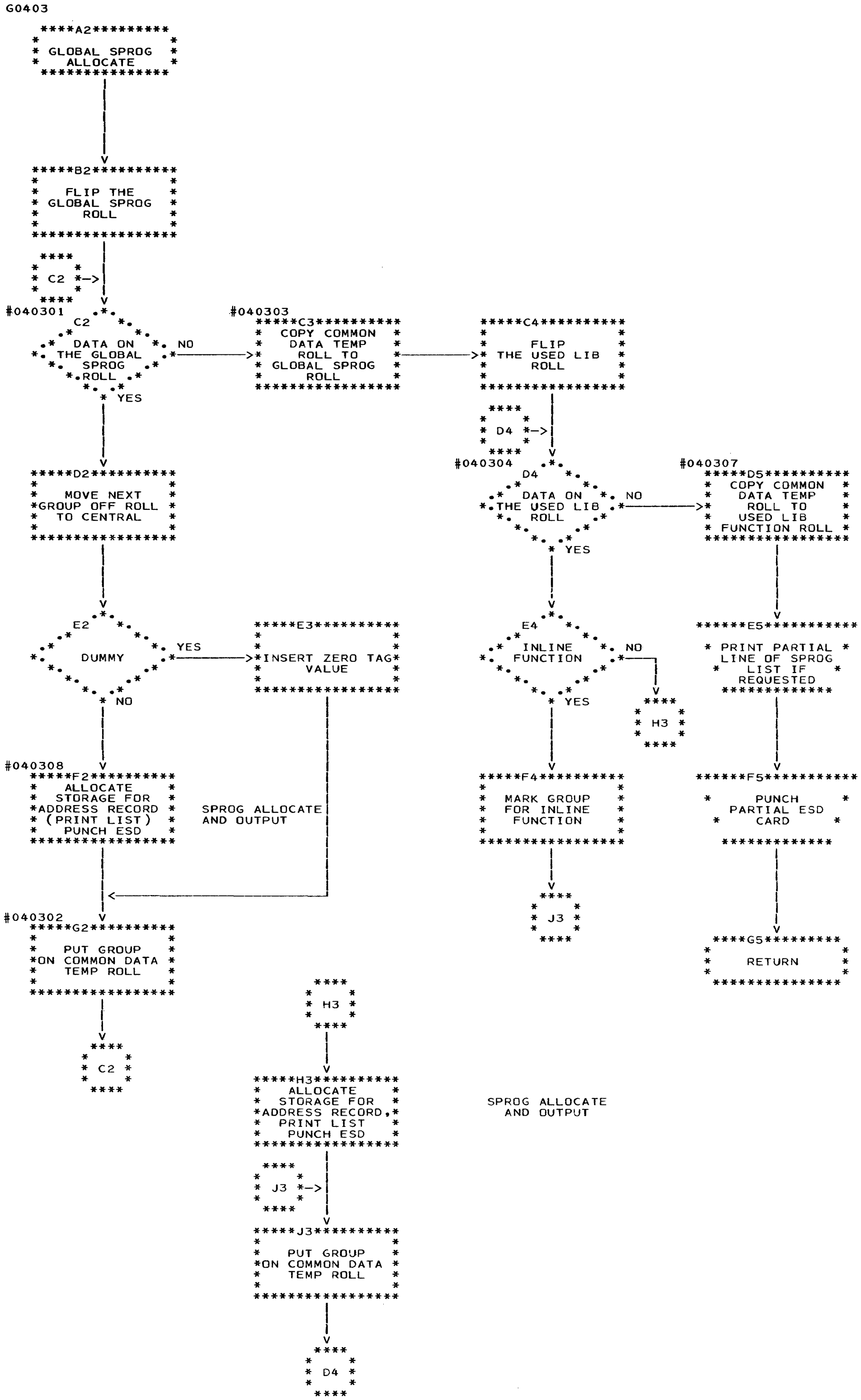


Chart CV. BUILD AND PUNCH NAMELIST TABLES

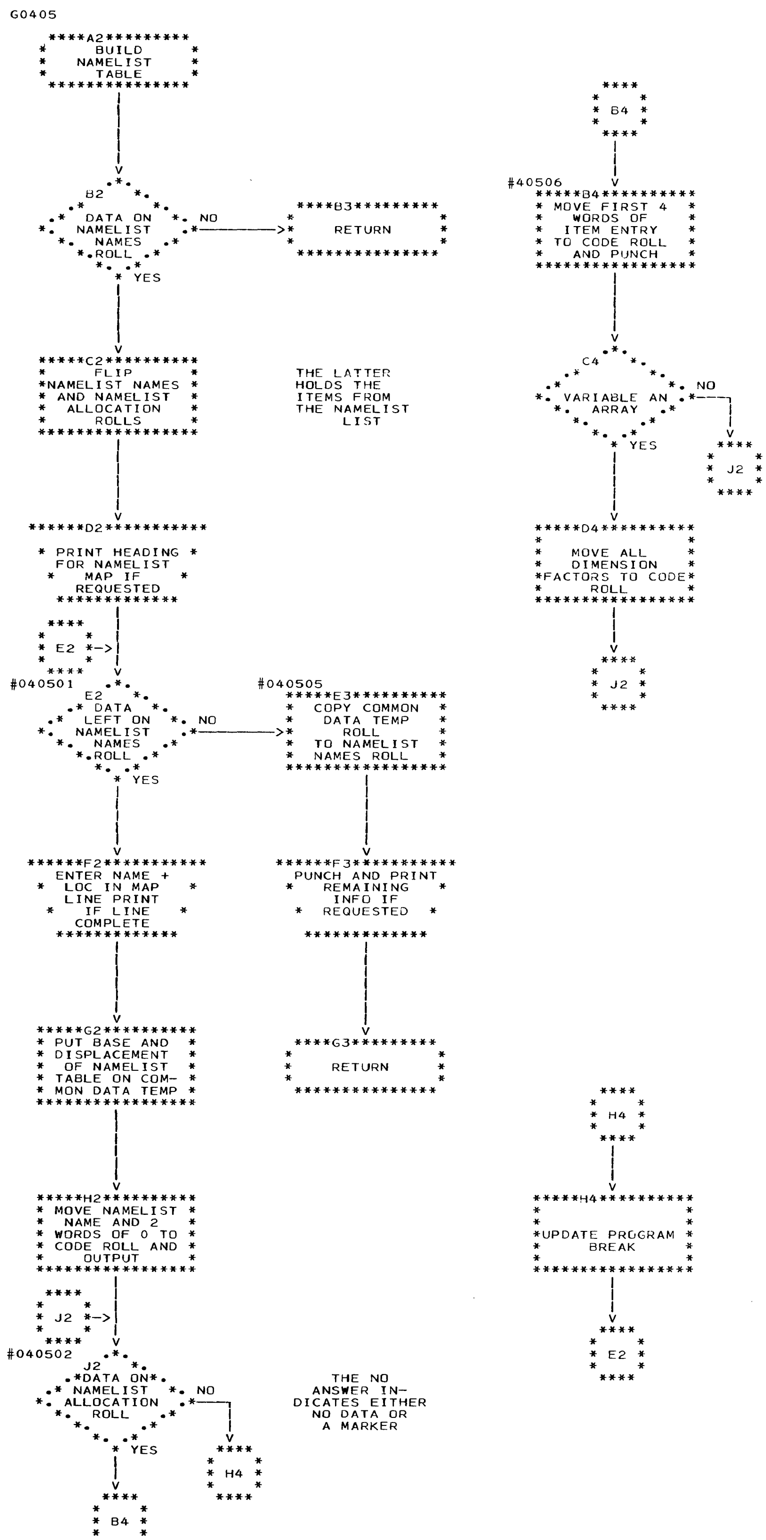


Chart CW. BUILD BASES

```
G0438
      ****A2*********
      *    BUILD     *
      *  ADDITIONAL  *
      *    BASES     *
      ***************
              |
    ****      |
   *    *     |
   * B2 *->|
   *    *     |
    ****      |
              V
     *****B2**********
     *               *
     *    OBTAIN     *
     *PRESENT PROGRAM*
     *   LOCATION    *
     *               *
     *****************
              |
              |
              V
             .*.
        C2 .*   *.
         .*       *.
       .*  MAX FOR  *. YES             ****C3*********
      *.  LAST BASE  .*----------->*     RETURN     *
         *.        .*                 *               *
           *.    .*                   ***************
             *. .*
              *  NO
              |
              |
              V
     *****D2**********
     *               *
     *   INCREMENT   *
     *BASE ALLOCATION*
     *               *
     *               *
     *****************
              |
              |
              V
     *****E2**********
     *               *
     *   REGISTER    *
     *   NEW BASE    *
     *  ALLOCATION   *
     *               *
     *****************
              |
              V
             ****
            *    *
            * B2 *
            *    *
             ****
```

Chart CX. DEBUG ALLOCATE

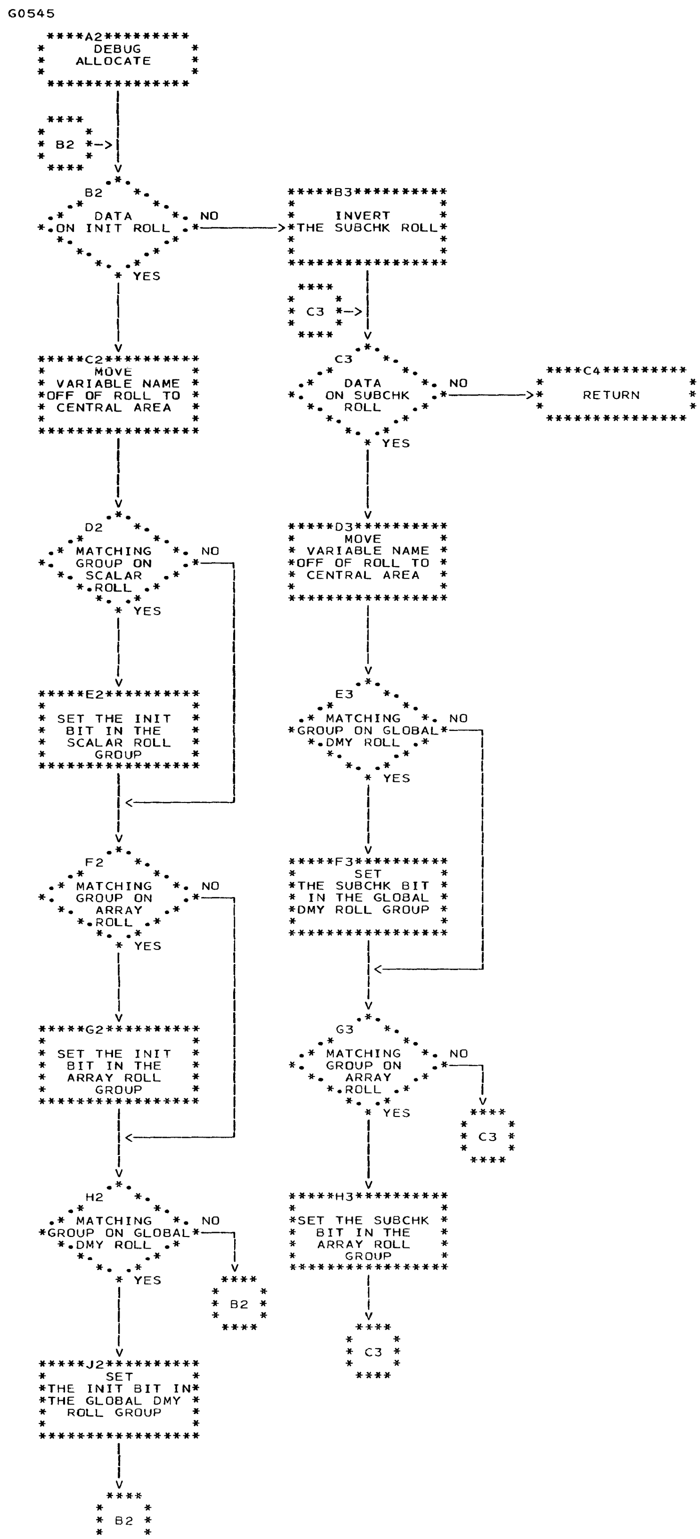


Chart 07. PHASE 3 - UNIFY

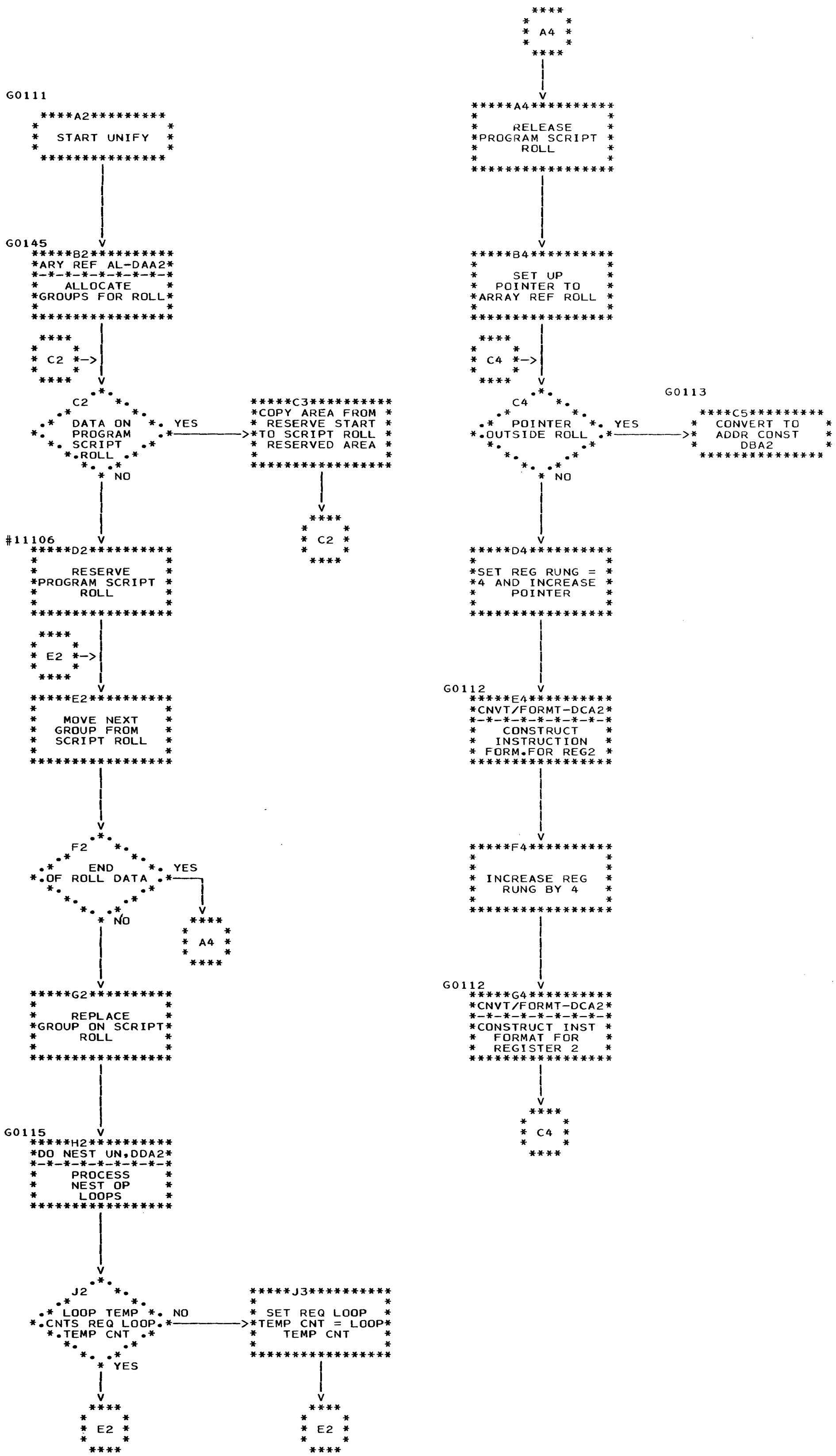


Chart DA. BUILD ARRAY REF ROLL

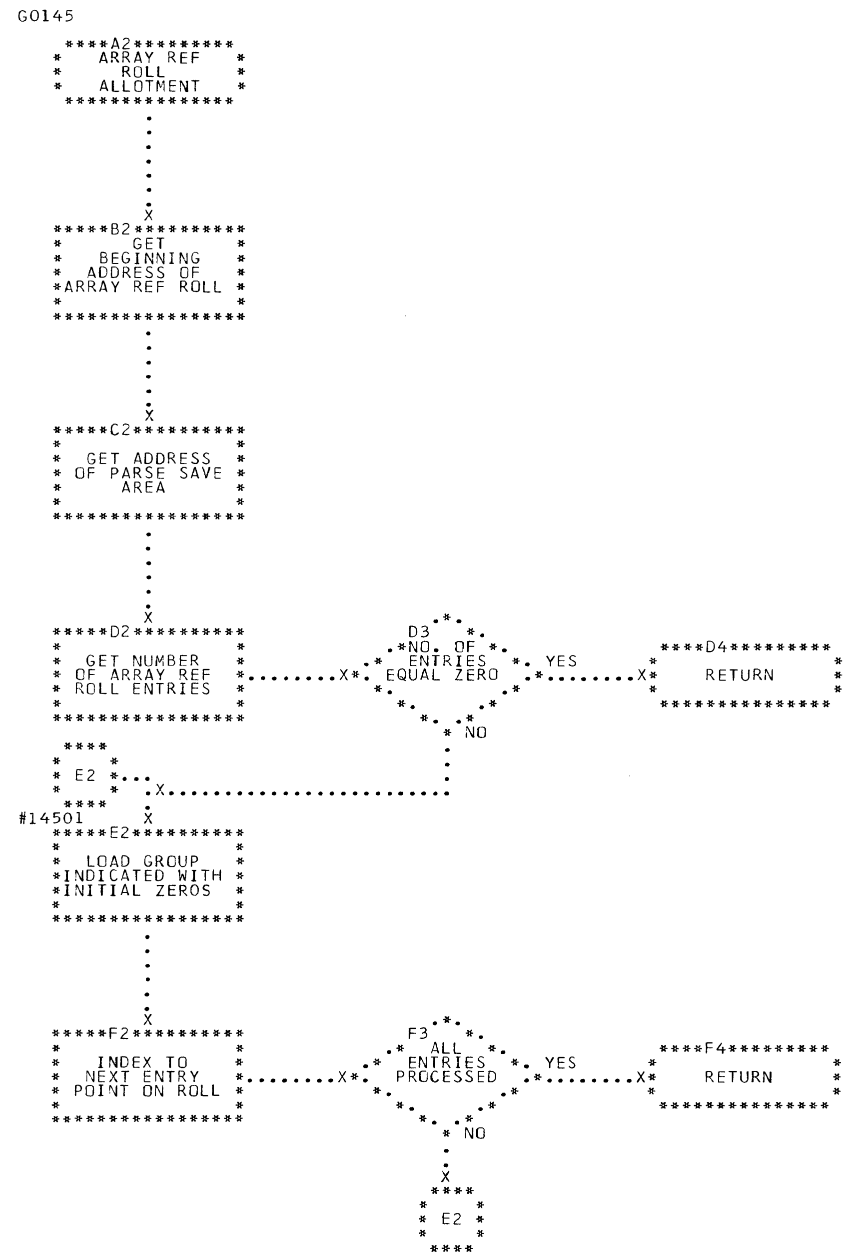


Chart DB. MAKE ADDRESS CONSTANTS

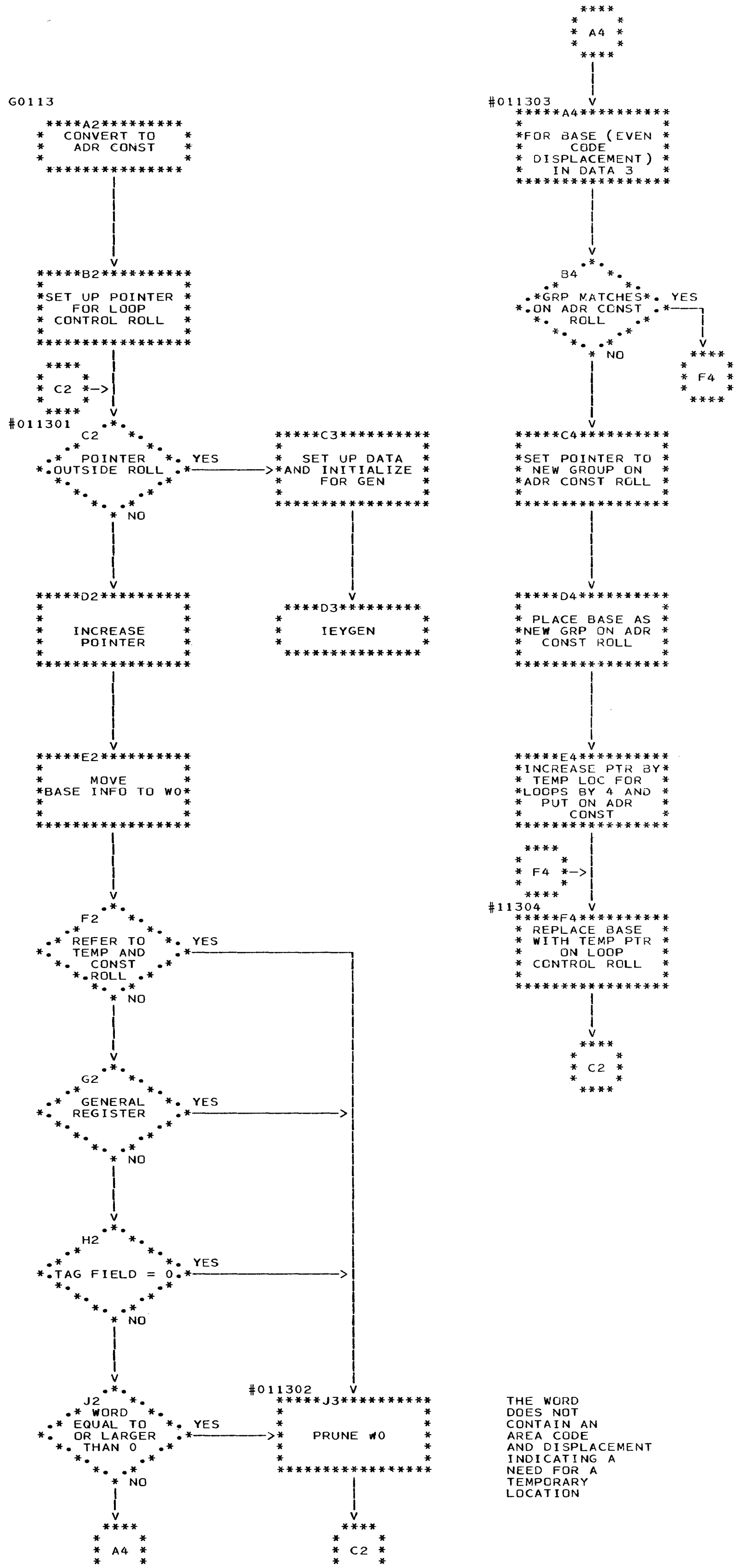


Chart DC. CONSTRUCT INSTRUCTIONS

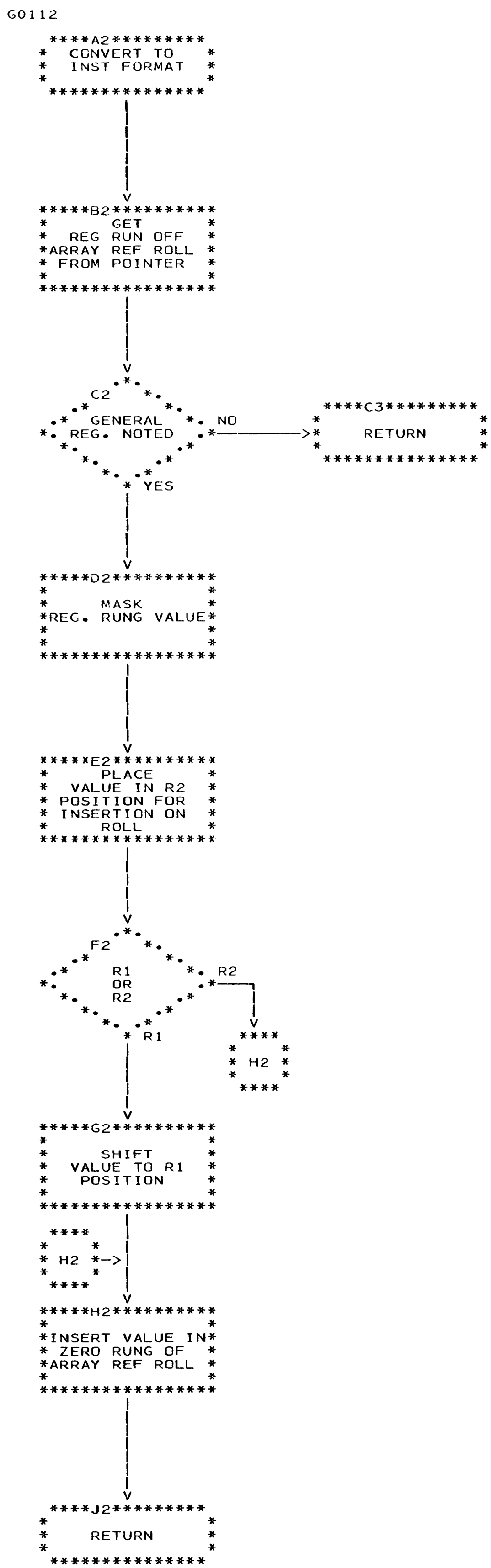


Chart DD. PROCESS NESTED LOOPS

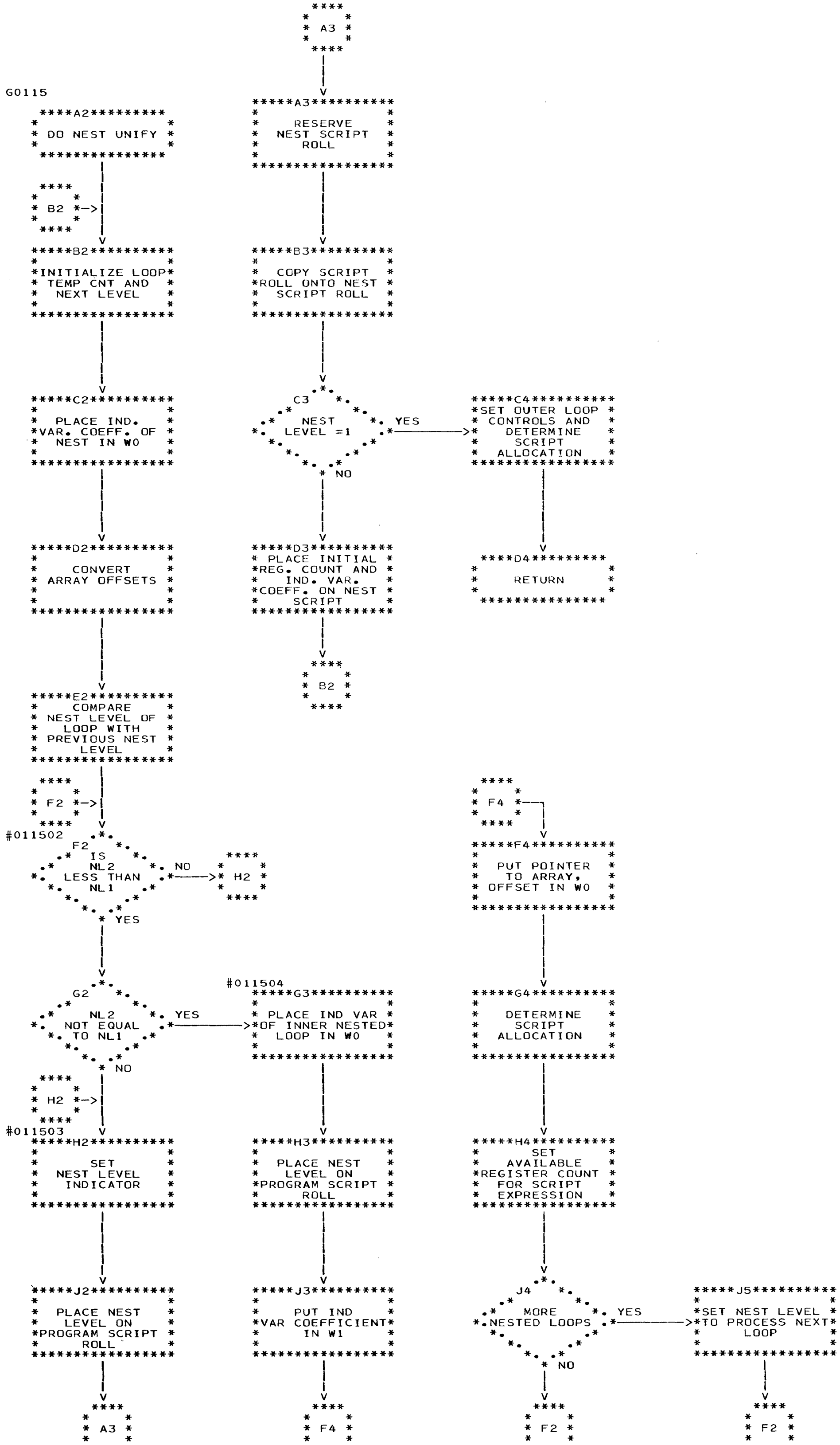


Chart 08. PHASE 4 - GEN

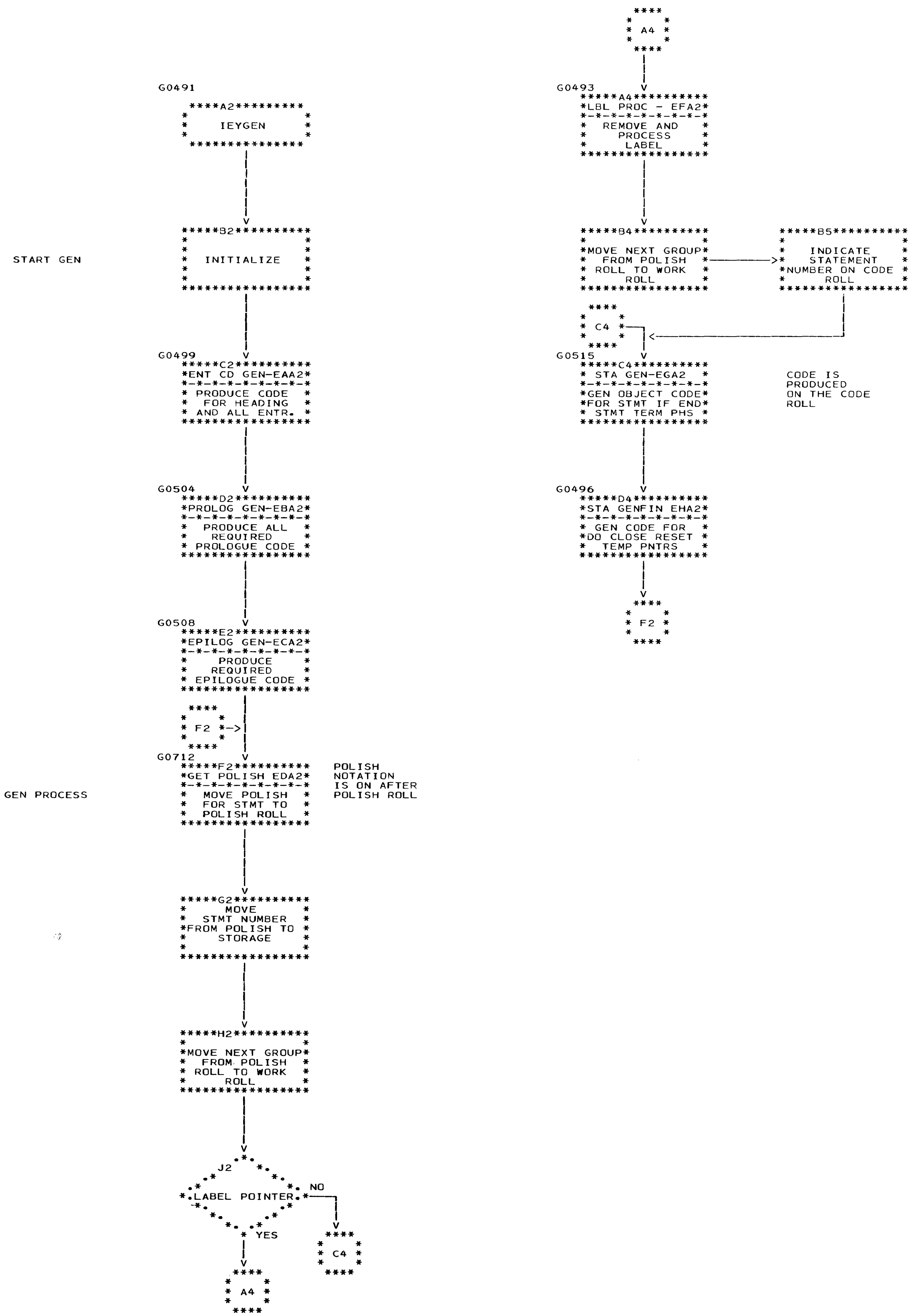


Chart EA. GENERATE ENTRY CODE

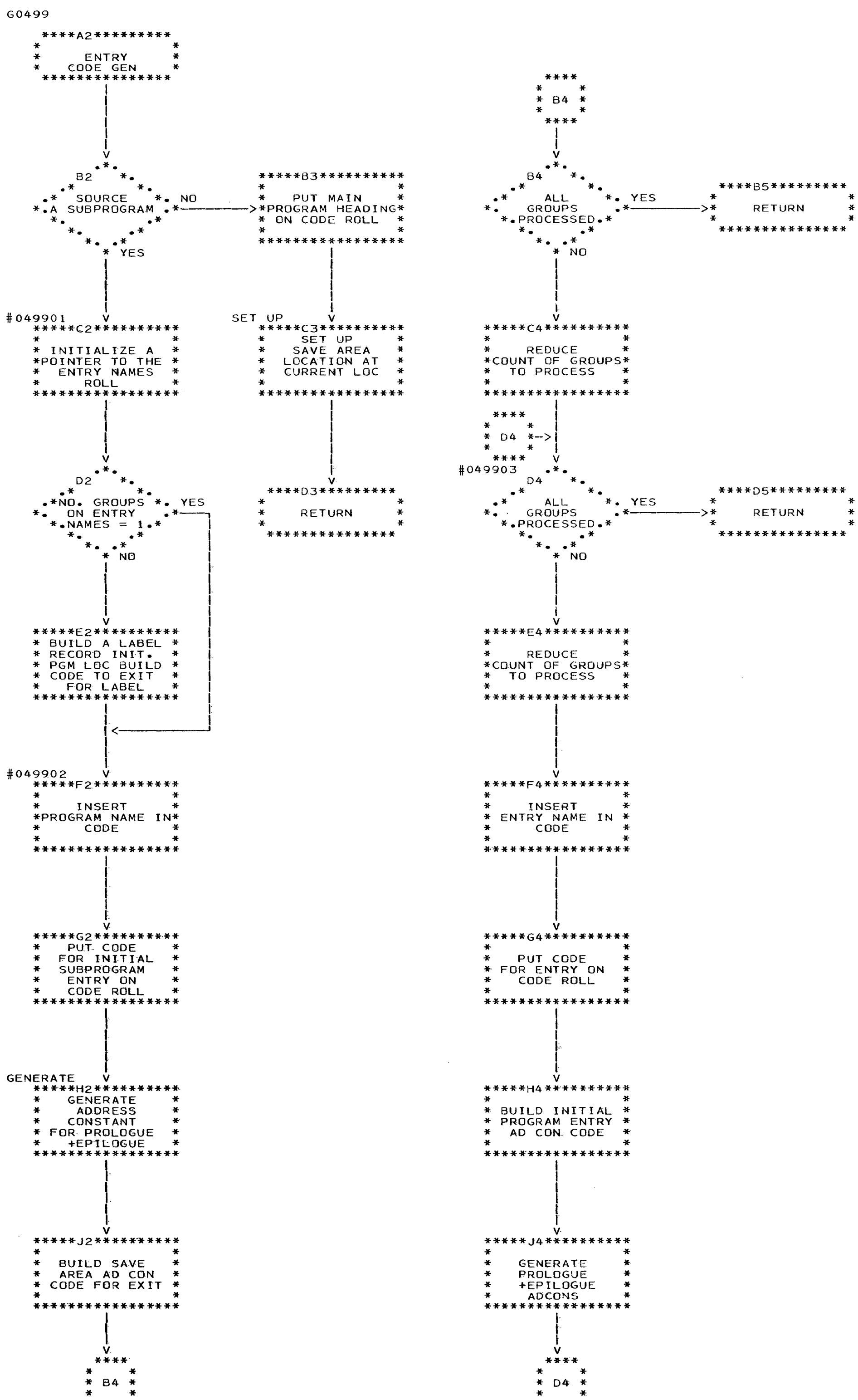


Chart EB. PROLOGUE CODE GENERATION

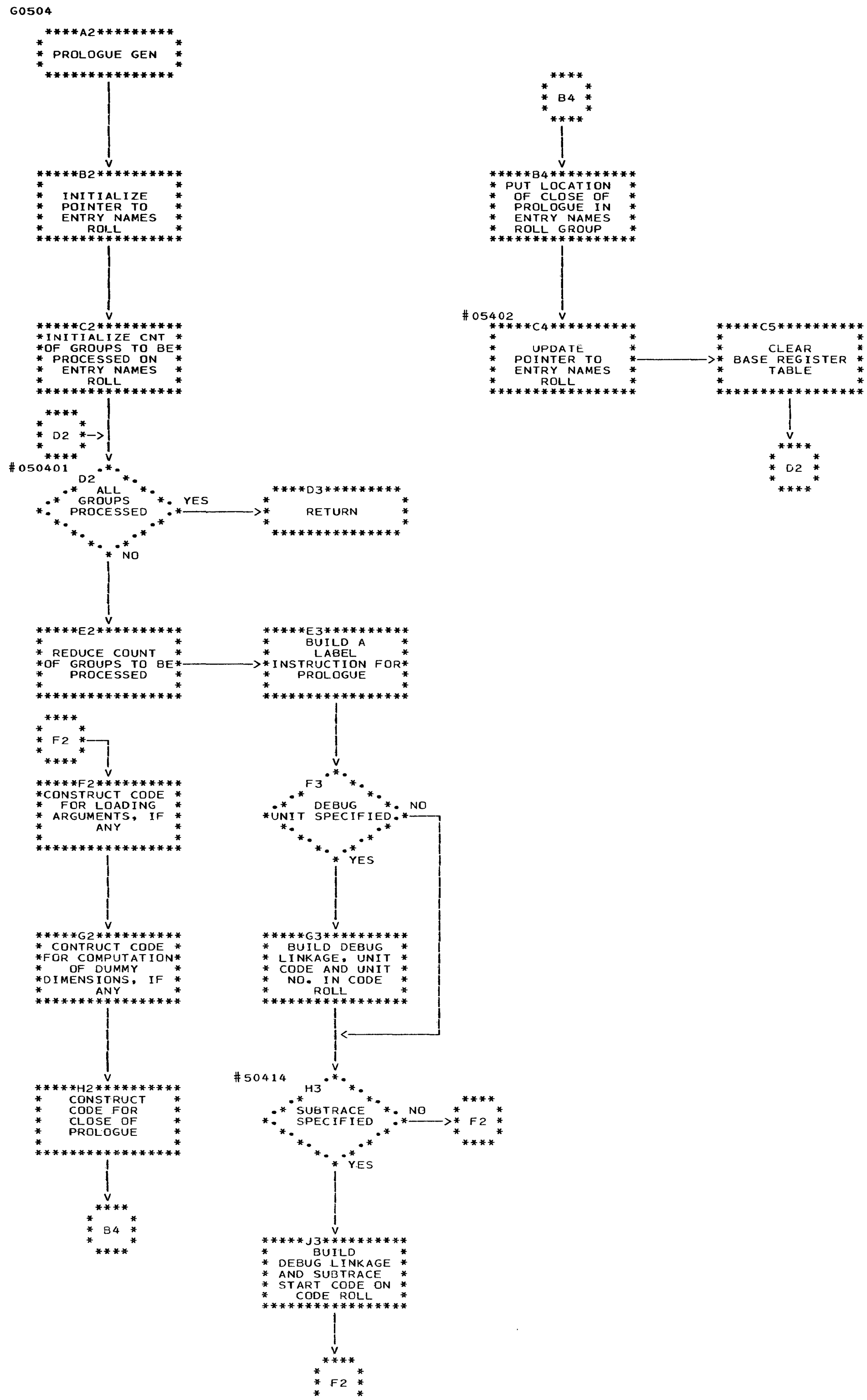


Chart EC. EPILOGUE CODE GENERATION

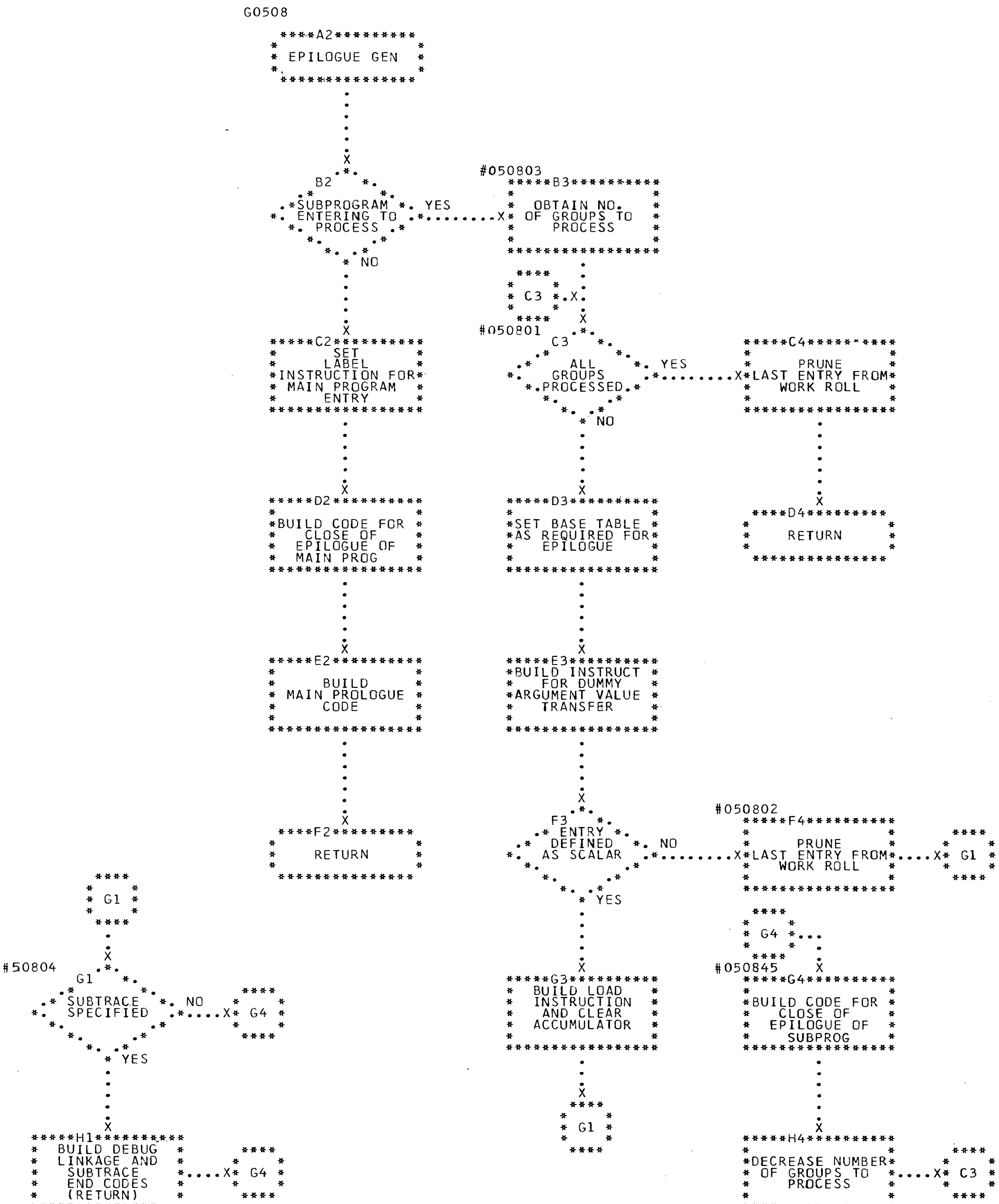


Chart ED. MOVE POLISH NOTATION

```
G0712
  ****A2*********
  *             *
  *  GET POLISH  *
  *             *
  ***************
         |
         |
         |
         V
*****B2**********
*               *
*SET UP POINTER *
*TO AFTER POLISH*
*      ROLL     *
*               *
*****************
         |
         |
         |
         V
*****C2**********
*               *
*  COPY POLISH  *
*  FOR STMT TO  *
*  POLISH ROLL  *
*               *
*****************
         |
         |
         |
         V
*****D2**********
*               *
*UPDATE CONTROLS*
*   FOR AFTER   *
*  POLISH ROLL  *
*               *
*****************
         |
         |
         |
         V
  ****E2*********
  *             *
  *   RETURN    *
  *             *
  ***************
```

Chart EF. PROCESS LABELS

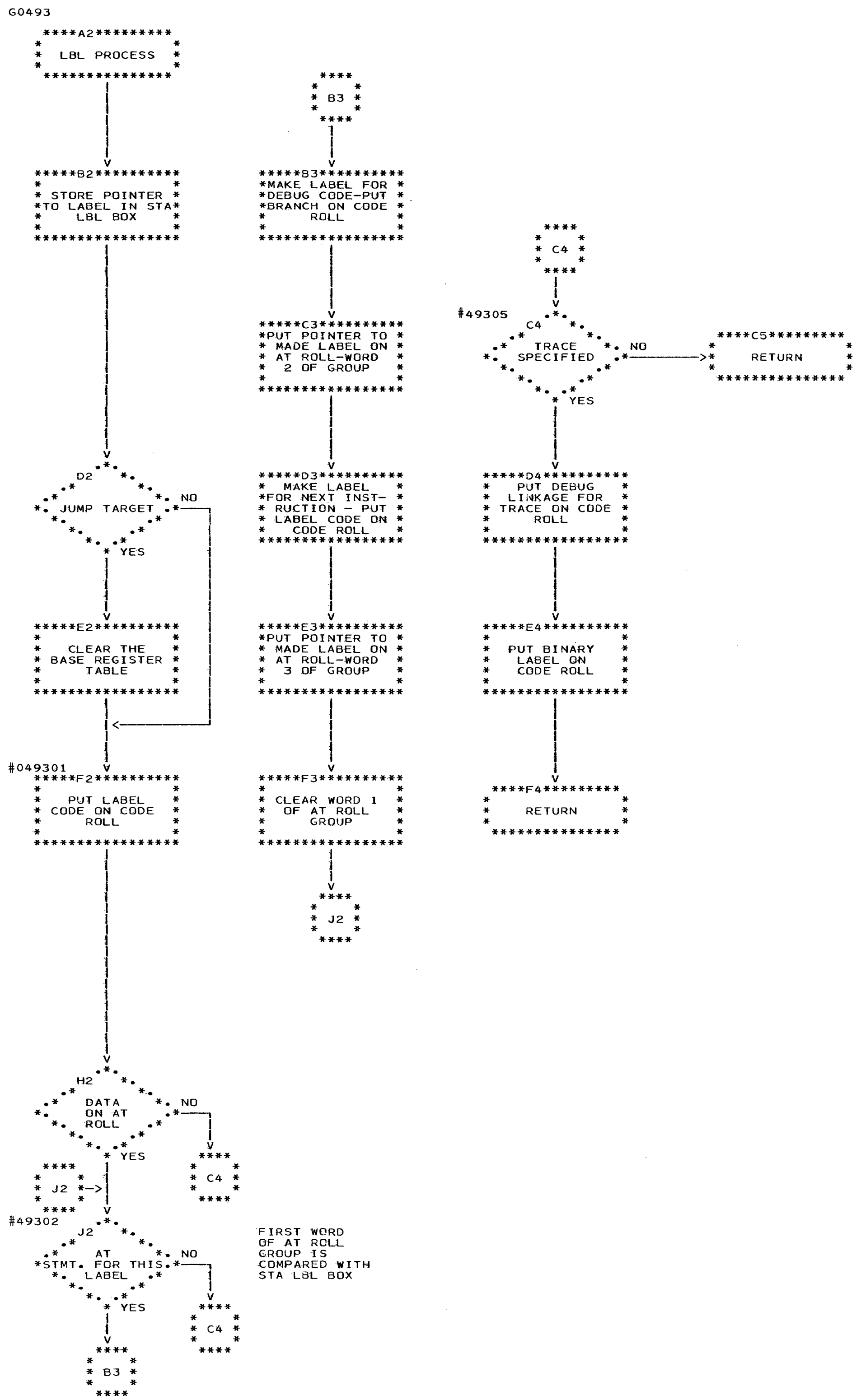


Chart EG. GENERATE STMT CODE

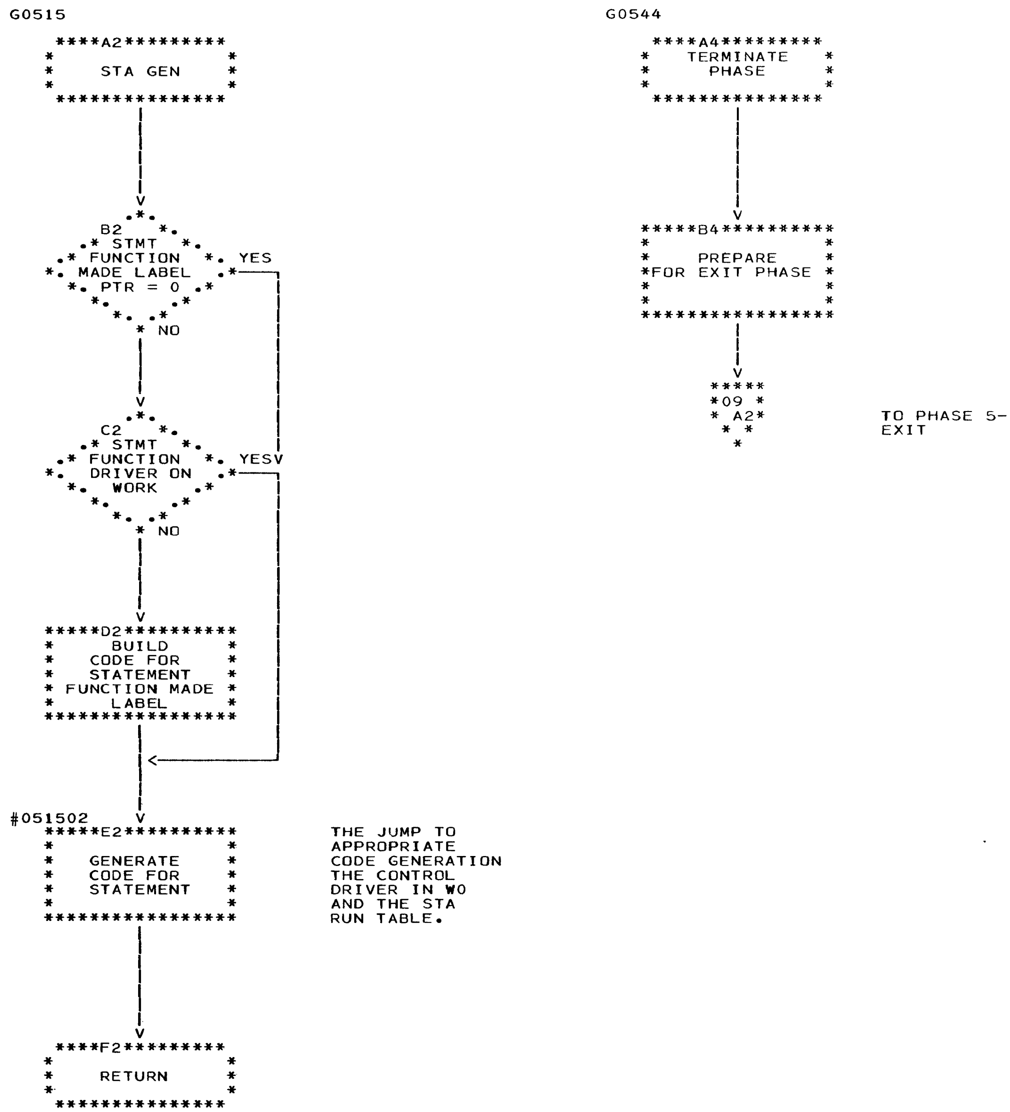


Chart EH. COMPLETE OBJECT CODE

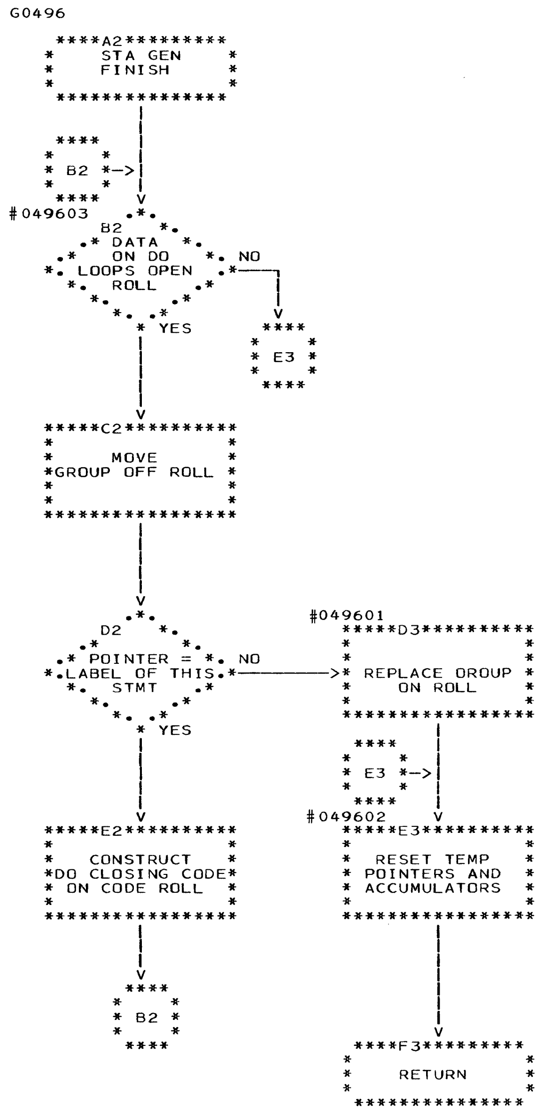


Chart 09. PHASE 5 - IEYEXT

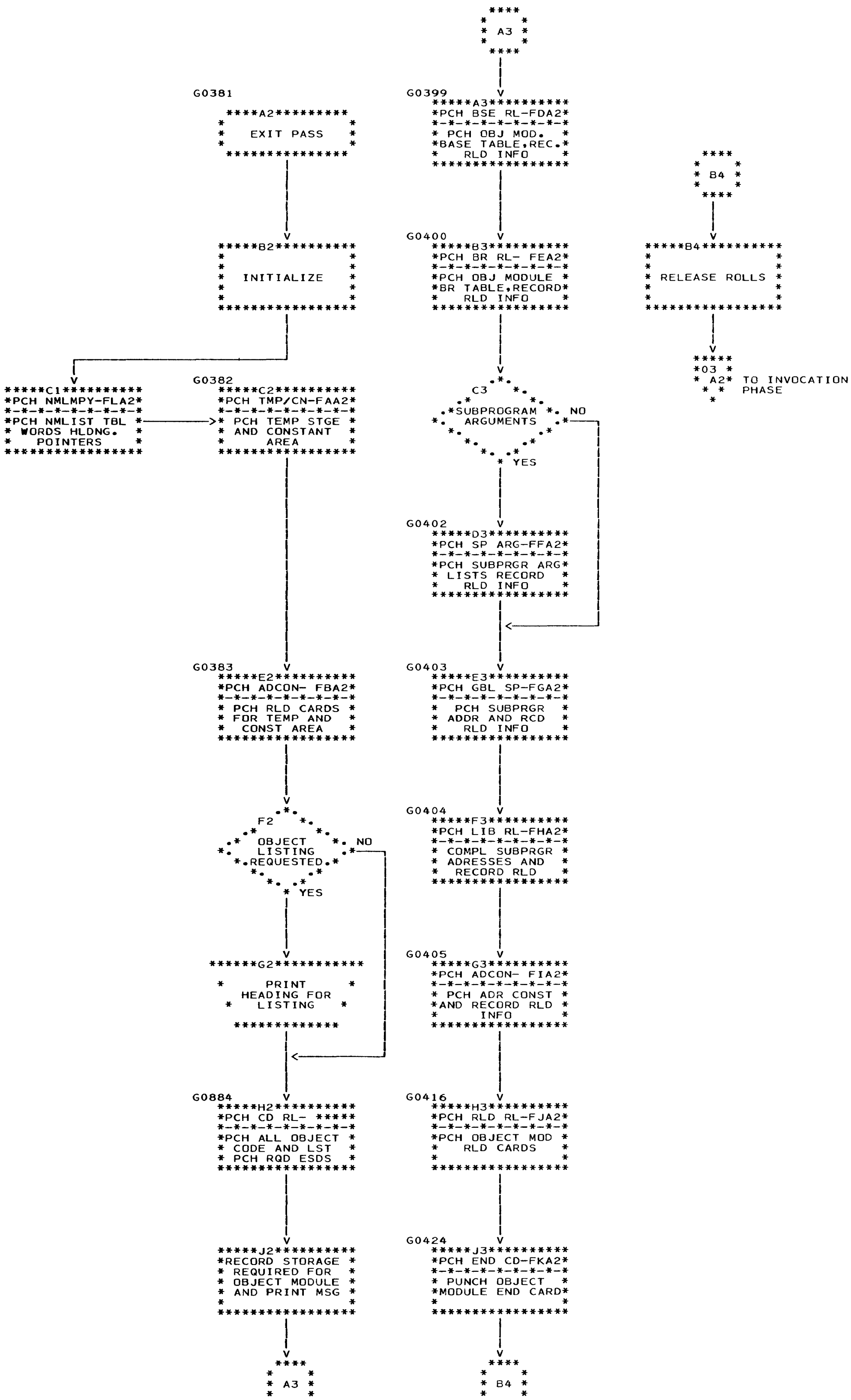


Chart FA. PUNCH CONSTANTS AND TEMP STORAGE

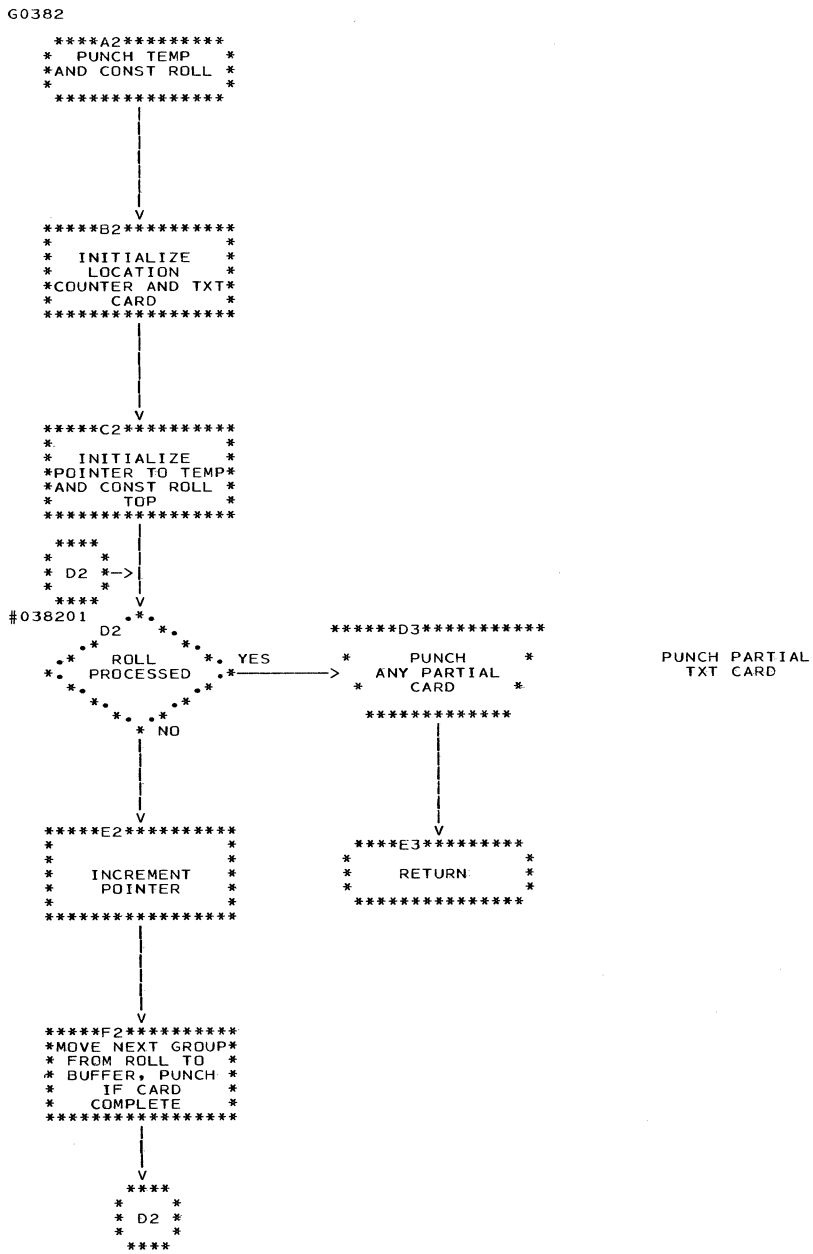


Chart FB. PUNCH ADR CONST ROLL

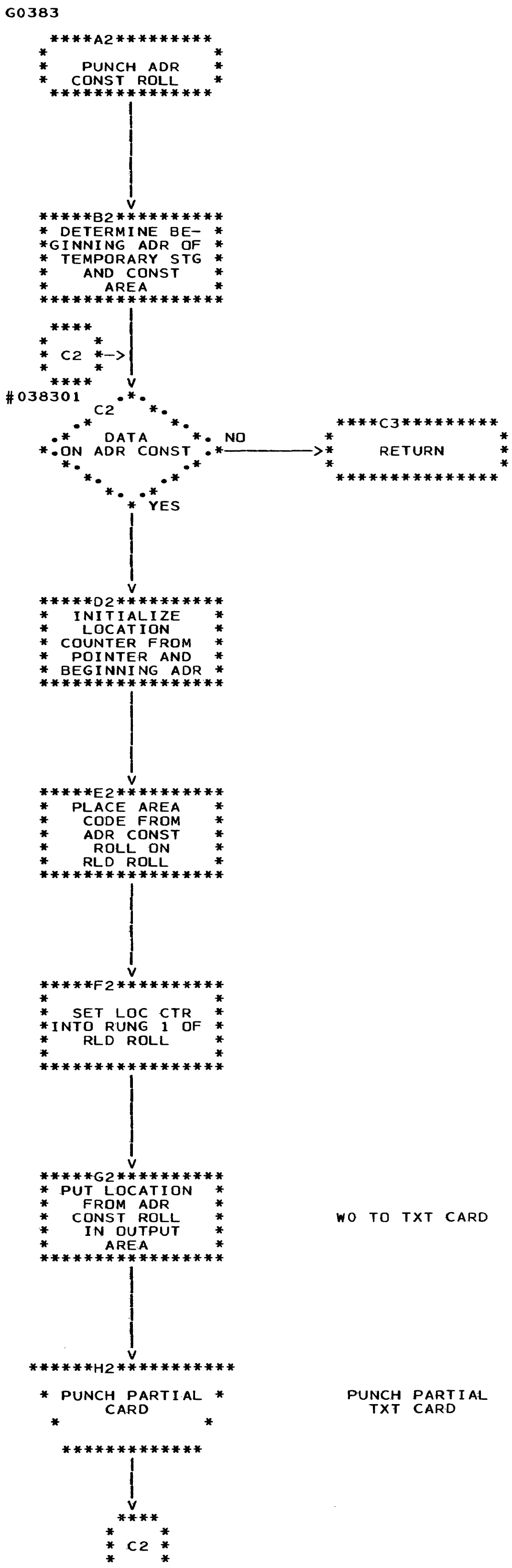
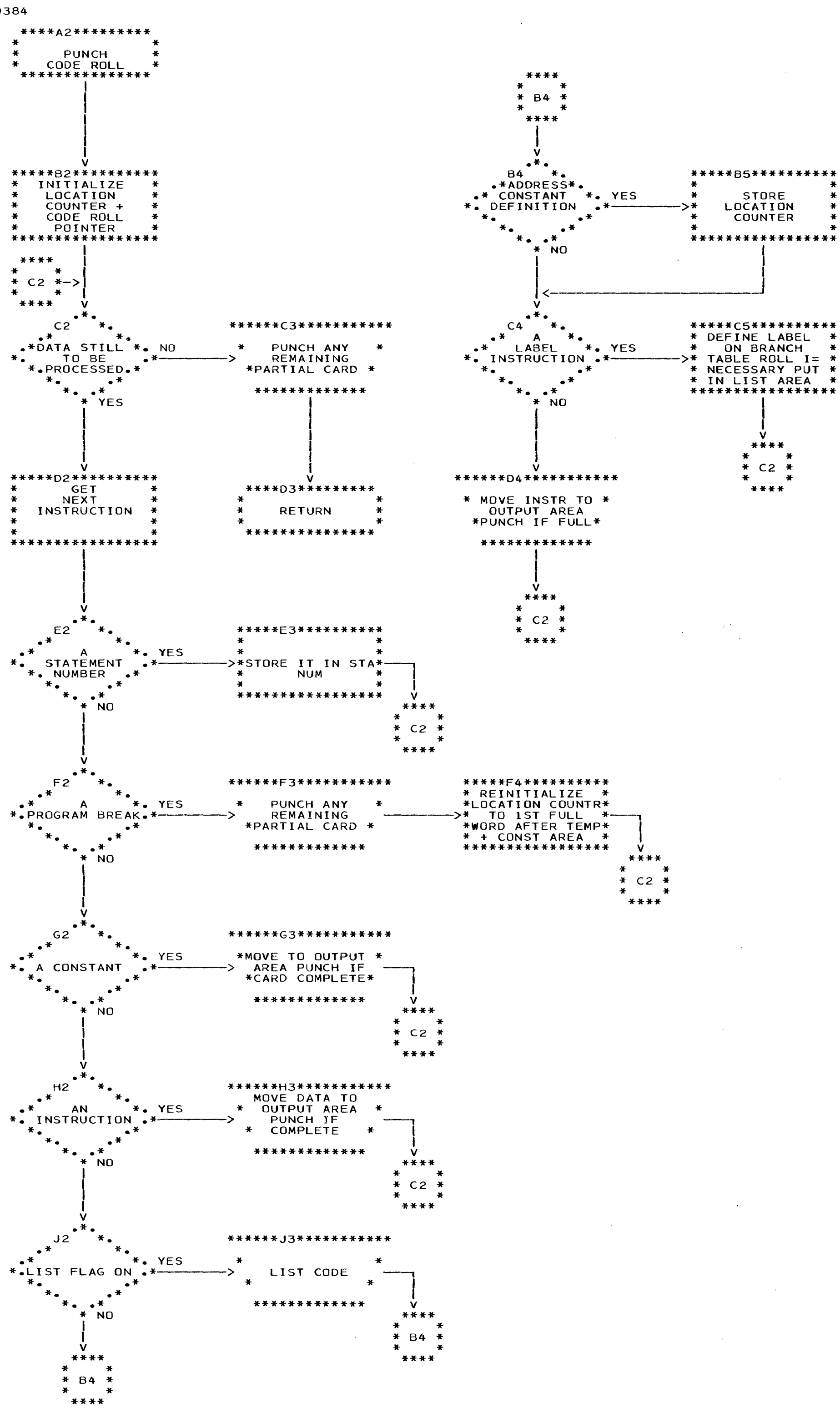


Chart FC. PUNCH OBJECT CODE



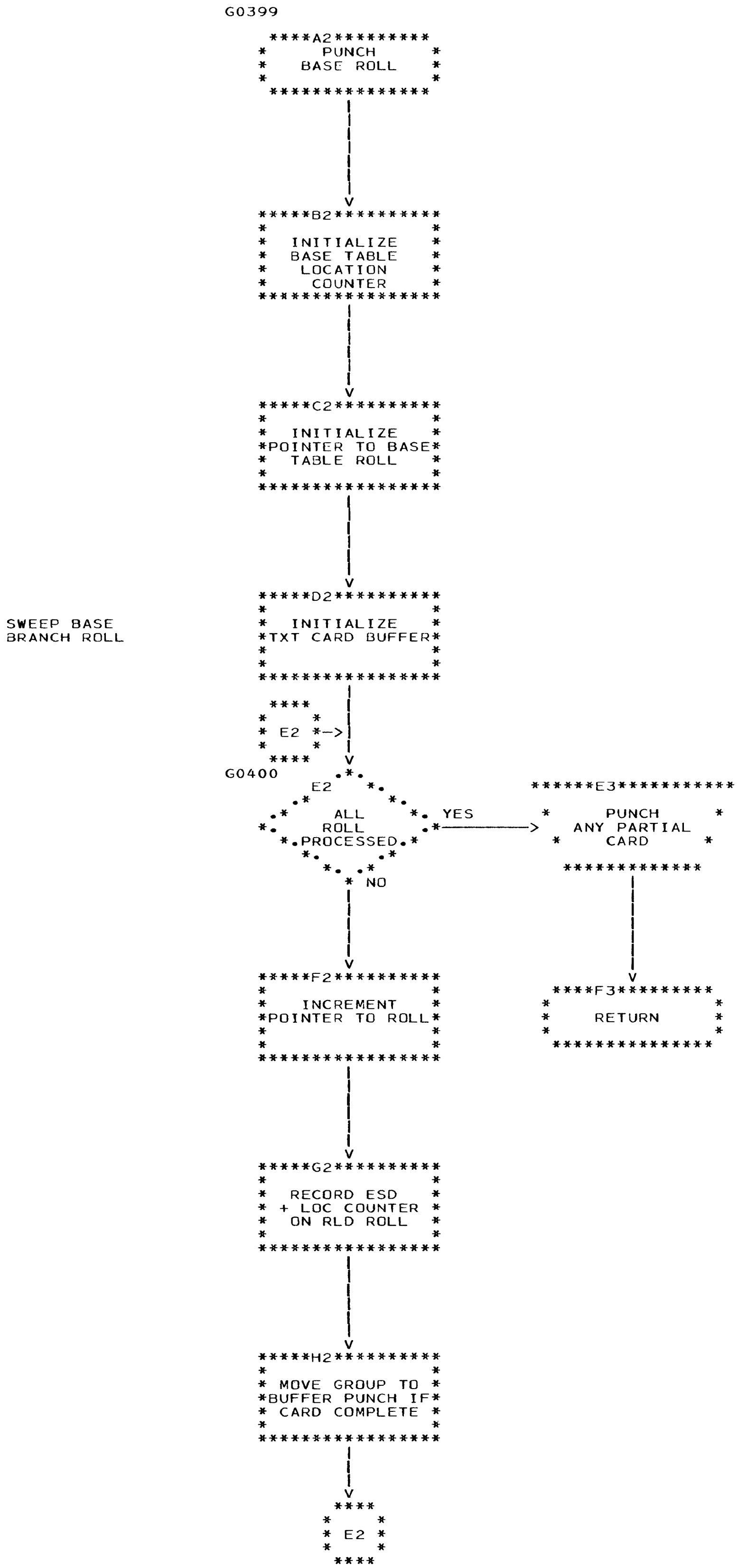

G0399
A2
PUNCH BASE ROLL
B2
INITIALIZE BASE TABLE LOCATION COUNTER
C2
INITIALIZE POINTER TO BASE TABLE ROLL
SWEEP BASE BRANCH ROLL
D2
INITIALIZE TXT CARD BUFFER
E2
G0400
E2
ALL ROLL PROCESSED
YES
NO
E3
PUNCH ANY PARTIAL CARD
F3
RETURN
F2
INCREMENT POINTER TO ROLL
G2
RECORD ESD + LOC COUNTER ON RLD ROLL
H2
MOVE GROUP TO BUFFER PUNCH IF CARD COMPLETE
E2

Chart FE. PUNCH BRANCH TABLE

```
                          G0400
                           ****A2*********
                           *    PUNCH      *
                           *  BRANCH ROLL  *
                           *               *
                           ***************
                                  |
                                  |
                                  V
                          *****B2**********
                          *                *
                          *   INITIALIZE   *
                          * BRANCH TABLE   *
                          *  LOC COUNTER   *
                          *                *
                          *****************
                                  |
                                  |
                                  V
                          *****C2**********
                          *                *
                          *   INITIALIZE   *
                          *   POINTER TO   *
                          * BRANCH TABLE   *
                          *      ROLL      *
                          *****************
                                  |
                                  |
                                  V
SWEEP BASE                *****D2**********
BRANCH ROLL               *                *
                          *   INITIALIZE   *
                          *TXT CARD BUFFER*
                          *                *
                          *                *
                          *****************
                                  |
                           ****   |
                          *    *  |
                          * E2 *->|
                          *    *  |
                           ****   V
                         #040001 .*.
                              E2    *.                         ******E3***********
                              .*       *.                      *     PUNCH         *
                          .*     ALL       *. YES              *  ANY PARTIAL     *
                          *.    ROLL      .*-------------->    *     CARD         *
                           *.PROCESSED.*                       *                  *
                              *.    .*                           *************
                               *. .*                                   |
                                 * NO                                  |
                                 |                                     V
                                 |                               ****F3*********
                                 V                               *              *
                          *****F2**********                      *    RETURN    *
                          *                *                     *              *
                          *   INCREMENT    *                      **************
                          *POINTER TO ROLL*
                          *                *
                          *                *
                          *****************
                                  |
                                  |
                                  V
                          *****G2**********
                          *                *
                          *   RECORD ESD   *
                          *AND LOC COUNTER*
                          *   ON RLD ROLL  *
                          *                *
                          *****************
                                  |
                                  |
                                  V
                          *****H2**********
                          *      MOVE      *
                          *    GROUP TO    *
                          * BUFFER, PUNCH  *
                          *    IF CARD     *
                          *    COMPLETE    *
                          *****************
                                  |
                                  V
                                 ****
                                *    *
                                * E2 *
                                *    *
                                 ****
```

Chart FF. PUNCH SUBPROGRAM ARGUMENT LISTS

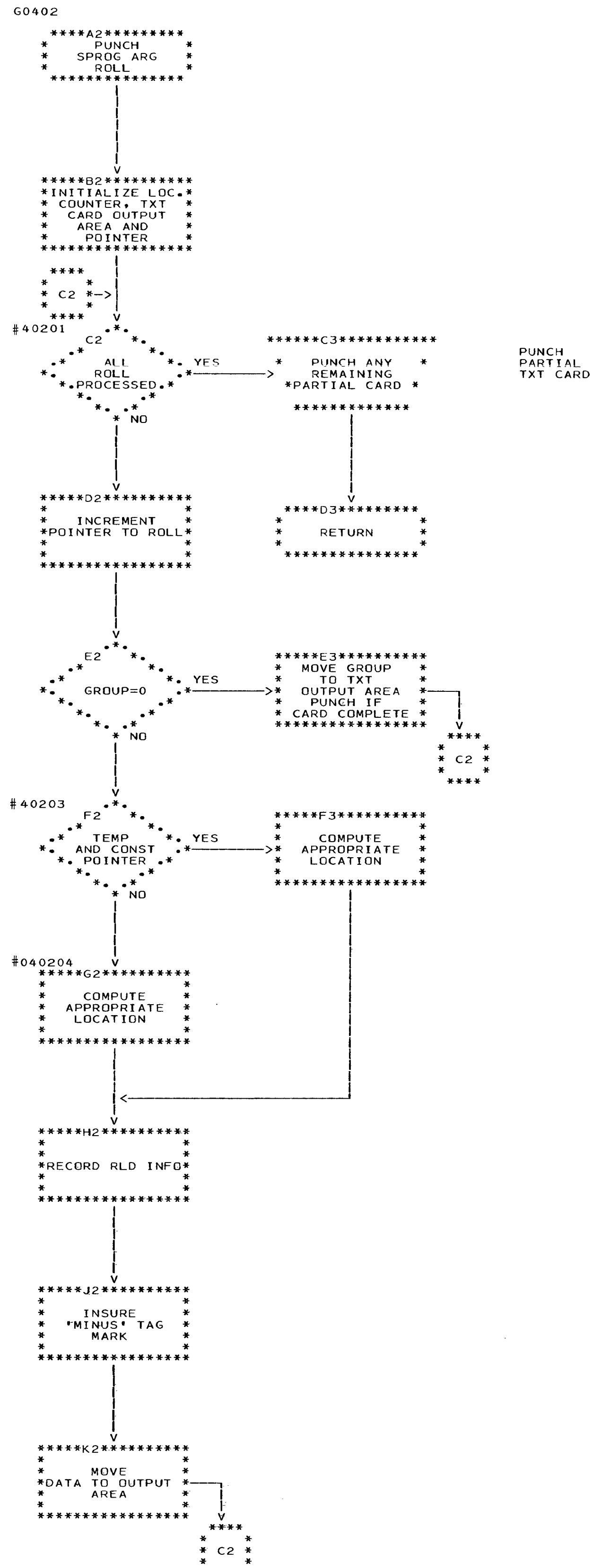


Chart FG. PUNCH SUBPROGRAM ADDRESSES

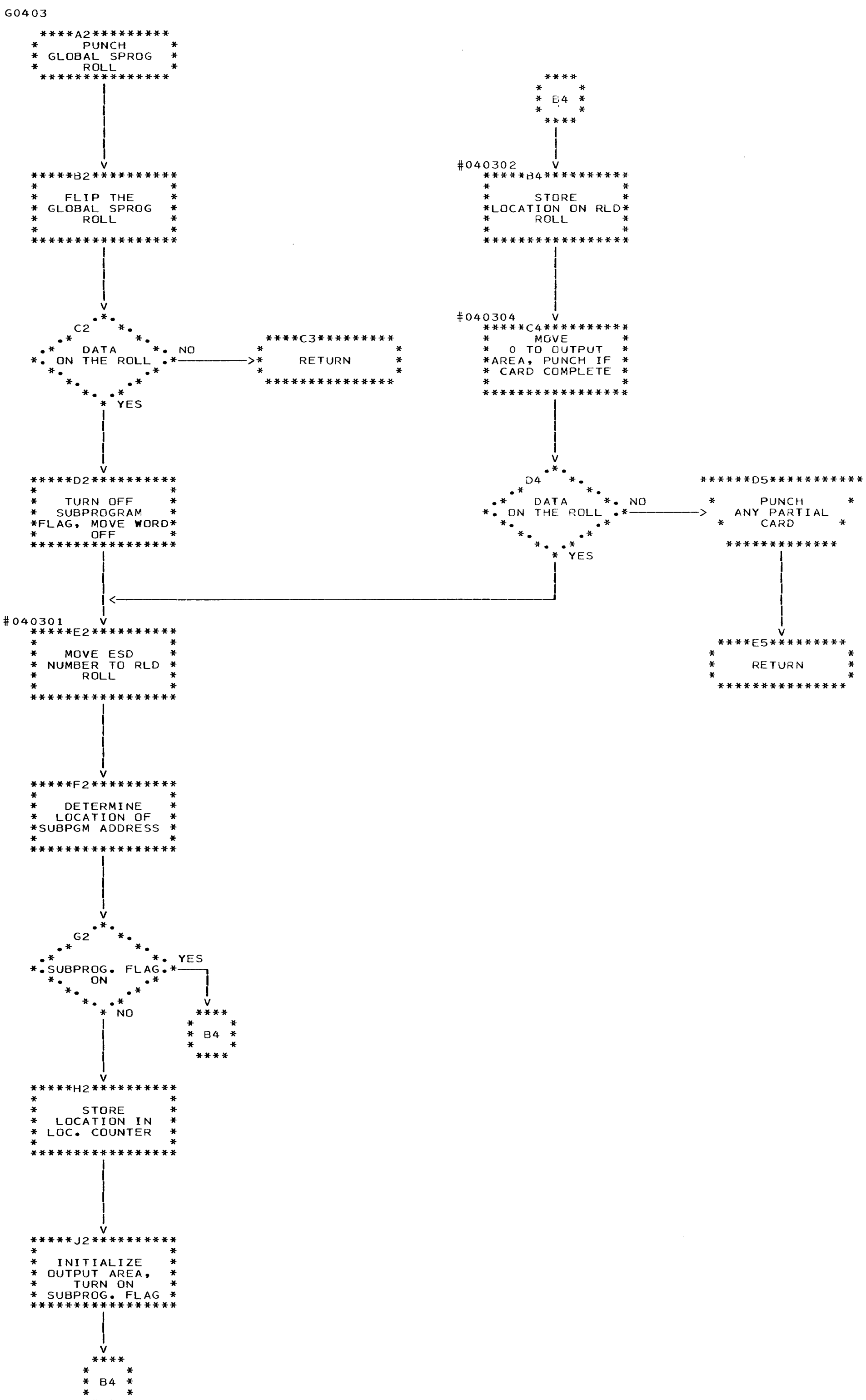


Chart FH. COMPLETE ADDRESSES FROM LIBRARY

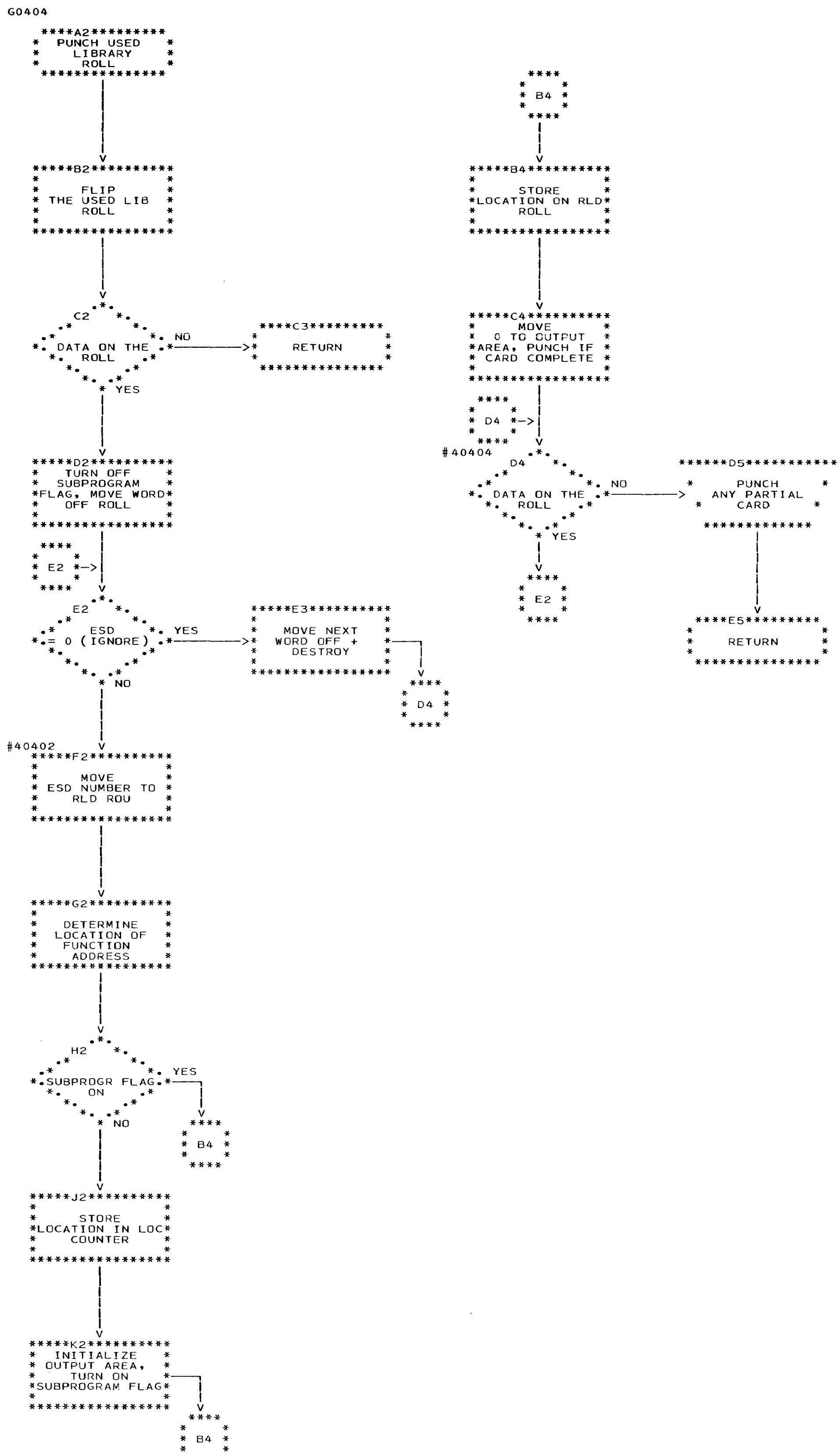


Chart FI. PUNCH ADDRESS CONSTANTS

```
G0405
  ****A2*********
  *             *
  *    PUNCH    *
  *  ADCON ROLL *
  ***************
         |
         |
  ****   |
 *    *  |
 * B2 *->|
 *    *  |
  ****   V
        .*.
   B2 .*   *.
    .*       *.
  .*           *. NO             ****B3*********
 *.DATA ON ROLL .*------------->*    RETURN    *
   *.         .*                *              *
     *.     .*                   ***************
       *. .*
         * YES
         |
         |
         |
         V
 *****C2**********
 *               *
 *   SET AREA    *
 *CODE FROM LAST *
 * WORD ON ROLL  *
 *               *
 *****************
         |
         |
         |
         V
 *****D2**********
 *  SET ADDRESS  *
 *  WHERE CONST  *
 *IS TO BE LOADED*
 *FROM NEXT WORD *
 *    ON ROLL    *
 *****************
         |
         |
         |
         V
******E2***********
 *   MOVE INFO   *
     TO OUTPUT
  AREA AND PUNCH*

  *************
         |
         |
         |
         V
 *****F2**********
 *               *
 *      SET      *
 *  UP RLD ROLL  *
 *     ENTRY     *
 *               *
 *****************
         |
         |
         V
        ****
       *    *
       * B2 *
       *    *
        ****
```

Chart FJ. PUNCH RLD CARDS

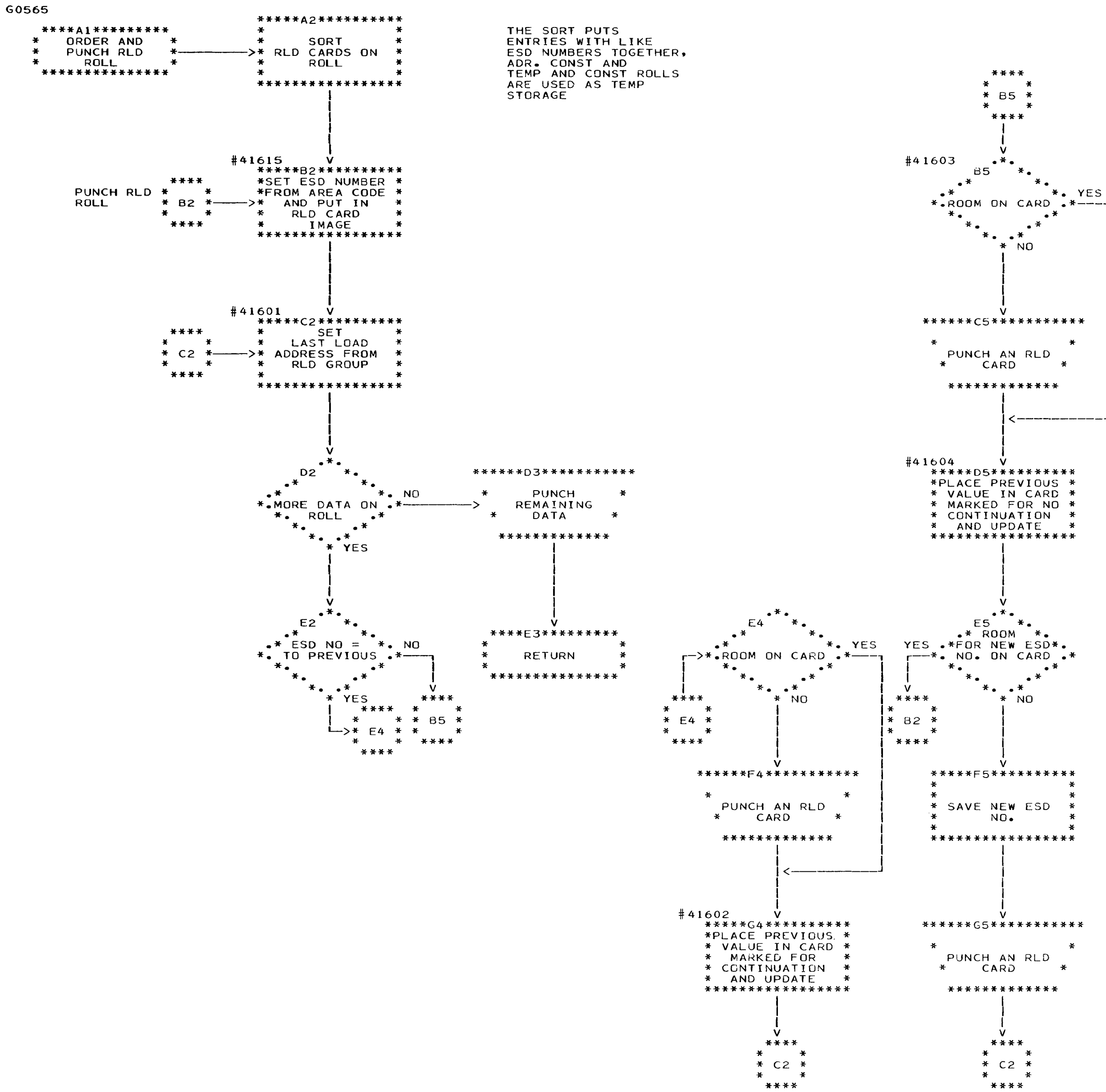


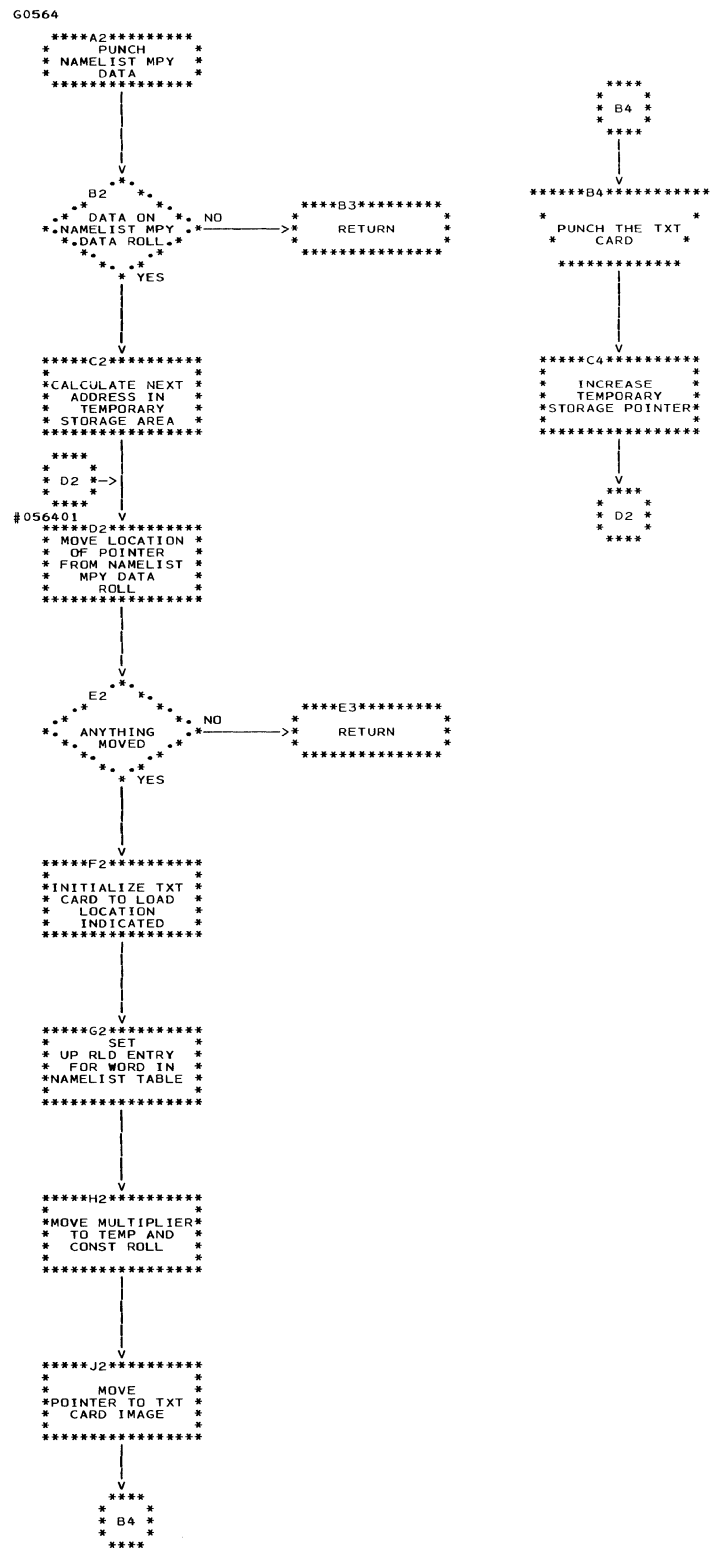
Chart FK. PUNCH END CARDS

```
G0424
    ****A2*********
   *               *
   *     PUNCH     *
   *   END CARD    *
    ***************
           |
           |
           |
           |
           |
           |
           V
   *****B2**********
   *               *
   *               *
   *SET UP END CARD*
   *               *
   *               *
   *****************
           |
           |
           |
           |
           |
           V
 ******C2***********

   *               *
    PUNCH END CARD
    *             *

     *************
           |
           |
           |
           |
           |
           |
           V
    ****D2*********
   *               *
   *     RETURN    *
   *               *
    ***************
```

Chart FL. PUNCH NAMELIST TABLE POINTERS



This appendix deals with the POP language, the language in which the FORTRAN IV (G) compiler is written. The parts of the appendix describe this language in the following way:

- The first part describes the POP instructions, which are grouped according to their functions.
- The second part describes the labels used in the routines of the compiler.
- The third part discusses the assembly and operation of the compiler, as it is affected by the use of the POP language. This part ends with a cross-reference list giving the mnemonic for each instruction, the hexadecimal code which represents it, and the instruction group in which it is described.

POP INSTRUCTIONS

For the purpose of describing their operation, the POP instructions have been divided into groups according to the primary function which they perform. Where a particular POP instruction pertains to more than one group, it is described in the group which discusses its most important functions.

In the descriptions of the instructions, the following notational conventions are employed:

1. Parentheses are used to indicate "the contents of;" thus (G) stands for the contents of storage address G, where all addresses are fullword addresses.
2. The arrow is used to indicate transmission in the direction of the arrow; (G) + 1 --> G reads: the contents of storage address G, plus one, are transmitted to storage address G.
3. Wn (n=1,2,3,...) refers to the BOTTOM, BOTTOM-1, ... etc., words on the WORK roll.

It should be noted that in many cases the address field, G, of the instruction contains a value other than a storage address (for instance, a roll name). In most of these cases, the symbolic reference which is used is defined in the program by means of an EQU card.

The mnemonic codes for the POP instructions are of the form IEYxxx. In the following discussion, the characters IEY are omitted from the mnemonics in the interest of ease of reading, and only the xxx portion of the code appears.

TRANSMISSIVE INSTRUCTIONS

The instructions described in this section are primarily involved in moving information from place to place in storage.

APH G: Assign and Prune Half

The upper halfword of (W0) --> the lower halfword of G, where G is a storage address; the upper halfword of G remains unaltered; the BOTTOM of the WORK roll is reduced by four, thus pruning W0.

ARK G: Assign Relative to Pointer and Keep

(W0) --> P + (G), where P is the address defined by the pointer in W1 and G is a storage address; the BOTTOM of the WORK roll is reduced by four, thus pruning the value assigned and keeping the pointer.

ARP G: Assign Relative to Pointer

(W0) --> P + (G), where P is the address defined by the pointer in W1 and G is a storage address; the BOTTOM of the WORK roll is reduced by eight, thus pruning the current W0 and W1.

ASK G: Assign to Storage and Keep

(W0) --> G, where G is a storage address; the BOTTOM of the WORK roll is unchanged.

ASP G: Assign to Storage and Prune

(W0) --> G, where G is a storage address; the BOTTOM of the WORK roll is reduced by four, thus pruning the current W0.

BOP G: Build on Polish

The control driver G is built on the POLISH roll, where the G field of the instruction is the lower eight bits of the ADDRESS portion of the desired driver. (The TAG field of the pointer contains zero, and the OPERATOR field contains 255.)

CAR G: Copy and Release

Copy roll G, where G is a roll number, to roll T, and release roll G (i.e., restore it to its condition before the last reserve); the number T is found in W0; the BOTTOM of the WORK roll is reduced by four. If roll G is in the reserved state when this instruction is executed, the instruction sets its BOTTOM to (TOP) minus four; if the roll is not reserved, BOTTOM is set to (BASE).

CLA G: Clear and Add

Clear W0; (G) --> W0, where G is a storage address; the BOTTOM of the WORK roll is unchanged.

CNT G: Count

The number of words on roll G --> W0, where G is a roll number; the BOTTOM of the WORK roll is increased by four.

CPO G: Copy Plex On

The plex pointed to by the pointer in W0 is copied to roll G, where G is the number of the target roll, except for the first word of the plex (which holds the number of words in the plex, exclusive of itself). The BOTTOM of the WORK roll is reduced by four, thus pruning the pointer. The BOTTOM of roll G is increased by four for each word moved; the BOTTOM of the original roll is unchanged.

CRP G: Copy Relative to Pointer

Copy roll S to roll G, where G is a roll number, beginning with the group indicated by the pointer in W0, to the BOTTOM of the roll. The roll number S is also provided by the pointer in W0. The BOTTOM of roll S is decreased by the number of bytes moved. The BOTTOM of roll G is increased by the number of bytes moved. The BOTTOM of the WORK roll is unchanged; thus, the pointer remains.

EAD G: Extract Address

The ADDRESS portion of (G) --> W0, where G is a storage address; the BOTTOM of the WORK roll is increased by four.

EAW G: Effective Address to Work

G --> W0, where G is a storage address; the BOTTOM of the WORK roll is increased by four.

ECW G: Effective Constant Address to Work

G --> W0, where G is a storage address which refers to a constant under a constant base. The BOTTOM of the WORK roll is increased by four.

EOP G: Extract Operator

The OPERATOR portion of (G) --> W0 (right adjusted), where G is a storage address; the BOTTOM of the WORK roll is increased by four.

ETA G: Extract Tag

TAG portion of (G) --> TAG portion of W0, where G is a storage address; the BOTTOM of the WORK roll is increased by four.

FET G: Fetch

(G) --> W0, where G is a storage address; the BOTTOM of the WORK roll is increased by four.

FLP G: Flip

Invert the order of roll G, where G is a roll number, word for word.

FRK G: Fetch Relative to Pointer and Keep

(P + (G)) --> W0, where P is the address defined by the pointer in W0 and G is a storage address; the BOTTOM of the WORK roll is increased by four; thus, the pointer remains in W1.

FRP G: Fetch Relative to Pointer

(P + (G)) --> W0, where P is the address defined by the pointer in W0 and G is a storage address; the BOTTOM of the WORK roll is unchanged; thus, the pointer is destroyed.

FTH G: Fetch Half

The lower halfword of (G) --> upper halfword of W0, where G is a storage address; the lower halfword of W0 is set to zero; the BOTTOM of the WORK roll is increased by four.

IAD G: Insert Address

The ADDRESS portion of (G) --> the ADDRESS portion of the pointer in W0, where G is a storage address; the BOTTOM of the WORK roll is unchanged.

IOP G: Insert Operator

G --> OPERATOR portion of the pointer in W0, where the G field of the instruction is the desired OPERATOR value; the BOTTOM of the WORK roll is unchanged.

ITA G: Insert Tag

TAG portion of (G) --> TAG portion of the pointer in W0, where G is a storage address; the BOTTOM of the WORK roll is unchanged.

ITM G: Insert Tag Mode

Mode portion of the TAG field of (G) --> mode portion of the TAG field of the pointer in W0, where G is a storage address; the BOTTOM of the WORK roll is unchanged.

LCE G: Last Character Error

The last character count and the address G --> ERROR roll, where G is the address of the message for the error. The count of errors of the severity associated with the message is increased by one, and the MAX STA ERROR NUMBER (which indicates the highest severity level of errors for the present statement) is updated as required.

LCF G: Last Character Error if False

If (ANSWER BOX) = false, the last character count and the address G --> ERROR roll, where G is the address of the message for the error. The count of errors of the severity associated with the message is increased by one, and the MAX STA ERROR NUMBER is updated as required. If (ANSWER BOX) = true, the instruction does nothing.

LCT G: Last Character Error if True

If (ANSWER BOX) = true, the last character count and the address G --> ERROR roll, where G is the address of the message for the error. The count of errors of the severity associated with the message is increased by one, and the MAX STA ERROR NUMBER is updated as required. If (ANSWER BOX) = false, the instruction does nothing.

LGP G: Load Group from Pointer

Loads the group specified by the pointer in W0 into SYMBOL 1, 2, and 3, DATA 0, 1, 2, 3, 4, and 5. The number G is the number of bytes to be loaded; if G=0, the entire group is loaded. The BOTTOM of the WORK roll is unchanged; hence, the pointer remains in W0.

LSS G: Load Symbol from Storage

Loads the (G and G+4), where G is a storage address, into SYMBOL 1, 2, and 3, and DATA 0.

MOC G: Move on Code

G halfwords, where G is an even number, are to be moved from the WORK roll to the CODE roll. A word containing a special value in the first two bytes and the number of <u>words</u> transferred in the last two bytes are first placed on the CODE roll. G/2 words of information are then moved from the WORK roll to the CODE roll; the BOTTOM of the CODE roll is increased by four for each word placed on the roll; the BOTTOM of the WORK roll is reduced by four for each word moved from the roll. A location counter is increased by the number of bytes of object code placed on the roll.

MON G: Move on

(W0) --> roll G, where G is the roll number; the BOTTOM of roll G is increased by four; the BOTTOM of the WORK roll is decreased by four.

NOG G: Number of Groups

The number of groups on roll G --> W0, where G is the roll number; the BOTTOM of the WORK roll is increased by four.

NOZ G: Nonzero

A nonzero value --> G, where G is a storage address.

PGO G: Place Group On

A group from SYMBOL 1, 2, and 3 and DATA 0, 1, 2, 3, 4, and 5 --> roll G, where G is the roll number, by group status; the BOTTOM of roll G is increased by group size.

PGP G: Place Group from Pointer

The group in SYMBOL 1, 2, 3, DATA 0, 1, 2, 3, 4, and 5 is placed on a roll according to the pointer in W0. The number G is the number of bytes to be moved; if G=0, an entire group is moved; the BOTTOM of the WORK roll is unchanged.

PLD G: Precision Load

(G and G+4) --> MPAC 1 and MPAC 2, where G is a storage address.

PNG G: Pointer to New Group

Builds a pointer to the first byte of the next group to be added to roll G, where G is the roll number, and places the pointer in W0; the BOTTOM of the WORK roll is increased by four.

POC G: Place on Code

The data located at storage address G+4 and following is to be moved to the CODE roll. The number of half-words to be moved is stored in location G and is an even number. A word containing a special value in the first two bytes and the number of <u>words</u> of data in the last two bytes is first placed on the CODE roll. The indicated data is then moved to the CODE roll, and the BOTTOM of the CODE roll is increased by four for each word placed on the roll. A location counter is increased by the number of bytes of object code placed on the roll.

PST G: Precision Store

(MPAC 1 and MPAC 2) --> G and G+4, where G is a storage address. This instruction performs a doubleword store.

SWT G: Switch

Interchanges (W0) and (G), where G is a storage address; the BOTTOM of the WORK roll is unchanged.

ZER G: Zero

0 --> G, where G is a storage address.

ARITHMETIC AND LOGICAL INSTRUCTIONS

The following instructions are primarily designed to perform arithmetic and logical manipulations.

ADD G: Add

(G) + (W0) --> W0, where G is a storage address; the BOTTOM of the WORK roll is unchanged; hence, the initial contents of W0 are destroyed.

AFS G: Add Four to Storage

(G) + 4 --> G, where G is a storage address.

AND G: And

(G) AND (W0) --> W0; that is, a logical product is formed between (G) and (W0), and the result is placed in W0. The BOTTOM of the WORK roll is unchanged; hence, the initial contents of W0 are destroyed.

DIM G: Diminish

(G) - 1 --> G, where G is a storage address.

DIV G: Divide

(W0) / (G) --> G, where G is a storage address; the remainder, if any, from the division is lost; a true answer is returned if there is no remainder; the BOTTOM of the WORK roll is unchanged; hence, the initial contents of W0 are destroyed.

IOR G: Inclusive Or

The inclusive OR of (W0) and (G), where G is a storage location, is formed, and the result is placed in W0. The BOTTOM of the WORK roll is unchanged; hence, the initial contents of W0 are destroyed.

LLS G: Logical Left Shift

(W0) are shifted left G places; the result is left in W0; bits shifted out at the left are lost, and vacated bit positions on the right are filled with zeros.

LRS G: Logical Right Shift

(W0) are shifted right G places; the result is left in W0; bits shifted out at the right are lost, and vacated bit positions on the left are filled with zeros.

MPY G: Multiply

(G) * (W0) --> W0, where G is a storage address; the BOTTOM of the WORK roll is unchanged; hence, the initial contents of W0 are destroyed.

PSP G: Product Sign and Prune

The exclusive OR of (W0) and (G), where G is a storage location, replace the contents of G; the BOTTOM of the WORK roll is reduced by four, thus pruning W0.

SUB G: Subtract

(W0) - (G) --> W0, where G is a storage address; the BOTTOM of the WORK roll is unchanged; hence, the initial contents of W0 are destroyed.

TLY G: Tally

(G) + 1 --> G, where G is a storage address.

DECISION MAKING INSTRUCTIONS

These instructions inspect certain conditions and return either a true or false answer in the ANSWER BOX. Some of the instructions also transmit stored information from place to place.

CSA G: Character Scan with Answer

If G = (CRRNT CHAR), the scan arrow is advanced and a true answer is returned; otherwise, the scan arrow is not advanced and a false answer is returned.

LGA G: Load Group with Answer

The group from the BOTTOM of roll G, where G is the roll number and roll G has been flipped, is loaded into SYMBOL 1, 2, 3, DATA 0, 1, 2, 3, 4, and 5 (as many words as necessary); if the roll is empty or if the group is a marker symbol, a false answer is returned; otherwise, a true answer is returned; the BOTTOM of roll G is reduced by group size.

MOA G: Move off with Answer

If roll G, where G is the roll number, is empty, a false answer is returned. Otherwise, the BOTTOM of roll G is reduced by four, pruning the word moved; the BOTTOM of the WORK roll is increased by four; a true answer is returned.

QSA G: Quote Scan with Answer

If the quotation mark (sequence of characters) beginning at storage address G (the first byte in the quotation mark is the number of bytes in the quotation mark) is equal to the quotation mark starting at the scan arrow, advance the scan arrow to the next active character following the quotation mark, and return a true answer; otherwise, do not advance the scan arrow and return a false answer.

SAD G: Set on Address

If G = ADDRESS portion of the pointer in W0, return a true answer; otherwise, return a false answer.

SBP G: Search by Stats from Pointer

Search the roll specified by the pointer in W0, beginning with the group following the one specified by the pointer for a group which is equal to the group in the central items SYMBOL 1, 2, 3, etc., according to the group stats values stored at locations G+4 and G+8 (these values are in the same order as those in the group stats tables). The roll number multiplied by four is stored at location G. If a match is found, return a true answer, replace the pointer in W0 with a pointer to the matching group, and continue in sequence. If no match is found, return a false answer, prune the pointer in W0, and continue in sequence. This instruction is used to continue a search of a roll according to group stats values other than those normally used for the roll.

SBS G: Search by Stats

If the roll, whose number multiplied by four is in storage at location G, is empty, return a false answer. Otherwise, search that roll against the central items SYMBOL 1, 2, and 3 and DATA 0, 1, 2, 3, 4, and 5, as defined by the group stats values stored at locations G+4 and G+8 (these values are in the same order as those in the group stats tables); if a match is found, place a pointer to the matching group in W0, increase the BOTTOM of the WORK roll, and return a true answer; if no match is found, return a false answer. This instruction is used to search a roll according to group stats values other than those normally used for that roll.

SCE G: Set if Character Equal

If G = (CRRNT CHAR), return a true answer; otherwise, return a false answer; in neither case is the scan arrow advanced.

SCK G: Set on Character Key

If (CRRNT CHAR) displays any of the character keys of G, where G is a character code whose bit settings describe a group of characters, return a true answer; otherwise, a false answer is returned; in neither case is the scan arrow advanced.

SFP G: Search from Pointer

Search the roll specified by the pointer in W0, beginning with the group following the one specified by the pointer in W0, for a group which is equal to the group in SYMBOL 1, 2, 3, DATA 0, 1..., etc., by roll statistics. If a match is found, return a true answer, replace the pointer in W0 with a pointer to the matching group, and jump to G, where G must be a local address. If no match is found, return a false answer, prune the pointer in W0 (reduce the BOTTOM of the WORK roll by four), and continue in sequence.

SLE G: Set if Less or Equal

If (W0) ≤ (G), where G is a storage address, a true answer is returned; otherwise, a false answer is returned. The comparison made considers the two values to be signed quantities.

SNE G: Set if Not Equal

If (W0) ≠ (G), where G is a storage address, a true answer is returned; otherwise, a false answer is returned.

SNZ G: Set if Nonzero

If (G) ≠ 0, where G is a storage address, return a true answer; otherwise, return a false answer.

SOP G: Set on Operator

If G = OPERATOR portion of the pointer in W0, return a true answer; otherwise, return a false answer.

SPM G: Set on Polish Mode

If the mode portion of the TAG field of the (G) = the mode portion of the TAG field of the pointer in P1, where G is a storage addess, return a true answer; otherwise, return a false answer.

SPT G: Set on Polish Tag

If the TAG field of the (G) = the TAG field of the pointer in P1, where G is a storage address, return a true answer; otherwise, return a false answer.

SRA G: Search

If roll G, where G is the roll number, is empty, return a false answer; otherwise, search roll G against the central items SYMBOL 1, 2, and 3 and DATA 0, 1, 2, 3, 4, and 5, as defined by the roll statistics; if a match is found, place a pointer to the matching group in W0, increase the BOTTOM of the WORK roll, and return a true answer; if no match is found, return a false answer.

SRD G: Set if Remaining Data

If roll G, where G is the roll number, is not empty, return a true answer; otherwise, return a false answer.

STA G: Set on Tag

If the TAG portion of (G) = the TAG portion of the pointer in W0, where G is a storage address, return a true answer; otherwise, return a false answer.

STM G: Set on Tag Mode

If the mode portion of the TAG field of the (G) = the mode portion of the TAG field of the pointer in W0, where G is a storage address, return a true answer; otherwise, return a false answer.

JUMP INSTRUCTIONS

The following instructions cause the normal sequential operation of the POP instructions to be altered, either unconditionally or conditionally. See the sections "Labels" and "Assembly and Operation" in this Appendix for further discussion of jump instructions.

CSF G: Character Scan or Fail

If G = (CRRNT CHAR), advance the scan arrow to the next active character; otherwise, jump to SYNTAX FAIL.

JAF G: Jump if Answer False

If (ANSWER BOX) = false, jump to G, where G is either a global or a local address; otherwise, continue in sequence. One of two operation codes is produced for this instruction depending on whether G is a global or local label.

JAT G: Jump if Answer True

If (ANSWER BOX) = true, jump to G, where G is either a global or a local address; otherwise, continue in sequence. One of two operation codes is produced for this instruction depending on whether G is a global or a local label.

JOW G: Jump on Work

If (W0) = 0, decrease the BOTTOM of the WORK roll by four and jump to G, where G is either a global or a local address; otherwise, reduce word 0 by one, --> W0, and continue in sequence. One of two operation codes is produced for this instruction, depending on whether G is a global or a local label.

JPE G: Jump and Prepare for Error

The following values are saved in storage: the location of the next instruction, the last character count, the BOTTOM of the EXIT roll, and the BOTTOM of the WORK roll. The JPE FLAG is set to nonzero, and a jump is taken to G, which may only be a local address.

JRD G: Jump Roll Down

This instruction manipulates a pointer in W0. If the ADDRESS field of that pointer is equal to 0 (pointing to the word preceding the beginning of a reserved area), the ADDRESS field is increased to four. If the ADDRESS field of the pointer is equal to any legitimate value within the roll, it is increased by group size. If the ADDRESS field of the pointer indicates a location beyond the BOTTOM of the roll, the pointer is pruned (the BOTTOM of the WORK roll is reduced by four), and a jump is made to the location G, which must be a global address.

JSB G: Jump to Subroutine

Return information is placed on the EXIT roll; jump to G, which is a global address.

JUN G: Jump Unconditional

Jump to G, which is either a global or a local address. One of two operation codes is produced for this instruction, depending on whether G is a global or a local label.

QSF G: Quote Scan or Fail

If the quotation mark (sequence of characters) beginning at storage address G (the value of the first byte in the quotation mark is the number of bytes in the quotation mark) is equal to the quotation mark starting at the scan arrow, advance the scan arrow to the first active character beyond the quotation mark; otherwise, jump to SYNTAX FAIL.

XIT : Exit

Exit from the interpreter; the code which follows is written in assembler language.

ROLL CONTROL INSTRUCTIONS

These instructions are concerned with the control of the rolls used in the compiler.

POW G: Prune off Work

Reduce the BOTTOM of the WORK roll by four times G, where G is an integer, thus pruning G words off the WORK roll.

REL G: Release

Restore roll G, where G is the roll number, to the condition preceding the last reserve; this sets BOTTOM to (TOP) reduced by four if the roll is reserved, or to (BASE) if the roll is not reserved; TOP is set to the value it had before the reserve.

RSV G: Reserve

Reserve roll G, where G is the roll number, by storing (TOP) - (BASE) on the roll, increasing BOTTOM by four, and setting TOP to (BOTTOM); this protects the area between BASE and TOP, and allows ascending addresses from TOP to be used as a new, empty roll.

CODE PRODUCING INSTRUCTIONS

These POP instructions construct object module code on the CODE roll. Each object module instruction constructed results in the placing of a 2-word group on the CODE roll. The instruction generated, in binary, is left justified in this group. In the case of halfword instructions, the remainder of the first word is filled with zero. The second word contains a pointer to the instruction operand, except in the case of 6-byte instructions when the last two bytes of the group contain the value zero.

BID G: Build Instruction Double

The instruction indicated by G, where G is an instruction number which indicates the exact instruction to be generated, is built on the CODE roll, where W0 contains a pointer to the first operand and W1 contains a pointer to the second operand. The BOTTOM of the CODE roll is increased by eight. The BOTTOM of the WORK roll is reduced by eight; thus, both pointers are pruned. A location counter is increased by one for each byte of the instruction.

BIM G: Build Instruction by Mode

The instruction indicated by G, where G is an instruction number which indicates the <u>class</u> of the instruction only. For example, LOAD INSTR as opposed to LE INSTR is built on the CODE roll, where W0 contains a pointer to the second operand. A pointer to the accumulator which holds the first operand is contained in the variable CRRNT ACC. The instruction mode is determined by inspecting the TAG fields of the pointers; the BOTTOM of the CODE roll is increased by eight; the BOTTOM of the WORK roll is reduced by four, thus pruning the pointer. A location counter is increased by one for each byte of the generated instruction.

BIN G: Build Instruction

The instruction indicated by G, where G is an instruction number which indicates the exact instruction to be built, is constructed on the CODE roll. The WORK roll holds from zero to three words of information required for producing the instruction. For instructions requiring no operands, nothing appears on the WORK roll. For instructions requiring one operand, a pointer to that operand appears in W0. For two operand instructions, a pointer to the first operand appears in W0 and a pointer to the second operand is in W1. For input/output instructions, W1 holds a constant which becomes part of the instruction. For storage-to-storage move instructions, W2 holds the length. The BOTTOM of the CODE roll is increased by eight to reflect the addition of the group. The BOTTOM of the WORK roll is reduced by four for each word of information found on that roll; thus, all the information is pruned. A location counter is increased by one for each byte of the instruction.

ADDRESS COMPUTATION INSTRUCTIONS

The POP instructions whose G fields require storage addresses may be used to refer to WORK roll groups, provided the storage address of the desired group is first computed. This computation must be performed at execution time, since the location of W0, for example, varies as the program is operated. The instructions in this category perform these computations and jump to the appropriate POP, which then operates using the computed address.

W0P G: W0 POP

Compute the address of the current W0 and jump to the POP indicated by G, where G is a POP instruction which normally accepts a storage address in its G field.

W1P G: W1 POP

Compute the address of the current W1 and jump to the POP indicated by G, where G is a POP instruction which normally accepts a storage address in its G field.

W2P G: W2 POP

Compute the address of the current W2 and jump to the POP indicated by G, where G is a POP instruction which normally accepts a storage address in its G field.

W3P G: W3 POP

Compute the address of the current W3 and jump to the POP indicated by G, where G is a POP instruction which normally accepts a storage address in its G field.

W4P G: W4 POP

Compute the address of the current W4 and jump to the POP indicated by G, where G is a POP instruction which normally accepts a storage address in its G field.

INDIRECT ADDRESSING INSTRUCTION

Indirect addressing is provided for POP instructions whose address fields normally require storage addresses by means of the following instruction.

IND G: Indirect

The address contained in the storage address INDIRECT BOX is transmitted to the POP indicated by G, where G is a POP instruction which requires a storage address in its G field, and a jump is made to that POP. The POP "G" operates in its normal fashion, using the transmitted address.

LABELS

In the POP language, storage locations containing instructions or data may be named with two types of labels, global labels and local labels. Global labels are unique within each phase of the compiler (but not from one phase to another); these labels may be referred to from any point in the phase. Local labels are also unique within each phase (but not between phases); however, these labels may be referred to only within the global area (that is, the area between two consecutive global labels) in which they are defined.

GLOBAL LABELS

The global labels which appear on a System/360 assembler listing of the compiler are distinguished from local labels in that the global labels do not begin with a pound sign. Most of the global labels are of the form Gdddd, where each d is a decimal digit and the 4-digit value dddd is unique for the global label. Labels of this form are generally assigned in ascending sequence to the compiler routines. All remaining global labels are limited to a length of seven characters.

In contrast, the routine and data names used throughout this publication are limited only to a length of 30 characters. A comment card containing the long name used here precedes the card on which each global label is defined. In addition, the longer name appears as a comment on any card containing a POP instruction which refers to the global label.

Example:

```
              G0336     STA GEN FINISH
G0336 IEYMOA G0494 MOA DO LOOPS OPEN ROLL
                 .
                 .
                 .
```

Explanation: The second card shown defines the global label G0336. The first card, a comment card, indicates the longer name of the routine, STA GEN FINISH. The second card contains a reference to the label G0494; the longer form of this label is DO LOOPS OPEN ROLL, as indicated by the comment.

Occasionally, several comment cards with identical address fields appear in sequence on the listing. This occurs when more than one long label has been applied to a single instruction or data value. The long labels are indicated in the comments fields of the cards.

Example:

```
*              ACTEST    AC TEST

*              ACTEST    TESTAC

ACTEST IEYSOP G0504 SOP FL AC OP MARK
          .
          .
          .
```

Explanation: The three cards shown define the global label ACTEST. One long form of this label is AC TEST, as indicated by the comment on the first card. The second card indicates that the name TESTAC has also been applied to this location, and that it also corresponds to ACTEST.

LOCAL LABELS

All local labels consist of a pound sign followed by six decimal digits. If the preceding global label is of the form Gdddd, the first four digits are identical to those in the global name. The remaining two digits of the local label do not follow any particular sequence; they are, however, unique in the global area.

The local label is defined by its appearance in the name field of a card containing a POP or assembler language instruction.

Example:

```
*              G0268    PROCESS SCALAR ROLL
G0268   IEYSRD G0432 SRD SCALAR ROLL
          .
          .
          .
#026811 IEYJOW #026821
#026802 IEYITA G0359 ITA CED TAG MARK
```

Explanation: The global label G0268 is defined by the second card in the sequence shown. The next two cards define, respectively, the local labels #026811 and #026802. In addition, the third card in the sequence contains a reference to the local label #026821, which is presumably defined elsewhere within the global area shown here.

ASSEMBLY AND OPERATION

The compiler is assembled with each POP instruction defined as a macro. Unless "Quick Link" output has been designated to the macro by means of the assembler instruction SETC 'QLK', the resulting code consists of two 1-byte address constants per POP instruction. This 16-bit value represents an 8-bit numeric operation code and an 8-bit operand or relative address.

The definition of the 8-bit operand or relative address varies according to the POP instruction used. Roll numbers appear in this field for instructions requiring them. For instructions which refer to storage locations relative to CBASE (see "Compiler Arrangement and General Register Usage") or to other base addresses, the word number relative to the appropriate base is used. The format for jump instructions is discussed in the following paragraphs.

When Quick Link is specified, machine language instructions are generated for the following POP instruction. (See "Assembler Language References to POP Subroutines.")

POP INTERPRETER

The assembled POP code is interpreted by a short machine language routine, POP SETUP, which appears with the POP subroutines at the beginning of the compiler.

POP SETUP inspects each pair of address constants in sequence, and, using the 8-bit operation code as an index into the POP jump table, a table which correlates operation codes for the POPs with the addresses of the POP subroutines, transfers control to the appropriate POP subroutine.

Thus, on encountering the hexadecimal value 081A, POP SETUP indexes into the POP jump table (labeled POPTABLE) at the eighth byte, counting from zero. The value found at this location is 0158 (hexadecimal); this is the address, relative to the base of the POP jump table, of the POP subroutine for the POP numbered 08 (IEYSUB). When this value is added to the beginning address of the POP jump table, the absolute address of IEYSUB is produced, and POP SETUP performs a branch to that location.

IEYSUB then operates, using the relative address 1A (which it finds in general register 7, ADDR), and returns via POPXIT, register 6; in this case the return is to POP SETUP, which then continues with the next POP in sequence. The register POPADR is used to keep track of the location of the POP being executed.

This sequential operation can be interrupted by means of POP jump (branch) instructions, which cause an instruction other than the next in sequence to be operated next. The XIT POP instruction also alters the sequence by causing the interpreter to release control, performing a branch to the assembler language instruction following the XIT. This device is employed to introduce assembler language coding into the compiler routines when this is more efficient than the use of POPs. Assembler language sequences sometimes terminate with a branch to POP SETUP, so that it may resume the execution of POP instructions.

ASSEMBLER LANGUAGE REFERENCES TO POP SUBROUTINES

In some of the routines of the compiler, the operation of POP SETUP is bypassed by assembler language instructions which make direct reference to the POP subroutines. In these sequences, a pair of machine language instructions performs the function of a single POP instruction. For example, the instructions

```
LA    ADDR,ONE-CBASE(0,0)
BAL   POPXIT,FETQ
```

accomplish the function of the POP instruction

```
IEYFET ONE
```

but bypass the operation of POP SETUP. The IEYFET routine, (referred to by its label FETQ) returns, via POPXIT, to the next instruction. Note that the first instruction of the pair sets ADDR to the correct value for the operand of the IEYFET operation; this would be done by POP SETUP if it interpreted IEYFET ONE.

GLOBAL JUMP INSTRUCTIONS

The labels referred to in POP global jump instructions, jump instructions which branch to global labels, always end with the character J. These global labels refer to the global jump table, a table whose fullword entries contain the relative addresses of global labels which are the targets of branches. Each phase of the compiler has a global jump table. The table is labeled JUMP TABLE.

References in POP global jump instructions to the global jump table are assembled as relative word addresses in that table. Each entry in the table contains the address, relative in bytes to CBASE, of the label whose spelling is identical to that of the global jump table entry except that it does not include the terminal J.

Thus, the instruction IEYJUN G0192J is assembled as 5002, for example, where the global jump table begins:

G0075J	5A0
G0111J	752
G0192J	B02
	.
	.
	.

On encountering this instruction, POP SETUP loads its address field (02), multiplied by four (08), into the register ADDR. It then jumps to the POP subroutine for IEYJUN.

The IEYJUN subroutine uses ADDR as an index into JUMP TABLE, finding the value B02. This value is placed in the register TMP and a branch is made to the location defined by the sum of the contents of TMP and the contents of CONSTR, which holds the location CBASE. Thus, if the location CBASE is 10B0, the location branched to is 1BB2, the location of the routine labeled G0192, and the instruction at that location is operated next.

Since the POP subroutines for global jumps branch directly to the target location, the instruction at that location must be a machine language instruction rather than a POP. Moreover, all jump target routines which contain local jumps must reset POPADR to reflect the new location. Thus, routines which are jump targets and which are written in POPs begin with the instruction

```
BALR  POPADR, POPPGB
```

which sets POPADR to the location of the first POP instruction in the routine and branches to POP BASE, the address of which is held in POPPGB. At POP BASE, the contents of POPADR are saved in LOCAL JUMP BASE, POPXIT is set to the beginning location of POP SETUP, and POP SETUP begins operating. For the sake of brevity, this instruction is coded as

```
BALR  A,B
```

in some routines.

Routines in which the POP instructions have been replaced by pairs of assembler language instructions and which contain local jumps begin with the instruction

```
BALR  A,0
or
BALR  POPADR,0
```

instead of the instruction given above, since the branch to POP SETUP is not desired.

Because global jump targets begin with this machine language code, it is not possible for POP instructions to continue in sequence into new global routines. When this operation is intended, an IEYXIT or an IEYJUN instruction terminates the first routine.

LOCAL JUMP INSTRUCTIONS

POP _local jump instructions_, jump instructions which transfer control out of the normal sequence to local labels, must occur in the same global area as the one in which the local label referred to is defined.

The address portions of POP local jump instructions are assembled to contain the distance in halfwords from the beginning of the global area plus two to the indicated local label. This value is a relative halfword address for the target, where the base used is the location of the first POP instruction in the global area.

Example:

Decimal Location	Label	Symbolic Instruction	Hexadecimal Instruction
100	G0245	BALR A,B	
102		IEYCLA G0566	062A
.			
.			
.			
120	#024503	IEYLGA G0338	9A12
.			
.			
.			
140		IEYJUN #024503	5809

Explanation: The local jump instruction illustrated at location 140 is assembled so that its address field contains the location of the label #024503 (120), relative in halfwords to the beginning location of the global area plus two (102). Thus, the address field of the IEYJUN instruction contains the value 09.

When the POP local jump instruction is interpreted, the contents of the location LOCAL JUMP BASE are added to the address field of the POP instruction to produce the absolute address of the jump target. LOCAL JUMP BASE is set to the beginning address of the global area plus two as a result of the BALR instruction which begins the global routine; this function is performed at POP BASE, as described in "Global Jump Instructions."

When local jumps are performed directly in machine language, the relative addressing described above is also used; in this case, however, the base address is in the register POPADR as a result of the BALR instruction heading the routine.

POP instruction mnemonics are listed in Table 8.

Table 8. POP Instruction Cross-Reference List

Mnemonic	Hex	Instruction Group	Mnemonic	Hex	Instruction Group
ADD	04	Arithmetic/Logical	LGA	9A	Decision Making
AFS	BC	Arithmetic/Logical	LGP	80	Transmissive
AND	B4	Arithmetic/Logical	LLS	98	Arithmetic/Logical
APH	A4	Transmissive	LRS	B6	Arithmetic/Logical
ARK	86	Transmissive	LSS	B0	Transmissive
ARP	0E	Transmissive	MOA	5C	Decision Making
ASK	12	Transmissive	MOC	9E	Transmissive
ASP	14	Transmissive	MON	5E	Transmissive
BID	7E	Code Producing	MPY	0A	Arithmetic/Logical
BIM	7C	Code Producing	NOG	1E	Transmissive
BIN	7A	Code Producing	NOZ	3E	Transmissive
BOP	60	Transmissive	PGO	22	Transmissive
CAR	1A	Transmissive	PGP	9C	Transmissive
CLA	06	Transmissive	PLD	90	Transmissive
CNT	1C	Transmissive	PNG	20	Transmissive
CPO	B2	Transmissive	POC	94	Transmissive
CRP	62	Transmissive	POW	16	Roll Control
CSA	24	Decision Making	PSP	92	Arithmetic/Logical
CSF	26	Jump	PST	8C	Transmissive
DIM	8E	Arithmetic/Logical	QSA	2A	Decision Making
DIV	B8	Arithmetic/Logical	QSF	2C	Jump
EAD	2E	Transmissive	REL	64	Roll Control
EAW	18	Transmissive	RSV	66	Roll Control
ECW	18	Transmissive	SAD	6A	Decision Making
EOP	30	Transmissive	SBP	BA	Decision Making
ETA	32	Transmissive	SBS	96	Decision Making
FET	34	Transmissive	SCE	28	Decision Making
FLP	46	Transmissive	SCK	6E	Decision Making
FRK	84	Transmissive	SFP	A6	Decision Making
FRP	10	Transmissive	SLE	70	Decision Making
FTH	AE	Transmissive	SNE	74	Decision Making
IAD	36	Transmissive	SNZ	72	Decision Making
IND	D2	Indirect Addressing	SOP	6C	Decision Making
IOP	38	Transmissive	SPM	A2	Decision Making
IOR	8A	Arithmetic/Logical	SPT	AC	Decision Making
ITA	3A	Transmissive	SRA	76	Decision Making
ITM	A0	Transmissive	SRD	78	Decision Making
JAF	4A	Jump (global)	STA	68	Decision Making
	56	Jump (local)	STM	3C	Decision Making
JAT	48	Jump (global)	SUB	08	Arithmetic/Logical
	54	Jump (local)	SWT	0C	Transmissive
JOW	4E	Jump (global)	TLY	42	Arithmetic/Logical
	5A	Jump (local)	W0P	C8	Address Computation
JPE	52	Jump	W1P	CA	Address Computation
JRD	82	Jump	W2P	CC	Address Computation
JSB	50	Jump	W3P	CE	Address Computation
JUN	4C	Jump (global)	W4P	DO	Address Computation
	58	Jump (local)	XIT	44	Jump
LCE	00	Transmissive	ZER	40	Transmissive
LCF	AA	Transmissive			
LCT	A8	Transmissive			

This appendix describes each of the rolls used in the compiler, giving the group size, the structure and content of the information in the group, and the roll number. Each roll is described as it appears in each of the phases of the compiler. This information is useful in observing the actions taken by the various phases, since a significant portion of the work performed by the compiler is the construction and manipulation of information on rolls.

The rolls are ordered in this appendix as they are in storage, by roll number. In some cases, a single, number is assigned to several rolls. In these cases, the rolls with identical numbers are presented chronologically, and the overlay of one roll on another indicates that the previous roll is no longer required when the new roll is used. The group stats values for rolls with the same number are always identical.

The roll number is the entry number in the roll statistics tables for the appropriate set of statistics; that is, the roll number multiplied by four is the relative address of the correct entry in the group stats, BASE, BOTTOM, and TOP tables.

ROLL 0: LIB ROLL

This roll contains one group for every name by which a library subprogram can be referred to in the source module. The roll is contained in IEYROL and remains unchanged in size and in content throughout compilation.

The group size for the LIB roll is twelve bytes. Each group has the form:

4 bytes

<---------------subprogram---------------			
--------name--------->		TAG	0
TAG	flag	no. arguments	

The TAG appearing in the seventh byte of the group provides the mode and size of the FUNCTION value, if the subprogram is a FUNCTION. The TAG in byte 9 indicates the mode and size of the arguments to the subprogram. For FUNCTIONs, the flag (byte 10) indicates either in-line (including which generation routine must be used) or that a call is to be generated (when the flag is equal to zero).

This roll is used and then destroyed by Allocate.

ROLL 1: SOURCE ROLL

This roll holds source module statements while they are being processed during the operations of Parse. The roll is not used by any later phase of the compiler.

Source statements appear on this roll one card column per byte. Thus, each card of a source statement occupies 20 groups on the roll. The group size is four bytes. The statement

A(I,J)=B(I,J)*2+C(I,J)**2

would therefore appear on the SOURCE roll as:

4 bytes

b	b	b	b
b	b	A	(
I	,	J	)
=	B	(	I
,	J	)	*
2	+	C	(
I	,	J	)
*	*	2	b
b	b	b	b
	. . .		
b	b	b	b

where b stands for the character blank, and a total of 20 words is occupied by the statement.

ROLL 2: IND VAR ROLL

This roll holds a pointer to the induction variable (the DO variable) used in each DO loop. The pointer specifies the appropriate group on the SCALAR roll. Each pointer is placed on the roll by Parse as the DO loop is encountered in the source module. When the loop is closed, the pointer is deleted.

The roll is not used in subsequent phases of the compiler. The group size for the IND VAR roll is four bytes.

ROLL 2: NONSTD SCRIPT ROLL

This roll exists only in Unify; the information held on it is taken from the SCRIPT roll. The group size for the NONSTD SCRIPT roll is variable, with a minimum of 20 bytes. Each group on the roll describes an array reference.

The format of the NONSTD SCRIPT roll group is:

4 bytes

traits	frequency
pointer to ARRAY REF roll	
pointer to the ARRAY roll	
offset	
induction variable coefficient	
. . .	
induction variable coefficient	

where the first byte of the first word contains the trait, which indicates either joined or not joined; the value of this item is always zero (not joined) for this roll. The joined value indicates that the subscript described must appear in a general register at the time of the reference. The remaining three bytes of the first word indicate the number of times this subscript expression is used.

The next two words contain pointers to rolls holding information on the array and the array reference to which this group refers. The fourth word holds the array offset; this value accounts for element size and includes all modification due to constant subscripts. The remaining words hold the induction variable coefficient used in this reference for each loop in the nest, beginning with nest level one (the outermost loop) and ending with the highest nest level at this array reference.

ROLL 3: NEST SCRIPT ROLL

This roll contains information concerning array references in nested DO loops. The information for this roll is taken from the SCRIPT roll as each nest of loops is encountered, one nest at a time. The roll exists only in Unify. The group size of the NEST SCRIPT roll is variable with a minimum of 20 bytes. The format of the NEST SCRIPT roll is as follows:

4 bytes

traits	frequency
pointer to ARRAY REF roll	
pointer to the ARRAY roll	
offset	
induction variable coefficient	
. . .	
induction variable coefficient	

where the first byte of the first word indicates joined or not joined. The remaining three bytes of the first word indicate the number of times that this subscript expression is used. The next two words of the group contain pointers to rolls which hold information on the array and the array reference to which this entry refers. The fourth word holds the actual adjusted offset for this array reference. The last words of the group contain the coefficients of induction variables used in the array reference, beginning with the nest level one variable and ending with the highest nest level.

ROLL 4: POLISH ROLL

This roll is used to hold the Polish notation generated by Parse, one statement at a time. (The Polish notation is moved to the AFTER POLISH roll at the end of each statement.) Therefore, the roll contains pointers, drivers, and an occasional constant. The terms P0 and P1 are used to refer to the bottom and next-to-bottom groups on the POLISH roll, respectively.

In Gen, the Polish notation is moved back onto the POLISH roll from the AFTER POLISH roll, one statement at a time. It is used in the production of object code.

The group size for the POLISH roll is four bytes. The format of the Polish notation which appears on this roll is described completely in Appendix C.

The POLISH roll is not used in the other phases of the compiler and no information is left on it through these phases.

ROLL 4: LOOP SCRIPT ROLL

This roll contains information on array references encountered in the source module. The group size for the LOOP SCRIPT roll is variable; the minimum is 20 bytes. Its format is:

4 bytes

traits	frequency
pointer to the ARRAY REF roll	
pointer to the ARRAY roll	
offset	
induction variable coefficient	
. . .	
induction variable coefficient	

All items are the same as described for the NEST SCRIPT roll (roll 3).

The LOOP SCRIPT roll exists only in Unify. It is used by this phase to further separate subscripts into two categories: standard, those which must appear in general registers at the time of reference, and nonstandard.

ROLL 5: LITERAL CONST ROLL

This roll holds literal constants, which are stored as plexes. The group size for the LITERAL CONST roll is variable. Each plex has the form:

4 bytes

			n
			k
c_1	c_2	c_3	c
. . .			
c			

where n is the number of words in the plex, exclusive of the word which holds n, k is the number of bytes in the literal constant, and c (the k character) may fall in any byte of the last word of the plex. If the literal constant appeared in a source module DATA or PAUSE statement, the high order bit of the second word of the plex (k) is set to one; otherwise, it is zero.

Entries are made on the LITERAL CONST roll only during Parse. It is used to hold the literal constants throughout the compiler; its format, therefore, does not vary.

ROLL 7: GLOBAL SPROG ROLL

In Parse this roll holds the names of all SUBROUTINEs and non-library FUNCTIONs referred to in the source module. It also holds the names of all subprograms listed in EXTERNAL statements in the source module, including library subprograms. In addition, the compiler itself generates calls to the library exponentiation routines; the names of these routines are entered on the GLOBAL SPROG roll.

The group size for the GLOBAL SPROG roll is eight bytes. All groups placed on the GLOBAL SPROG roll by Parse have the following format:

4 bytes

<--------------subprogram----------------		
--------name-------->	TAG	0

The TAG appearing in the seventh byte of the group indicates the mode and size of the FUNCTION value for FUNCTIONs; it has no meaning for SUBROUTINEs.

In Allocate, the information on the roll is altered to:

4 bytes

ESD number	displacement
base table pointer	

The ESD number is the one assigned to the subprogram. The displacement and the base table pointer, taken together, indicate the location assigned by Allocate to hold the address of the subprogram. The specified BASE TABLE roll group holds an address; the displacement is the distance in bytes from that address to the location at which the address of the subprogram will be stored in the object module.

In Gen, the GLOBAL SPROG roll is used in the construction of object code, but it is not altered.

In Exit, the roll is used in the production of RLD cards, but is not altered.

ROLL 8: FX CONST ROLL

This roll holds the fullword integer constants which are used in the source module or generated by the compiler. The constants are held on the roll in binary, one constant per group. The group size for the FX CONST roll is four bytes.

The format of the FX CONST roll is identical for all phases of the compiler. The roll remains in the roll area for all phases, even though it is not actually used in Allocate and Unify.

ROLL 9: FL CONST ROLL

This roll holds the single-precision real (floating point) constants used in the source module or generated by the compiler. Constants are recorded on the roll in binary (floating point format), each constant occupying one group. The group size for the FL CONST roll is four bytes.

The FL CONST roll remains in the roll area for all phases of the compiler, although it is not actually used in Allocate or Unify. The format of this roll is identical for all phases.

ROLL 10: DP CONST ROLL

This roll holds the double-precision (8-byte) real constants used in the source module or defined by the compiler.

The constants are recorded in binary (double-precision floating point format), one constant per group. The group size for the DP CONST roll is eight bytes.

The DP CONST roll is present in this format through all phases of the compiler.

ROLL 11: COMPLEX CONST ROLL

This roll holds the complex constants of standard size (eight bytes) used in the source module or generated by the compiler. Each complex constant is stored on the roll as a pair of 4-byte binary floating-point numbers, the first represents the real part of the constant and the second represents the imaginary part.

The COMPLEX CONST roll exists in the format described above for all phases of the compiler. The group size is eight bytes.

ROLL 12: DP COMPLEX CONST ROLL

This roll holds the complex constants of optional size (16 bytes) which are used in the source module or generated by the compiler. Each constant is stored as a pair of double-precision binary floating point values. The first value represents the real part of the constant; the second value represents the imaginary part. The group size for the DP COMPLEX CONST roll is 16 bytes.

The DP COMPLEX CONST roll exists in this format for all phases of the compiler.

ROLL 13: TEMP NAME ROLL

This roll is used as temporary storage for names which are to be placed on the ARRAY or EQUIVALENCE roll. The group size for the TEMP NAME roll is eight bytes. The format of the group is:

4 bytes

<----------------name		
------------------>	TAG	0

The TAG appearing in the seventh byte of the group indicates, in the format of the TAG field of a pointer, the mode and size of the variable.

The TEMP NAME roll is used only during Parse and Allocate; it does not appear in any later phase of the compiler.

ROLL 13: STD SCRIPT ROLL

The information on this roll pertains to array references for which the subscript expression must appear in a general register (joined).

The roll exists only in Unify and the information contained therein is taken from the SCRIPT roll. Its structure and contents are identical to those of the NONSTD SCRIPT roll (roll 2) with the exception that the traits on this roll always indicate joined. The group size is variable with a minimum of 20 bytes.

ROLL 14: TEMP ROLL

This roll is used as temporary storage in Parse and is not used in any later phase of the compiler. The group size for the TEMP roll is four bytes.

This roll is used as temporary storage for error information in Parse and is not used in the other phases of the compiler. The group size for the ERROR TEMP roll is four bytes.

ROLL 15: DO LOOPS OPEN ROLL

In Parse, as DO statements are encountered, pointers to the target labels of the DO statements are placed on this roll. When the target statement itself is encountered, the pointer is removed.

In Allocate, the roll may contain some pointers left from Parse; if any are present, they indicate unclosed DO loops; the roll is checked by Allocate and any information on it is removed.

This roll is not used after Allocate. The group size for the DO LOOPS OPEN roll is four bytes.

ROLL 15: LOOPS OPEN ROLL

This roll contains the increment and terminal values of the induction variable used in a DO loop and transfer data for the reiteration of the loop.

Gen creates the roll by establishing an entry each time a DO loop is encountered. The information is used in generating the object code. As a loop is closed, the bottom group from the LOOPS OPEN roll is pruned.

The group size is four bytes. Four groups are placed in the roll at one time. The configuration of a LOOPS OPEN roll group is as follows:

4 bytes

pointer to n_3 (increment)
pointer to n_2 (terminal value)
LOOP DATA pointer
pointer to return point made label

ROLL 16: ERROR MESSAGE ROLL

This roll is used only in Parse. It is used during the printing of the error messages for a single card of the source module. Each group holds the beginning address of an error message required for the card. It is used in conjunction with the ERROR CHAR roll, whose corresponding group holds the column number in the card with which the error is associated. The group size for the ERROR MESSAGE roll is four bytes.

ROLL 16: TEMP AND CONST ROLL

This roll is produced in Gen and is used in Gen and Exit. It holds all constants required for the object module and zeros for all temporary storage locations required in the object module.

Binary constants are moved to this roll by Gen from the various CONST rolls. This roll becomes the object module's temporary storage and constant area. The group size for the TEMP AND CONST roll is four bytes.

ROLL 17: ERROR CHAR ROLL

This roll is used only during Parse, and is not used in any subsequent phase of the compiler.

While a single source module card and its error messages are being prepared for output, this roll holds the column number with which an error message is to be associated. The address of the error message is held in the corresponding group on the ERROR MESSAGE roll. The group size for the ERROR CHAR roll is four bytes.

ROLL 17: ADCON ROLL

This roll is used only in Exit, and is not used in previous phases of the compiler. It holds address constants, the locations at which they are to be stored, and relocation information. The group size is 16 bytes. The first word of the group holds an area code, indicating the control section in which the constant exists. The second word of the group holds the address into which the constant is to be placed; the third holds the constant. The last word of the group indicates the relocation factor (ESD number) to be used for the constant.

ROLL 18: INIT ROLL

The group size for the INIT roll is eight bytes. The roll is initialized in Parse, and used and destroyed in Allocate. Each group on the roll holds the name of a scalar variable or array listed in the INIT option of a DEBUG statement in the source module. The format of the group is:

```
                 4 bytes
┌─────────────────────────────────────────┐
|<───────────variable name───────────────|
├────────────────────┬────────────────────┤
|───────────────────>|         0          |
└────────────────────┴────────────────────┘
```

ROLL 18: DATA SAVE ROLL

This roll is used only in Gen, where it holds the Polish notation for portions of DATA statements or Explicit specification statements which refer to control sections different from the control section presently in process. The roll is a temporary storage location for this information, since data values are written out for one control section at a time. The group size is four bytes.

ROLL 19: XTEND LABEL (XTEND LBL) ROLL

This roll is used only by Parse. It holds the pointers to the LABEL roll for all labels defined within the innermost DO loops that are possible extended range candidates. The group size of the XTEND LABEL roll is four bytes. Each group holds a pointer to the LABEL roll. The format of the group on the roll is:

```
 1 byte        3 bytes
┌────────┬──────────────────────────────────┐
|TAG     |LABEL roll pointer                |
└────────┴──────────────────────────────────┘
```

If the label is a possible re-entry point from the extended range of a DO loop, the TAG byte contains a X'05'. Otherwise, the TAG byte contains a X'00'.

ROLL 19: EQUIVALENCE TEMP (EQUIV TEMP) ROLL

This roll is used to hold EQUIVALENCE roll data temporarily in Allocate, and is not used in any other phase of the compiler. The group size for the EQUIVALENCE TEMP or EQUIV TEMP roll is twelve bytes. The format of the group on the roll is:

```
                 4 bytes
┌─────────────────────────────────────────┐
|<──────────────variable─────────────────|
├────────────────────┬────────────────────┤
|───────name───────>|         0          |
├────────────────────┴────────────────────┤
|                 offset                  |
└─────────────────────────────────────────┘
```

The offset is the relative address of the beginning of the variable within the EQUIVALENCE group (set) of which it is a member. This roll holds this information during the allocation of storage for EQUIVALENCE variables.

ROLL 20: XTEND TARGET LABEL (XTEND TARG LBL) ROLL

This roll is used only by Parse. The group size of the XTEND TARGET LABEL roll is four bytes. Each group holds a pointer to the LABEL roll for each label that appears in any transfer statement (e.g., GO TO, Arithmetic IF statements) within a DO loop. These groups indicate transfers out of an innermost DO loop and a possible extended range. The format of the group is the same as Roll 19, XTEND LABEL roll.

1 byte	3 bytes
TAG	LABEL roll pointer

If the TAG byte contains a X'40', this indicates that the target label also appears in a transfer statement outside the DO loop and may be a possible re-entry point (if the label is defined within the loop). Otherwise, the TAG byte contains a X'00'.

ROLL 20: EQUIVALENCE HOLD (EQUIV HOLD) ROLL

This roll is used to hold EQUIVALENCE roll data temporarily in Allocate, and is not used in any other phase of the compiler. The group size for the EQUIVALENCE HOLD roll is twelve bytes. The format of the group on the roll is:

4 bytes

<---------------variable-----------------	
-------name-------->	0
offset	

The offset is the relative address of the beginning of the variable within the EQUIVALENCE group (set) of which it is a member. This roll holds this information during the allocation of storage for EQUIVALENCE variables.

ROLL 20: REG ROLL

This roll contains information concerning general registers required in the execution of DO loops in the object module.

The group size of the REG roll is twelve bytes. The roll is used only in Unify. Each group has the following format:

4 bytes

traits	frequency
ARRAY REF pointer	
LOOP CONTROL pointer	

The frequency indicates how many times within a loop the register is used. The registers are symbolic registers that are converted to real registers and/or temporary storage locations. The pointer to the ARRAY REF roll is actually a thread which indicates each place that this register is required in the loop. The last word, the pointer to the LOOP CONTROL roll, designates where the register in question was initialized. (The particular information is contained in the second word of the entry on the LOOP CONTROL roll.)

ROLL 21: BASE TABLE ROLL

This roll is constructed by Allocate, and remains in the roll area for all remaining phases of the compiler. The BASE TABLE roll becomes the object module base table, which holds the base addresses used in referring to data in the object module.

The group size for this roll is eight bytes. One group at a time is added to this roll by Allocate. The first word holds the area code which indicates the relocation factor by which the base table entry must be modified at object time; each unique area code also defines an object module control section. The second word holds a relative address within the control section defined by the area code; this is the value which is in the corresponding base table entry prior to modification by the linkage editor.

The entire BASE TABLE roll is constructed by Allocate.

ROLL 22: ARRAY ROLL

This roll is used throughout the compiler to hold the required information describing arrays defined in the source module.

In Parse, the name and dimension information is added to the roll for each array definition encountered. The group size for the ARRAY roll is 20 bytes. The format of the group is:

4 bytes

<--------------array name----------------		
------------------>	TAG	0
ARRAY DIMENSION pointer		
number of elements		
array offset		

The TAG appearing in the seventh byte of the group indicates, in the format of the TAG field of a pointer, the mode and size of the array variable. The pointer in the third word of the group points to the beginning of the plex on the ARRAY DIMENSION roll, which describes the dimensions of the array. The number of elements in the array is a constant, unless the array has dummy dimensions; in the latter case, Parse puts a dummy pointer to a temporary location in this word of the group.

The array offset is the summation of the multipliers for the array subscripts. If the array dimensions are n1, n2,...n7, then the multipliers are 1, n1, n1*n2, n1*n2*n3, ...n1*n2*n3*n4*n5*n6, where the size of the element of the array is not considered. This value, after it is multiplied by the element size, is used as a subtractive offset for array references. The offset is placed on the roll as a constant unless the array has dummy dimensions; in the latter case, a dummy pointer to a temporary location is placed in the last word of the group.

In Allocate, the first two words of the ARRAY roll group are replaced with the following:

4 bytes

TAG	DBG/CEAD	displacement
base table pointer		

The TAG is unchanged, except in location, from Parse. The DBG/CEAD flag is logically

split into two hexadecimal values. The first of these indicates debug references to the variable; its value is 1 for INIT, 2 for SUBCHK, 0 for neither, and 3 for both. The second hexadecimal value is nonzero if the array is in COMMON, a member of an EQUIVALENCE set, used as an argument to a subprogram, or a dummy; it is zero otherwise. The displacement and the base table pointer, taken together, indicate the beginning address of the array. The base table pointer specifies the BASE TABLE roll group to be used in references to the array; the displacement is the distance in bytes from the address held in that group to the location at which the array begins. If the array is a dummy, the base table pointer is replaced by a pointer to the GLOBAL DMY roll group defining the array, and the displacement is zero.

The third, fourth, and fifth words of the ARRAY roll group are not modified by Allocate.

The ARRAY roll remains in storage throughout the compiler, and it is consulted, but not modified, by the phases following Allocate.

ROLL 23: DMY DIMENSION ROLL

This roll is used first in Allocate, where it holds pointers to the array definition and the entry statement with which dummy array dimensions are associated. The group size of the DMY DIMENSION roll is four bytes. Two groups are added to the roll at a time to accommodate this information; the format is:

4 bytes
ARRAY pointer
ENTRY NAMES pointer

In Gen, the DMY DIMENSION roll is used in the generation of temporary locations for the dummy dimensions. This operation is performed when code is being produced for the prologue with which the dummy dimension is associated.

The DMY DIMENSION roll is not used by later phases of the compiler.

ROLL 23: SPROG ARG ROLL

This roll becomes the subprogram argument list area of the object module. The roll is constructed by Gen and holds pointers to the arguments to subprograms in the order in which they are presented in the subprogram reference. These pointers may, therefore, point to the SCALAR, ARRAY, GLOBAL SPROG, or TEMP AND CONST rolls (the last roll holds arguments which are expressions or constants). The value zero is placed on this roll for arguments whose addresses are computed and stored in the object module argument list area.

The TAG fields of the pointers on this roll contain the value zero except for the TAG field of the last pointer for a single subprogram reference; this field contains the value 80.

The contents of the SPROG ARG roll are punched by Exit. The group size for the SPROG ARG roll is four bytes.

ROLL 24: ENTRY NAMES ROLL

In Parse, this roll holds all ENTRY names defined in the source subprogram, and pointers to the locations on the GLOBAL DMY roll at which the definitions of the dummy arguments corresponding to the ENTRY begin. The group size for the ENTRY NAMES roll is 16 bytes. The format of the group is:

4 bytes

<--------ENTRY name	---------
-------------------->	0
dummy pointer	
0	

The dummy arguments corresponding to the ENTRY are listed on the GLOBAL DMY roll in the order in which they are presented in the ENTRY statement.

In Allocate, the ENTRY NAMES roll is used in the check to determine that scalars with the same names as all ENTRYs have been set. A pointer to the scalar is placed in the fourth word of the group by this phase.

In Gen, during the production of the initialization code (the object module heading), the first word of the group is replaced by a pointer to the ADCON roll indicating the location of the prologue, and the second word is replaced by a pointer to the ADCON roll indicating the location of the epilogue. During the production of code for the prologue, the first pointer (the first word of the group) is replaced by a pointer to the ADCON roll which indicates the entry point for the ENTRY.

This roll is not required after the Gen phase.

ROLL 25: GLOBAL DMY ROLL

In Parse, each group on the roll contains the name of a dummy listed in a dummy argument list for the principle entry or for an ENTRY statement in a source subprogram. A flag also appears in each group which indicates whether the dummy is a "call by name" or a "call by value" dummy. The group size is eight bytes. The format of the group in Parse is:

```
                4 bytes
┌─────────────────────────────────────────┐
|<──────────────dummy name────────────────|
├────────────────────┬────────────────────┤
|───────────────────>|        flag        |
└────────────────────┴────────────────────┘
```

where the dummy name occupies the first six bytes of the group.

Label dummies, indicated by asterisks in the source module, are not listed on this roll. With this exception, however, the dummy lists from the source subprogram are entered on this roll as they appear in the source statements. The end of each dummy list is signaled by a marker symbol on the roll. Since each of the dummy lists is represented on the roll, the name of a single dummy may appear more than once.

In Allocate, the information in each group is replaced by:

```
                4 bytes
┌─────────┬──────────┬────────────────────┐
|   TAG   | DBG/flag |    displacement    |
├─────────┴──────────┴────────────────────┤
|           base table pointer            |
└─────────────────────────────────────────┘
```

where the base table pointer indicates the group on the BASE TABLE roll to be used for references to the dummy, and the displacement (in the third and fourth bytes) indicates the distance in bytes from the address stored in that BASE TABLE roll group to the location of the dummy. The "flag" occupies the second hexadecimal character of the second byte and is unchanged from Parse, indicating call by name if it is on. The first hexadecimal value in that byte indicates debug references to the variable; its value is 1 for INIT, 2 for SUBCHK, 0 for neither, and 3 for both. The TAG indicates the mode and size of the dummy.

The GLOBAL DMY roll is used but unmodified in Gen and Exit.

ROLL 26: ERROR ROLL

This roll is used only in Parse and holds the location within the statement of an error, and the address of the error message for all errors encountered within a single statement. As the statement is written on the source listing, the information in the ERROR roll groups is removed, leaving the roll empty for the processing of the next statement.

The group size is four bytes. Two groups are added to this roll at a time: (1) the column number of the error, counting from one at the beginning of the source statement and increasing by one for every card column in the statement, and (2) the address of the message associated with the particular error encountered.

ROLL 26: ERROR LBL ROLL

This roll is used only in Allocate, where it holds labels which are referred to in the source module, but which are undefined. These labels are held on this roll prior to being written out as undefined labels or unclosed DO loops. The group size for the ERROR LBL roll is four bytes.

ROLL 27: LOCAL DMY ROLL

This roll holds the names of the dummy arguments to a statement function while the statement function is being processed by Parse. The group size is eight bytes. The format of the group is:

```
                4 bytes
┌─────────────────────────────────────────┐
|<──────────────dummy name────────────────|
├────────────────────┬────────────────────┤
|───────────────────>|         0          |
└────────────────────┴────────────────────┘
```

The information is removed from the roll when the processing of the statement function is complete.

This roll does not appear in any subsequent phase of the compiler; however, pointers to it appear in the Polish notation produced by Parse and these pointers are, therefore, processed by Gen.

ROLL 28: LOCAL SPROG ROLL

In Parse, the roll holds the names of all statement functions as they are encountered in the source module. The group size for the LOCAL SPROG roll is eight bytes. The format of the group is:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<───────────stmt. function──────────────|
├───────────────────┬──────────┬──────────┤
|───────name───────>|   TAG    |    0     |
└───────────────────┴──────────┴──────────┘
```

The TAG appearing in the seventh byte of the group indicates, in the format of the TAG field of a pointer, the mode and size of the function value.

In Allocate, the first four bytes of each group are replaced by a pointer to the BRANCH TABLE roll group which has been assigned to hold the address of the statement function.

The LOCAL SPROG roll is used by Gen and Exit, but it is not modified in those phases.

ROLL 29: EXPLICIT ROLL

This roll is used in Parse and Allocate, where it holds the names of all variables defined by Explicit specification statements. The group size for the EXPLICIT roll is eight bytes. The format of the group in both phases is:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<───────────variable name───────────────|
├───────────────────┬──────────┬──────────┤
|──────────────────>|   TAG    |    0     |
└───────────────────┴──────────┴──────────┘
```

where the TAG (seventh byte) indicates the mode and size of the variable.

Groups are entered on this roll by Parse; the roll is consulted by Allocate, but not altered.

ROLL 30: CALL LBL ROLL

This roll is used only in Parse, where it holds pointers to the LBL roll groups defining labels which are passed as arguments in source module CALL statements. The pointers are held on this roll only temporarily, and are packed two pointers to a group. Pointers are added to the roll when the labels are found as arguments in CALL statements. The group size for the CALL LBL is eight bytes.

ROLL 30: ERROR SYMBOL ROLL

This roll is used only in Allocate, where it holds any symbol which is in error, in preparation for printing. The group size for the ERROR SYMBOL roll is eight bytes. The symbol (variable name, subprogram name) occupies the first six bytes of the group. The remaining two bytes are set to zero.

ROLL 31: NAMELIST NAMES ROLL

In Parse, this roll holds the NAMELIST names defined in the NAMELIST statement by the source module. The group size for the NAMELIST NAMES roll is twelve bytes. These groups are placed on the roll in the following format:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<──────────────NAMELIST─────────────────|
├───────────────────┬─────────────────────┤
|───────name───────>|          0          |
├───────────────────┴─────────────────────┤
|      pointer to NAMELIST items          |
└─────────────────────────────────────────┘
```

where the pointer indicates the first variable in the list associated with the NAMELIST name. In Allocate, the content of the group on the NAMELIST NAMES roll is changed to reflect the placement of the corresponding NAMELIST table in the object module. The format of the first two words of the modified group is:

```
                  4 bytes
┌───────────────────┬─────────────────────┐
|         0         |    displacement     |
├───────────────────┴─────────────────────┤
|           base table pointer            |
└─────────────────────────────────────────┘
```

where the base table pointer indicates the group on the BASE TABLE roll to be used for references to the NAMELIST table, and the displacement (bytes 3 and 4) indicates the distance in bytes from the address in that BASE TABLE roll group to the location of the beginning of the NAMELIST table.

This roll is used, but not modified, in Gen and Exit.

ROLL 32: NAMELIST ITEMS ROLL

This roll holds the variable names listed in the namelists defined by the source module. The group size for the NAMELIST ITEMS roll is eight bytes. Information is placed on the roll by Parse in the following form:

```
                  4 bytes
┌──────────────────────────────────────────┐
│<----------------variable-----------------│
├────────────────────┬─────────────────────┤
│--------name-------->│         0           │
└────────────────────┴─────────────────────┘
```

A marker symbol separates namelists on the roll.

The roll is used in this format by Allocate and is destroyed. It does not appear in later phases.

ROLL 33: ARRAY DIMENSION ROLL

This roll is used to hold dimension information for the arrays defined in the source module. The group size for the ARRAY DIMENSION roll is variable. The information is placed on the roll by Parse in the form of a plex, as follows:

```
                  4 bytes
┌──────────────────────────────────────────┐
│                    n                     │
├──────────────────────────────────────────┤
│                dimension                 │
├──────────────────────────────────────────┤
│                multiplier                │
├──────────────────────────────────────────┤
│                dimension                 │
├──────────────────────────────────────────┤
│                multiplier                │
├──────────────────────────────────────────┤
│                    .                     │
│                    .                     │
│                    .                     │
├──────────────────────────────────────────┤
│                dimension                 │
├──────────────────────────────────────────┤
│                multiplier                │
└──────────────────────────────────────────┘
```

where n is the number of words in the plex, exclusive of itself. As many dimensions and corresponding multipliers appear as there are dimensions declared for the array.

Unless the array is a dummy and has dummy dimensions, each dimension and multiplier is a constant. When dummy dimensions do appear in the array definition, the corresponding dimension on this roll is a pointer to the dummy dimension variable on the SCALAR roll, and all affected multipliers are pointers to temporary locations (on the TEMP AND CONST roll). The multipliers for an array with dimensions n1, n2, n3,..., n7 are 1, n1, n1*n2,..., n1*n2*n3*n4*n5*n6.

The ARRAY DIMENSION roll is present, but not modified in Unify, Gen, and Exit.

ROLL 34: BRANCH TABLE ROLL

This roll becomes the object module branch table. During Allocate, where the roll is first used, the size of the roll is determined, and some groups are actually placed on it. These groups contain the value zero, and each group refers to a source module label.

In Gen, the information for the BRANCH TABLE roll groups is supplied as each labeled statement is processed. The group size for the BRANCH TABLE roll is eight bytes. The format of the group is:

```
                  4 bytes
┌──────────────────────────────────────────┐
│                area code                 │
├──────────────────────────────────────────┤
│             relative address             │
└──────────────────────────────────────────┘
```

where the area code provides the reference for linkage editor modification of the corresponding branch table word, and the relative address is the relative location of the label in the control section (area) in which it appears. Branch table (and, hence, BRANCH TABLE roll) entries are provided for all branch target labels, statement functions, and made labels (labels constructed by the compiler to refer to return points in DO loops and to the statements following Logical IF statements).

The roll is retained in the Gen format until it is written out by Exit.

ROLL 35: TEMP DATA NAME ROLL

This roll is used only in Parse, where it holds pointers and size information for variables listed in DATA statements or in Explicit specification statements which specify initial values. Information is held on this roll while the statement is being processed.

The group size for the TEMP DATA NAME roll is four bytes. Four groups are added to the TEMP DATA NAME roll for each variable listed in the statement being scanned. They are in the following sequence:

4 bytes

element size (bytes)
pointer to variable
number elements set
element number

The third group specifies the number of elements of the variable being set by the DATA statement or the Explicit specification statement. If a full array is set, this is the number of elements in the array; if a specific array element is set, this word contains the value one.

The fourth group indicates the first element number being set. If a full array is being set, this word holds the value zero; otherwise, it holds the element number.

ROLL 36: TEMP POLISH ROLL

This roll is used only in Parse, where it holds the Polish notation for a single DATA group during the scanning of that group. In an Explicit specification statement, a DATA group is defined to be a single variable and the associated constants; in a DATA statement, a DATA group is the set of variables listed between a pair of slash characters and the constants associated with that set.

This roll is used because any error encountered in a DATA group will cause the Polish notation for the entire group to be canceled. In an Explicit specification statement, the type information on the variable is retained when the data is bad; if, however, the type information is bad, the data is also lost. The group size is four bytes.

ROLL 36: FX AC ROLL

This roll is used in Gen only and is a fixed length roll of 16 groups. The groups refer to the 16 general registers in order.

The group size for the FX AC roll is four bytes. Each group on the roll contains a pointer to the value which is held in the corresponding general register at the present point in the object module; as the contents of the general registers are changed, the pointers are changed. The pointers are used primarily to indicate that the general register is in use and the mode of the value in it. They are used for optimizing only in the case of the general registers which are loaded from the base table and the general registers used for indexing. If the general register corresponding to a specific group is not in use, the group holds the value zero.

ROLL 37: EQUIVALENCE ROLL

In Parse, this roll holds the names of all variables listed in source module EQUIVALENCE statements. One group is used for each variable name listed in the source statement, and EQUIVALENCE sets are separated from each other by a marker symbol. The group size for the EQUIVALENCE roll is twelve bytes. The format of the group is:

4 bytes

<---------------variable------------------	
---name------------>	0
EQUIVALENCE OFFSET pointer	

The pointer to the EQUIVALENCE OFFSET roll points to the first word of a plex on that roll which holds the subscript information supplied in the EQUIVALENCE statement. If no subscript was used on the variable in the EQUIVALENCE statement, the value zero appears in the third word of the group on the EQUIVALENCE roll.

The roll is used and destroyed in Allocate, during the assignment of storage for EQUIVALENCE variables.

ROLL 37: BYTE SCALAR ROLL

This roll is used only in Allocate, where it holds (temporarily) the names of 1-byte scalar variables. The group size for the BYTE SCALAR roll is eight bytes. The format of the group is:

```
                  4 bytes
┌────────────────────────────────────────┐
|<────────────scalar name────────────────|
├──────────────────────┬─────────┬───────┤
|─────────────────────>|   TAG   |   0   |
└──────────────────────┴─────────┴───────┘
```

where the TAG field indicates the mode and size of the variable.

ROLL 38: USED LIB FUNCTION ROLL

In Parse, the roll holds the names and other information for all library FUNCTIONs which are actually referenced in the source module. The group size for the USED LIB FUNCTION roll is twelve bytes. The information is placed on the roll in the following format:

```
                  4 bytes
┌────────────────────────────────────────┐
|<──────────────FUNCTION─────────────────|
├──────────────────────┬─────────┬───────┤
|───────name──────────>|   TAG   |   0   |
├──────────┬───────────┼─────────┴───────┤
|TAG       |  flag     |  no. arguments  |
└──────────┴───────────┴─────────────────┘
```

The TAG appearing in byte 7 indicates the mode and size of the function value. The TAG appearing in byte 9 indicates the mode and size of the arguments to the FUNCTION. The flag in byte 10 indicates whether the FUNCTION is in-line and, if it is, which generation routine should be used. If the flag is zero, a call is to be generated. The last two bytes hold the number of arguments to the FUNCTION. The maximum number of arguments allowed for the MIN and MAX FUNCTIONs is 16,000.

In Allocate, the information in the first two words of the group is altered to:

```
                  4 bytes
┌──────────┬───────────┬─────────────────┐
|   TAG    |     0     |   displacement  |
├──────────┴───────────┴─────────────────┤
|           base table pointer           |
└────────────────────────────────────────┘
```

where the base table pointer indicates the group on the BASE TABLE roll to be used in referring to the address of the subprogram. The displacement is the distance in bytes from the contents of the base table entry to the location at which the address of the subprogram will be stored. The TAG byte is unchanged, except in location, from Parse.

The USED LIB FUNCTION roll is consulted by Gen in the construction of object code, but it is not modified. It is also present, but not modified, in Exit.

ROLL 39: COMMON DATA ROLL

This roll holds the names of all COMMON variables as defined in source module COMMON statements. A marker symbol separates COMMON blocks on this roll. All information is placed on this roll in Parse.

The group size is eight bytes. The first six bytes of each group hold the nameof the COMMON variable; the remaining two bytes are set to zero, as follows:

```
                  4 bytes
┌────────────────────────────────────────┐
|<───────────variable name───────────────|
├──────────────────┬─────────────────────┤
|─────────────────>|          0          |
└──────────────────┴─────────────────────┘
```

In Allocate, the information on this roll is used and destroyed. The roll is not used in later phases.

ROLL 39: HALF WORD SCALAR ROLL

The roll is used only in Allocate, where it holds (temporarily) the names of half-word scalar variables defined in the source module. The group size for the HALF WORD SCALAR roll is eight bytes. The format of the group is:

```
                  4 bytes
┌────────────────────────────────────────┐
|<────────────scalar name────────────────|
├──────────────────┬──────────┬──────────┤
|─────────────────>|   TAG    |    0     |
└──────────────────┴──────────┴──────────┘
```

where the TAG indicates the mode and size of the variable.

ROLL 40: COMMON NAME ROLL

In Parse, this roll holds the name of each COMMON block, and a pointer to the location on the COMMON DATA roll at which the specification of the variables in that block begins. The group size for the COMMON NAME roll is twelve bytes. The format of the group is:

```
                  4 bytes
┌────────────────────────────────────────┐
|<──────────────block name───────────────|
├──────────────────┬─────────────────────┤
|─────────────────>|          0          |
├──────────────────┴─────────────────────┤
|           COMMON DATA pointer          |
└────────────────────────────────────────┘
```

The pointer points to the first variable in the list of names which follows the block name in the COMMON statement; since a single COMMON block may be mentioned more than once in source module COMMON statements, the same COMMON name may appear more than once on this roll. The information is placed on this roll as COMMON statements are processed by Parse.

In Allocate, the roll is rearranged and the last word of each group is replaced by the size of the COMMON block in bytes, after duplicate COMMON names have been eliminated. The size is written out by Allocate and the roll is destroyed.

ROLL 40: TEMP PNTR ROLL

The group size for the TEMP PNTR roll is four bytes. This roll is used only in Gen, and holds pointers to those groups on the TEMP AND CONST roll that represent object module temporary storage locations. The information recorded on this roll is maintained so that temporary storage created for one statement can be reused by subsequent statements.

ROLL 41: IMPLICIT ROLL

This roll is used only in Parse and Allocate, where it holds the information supplied by the source module IMPLICIT statement. The group size for the IMPLICIT roll is four bytes. Its format is:

1 byte	1 byte	1 byte	1 byte
letter	0	TAG	0

This information is placed on the roll by Parse. The TAG field in the third byte of the group indicates, in the format of the TAG field of a pointer, the mode and size assigned to the letter by means of the IMPLICIT statement.

The IMPLICIT roll is used by Allocate, and destroyed.

ROLL 42: EQUIVALENCE OFFSET ROLL

This roll is constructed during the operation of Parse and holds the subscripts from EQUIVALENCE variables in the form of plexes. The group size for the EQUIVALENCE OFFSET roll is variable. Each plex has the form:

4 bytes
n
subscript 1
subscript 2
. . .
subscript n

where n is the number of words in the plex exclusive of itself and, therefore, also the number of subscripts. Each subscript is recorded as an integer constant.

The connection between a plex on this roll and the corresponding EQUIVALENCE variable is made by a pointer which appears on the EQUIVALENCE roll and points to the first word of the appropriate plex on this roll.

In Allocate, the EQUIVALENCE OFFSET roll is used in the allocation of storage for EQUIVALENCE variables. It is destroyed during this phase, and does not appear in the later phases of the compiler.

ROLL 42: FL AC ROLL

This roll is used in Gen only, and is a fixed length roll of four groups. The groups refer to the four floating-point registers, in order.

The group size for the FL AC roll is four bytes. Each group on the roll contains a pointer to the value which is held in the register at the present point in the object program; as the contents of the registers change, the pointers are changed. These pointers are used primarily to indicate that the register is in use and the mode of the value in it. If the register is not in use, the corresponding group on this roll contains zero.

ROLL 43: LBL ROLL

This roll holds all labels used and/or defined in the source module. Each label is entered on the roll by Parse when it is first encountered, whether in the label field or within a statement.

The group size for the LBL roll is four bytes. In Parse, the format of the LBL roll group is:

1 byte	3 bytes
TAG	binary label

where the first byte is treated as the TAG field of a pointer, and the remaining three bytes contain the label, converted to a binary integer.

In the TAG field, the mode portion (the first four bits) is used to indicate whether the label has been defined; the remainder of the TAG field is used to indicate whether the label is the target of a jump, the label of a FORMAT, or neither.

The leftmost four bits of the TAG byte are used as follows:

8 = Label is defined

0 = Label is undefined

The rightmost four bits of the TAG byte indicate the following:

1 = This is the label of the target of a jump (GO TO) statement.

3 = This is the label of a FORMAT statement.

5 = This label is a possible re-entry point within an innermost DO loop that may have a possible extended range. (Parse inserts the hexadecimal 5 to indicate to Gen that the label is a possible re-entry point; the Gen phase then restores those registers that were saved before the extended range was entered.)

0 = None of the above conditions.

In Allocate, the lower three bytes of each LBL roll group defining a jump target label are replaced by the lower three bytes of a pointer to the BRANCH TABLE roll group, which will hold the location of the label at object time. Each group defining a FORMAT statement label is replaced (lower three bytes only) with a pointer to the FORMAT roll group which holds the base pointer and displacement for the FORMAT. Groups defining the targets of unclosed DO loops are cleared to zero.

In Gen, the LBL roll is used to find the pointers to the BRANCH TABLE and FORMAT rolls, but it is not altered.

ROLL 44: SCALAR ROLL

In Parse, the names of all unsubscripted variables which are not dummy arguments to statement functions are listed on the roll in the order of their appearance in active (non-specification) statements in the source module. Variables which are defined in specification statements, but which are never used in the source module, are not entered on the roll. The group size for the SCALAR roll is eight bytes. The format of the group is:

```
                    4 bytes
┌──────────────────────────────────────────┐
│<─────────────scalar name─────────────────│
├───────────────────┬───────────┬──────────┤
│──────────────────>│    TAG    │    0     │
└───────────────────┴───────────┴──────────┘
```

The TAG field appearing in the seventh byte of the group indicates the mode and size of the variable in the format of the TAG field of a pointer.

In Allocate, the information left on the SCALAR roll by Parse is replaced by information indicating the storage assigned for the variable. The resulting format of the group is:

```
                    4 bytes
┌─────────┬──────────┬─────────────────────┐
│   TAG   │DBG/CEAD  │    displacement     │
├─────────┴──────────┴─────────────────────┤
│            base table pointer            │
└──────────────────────────────────────────┘
```

The TAG field appearing in the first byte is unchanged, except in location, from the TAG field held in the SCALAR roll group during Parse. The DBG/CEAD flag (in the second byte) is logically split into two hexadecimal values. The first of these indicates debug references to the variable; the value is 1 for a scalar referred to in the INIT option; otherwise, the value is zero. The second hexadecimal value is nonzero if the variable is in COMMON, a member of an EQUIVALENCE set, or an argument to a subprogram or a global dummy; otherwise, it is zero. The displacement in bytes 3 and 4, and the base table pointer in the second word, function together to indicate the storage location assigned for the variable. The base table pointer specifies a BASE TABLE roll group; the displacement is the distance in bytes from the location contained in that group to the location of the scalar variable. If the scalar is a call by name dummy, the base table pointer is replaced by a pointer to the GLOBAL DMY roll group defining it, and the displacement is zero.

The SCALAR roll is checked, but modified, during Unify, Gen, and Exit.

ROLL 44: HEX CONST ROLL

This roll holds the hexadecimal constants used in source module DATA statements.

The format of the roll is identical for all phases of the compiler. The group size is 16 bytes. Two hexadecimal characters are packed to a byte, and constants which occupy fewer than 16 characters are right-adjusted in the group with leading zeros.

ROLL 45: DATA VAR ROLL

In Parse, this roll holds the names of variables listed in DATA statements and variables for which data values are provided in Explicit specification statements. The names are entered on the roll when they are found in these statements. The group size for this roll is eight bytes. The groups have the following form:

```
                    4 bytes
┌─────────────────────────────────────────┐
|<------------variable name---------------|
├────────────────────┬────────────────────┤
|------------------->|         0          |
└────────────────────┴────────────────────┘
```

This information is used to ensure that no data values are provided in the source module for dummy variables. The information is left on the roll throughout Parse, but is cleared before Allocate operates.

In Allocate, binary labels and the names of statement functions, scalar variables, arrays, global subprograms, and used library functions are placed on the roll in order. The group size for this roll is four bytes. Each label entered on the roll occupies one word; the names occupy two words each and are left-justified, leaving the last two bytes of each name group unused.

The encoded information is placed on this roll by Allocate as its operations modify the rolls on which the information was originally recorded by Parse. Thus, all the labels appear first, in the order of their appearance on the LBL roll, etc. The information is used by the Exit phase in producing the object module listing (if the LIST option is specified by the user).

ROLL 46: LITERAL TEMP (TEMP LITERAL) ROLL

This roll is used only in Parse, where it holds literal constants temporarily while they are being scanned. The group size for the LITERAL TEMP or TEMP LITERAL roll is four bytes. Literal constants are placed on the roll one character per byte, or four characters per group.

ROLL 47: COMMON DATA TEMP ROLL

This roll holds the information from the COMMON DATA roll temporarily during the operation of Allocate, which is the only phase in which this roll is used. The group size for the COMMON DATA TEMP roll is eight bytes. The format of the group is identical to that of the COMMON DATA roll, namely:

```
                    4 bytes
┌─────────────────────────────────────────┐
|<--------------variable------------------|
├────────────────────┬────────────────────┤
|-------name-------->|         0          |
└────────────────────┴────────────────────┘
```

ROLL 47: FULL WORD SCALAR ROLL

This roll is used only in Allocate, where it holds the names of all fullword scalar variables defined by the source module. The group size is eight bytes. The format of the group on the roll is:

```
                    4 bytes
┌─────────────────────────────────────────┐
|<-------------scalar name----------------|
├────────────────────┬──────────┬─────────┤
|------------------->|   TAG    |    0    |
└────────────────────┴──────────┴─────────┘
```

where the TAG indicates the mode and size of the variable. This information is held on this roll only temporarily during the assignment of storage for scalar variables.

ROLL 48: COMMON AREA ROLL

This roll is used only in Allocate, where it holds COMMON block names and sizes temporarily during the allocation of COMMON storage. The group size for the COMMON AREA roll is twelve bytes. The format of the group on the roll is:

```
                    4 bytes
┌─────────────────────────────────────────┐
|<-------------block name-----------------|
├────────────────────┬────────────────────┤
|------------------->|         0          |
├────────────────────┴────────────────────┤
|           block size (bytes)            |
└─────────────────────────────────────────┘
```

ROLL 48: NAMELIST ALLOCATION ROLL

This roll is used only in Allocate, where it holds information regarding NAMELIST items temporarily during the allocation of storage for the NAMELIST tables. The group size for this roll is twelve bytes. The format of the group is:

```
                    4 bytes
┌─────────────────────────────────────────┐
|<------------variable name---------------|
├────────────────────┬────────────────────┤
|------------------->|         0          |
├────────────────────┴────────────────────┤
|                pointer                  |
└─────────────────────────────────────────┘
```

where the pointer indicates the group defining the variable on either the SCALAR or ARRAY roll.

ROLL 49: COMMON NAME TEMP ROLL

This roll is used only in Allocate, where it holds the information from the COMMON NAME roll temporarily. The group size for the COMMON NAME TEMP roll is twelve bytes. The format of the group is therefore identical to that of the COMMON NAME roll:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<-------------block name---------------->|
├───────────────────┬─────────────────────┤
|------------------>|          0          |
├───────────────────┴─────────────────────┤
|           COMMON DATA pointer           |
└─────────────────────────────────────────┘
```

where the COMMON DATA pointer points to the list of variables in the COMMON block.

ROLL 50: EQUIV ALLOCATION ROLL

This roll is used only during Allocate, and is not used in any other phase of the compiler. When the allocation of storage for EQUIVALENCE variables has been completed, the information which has been produced on the GENERAL ALLOCATION roll is moved to this roll. The group size for the EQUIV ALLOCATION roll is twelve bytes. The format of the group is, therefore, identical to that on the GENERAL ALLOCATION roll:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<--------------variable----------------->|
├───────────────────┬─────────────────────┤
|-------name------->|    displacement     |
├───────────────────┴─────────────────────┤
|           base table pointer            |
└─────────────────────────────────────────┘
```

where the base table pointer indicates the group on the BASE TABLE roll which will be used for references to the variable. The displacement is the distance in bytes from the location indicated in the BASE TABLE roll group to the location of the variable.

ROLL 51: RLD ROLL

This roll is used only in Allocate and Exit; it is not used in Parse. In both Allocate and Exit, the roll holds the information required for the production of RLD cards. The group size for the RLD roll is eight bytes. The group format is:

```
                  4 bytes
┌───────────────────────────────┬─────────┐
|           area code           |  ESD #  |
├───────────────────────────────┴─────────┤
|           address                       |
└─────────────────────────────────────────┘
```

where the area code indicates the control section in which the variable or constant is contained. The ESD number governs the modification of the location by the linkage editor, and the address is the location requiring modification.

Information is placed on this roll by both Allocate and Exit, and the RLD cards are written from the information by Exit. The entries made on the RLD roll by Allocate concern the NAMELIST tables; all remaining entries are made by Exit.

ROLL 52: COMMON ALLOCATION ROLL

This roll is used only in Allocate and is not used in any other phase of the compiler. When the allocation of COMMON storage has been completed, the information which has been produced on the GENERAL ALLOCATION roll is moved to this roll. The group size for the COMMON ALLOCATION roll is twelve bytes. The format of the group is, therefore, identical to that on the GENERAL ALLOCATION roll:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<--------------variable----------------->|
├───────────────────┬─────────────────────┤
|-------name------->|    displacement     |
├───────────────────┴─────────────────────┤
|           base table pointer            |
└─────────────────────────────────────────┘
```

where the base table pointer indicates the group on the BASE TABLE roll which will be used for references to the variable.

The displacement is the distance in bytes from the location indicated in the BASE TABLE roll group to the location of the variable.

ROLL 52: LOOP CONTROL ROLL

This roll is created by Unify and is used by Gen. The information contained on the roll indicates the control of a loop.

The group size for the LOOP CONTROL roll is twelve bytes. The format of the LOOP CONTROL roll group in Unify and Gen is:

4 bytes

traits	coefficient
register (this loop)	
base or register (outer loop)	

where the first byte of the first word (traits) indicates whether the coefficient is initiated by a direct load. The remaining three bytes is the coefficient, which is the multiplier for the induction variable. The second four bytes is the register where the coefficient is required. The base is the source of initialization of the register; it can be either a constant, register, or an address.

ROLL 53: FORMAT ROLL

This roll is first used in Parse, where the FORMAT statements are placed on it. See Appendix D for the description of the encoding of the FORMAT statement.

Each group of the FORMAT roll is in the form of a plex (the group size is given in word 0). The configuration of a FORMAT group in Parse is:

4 bytes

size of the group
pointer to the LBL roll
number of bytes in the FORMAT
. . .

Word 0 contains a value which indicates the number of words in the group on the roll. The pointer to the LBL roll points to the label of the corresponding FORMAT statement. The next word gives the number of bytes of storage occupied by this particular FORMAT statement. The ellipses denote that the encoded FORMAT follows this control information.

In Allocate, the FORMATs are replaced by the following:

4 bytes

0	displacement
base table pointer	

which, taken together, indicate the beginning location of the FORMAT statement. These groups are packed to the BASE of the roll; that is, this information for the first FORMAT appears in the first two words on the roll, the information for the second FORMAT appears in words 3 and 4, etc.

The LBL roll group which defines the label of the FORMAT statement holds a pointer to the displacement recorded for the statement on this roll.

The FORMAT roll is retained in this form for the remainder of the compilation.

ROLL 54: SCRIPT ROLL

This roll is created by Parse as each appropriate array reference is encountered. The array reference indicated includes subscripts (one or more) which use the instruction variable in a linear fashion. Unify uses the contents of the roll.

The group size of the SCRIPT roll is 16 bytes, plus an additional 4 bytes for each DO loop that is open at the point of the array reference represented by the entry. The group format of the SCRIPT roll in Parse and Unify is as described for the NONSTD SCRIPT roll.

ROLL 55: LOOP DATA ROLL

This roll contains the initializing and terminating data, and indicates the induction variable and the nesting level of the particular loop from which this entry was created.

The roll is created in Parse at the time that the loop is encountered. The group size of the LOOP DATA roll is 20 bytes. The group format of the roll in Parse is:

4 bytes

TAG	nest level
pointer to induction variable	
pointer to n_1 (initial value)	

where the TAG byte contains a X'80' when an inner DO loop contains a possible extended range. The X'80' is placed there by Parse and tested by Gen. The Gen phase then produces object code to save general registers 4 through 7 at the beginning of this DO loop so that the registers are not altered in the extended range. The next three bytes indicate the nest level of the loop. The second word is a pointer to the SCALAR roll group which describes the induction variable. The third word of the group points to the initializing value for the induction variable, which may be represented on the FX CONST roll or the SCALAR roll.

During the operation of the Unify phase, the roll is completed with pointers to the LOOP CONTROL roll. During Unify, the LOOP CONTROL roll is also created; therefore, insertion of the pointers is done while the loop control data is being established.

The following illustration shows the configuration of the LOOP DATA roll as it is used in Unify:

4 bytes

nest level
SCALAR pointer (induction variable)
FX CONST pointer or SCALAR pointer
LOOP CONTROL pointer (start init.)
LOOP CONTROL pointer (end init.)

The last two words (eight bytes) of the group are inserted by Unify. These pointers point to the first and last LOOP CONTROL roll groups concerned with this loop.

ROLL 56: PROGRAM SCRIPT ROLL

This roll is a duplicate of the SCRIPT roll. The contents of the SCRIPT roll are transferred to the PROGRAM SCRIPT roll in Parse as each loop is closed. Each loop is represented by a reserved block on the roll.

The group size of the PROGRAM SCRIPT roll is 16 bytes, plus an additional 4 bytes for each nest level up to and including the one containing the reference represented by the entry. The format of the PROGRAM SCRIPT roll group in Parse and Unify is as follows:

4 bytes

traits	frequency
ARRAY REF pointer	
ARRAY pointer	
ARRAY offset pointer	
induction variable coefficient	
induction variable coefficient (nest level = 2)	
. . .	
induction variable coefficient (nest level = n)	

See the NONSTD SCRIPT roll for further description.

ROLL 56: ARRAY PLEX ROLL

This roll is used only in Gen, where it handles subscripts (array references) which are not handled by Unify. The group size for the ARRAY PLEX roll is twelve bytes. The format of the group on the roll is:

4 bytes

pointer to array
pointer to index
displacement

The pointer in the first word of the group points to the ARRAY REF roll when the subscript used contains DO dependent linear subscripts (which are handled by Unify) and non-linear variables. Otherwise, the pointer refers to the ARRAY roll.

The second word of the group holds a pointer to the index value to be used in the subscripted array reference. This pointer points to general register 9 on the FX AC roll if the index value has been loaded into that register; if the index value has been stored in a temporary location, the pointer indicates the proper location on the TEMP AND CONST roll; if the index value is a fixed constant, the pointer indicates the proper group on the FX CONST roll. When the information in this word has been used to construct the proper instruction for the array reference, the word is cleared to zero.

The displacement, in the third word of the group, appears only when the first word of the group holds a pointer to the ARRAY roll. Otherwise, the displacement is on the ARRAY REF roll in the group indicated by the pointer in the first word, and this word contains the value zero. This value is the displacement value to be used in the instruction generated for the array reference.

ROLL 57: ARRAY REF ROLL

Pointers to this roll are inserted into the Polish notation by Parse. At the time that these pointers are established, the ARRAY REF roll is empty. The pointer is inserted into the Polish notation when an array reference includes linear loop-controlled subscripts.

The roll is initially created by Unify and completed by Gen. The group size of the ARRAY REF roll is 16 bytes. The format of the ARRAY REF roll group as it appears in Unify is as follows:

0 11	12 15	16 19	20 31
	R_1	R_2	offset
pointer to register (R_1) or to the TEMP AND CONST roll			
pointer to register (R_2) or to the TEMP AND CONST roll			
pointer to the ARRAY roll			

The first word of the group contains the low 20 bits of an instruction which is being formatted by the compiler. R_1 and R_2 are the two register fields to be filled with the numbers of the registers to be used for the array reference. Word 2 of the group contains the pointer indicating the register to be assigned for R_1. Word 3 of the group indicates the register R_2. When R_1 and R_2 have been assigned, the second and third words are set to zero.

Gen completes the entry by adding the operation code to the instruction that is being built. The format of an ARRAY REF roll group in Gen is:

0 11	12 15	16 19	20 31
op code	R_1	R_2	offset
0 or TEMP AND CONST roll pointer			
0 or TEMP AND CONST roll pointer			
ARRAY pointer			

ROLL 58: ADR CONST ROLL

This roll contains relocatable information that is to be used by Exit.

Unify creates the roll which contains a pointer to the TEMP AND CONST roll and an area code and displacement. The pointer indicates an entry on the TEMP AND CONST roll which must be relocated according to the area code. The displacement is the value to be placed in that temporary storage and constant area location.

The group size of the ADR CONST roll is eight bytes. The format of the ADR CONST roll group in Unify is:

4 bytes

area code	displacement
TEMP AND CONST pointer	

These groups are constructed by Unify to provide additional base table values for indexing.

ROLL 59: AT ROLL

This roll is constructed in Parse and used in Gen. It is not used in the remaining phases. The group size for this roll is twelve bytes. The format of the group is:

4 bytes

AT label pointer
debug label pointer
return label pointer

All three of the pointers in the group point to the LBL roll. The first points to the label indicated in the source module AT statement. The second points to the made label supplied by the compiler for the code it has written to perform the debugging operations. The third label pointer indicates the made label supplied for the point in the code to which the debug code returns; that is, the code which follows the branch to the debugging code.

ROLL 60: SUBCHK ROLL

This roll is initialized in Parse and used in Allocate. It does not appear in later phases. The group size for this roll is eight bytes. The format of the group is:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<------------variable name---------------|
├────────────────────┬────────────────────┤
|------------------->|         0          |
└────────────────────┴────────────────────┘
```

Each group holds the name of an array listed in the SUBCHK option of a source module DEBUG statement.

ROLL 60: NAMELIST MPY DATA ROLL

This roll is set up during the construction of the NAMELIST tables in Allocate. In Exit, the roll is used to complete the information in the NAMELIST tables. The roll is not used in the other phases of the compiler. The group size for the NAMELIST MPY DATA roll is eight bytes. The format of the group on this roll is:

```
                  4 bytes
┌─────────────────────────────────────────┐
|          multiplier constant            |
├─────────────────────────────────────────┤
|          address                        |
└─────────────────────────────────────────┘
```

The multiplier constant refers to an array dimension for an array mentioned in a NAMELIST list. The address is the location in a NAMELIST table at which a pointer to the multiplier constant must appear. In Exit, the constant is placed in the temporary storage and constant area of the object module, and a TXT card is punched to load its address into the location specified in the second word of the group.

ROLL 62: GENERAL ALLOCATION ROLL

This roll is used only during Allocate, and is not used in any other phase of the compiler. During the various allocation operations performed by this phase, the roll holds the information which ultimately resides on the remaining ALLOCATION rolls. The group size for the GENERAL ALLOCATION roll is twelve bytes. The format of the group is:

```
                  4 bytes
┌─────────────────────────────────────────┐
|<---------------variable-----------------|
├────────────────────┬────────────────────┤
|-------name-------->|    displacement    |
├────────────────────┴────────────────────┤
|           base table pointer            |
└─────────────────────────────────────────┘
```

where the base table pointer indicates the group on the BASE TABLE roll which will be used for references to the variable.

The displacement is the distance in bytes from the location indicated in the BASE TABLE roll group to the location of the variable.

During the allocation of COMMON, the third word of each group holds a relative address until all of a COMMON block has been allocated, when the relative address is replaced by the pointer as indicated above. During the allocation of EQUIVALENCE variables, relative addresses within the EQUIVALENCE variables are used and then replaced by pointers as for COMMON.

ROLL 62: CODE ROLL

This roll holds the object code generated by the compiler, in binary. This roll is first used in Gen, where the object code for the entire source module is built up on the roll.

The group size for the CODE roll is eight bytes. Two types of groups are placed on the roll during the operations of Gen. The first type of group is added to the roll by the instructions IEYBIN, IEYBIM and IEYBID. In this type of group, the binary instruction is left-justified in the eight bytes. When the instruction occupiesonly two bytes, the first word is completed with zeros. When the instruction occupies two or four bytes, the second word of the group holds a pointer to the defining group for the operand of the instruction. When the instruction is a 6-byte instruction, the last two bytes of the group contain zero, and no pointer to the operand appears. A unique value is placed on the CODE roll by these instructions to indicate the beginning of a new control section.

The second type of group entered on the CODE roll appears as a result of the operation of one of the instructions IEYPOC and IEYMOC. These groups do not observe the 8-byte group size of the roll, but rather begin with a word containing a special value in the upper two bytes; this value indicates an unusual group. The lower two bytes of this word contain the number of words in the following information. This word is followed by the binary instructions.

The object module code is written out from this roll by the Exit phase of the compiler.

ROLL 63: AFTER POLISH ROLL

This roll is constructed in Parse, remains untouched until Gen, and is destroyed in that phase.

The AFTER POLISH roll holds the Polish notation produced by Parse. The Polish for one statement is moved off of the POLISH roll and added to this roll when it is completed; thus, at the end of Parse, the Polish notation for the entire source module is on this roll.

In Gen, the Polish notation is returned to the POLISH roll from the AFTER POLISH roll for the production of object code. At the conclusion of the Gen phase, the roll is empty and is no longer required by the compiler. The group size for this roll is four bytes.

WORK AND EXIT ROLLS

Because of the nature and frequency of their use, the WORK roll and the EXIT roll are assigned permanent storage locations in IEYROL, which is distinct from the storage area reserved for all other rolls. As a result, these rolls may never be reserved and are manipulated differently by the POP instructions. The group stats and the items BASE and TOP are not maintained for these rolls. The only control item maintained for these rolls corresponds to the item BOTTOM, and is carried in the general register WRKADR (register 4) for the WORK roll and EXTADR (register 5) for the EXIT roll.

WORK ROLL

The WORK roll is often used to hold intermediate values. The group size for this roll is four bytes. The name W0 is applied to the bottom of the WORK roll (the last meaningful word), W1 refers to the next-to-bottom group on the WORK roll, etc. In the POP instructions these names are used liberally, and must be interpreted with care. Loading a value into W0 is storage into the next available word, (WRKADR) + 4, unless specifically otherwise indicated, while storage from W0 to another location involves access to the contents of the last word on the roll, (WRKADR). WRKADR is normally incremented following a load operation and decremented following a store.

EXIT ROLL

The EXIT roll holds exit addresses for subroutines and, thereby, provides for the recursion used throughout the compiler. The ANSWER BOX is also recorded on the EXIT roll. The group size for the EXIT roll is twelve bytes. The first byte is the ANSWER BOX. The remaining information on the roll is recorded when a subroutine jump is performed in the compiler code; it is used to return to the instruction following the jump when the subroutine has completed its operation.

The values placed on the EXIT roll differ, depending on the way in which the subroutine jump is performed. As a result of the interpretation of the IEYJSB POP instruction, the last three bytes of the first word contain the location of the IEYJSB plus two (the location of the POP instruction following the IEYJSB, the return point); the second word of the group holds an address within the IEYJSB subroutine; the third word contains the location of the global label for the routine from which the subroutine jump was made plus two (the value of LOCAL JUMP BASE in that routine).

As an example of how a subroutine jump is accomplished by means of machine language instructions, the following instructions are used:

```
L       TMP,G0052J

BAL     ADDR,JSB STORE IN EXIT
```

to replace the POP instruction

```
IEYJSB  G0052J
```

In this case, no value is placed in the last three bytes of the first word; the second word holds the address of the instruction following the BAL; the third word holds the location of the global label immediately preceding the BAL plus two (the value of POPADR when the jump is taken, which is also the value of LOCAL JUMP BASE, the base address to be used for local jumps in the routine from which the subroutine jump was made).

On return from a subroutine, these values are used to restore POPADR and LOCAL JUMP BASE and they are pruned from the EXIT roll.

This appendix shows the format of the Polish notation which is generated by the compiler for each type of statement in the FORTRAN IV (G) language.

GENERAL FORM

The format of the Polish notation depends on the statement type, but always terminates with the control driver which indicates the type of statement:

```
              4 bytes
┌──────────────────────────────┐┐
├──────────────────────────────┤│
|              .               ││ Polish for
|              .               │> statement
|              .               ││
├──────────────────────────────┤│
├──────────────────────────────┤┘
|control driver                |
├──────────────────────────────┤
|statement number              |
└──────────────────────────────┘
```

The statement number is an integer whose value is increased by one for each statement processed. This value is used only within the compiler.

LABELED STATEMENTS

For labeled statements, a pointer to the label is inserted between the control driver and the statement number:

```
              4 bytes
┌──────────────────────────────┐┐
├──────────────────────────────┤│
|              .               ││
|              .               │> Polish for
|              .               ││ statement
├──────────────────────────────┤│
├──────────────────────────────┤┘
|control driver                |
├──────────────────────────────┤
|label                         |
├──────────────────────────────┤
|pointer to statement label    |
├──────────────────────────────┤
|statement number              |
└──────────────────────────────┘
```

The label information is not included in the following descriptions of the Polish notation for individual statement types.

ARRAY REFERENCES

The Polish notation for an array reference whose subscripts are all linear functions of DO variables consists simply of a pointer to the appropriate group on the ARRAY REF roll. The Polish notation generated for all other references to an array element is:

```
              4 bytes
┌──────────────────────────────┐
|array driver                  |
├──────────────────────────────┤
|pointer to array              |
├──────────────────────────────┤┐
├──────────────────────────────┤│
|              .               ││ Polish for
|              .               │> subscript 1
|              .               ││
├──────────────────────────────┤│
├──────────────────────────────┤┘
|multiplier                    |
├──────────────────────────────┤
|argument driver               |
├──────────────────────────────┤┐
├──────────────────────────────┤│
|              .               ││ Polish for
|              .               │> subscript 2
|              .               ││
├──────────────────────────────┤│
├──────────────────────────────┤┘
|multiplier                    |
├──────────────────────────────┤
|argument driver               |
├──────────────────────────────┤
|              .               |
|              .               |
|              .               |
├──────────────────────────────┤┐
├──────────────────────────────┤│
|              .               ││ Polish for
|              .               │> subscript 7
|              .               ││
├──────────────────────────────┤│
├──────────────────────────────┤┘
|multiplier                    |
├──────────────────────────────┤
|argument driver               |
├──────────────────────────────┤
|dummy array pointer           |
└──────────────────────────────┘
```

The pointer to the array may indicate either (1) the ARRAY roll, when none of the subscripts used in the array reference are linear functions of DO variables, or (2) the ARRAY REF roll, when some, but not all, of the subscripts are linear functions of DO variables. The subscripts for which Polish notation appears are those which are not linear functions of DO variables. Only the required number of subscripts appear.

The multiplier following each subscript is the multiplier for the corresponding array dimension. This value is an integer unless the array is a dummy including dummy dimensions which affect this array dimension; in this case, the multiplier is represented by a pointer to the TEMP AND CONST roll.

ENTRY STATEMENT

The Polish notation generated for the ENTRY statement is:

4 bytes

pointer to ENTRY name
ENTRY driver
statement number

The pointer points to the ENTRY NAMES roll.

ASSIGN STATEMENT

The Polish notation generated for the ASSIGN statement is:

4 bytes

pointer to label
pointer to variable
ASSIGN driver
statement number

ASSIGNED GO TO STATEMENT

The Polish notation generated for this statement is:

4 bytes

pointer to variable
assigned GO TO driver
statement number

LOGICAL IF STATEMENT

The Polish notation generated for this statement is:

4 bytes

	Polish for logical expression
.	
.	
.	
	Polish for statement "S"
.	
.	
.	
logical IF driver	
statement number	

RETURN STATEMENT

The following Polish notation is produced for the RETURN statement:

4 bytes

pointer to I
RETURN driver
statement number

The pointer to I does not appear if the statement is of the form RETURN.

ARITHMETIC AND LOGICAL ASSIGNMENT STATEMENT

The Polish notation produced for this statement is:

4 bytes

pointer to variable to be set	
	Polish for right side
.	
.	
.	
assignment driver	
statement number	

The Polish notation for the right side of the assignment statement is in the proper form for an expression, and includes array references where they appear in the source statement. The variable to be set may also be an array element; in this case, the pointer to the variable to be set is replaced by the Polish notation for an array reference.

UNCONDITIONAL GO TO STATEMENT

The Polish notation produced for this statement is:

4 bytes
pointer to label
GO TO driver
statement number

COMPUTED GO TO STATEMENT

The following Polish notation is produced for this statement:

4 bytes	
pointer to x1	branch points
pointer to x2	
. . .	
pointer to xn	
number of branch points	
pointer to variable	
computed GO TO driver	
statement number	

ARITHMETIC IF STATEMENT

The following Polish notation is produced for this statement:

4 bytes	
	Polish for expression
. . .	
pointer to x1	branch points
pointer to x2	
pointer to x3	
pointer to label next stmt.	
IF driver	
statement number	

The label of the next statement is inserted following the IF driver because the next statement may be one of the branch points referenced; if it is, code will be generated to fall through to that statement in the appropriate case(s).

DO STATEMENT

The following is the Polish notation produced for the statement DO x i = m1, m2, m3:

4 bytes
pointer to M_2 (test value)
pointer to M_3 (increment)
pointer to LOOP DATA roll
pointer to LBL roll
DO driver
statement number

The pointer to m3 appears, even if the increment value is implied.

CONTINUE STATEMENT

The Polish notation produced for this statement is:

4 bytes

CONTINUE driver
statement number

PAUSE AND STOP STATEMENTS

The Polish notation produced for these statements is:

4 bytes

pointer to constant
PAUSE or STOP driver
statement number

For both the PAUSE statement and the STOP statement, the constant appears on the LITERAL CONST roll, regardless of its nature in the source statement. If no constant appears in the statement, the pointer to the constant points to the literal constant zero.

END STATEMENT

The Polish notation generated for the END statement is:

4 bytes

END driver
statement number

BLOCK DATA STATEMENT

The Polish notation generated for the BLOCK DATA statement is:

4 bytes

BLOCK DATA driver
statement number

DATA STATEMENT AND DATA IN EXPLICIT SPECIFICATION STATEMENTS

For each statement (DATA or Explicit specification) in which data values for variables are specified, a Polish record is produced. This record ends with a DATA driver and a statement number. For each variable initialized by the statement, the following appears:

4 bytes

pointer to variable
offset

The offset is the element number at which initialization begins; if it does not apply, this word contains the value zero.

This information is followed by the pair of groups

4 bytes

repetition count
pointer to constant

or, when the constant is literal, the three groups

4 bytes

repetition count
pointer to constant
number of elements

where the last group indicates the number of elements of an array to be filled by the literal constant. For array initialization, one or more of the "constant" groups may appear.

I/O LIST

The Polish notation for an I/O List contains pointers to the variables in the list, Polish notation for array references where they appear, and pointers and drivers to indicate implied DO loops.

The I/O list

((C(I),I=1,10),A,B)

for example, results in the following Polish notation:

4 bytes
pointer to M_2 (test value)
pointer to M_3 (increment)
pointer to LOOP DATA roll
implied DO driver
pointer to C
1 (number of subscripts)
pointer to I (subscript)
argument driver
array driver
IOL DO Close driver
pointer to A
pointer to B

The area between, and including, the implied DO driver and the array driver is an array reference, as it would appear wherever C(I) was referred to in source module statements.

INPUT STATEMENTS

The following paragraphs discuss the Polish notation produced for all forms of the READ statement except direct access.

FORMATTED READ

For the form READ (a,b) list, the formatted READ, the Polish notation generated is:

4 bytes	
pointer to a (data set)	
FORMAT driver	
pointer to FORMAT	
END= driver	
pointer to END label	
ERR= driver	
pointer to ERR label	
IOL driver	
. . .	Polish for I/O list
code word	
IBCOM entry, formatted READ	
pointer to IBCOM	
READ WRITE driver	
statement number	

The pointer to the FORMAT points either to the label of the FORMAT statement or to the array in which the FORMAT is stored. The END= and ERR= drivers and the pointers following them appear only if the END and ERR options are used in the statement; either one or both may appear, and in any order with respect to each other. If no I/O list appears in the statement, the Polish for the I/O list is omitted, but the IOL driver appears nonetheless.

The code word contains zero in its high-order three bytes, and, in its low-order byte, a unique code specifying the operation and unit for the input/output statement. This code word distinguishes among the various READ statements and is inserted in the code produced for them.

Input/output operations are performed by the RUNTIME routines. IBCOM is a transfer routine in RUNTIME through which all input/output except NAMELIST is performed. The IBCOM entry for formatted READ indicates an entry point to this routine. (See Appendix D for further discussion of IBCOM.) The pointer to IBCOM points to the routine on the GLOBAL SPROG roll.

NAMELIST READ

For the form READ (a,x), the NAMELIST READ, the following changes are made to the Polish notation given above:

1. The FORMAT driver is replaced by a NAMELIST driver.

2. The pointer to the FORMAT is replaced by a pointer to the NAMELIST.

3. The code word value is changed.

4. The IBCOM entry is replaced by the value zero, since NAMELIST input/output is not handled through IBCOM.

5. The pointer to IBCOM is replaced by a pointer to the NAMELIST READ routine.

6. No I/O list may appear.

UNFORMATTED READ

For the form READ (a) list, the unformatted READ, the following changes are made to the Polish notation given above:

1. The FORMAT driver is removed.

2. The pointer to the FORMAT is removed.

3. The IBCOM entry, formatted READ, is replaced by the IBCOM entry, unformatted READ.

READ STANDARD UNIT

For the form READ b, list, the standard unit READ statement, the following changes are made to the Polish notation given above:

1. No END= or ERR= drivers may appear, nor may the corresponding pointers to labels.

2. The code word value is changed.

OUTPUT STATEMENTS

The following paragraphs discuss the Polish notation produced for all forms of the WRITE statement except direct access, and for the PRINT and PUNCH statements.

FORMATTED WRITE

For the form WRITE (a,b) list, the formatted WRITE, the Polish notation generated is:

4 bytes	
pointer to a data set	
FORMAT driver	
pointer to FORMAT	
END= driver	
pointer to END label	
ERR= driver	
pointer to ERR label	
IOL driver	
. . .	Polish for I/O list
code word	
IBCOM entry, formatted WRITE	
pointer to IBCOM	
READ WRITE driver	
statement number	

The pointer to the FORMAT points either to the label of the FORMAT statement or to the array in which the FORMAT is stored. The END= and the ERR= drivers and the pointers following them appear only if the END and ERR options are used in the statement; either one or both may appear, and in any order relative to each other. If no I/O list appears in the statement, the Polish for the I/O list is omitted, but the IOL driver appears nonetheless.

The code word contains zero in its high-order three bytes, and, in its low-order byte, a unique code specifying the operation and unit for the input/output statement. This code word distinguishes among the various output statements and is inserted in the code produced for them.

Input/output operations are performed by the runtime routines. IBCOM# is the initial entry of a transfer vector in IHCFCOMH through which all input/output except NAMELIST is performed. (IHCFCOMH is further discussed in Appendix F.) The pointer to

IBCOM# points to the routine on the GLOBAL SPROG roll.

NAMELIST WRITE

For the form WRITE (a, x), the NAMELIST WRITE, the following changes are made to the Polish notation given above:

1. The FORMAT driver is replaced by a NAMELIST driver.
2. The pointer to the FORMAT is replaced by a pointer to the NAMELIST.
3. The code word value is changed.
4. The IBCOM# entry is replaced by the value zero, since NAMELIST input/output is not handled through IBCOM.
5. The pointer to IBCOM# is replaced by a pointer to the NAMELIST WRITE routine.
6. No I/O list may appear.

UNFORMATTED WRITE

For the form WRITE (a) list, the unformatted WRITE, the following changes are made to the Polish notation given above:

1. The FORMAT driver is removed.
2. The pointer to the FORMAT is removed.
3. The IBCOM# entry, formatted WRITE, is replaced by the IBCOM# entry, unformatted WRITE.

PRINT

The Polish notation generated for the form PRINT b, list is identical to that given for the formatted WRITE statement, with the following changes:

1. No END= or ERR= drivers may appear, nor may the corresponding pointers to labels.
2. The code word value is changed.

PUNCH

The Polish notation for the statement PUNCH b, list is as given for the formatted WRITE with the following changes:

1. No END= or ERR= drivers may appear, nor may the corresponding pointers to labels.
2. The code word value is changed.

<u>DIRECT ACCESS STATEMENTS</u>

The following paragraphs discuss the Polish notation produced for the direct access input/output statements.

READ, DIRECT ACCESS

For the forms READ (a'b,b) list and READ (a'r) list, the following Polish notation is generated:

```
            4 bytes
┌──────────────────────────────┐
│pointer to a                  │
├──────────────────────────────┤
│direct IO driver              │
├──────────────────────────────┤
│                              │ \
├──────────────────────────────┤  |
│              .               │  |  Polish for
│              .               │  >  r
│              .               │  |
├──────────────────────────────┤  |
│                              │ /
├──────────────────────────────┤
│expression driver             │
├──────────────────────────────┤
│pointer to b                  │
├──────────────────────────────┤
│ERR= driver                   │
├──────────────────────────────┤
│pointer to ERR label          │
├──────────────────────────────┤
│IOL driver                    │
├──────────────────────────────┤
│                              │ \
├──────────────────────────────┤  |
│              .               │  |  Polish for
│              .               │  >  I/O list
│              .               │  |
├──────────────────────────────┤  |
│                              │ /
├──────────────────────────────┤
│code word                     │
├──────────────────────────────┤
│IBCOM entry, READ             │
├──────────────────────────────┤
│pointer to IBCOM#             │
├──────────────────────────────┤
│READ WRITE driver             │
├──────────────────────────────┤
│statement number              │
└──────────────────────────────┘
```

The END= and ERR= drivers and the pointers following them appear only if the END and ERR options are used in the source statement; either one or both may appear, and in any order with respect to each other. If b does not appear in the source statement (the second form), the corresponding pointer does not appear in the Polish notation. If the I/O list does not appear in the source statement, the Polish notation for the I/O list is omitted from the Polish, but the IOL driver appears nonetheless.

The code word contains zero in its high-order three bytes, and, in its low-order byte, a unique code specifying the operation and unit for the input/output statement. This code word distinguishes the direct access statements from other input/output statements and is inserted in the code produced for them.

WRITE, DIRECT ACCESS

The Polish notation produced for the forms WRITE (a'r,b) list and WRITE (a'r) list is identical to that produced for the corresponding forms of the READ, direct access statement with the following exceptions:

1. The IBCOM entry, READ is replaced by the appropriate IBCOM entry, WRITE.
2. The value of the code word is changed.

FIND

The Polish notation produced for this statement is identical to that for an unformatted direct access READ statement given above, with the exception that the code word is changed to indicate the FIND statement.

DEFINE FILE

The form of this statement is:

DEFINE FILE a1 (m1,11,f1,v1),a2 (m2,12,f2,v2),...,an(mn,1n,fn,vn)

The Polish notation produced for it is:

4 bytes	
pointer to a1	file 1 data
pointer to m1	
pointer to 11	
E, L or U	
pointer to v1	
pointer to a2	file 2 data
. . .	
pointer to v2	
. . .	
pointer to an	file n data
. . .	
pointer to vn	
DEFINE FILE driver	
statement number	

where the fourth word of each set of file data holds the BCD character E, L, or U in the high-order byte and zeros in the remaining bytes.

<u>END FILE STATEMENT</u>

The Polish notation produced for END FILE is:

4 bytes
pointer to a (data set)
IBCOM entry for END FILE
pointer to IBCOM
BSREF driver
statement number

REWIND STATEMENT

The Polish notation produced for the REWIND statement is identical to that for the END FILE statement with the exception that the IBCOM entry for END FILE is replaced by the IBCOM entry for REWIND.

BACKSPACE STATEMENT

The Polish notation produced for the BACKSPACE statement is identical to that for the END FILE statement, except that the IBCOM entry for END FILE is replaced by the IBCOM entry for BACKSPACE.

STATEMENT FUNCTION

The Polish notation generated for a statement function is:

4 bytes	
pointer to function name	
	Polish for right side
. . .	
statement function driver	
statement number	

FUNCTION STATEMENT

The Polish notation produced for the FUNCTION statement is:

4 bytes
pointer to ENTRY name
FUNCTION driver
statement number

where the pointer points to the ENTRY NAMES roll.

FUNCTION (STATEMENT OR SUBPROGRAM) REFERENCE

The Polish notation generated for a reference to a function is:

4 bytes	
subprogram driver	
pointer to function name	
number of arguments	
expression driver	
	Polish for argument 1
. . .	
expression driver	
	Polish for argument 2
. . .	
expression driver	
. . .	
	Polish for argument n
. . .	
expression driver	
pointer to function name	

This Polish notation is part of the Polish notation for the expression in which the function reference occurs.

SUBROUTINE STATEMENT

The Polish notation generated for the SUBROUTINE statement is:

4 bytes
pointer to ENTRY name
SUBROUTINE driver
statement number

where the pointer points to the ENTRY NAMES roll.

CALL STATEMENT

The Polish notation for the CALL statement is:

4 bytes

subprogram driver	
pointer to subprogram name	
number of arguments	
expression driver	
	Polish for argument 1
. . .	
expression driver	
	Polish for argument 2
. . .	
expression driver	
. . .	
	Polish for argument n
. . .	
expression driver	
pointer to subprogram name	
pointer to x1	label arguments
pointer to x2	
. . .	
pointer to xn	
number of label arguments	
computed GO TO driver	
CALL driver	
statement number	

Label arguments are not counted in the "number of arguments" which appears as the third word of the Polish notation, and no representation of them appears in the Polish notation for the arguments. All label arguments are grouped together at the bottom of the Polish as indicated. If no label arguments exist, the section from the "pointer to x1" to and including the "computed GO TO driver" does not appear.

DEBUG FACILITY STATEMENTS

The following paragraphs describe the Polish notation produced for the statements of the debug facility.

AT

The Polish notation generated for the AT statement is:

4 bytes

pointer to AT group
AT driver
statement number

The pointer points to the AT roll group which contains the information relating to the AT statement represented by the Polish notation.

TRACE ON

The Polish notation generated for the TRACE ON statement is:

4 bytes

TRACE ON driver
statement number

TRACE OFF

The Polish notation generated for the TRACE OFF statement is:

4 bytes

TRACE OFF driver
statement number

DISPLAY

The Polish notation generated for the DISPLAY statement is:

4 bytes

pointer to NAMELIST WRITE
0
NAMELIST pointer
DISPLAY driver
statement number

where the pointer to NAMELIST WRITE points to this routine on the GLOBAL SPROG roll; the value zero is placed on the roll for conformity with other NAMELIST input/output statements; the NAMELIST pointer points to a group constructed for the DISPLAY statement on the NAMELIST NAMES roll.

This appendix describes the code produced by the FORTRAN IV (G) compiler for various types of source module statements.

BRANCHES

All branch instructions in the object module consist of a load from the branch table, followed by a BCR instruction, either conditional or unconditional, which uses the branch table value as its target.

The production of this code depends on the operation of Allocate, which replaces all jump target labels on the LBL roll with pointers to entries in the object module branch table. Using this information, Gen can write the load and branch instructions even though the address of the target may not yet be known.

When Gen encounters a labeled statement which is a jump target, it sets the appropriate entry in the branch table to the address of the first instruction it produces for that statement.

COMPUTED GO TO STATEMENT

The following code is generated for the Computed Go To statement:

```
L       15,variable
SLL     15,2
BALR    14,0
LTR     15,15
BNH     4n+22(0,14)
LA      1,4n(0,0)
CR      15,1
BH      4n+22(0,14)
L       1,18(15,14)
BR      1
.
.
.
n address constants
.
.
.
```

where variable is the Computed Go To variable, n is the number of branch points, and 4n is the length of the list of n address constants.

DO STATEMENT

The use of a DO loop in a FORTRAN program can be described by the following example:

```
DO 5 I = m1,m2,m3
       .
       .
       .
5 CONTINUE
```

When the DO statement is processed during phase 4, the following takes place:

1. The code

   ```
      L    R0,m1
   A  ST   R0,I
   ```

   is generated, where the label A is constructed by Gen.

2. The address of the instruction labeled A is placed in the branch table.

3. An entry is made on the DO LOOPS OPEN roll which contains pointers to m2, m3, the label A, I, and the label 5.

On receiving the Polish notation for the CONTINUE statement in the example, phase 4 produces the following code:

```
L       R0,I
L       R1, branch table
L       R2,m3
L       R3,m2
BXLE    R0,R2,0(R1)
```

where the load from the branch table sets R1 to the address of the made label A. When this code has been completed, phase 4 removes the bottom entry from the DO LOOPS OPEN roll.

STATEMENT FUNCTIONS

The following code is generated at the beginning of each statement function:

```
STM     14,6,14(0,15)
LR      7,14
LR      9,1
LR      6,15
B       42(0,15)
.
.
.
seven word buffer
.
.
.
```

The buffer is followed by the code for the statement function itself, including the code to load the return value. The following code closes the statement function:

```
LR      14,7
LM      6,12,14(6)
BR      14
```

SUBROUTINE AND FUNCTION SUBPROGRAMS

The following code is generated to save required information at the main entry to each SUBROUTINE and FUNCTION subprogram:

```
B       X(0,15)
DC      AL1(length of Ident)
DC      CLn(Ident)
STM     14,12,12(13)
LM      2,3,40(15)
LR      4,13
L       13,36(0,15)
ST      13,8(0,4)
STM     3,4,0(13)
BR      2
DC      (ADDRESS SAVE AREA)
DC      (ADDRESS PROLOGUE)
DC      (ADDRESS EPILOGUE)
```

This code is followed by the following code for saving required information for each of the ENTRYs to the subprogram (the sequence of code appears once for each ENTRY, in the order of the ENTRYs):

```
B       X(0,15)
DC      AL1(length of ident)
DC      CLn(Ident)
STM     14,12,12(13)
LM      2,3,32(15)
L       15,28(0,15)
B       20(0,15)
DC      (ADDRESS PROLOGUE)
DC      (ADDRESS EPILOGUE)
```

The save code for the ENTRYs to the subprogram is followed by a PROLOGUE, which transfers arguments to the subprogram, and an EPILOGUE, which returns arguments to the calling routine for the main entry to the subprogram and for each ENTRY to the subprogram.

The following code is produced for the RETURN statement:

```
SR      15,15
L       14,0(0,13)
BR      14
```

which branches to the appropriate EPILOGUE.

The following code is produced for the RETURN I statement:

```
L       15,I
SLL     15,2
L       14,0(0,13)
BR      14
```

which also branches to the appropriate EPILOGUE.

The PROLOGUE code generated for each entry point to the subprogram moves arguments as required and branches to the entry. The following code is generated to move each call by name argument:

```
L       2,n(0,1)
ST      2,global dmy
```

where n is the argument number (the arguments for each entry point are numbered from one) multiplied by four.

The following code is generated to move each call by value argument:

```
L       2,n(0,1)
MVC     global dmy(x),0(2)
```

where n is the argument number multiplied by four, and x is the size of the dummy.

Code to calculate dummy dimensions follows the code to move arguments.

The following code is generated at the close of all PROLOGUEs:

```
BALR    2,0
L       3,6(0,2)
BR      3
DC      (ADDRESS OF CODE ENTRY POINT)
```

The EPILOGUE code generated for each entry point to a subprogram moves arguments back to the calling routine and returns to it, as dictated by the RETURN or RETURN I statement.

The first instructions in each EPILOGUE are:

```
L       1,4(0,13)
L       1,24(0,1)
```

The following code is generated to return each call by value argument:

```
L       2,n(0,1)
MVC     0(x,2),global dmy
```

where n is the argument number multiplied by four and x is the size of the dummy.

For FUNCTION subprograms, the following instruction is generated:

```
Lx      0,entry name
```

where x is the instruction mode. If the FUNCTION is complex, two load instructions are required.

The following code is generated for the closing of each EPILOGUE:

```
L       13,4(0,13)
L       14,12(0,13)
LM      2,12,28(13)
MVI     12(13),255
BR      14
```

INPUT/OUTPUT OPERATIONS

The following paragraphs describe the code produced for the FORTRAN input/output statements. The generated instructions set up necessary parameters and branch into the IBCOM# transfer table. This table has the following format:

```
IBCOM#     Main entry, formatted READ
       +4  Main entry, formatted WRITE
       +8  Second list item, formatted
      +12  Second list array, formatted
      +16  Final entry, end of I/O list
      +20  Main entry, unformatted READ
      +24  Main entry, unformatted WRITE
      +28  Second list item, unformatted
      +32  Second list array, unformatted
      +36  Final entry, end of I/O list
      +40  Backspace tape
      +44  Rewind tape
      +48  Write tapemark
      +52  STOP
      +56  PAUSE
      +60  IBERR execution error monitor
      +64  IBFINT interruption processor
      +68  IBEXIT job termination
```

FORMATTED READ AND WRITE STATEMENTS

The code produced for these statements is:

```
CNOP    0,4
L       15,=V(IBCOM#)
BAL     14,N(15)
DC      XL0.4'PI',XL0.4'UI',AL3(UNIT)
DC      AL1(FI),AL3(FORMAT)
DC      AL4(EOFADD)             "optional"
DC      AL4(ERRADD)             "optional"
```

where:

PI = 0 if neither EOF nor ERR is specified
= 1 if EOF only is specified
= 2 if ERR only is specified
= 3 if both EOF and ERR are specified

UI = 0 if unit is an integer constant
= 1 if unit is a variable name
= 4 if unit is the standard system unit

FI = X'00' if FORMAT is a statement label
= X'01' if FORMAT is an array name

N = 0 for READ
= 4 for WRITE

UI = 4 is used for debug and for READ b, list, PRINT b, list and PUNCH b, list.

SECOND LIST ITEM, FORMATTED

The code produced is:

```
L       15,=V(IBCOM#)
BAL     14,8(15)
DC      XL1'L',LX0.4'T'.XL0.4'X'
        XL0.4'B',XL1.4'D'
```

where:

L = the size in bytes of the item

T = 2 for a logical 1-byte item
= 3 for a logical fullword item
= 4 for a halfword integer item
= 5 for a fullword integer item
= 6 for a double-precision real item
= 7 for a single-precision real item
= 8 for a double-precision complex item
= 9 for a single-precision complex item
= A for a literal item (not currently compiler-generated)

X, B, and D are, respectively, the index, base, and displacement which specify the item address.

SECOND LIST ARRAY, FORMATTED

The code produced is:

```
L       15,=V(IBCOM#)
BAL     14,12(15)
DC      LX1'SPAN',AL3(ADDRESS)
DC      XL1'L',XL0.4'T',XL2.4'ELEMENTS'
```

where:

SPAN (not used)

ADDRESS = the beginning location of the array

L = the size in bytes of the array element

T = the values given for items

ELEMENTS = the number of elements in the array

FINAL LIST ENTRY, FORMATTED

The code produced is:

```
L       15,=V(IBCOM#)
BAL     14,16(15)
```

UNFORMATTED READ AND WRITE STATEMENTS

The code produced for these statements is:

```
CNOP    0,4
L       15,=V(IBCOM#)
BAL     14,N(15)
DC      XL0.4'PI',XL0.4'UI,AL3(UNIT)
DC      AL4(EOFADD)            "optional"
DC      AL4(ERRADD)            "optional"
```

where:

PI, UI, UNIT, EOFADD and ERRADD have the same values as those given in the formatted READ/WRITE definition.

N = 20 for READ
  = 24 for WRITE

SECOND LIST ITEM, UNFORMATTED

The code produced is:

```
L       15,=V(IBCOM#)
BAL     14,28(15)
DC      XL1'L',XL0.4'0',XL0.4'X',
        XL0.4'B',XL1.4'D'
```

where:

L = the size in bytes of the item

X, B and D are, respectively, the index, base, and displacement which specify the address of the item.

SECOND LIST ARRAY, UNFORMATTED

The code produced is:

```
L       15,=V(IBCOM#)
BAL     14,32(L)
DC      XL1'SPAN',AL3(ADDRESS)
DC      XL1'L',AL3(ELEMENTS)
```

where SPAN, ADDRESS, L, and ELEMENTS have the meanings described in second list array, formatted.

FINAL LIST ENTRY, UNFORMATTED

The code produced is:

```
L       15,=V(IBCOM#)
BAL     14,36(15)
```

BACKSPACE, REWIND, AND WRITE TAPEMARK

The code produced is:

```
CNOP    0,4
L       15,=V(IBCOM#)
BAL     14,N(15)
DC      XL1'FLAG', AL3(UNIT)
```

where:

FLAG = 0 if unit is an integer
     = any other bit pattern if unit is a variable.

N = 40 for BACKSPACE
  = 44 for REWIND
  = 48 for write tapemark

STOP AND PAUSE STATEMENTS

The code produced for these statements is:

```
L       15,=V(IBCOM#)
BAL     14,N(15)
DC      AL1(LENGTH)
DC      C'TEXT'
```

where:

LENGTH is the number of bytes in the 'TEXT' message

TEXT is an alphameric number or message (TEXT = '40404040F0' if the STOP or PAUSE message is blank).

N = 52 for STOP
  = 56 for PAUSE

NAMELIST READ AND WRITE

The code produced is:*

```
CNOP    0,4
L       15,=V(FWRNL#)
BAL     14,0(15)
DC      XL0.4'PI',XL0.4'UI',AL3(UNIT)
DC      AL4(NAMELIST)
DC      AL4(EOFADD)
DC      AL4(ERRADD)
```

where:

PI, UI, and UNIT are as described for formatted READ and WRITE

* The "L 15,=V(FWRNL#)" shown is for write; the code produced for read is "L 15,+V(FRDNL#)."

DEFINE FILE STATEMENT

The form of the parameters specified in the statement is:

$a_1(m_1,f_1,r_1,v_1),\ldots a_n(m_n,f_n,r_n,v_n)$

The following code is generated in the object module prologue:

```
LA      R1,LIST
L       L,=V(DIOCS#)
BALR    R2,L
```

where:

R_1 = 1

L = 15

R_2 = 14

The following parameter list is also generated:

```
DC      X'a1',AL3(m1)
DC      C'f1',AL3(r1)
DC      X'00',AL3(v1)
           .
           .
           .
DC      X'an',AL3(mn)
DC      C'fn',AL3(rn)
DC      X'80',AL3(vn)
```

The third DC in the group is changed to

```
DC          X'01',AL3(vi)
```

if the associated variable is a halfword variable. In the last group, it becomes X'81',AL3(v_n) in this case.

FIND STATEMENT

The code produced is:

```
CNOP        0,4
L           15,=V(IBCOM#)
BAL         14,20(15)
DC          XL0.4'PI',XL0.4'UI',AL3(UNIT)
DC          XL1'VI',AL3(r)
```

PI = C

UI = 0 if the unit is a constant
   = 1 if the unit is a variable name

VI = 00 if the record number is a constant
   = 01 if the record number is a variable name

Note that 20 is the IBCOM entry point for an unformatted READ.

DIRECT ACCESS READ AND WRITE STATEMENTS

The code produced for these statements is:

```
CNOP    0.4
L       15,=V(IBCOM#)
BAL     14,N(15)
DC      XL0.4'PI',XL0.4'UI'AL3(UNIT)
DC      AL1(FI),AL3(FORMAT)
DC      AL1(VI),AL3(r)
DC      AL4(ERRADD) "may only appear for
        READ"
```

where:

PI = 8 if ERR is not specified
 = A if ERR is specified, which is only possible for READ

UI = 0 if the unit is an integer constant
 = 1 if the unit is a variable name

FI = 00 if the FORMAT is a statement label
 = 01 if the FORMAT is an array name

VI = 00 if r (the record number) is a constant
 = 01 if r is a variable name

The entry points which may appear (N) are 0, 4, 20, or 24. If 20 or 24 appears (indicating an unformatted operation), the second DC does not appear.

FORMAT STATEMENTS

FORMAT statements are stored after literal constants in the object module.

The FORMAT specifications are recoded from their source module form so that each unit of information in the FORMAT statement occupies one byte of storage. Each integer which appears in the FORMAT statement (i.e., a scale factor, field width, number of fractional digits, repetition count) is converted to a 1-byte binary value. Decimal points used to separate field width from the number of fractional digits in the source module FORMAT statement are dropped; all other characters appearing in the source module statement are represented by 1-byte hexadecimal codes. The following sections describe the encoding scheme which is used.

FORMAT Beginning and Ending Parentheses

The beginning and ending parentheses of the FORMAT statement are represented by the hexadecimal codes 02 and 22, respectively.

Slashes

The slashes appearing in the FORMAT statement are represented by the hexadecimal code 1E.

Internal Parentheses

Parentheses used to enclose groups of FORMAT specifications within the FORMAT statement are represented by the codes 04 and 1C for the left and right parenthesis, respectively. The code for the left parenthesis is always followed by the 1-byte value of the repetition count which preceded the parenthesis in the source module statement. A value of one is inserted if no repetition count appeared.

Repetition of Individual FORMAT Specifications

Whenever the source module FORMAT statement contains a field specification of the form aIw, aFw.d, aEw.d, aDw.d, or aAw, where the repetition count "a" is present, the hexadecimal code 06 is produced to indicate the field repetition. This code is followed by the 1-byte value of "a".

I,F,E, and D FORMAT Codes

The I and F FORMAT codes are represented by the hexadecimal values 10 and 0A, respectively. The I code is followed by the 1-byte field width value; the F code is followed by two bytes, the first containing the field width (w) and the second containing the number of fractional digits (d).

E and D FORMAT codes are represented by the hexadecimal values 0C and 0E, respectively. This value is always followed by two bytes which represent the field width and the number of fractional digits, respectively.

A FORMAT Code

The A FORMAT code is represented by the hexadecimal value 14. This representation is always followed by the 1-byte value of w, the number of characters of data.

Literal Data

The H FORMAT code and the quotation marks used to enclose literal data are both represented by the hexadecimal value 1A. This code is followed by the character count (w in the case of the H specification, the number of characters enclosed in quotation marks in the case of the use of quotation marks). The literal data follows the character count.

X FORMAT Code

The specification wX results in the production of the hexadecimal code 18 for the X; this is followed by the 1-byte value of w.

T FORMAT Code

The T FORMAT code is represented by the value 12. The print position, w, is represented by a 1-byte binary value.

Scale Factor-P

The P scale factor in the source module FORMAT statement is represented by the hexadecimal value 08. This code is followed by the value of the scale factor, if it was positive. If the scale factor was negative, 128_1 is added to it before it is stored following the P representation.

G FORMAT Code

The G FORMAT Code is represented by the hexadecimal value 20. This value is always followed by two bytes which represent the field width and the number of significant digits, respectively.

L FORMAT Code

The L FORMAT code is represented by the hexadecimal value 16. This value is followed by the 1-byte field width.

Z FORMAT Code

The Z FORMAT code is represented by the hexadecimal value 24. This value is followed by the 1-byte field width.

DEBUG FACILITY

The following paragraphs describe the code produced for the FORTRAN Debug Facility statements. The generated instructions set up parameters and branch into the DEBUG# transfer table. The object-time routines which support the Debug Facility are described in Appendix E.

DEBUG STATEMENT

When the source module includes a DEBUG statement, debug calls are generated before and after each sequence of calls to IBCOM for source module input/output statements. Additional debug calls are generated to satisfy the options listed in the DEBUG statement.

Beginning of Input/Output

The following code appears before the first call to IBCOM for an input or output operation:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,44(0,15)
```

End of Input/Output

The following code appears after the last call to IBCOM for an input or output operation:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,48(0,15)
```

UNIT Option

When the DEBUG statement does not include the UNIT option, the object-time debug routine automatically writes debug output on SYSOUT. When UNIT is specified, the following code is generated at the beginning of the object module:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,12(0,15)
DC      F'DSRN'
```

where DSRN is the data set reference number to be used for all subsequent debug output.

TRACE Option

When the TRACE option is specified in the source module DEBUG statement, the TRACE call is inserted immediately before the code for every labeled statement. The code is:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,0(0,15)
DC      F'LABEL'
```

where LABEL is the label of the following statement.

SUBTRACE Option

When the SUBTRACE option is listed in the source DEBUG statement, two sequences of code are produced: one at the entry to the object module, and one prior to each RETURN.

SUBTRACE ENTRY: The debug call is made at the beginning of the object module. The call is:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,4(0,15)
```

At the time of the call, register 13 contains the address of the SAVE AREA, the fifth word of which contains the address of the subprogram identification. Bytes 6 through 11 of the subprogram identification are the subprogram name.

SUBTRACE RETURN: The debug call is made immediately before the RETURN statement. The call is:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,8(0,15)
```

INIT Option

When the INIT option is given in the source module DEBUG statement, a debug call is produced for every assignment to a variable, or to a listed variable if a list is provided. The call immediately follows each assignment, including those which occur as a result of a READ statement or a subprogram call. Three calls may occur, depending on the type of variable (scalar or array) and the method of assignment.

INIT SCALAR VARIABLE: The following code is produced after each assignment of value to a scalar variable covered by the INIT option:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,16(0,15)
DC      CL6'NAME',CL2'  '
DC      XL1'L',XL0.4'T',XL0.4'X',XL0.4'B',
                                    XL1.4'D'
```

where:

NAME is the name of the variable which was set.

L is the length of the variable in bytes.

T is the type code for the variable:

- = 2 for a logical 1-byte item
- = 3 for a logical fullword item
- = 4 for a halfword integer item
- = 5 for a fullword integer item
- = 6 for a double-precision real item
- = 7 for a single-precision real item
- = 8 for a double-precision complex item
- = 9 for a single-precision complex item
- = A for a literal item (not currently compiler generated)

X, B, and D are, respectively, the index, base, and displacement which locate the item.

INIT ARRAY ITEM: The following code is produced after each assignment of value to an array element:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,20(0,15)
DC      CL6'NAME',CL2'  '
DC      XL1'L',XL0.4'T',XL0.4'X',XL0.4'B',
                                    XL1.4'D'
DC      XL1'TAG',AL3(ADDRESS)
```

where:

ADDRESS IS THE LOCATION OF THE FIRST array element if TAG = 0, or ADDRESS is a pointer to the location of the first array element if TAG ≠ 0.

NAME, L, T, X, B, and D are as described for a scalar variable.

INIT FULL ARRAY: The following code is produced when a full array is set by means of an input statement specifying the array name or when the array name appears as an argument to a subprogram:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,24(0,15)
DC      CL6'NAME',CL2'  '
DC      A(ADDRESS)
DC      XL1'L',XL0.4'T',XL2.4'00000'
DC      A(ELEMENTS)
```

where:

ADDRESS is the location of the first array element.

ELEMENTS is a pointer to a word containing the number of elements in the array.

NAME, L, and T are as described for a scalar variable.

SUBCHK Option

A debug call is produced for each reference to an array element when the SUBCHK option appears without a list of array names; when the list is given, only references to the listed arrays produce debug calls. The debug call appears before the reference to the array, and is:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,28(0,15)
DC      CL6'NAME',CL2'  '
DC      XL1'TAG',AL3(ADDRESS)
DC      AL4(ELEMENTS)
```

where:

NAME is the array name.

ADDRESS is the location of the first array element if TAG = 0, or ADDRESS is a pointer to the location of the first array element if TAG ≠ 0.

ELEMENTS is a pointer to a word containing the number of elements in the array.

AT STATEMENT

The AT statement specifies the label, L, of a statement whose operation should be immediately preceded by the operation of the statements following the AT. As a result of the AT statement, an unconditional branch to the location of the first statement following the AT is inserted before the first instruction generated for the statement labeled L. This branch precedes any TRACE or SUBTRACE calls which may be written for statement L.

The branch, like all branches performed in the object module, consists of a load from the branch table, followed by a BCR instruction. The branch table entry referred to is one constructed for a label which the compiler provides for the statement following the AT.

TRACE ON STATEMENT

The debug call produced for the TRACE ON statement appears at the location of the TRACE ON statement itself; the call is:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,32(0,15)
```

TRACE OFF STATEMENT

The debug call produced for the TRACE OFF statement appears at the location of the TRACE OFF statement itself; the call is:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,36(0,15)
```

DISPLAY STATEMENT

The code for the DISPLAY statement is:

```
L       15,=V(DEBUG#)
CNOP    0,4
BAL     14,40(0,15)
DC      A(NAMELIST)
DC      A(FWRNL#)
```

where NAMELIST is the address of the NAMELIST table generated from the DISPLAY list by the compiler. This code appears at the location of the DISPLAY statement itself.

The information provided in this appendix has its primary use in connection with a listing of the compiler. The label lists indicate the chart on which a specific label can be found, or, for routines which are not flowcharted, they provide a description of the routine.

PARSE LABEL LIST

The labels enumerated in the following list are used in the flowcharts provided for the illustration of the major routines used in Parse.

Label	Chart ID	Routine Name
G0630	04	START COMPILER
G0631	04	STATEMENT PROCESS
G0837	BA	PRINT AND READ SOURCE
G0632	BB	STA INIT
G0635	BC	LBL FIELD XLATE
G0636	BD	STA XLATE
G0633	BE	STA FINAL
G0642	BF	ACTIVE END STA XLATE
G0844	BG	PROCESS POLISH

SUPPLEMENTARY PARSE LABEL LIST

The routines described in this section are listed by G number labels which are presented in ascending order. These routines are those used in the operation of Parse which are not shown in the section of flowcharts for the phase.

Label	Routine Name	Comments
G0287	REASSIGN MEMORY	Obtains additional core storage, if possible, for a specific roll by pushing up the rolls that precede the requesting roll in the block of storage. If this is not possible, it requests more core storage and, if none is available, enters PRESS MEMORY.
G0637	ASSIGNMENT STA XLATE	Constructs the Polish notation for an assignment statement.
G0638	ARITH FUN DEF STA XLATE	Constructs the Polish notation for an arithmetic function definition statement.
G0639	ASSIGNMENT VAR CHECK	Checks the mode of assignment variable and the expression for conflict in type specification.
G0640	LITERAL TEST	Determines the statement type and transfers to the indicated statement processing routine.
G0641	END STA XLATE	Determines the nature of the statement and transfers to the appropriate translation routine for non-END; translates END.
G0643	DO STA XLATE	Constructs the Polish notation for the DO statement. Locates the innermost DO statement in a nest of DO's, and sets up extended range checking.
G0644	DO STA CONTROL XLATE	Interprets the loop control specification in the DO statement and constructs the Polish notation for these controls.
G0645	DIMENSION STA XLATE	Determines the validity of the specifications in the DIMENSION statement and constructs roll entries.
G0646	GOTO STA XLATE	Determines the type of GO TO statement, and constructs the Polish notation for a GO TO statement.
G0647	CGOTO STA	Constructs the Polish notation for a Computed GO TO statement.
G0648	ASSIGNED GOTO STA XLATE	Constructs the Polish notation for an Assigned GO TO statement.
G0649	ASSIGN STA XLATE	Controls the constructions of the Polish notation for an ASSIGN statement.

Label	Routine Name	Comments
G0650	IF STA XLATE	Constructs the Polish notation for an IF statement.
G0651	LOGICAL IF STA XLATE	Constructs the Polish notation for a logical IF statement.
G0652	IMPLICIT STA XLATE	Checks the IMPLICIT statement and controls the construction of the roll entries for the statement.
G0653	REGISTER RANGE	Controls character entries for an IMPLICIT statement.
G0654	REGISTER IMPLICIT CHAR	Places the characters in the IMPLICIT statement on the IMPLICIT roll.
G0655	SCAN FOR TYPE QT AND SIZE	Determines the mode and size of the variables in specification statements.
G0656	CONTINUE STA XLATE	Constructs the Polish notation for a continue statement.
G0657	CALL STA XLATE	Constructs the Polish notation for a CALL statement.
G0658	EXTERNAL STA XLATE	Validates the use of the EXTERNAL statement and constructs roll entries.
G0659	FORMAT STA XLATE	Validates the use of the FORMAT statement and controls the construction of the Polish notation for the statement.
G0660	FORMAT STA END	Builds the FORMAT roll from the information obtained from the processing of the statement.
G0661	FORMAT LIST SCAN	Checks the form of the literal content of the FORMAT statement.
G0662	FORMAT BASIC SCAN	Interprets the FORMAT list and constructs the Polish notation for the list.
G0663	ISCAN TEST	Checks the size of the inteter constant or variable specified.
G0664	PACK H CODE	Interprets the specification for the H format code.
G0665	PACK FORMAT QUOTE	Controls the registering of the contents of a literal quote specified in a FORMAT statement.
G0666	REWIND STA XLATE	Constructs the Polish notation for a REWIND statement.
G0667	BACKSPACE STA XLATE	Constructs the Polish notation for a BACKSPACE statement.
G0668	END FILE STA XLATE	Constructs the Polish notation for an END FILE statement.
G0669	END FILE END	Completes the Polish notation for input/output control statements.
G0670	BLOCK DATA STA XLATE	Validates the use of the BLOCK DATA statement.
G0671	STOP STA XLATE	Sets up the Polish notation for the STOP statement.
G0672	STOP CODE ENTRY	Sets up the Polish notation for the STOP statement.
G0673	PAUSE STA XLATE	Controls the interpretation of the PAUSE statement.
G0674	PAUSE STOP COMMON	Checks the form of the specified statement and controls the construction of the Polish notation for the statement.
G0675	PAUSE STOP END	Registers the constructed Polish notation on the POLISH roll.
G0676	INIT LITERAL FOR STOP PAUSE	Controls the interpretation of the message specified in the PAUSE statement.
G0677	NAMELIST STA XLATE	Constructs the roll entries for the NAMELIST statement.
G0678	COMMON STA XLATE	Constructs the roll entries for the COMMON specification.

Label	Routine Name	Comments
G0679	TEST ID ARRAY OR SCALAR	Validates the identification of the array or scalar used in COMMON.
G0680	DOUBLE PRE STA XLATE	Checks the use of the DOUBLE PRECISION statement and controls the interpretation of the statement.
G0681	TYPE STA XLATE	Interprets and constructs the roll entries for the type specification statement.
G0682	SCAN FOR SIZE	Checks the size specification for the variables in type statements.
G0683	TYPE SEARCH TEST AND REG	Checks the identification of the variables in the type specification in statement for previous definition and defines if correct.
G0684	ENTRY STA XLATE	Constructs the Polish notation and roll entries for an ENTRY statement.
G0685	FUNCTION STA XLATE	These routines control the construction of the Polish notation for a FUNCTION subprogram by invoking the routines which interpret the contents of the statement.
G0686	TYPED FUNCTION STA XLATE	
G0687	FUNCTION ENTRY STA XLATE XLATE	
G0688	SUBROUTINE STA XLATE	These routines control the construction of the Polish notation for a SUBROUTINE subprogram by invoking the routine which interprets the contents of the statement.
G0689	SUBROUTINE ENTRY STA XLATE	
G0690	SUBPROGRAM END	Common closing routine for ENTRY, FUNCTION, and SUBROUTINE statements.
G0691	SPROG NAME SCAN AND REG	Checks the identification of the SUBROUTINE or FUNCTION subprogram for conflicts in definition.
G0692	TEST ORDER	Checks the order in which the SUBROUTINE or FUNCTION statement appears in the source module.
G0693	DMY SEQ SCAN	Checks the designation of the dummy variables for call by name or call by value.
G0694	GLOBAL DMY SCAN AND TEST	Checks the identification of the global dummy for a possible conflict in definition.
G0695	DEFINE FILE STA XLATE	Constructs the Polish notation for the DEFINE FILE statement.
G0696	DATA STA XLATE	Constructs the Polish notation and roll entries for the DATA statement.
G0697	DATA CONST XLATE	Interprets the constants specified in the DATA statement.
G0698	INIT DATA VAR GROUP	Determines and sets up the number of elements specified in the DATA statement.
G0699	DATA CONST ANALYSIS	Validates the specification of the constants used in the DATA statement.
G0700	DATA VAR TEST AND SIZE	Checks the definition of the variables specified in the DATA statement for usage conflict, and registers the variables if no conflict is found.
G0701	MOVE TO TEMP POLISH ROLL	Moves information for DATA statement to TEMP POLISH roll from WORK roll.
G0702	READ STA XLATE	Checks the type of READ statement and controls the interpretation of the statement.
G0704	READ WRITE STA XLATE	Interprets the elements of the READ or WRITE statement and constructs the Polish notation for the statement.
G0705	END QT XLATE	Constructs the Polish notation for the END= quote.

Label	Routine Name	Comments
G0706	ERR QT XLATE	Constructs the Polish notation for the ERR= quote in the READ statement.
G0707	REGISTER IBCOM	Inserts a roll entry for a call to IBCOM.
G0708	REGISTER ERROR LINK	Sets the roll entry for the generation of error linkage.
G0709	READ B STA XLATE	Initialize for the construction of the Polish notation for the indicated statement.
G0710	PUNCH STA XLATE	
G0711	PRINT STA XLATE	
G0712	F2 IO XLATE	Constructs the Polish notation for the indicated input/output statement and interprets FORMAT designations associated with the input/output statement.
G0713	IOL LIST XLATE	Interprets and constructs the Polish notation for the list associated with the indicated input/output statement.
G0714	FIND STA XLATE	Constructs the Polish notation for the FIND statement.
G0715	RETURN STA XLATE	Constructs the Polish notation for the RETURN statement.
G0716	EQUIVALENCE STA XLATE	Constructs the roll entries for the EQUIVALENCE statement
G0717	DIMENSION SEQ XLATE	Constructs the roll entries for the dimensions designated for an array.
G0718	TEMP MAKER	Increments pointer for temporary locations used for dummy dimensions.
G0719	SPECIFICATION STA EXIT	Set flags and return.
G0720	JUMP END	
G0721	ACTIVE END	
G0722	HEAD STA EXIT	

Label	Routine Name	Comments
G0723	STA XLATE EXIT	Replaces the Polish notation for a statement with error linkage if indicated.
G0724	ILLEGAL STA FAIL	These routines set up diagnostic messages for the type of error indicated by the routine name.
G0725	ORDER FAIL	
G0726	ALLOCATION FAIL	
G0727	ILLEGAL NUMBER FAIL	
G0728	SUBSCRIPT FAIL	
G0729	ID CONFLICT FAIL	
G0730	TYPE CONFLICT FAIL	
G0731	VAR SCAN	Checks definition of variables in the source module; defines as scalar if undefined.
G0732	ARRAY SCAN	Constructs the Polish notation and roll entries for array references.
G0733	SUBSCRIPT ANALYSIS	Determines the nature of an array reference for purposes of subscript optimization.
G0734	SCRIPT ITEM ANALYSIS	Determines whether a subscript expression is a linear function of a DO variable, and sets ANSWER BOX.
G0735	NOTE LINEAR SCRIPT	Registers a linear subscript expression on SCRIPT roll.
G0736	RESTORE NONLINEAR SCRIPT	Builds the Polish notation for a nonlinear subscript expression on Polish roll.
G0737	MOVE ON EXIT FALSE	Moves one group from WORK roll to POLISH roll, sets ANSWER BOX to false, and returns.
G0738	SCRIPT SCALAR ANALYSIS	Determines whether a scalar used in a subscript is a DO variable and sets ANSWER BOX.

Label	Routine Name	Comments
G0739	SCRIPT CONST ANALYSIS	Separates constant used in a subscript expression as either induction variable coefficient or additive constant.
G0740	DEFINE SCRIPT GROUP	Creates new group containing zeros on the SCRIPT roll.
G0741	REGISTER SCRIPT GROUP	Defines a subscript expression on the SCRIPT roll by setting the traits, displacement, and array reference.
G0744	TERM SCAN	Initializes the construction of Polish notation for a new term in an expression.
G0745	ELEMENT OP SEQ SCAN	Constructs the Polish notation for a term in an arithmetic expression.
G0746	UNAPPENDED SPROG ARG	Exits from expression scanning on finding an array or subprogram name not followed by a left parenthesis; ensures reference is correct.
G0747	FUNCTION ELEMENT	Determines whether a function call in an expression is to a statement function, a library function, or a global subprogram; calls SPROG ARG SEQ SCAN to scan arguments.
G0748	CONST ELEMENT	Scanning expression, if compiler finds non-letter, non-left parenthesis, it goes here; determines if really a constant.
G0749	SCALAR ELEMENT	Ensures that scalar is registered.
G0750	ELEMENT MOVE	Moves pointer to POLISH roll for any element in expression.
G0751	OP SCAN CHECK DEPOSIT	Determines the operation indicated in an expression, sets up the appropriate driver, and falls through to OP CHECK AND DEPOSIT.
G0752	OP CHECK AND DEPOSIT	The current and previous operations are set up according to a precedence, and a Polish notation is constructed.
G0753	GEN AND REG EXPON SPROG	Determines the nature of an exponentiation, and records the required subprogram on the GLOBAL SPROG roll.
G0754	REG COMPLEX SPROG	Determines the nature of an operation involving complex variables and registers the appropriate routine on the GLOBAL SPROG roll.
G0755	A MODE PICK AND CHECK	Checks and sets mode of operator by inspecting the first of a pair of operands.
G0756	MODE PICK	Actually places mode field in driver.
G0757	B MODE PICK AND CHECK	With second operand and driver set by A MODE PICK AND CHECK, resets driver mode; if complex raised to a power, ensures power is integer.
G0758	MODE CHECK	Determines whether modes of operands are valid in relational and logical operations.
G0759	NUMERIC EXP CHECK	Determines that an operation or an expression is numeric, as opposed to logical, for compatibility.
G0760	NUMERIC EXP CHECK AND PRUNE	Uses NUMERIC EXP CHECK, then prunes bottom of POLISH roll.
G0761	SPROG ARG SEQ SCAN	Constructs the Polish notation for the argument list designated for a subprogram.
G0762	ARG TEST AND PRUNE	Tests the number and type of arguments to library routine; moves label arguments to CALL LBL roll.
G0763	TEST FOR ALTERABLE	Determines whether a scalar has been passed as a subprogram argument.

Label	Routine Name	Comments
G0764	ID SCAN NO USE	Sets a flag tested in MODE SET so that low-order bits of roll are not altered when variable is defined; statement does not use variable.
G0765	ID CLASSIFY NO USE	Goes to ID CLASSIFY after setting flag to indicate variable has not been used and mode should not be set.
G0766	ID SCAN	Compiles name from source in central area and goes to ID CLASSIFY.
G0767	ID CLASSIFY	Determines the classification of a name -- scalar, array, subprogram, etc., and leaves pointer in W0; exits false if name not defined.
G0768	REGISTER SCALAR	Records new name on SCALAR roll.
G0769	REGISTER GLOBAL SPROG REGISTER RUNTIME GS	Determines if name is already a defined subprogram; if not records it on GLOBAL SPROG roll.
G0770	REGISTER GLOBAL SPROG ROLL	Records name on GLOBAL SPROG roll.
G0771	MODE SET	Determines the mode of the indicated variable, logical, integer, complex, etc., and inserts code in pointer in W0.
G0772	CONST SCAN	Controls the translation and recording of constants.
G0773	REGISTER COMPLEX CONST	Records complex and double-precision complex constants not previously defined on appropriate roll.
G0774	REGISTER FL CONST	Records single- and double-precision real constants on appropriate roll when not previously defined.
G0775	REGISTER WORK CONST	Records constant in W0 as new integer constant if not defined.
G0776	REGISTER FX CONST	Records new integer constant if not previosuly defined.
G0777	CONST ANALYSIS	Determines the type of a constant and jumps to proper conversion routine.
G0778	CPLX CONST ANALYSIS	Converts a complex constant.
G0779	CHECK CONST SIGN	Checks for unary minus sign on constant.
G0780	SCAN CONST SIGN	Scans first character of a constant for a sign; sets up driver if unary minus.
G0782	HEXADECIMAL CONST SCAN	Converts a hexadecimal constant.
G0783	REGISTER HEX CONST	Records new c onstant on HEX CONST roll if not previously defined.
G0784	LBL ARG SCAN	Checks validity of a label argument to a subprogram and records label as jump target.
G0785	SCAN HOLLERITH ARGUMENT	Scans an IBM card code argument to a subprogram, and records as literal constant.
G0786	LITERAL CONST SCAN	Distinguishes literal constants from logical; converts and records.
G0787	LITERAL CONST SCAN PAUSE	Packs a literal constant.
G0788	REGISTER LITERAL CONST	Records literal constant on LITERAL CONST roll if not previously defined.
G0789	INIT PACK LITERAL	Initializes for conversion of a literal constant.
G0790	PACK LITERAL COMPLETE	Moves literal constant onto TEMP LITERAL roll if packed.
G0791	PACK LITERAL CONST	Converts a literal constant from source input.
G0792	LOOK FOR ONE QUOTE	Checks for a quotation mark not followed by a second quotation mark; sets ANSWER BOX.

Label	Routine Name	Comments
G0793	PACK TWO FROM WORK	Packs low-order byte from last one or two groups on WORK roll onto LITERAL TEMP roll.
G0794	PACK ONE FROM WORK	
G0795	PACK CRRNT CHAR	Packs current character onto LITERAL TEMP roll.
G0796	PACK CHAR	General routine to actually place a byte in a word which, when complete, is placed on the LITERAL TEMP roll.
G0797	SYMBOL SCAN	Assembles identifier from input in SYMBOL 1, 2, and 3, and returns.
G0798	LOGICAL CONST SCAN	Scans logical constants from source input and records as integers.
G0799	JUMP LBL SCAN AND MOVE	Scans label, defines it as jump target and pointer on POLISH roll. Locates transfers from innermost DO loops that are possible extended range candidates. Also checks for possible re-entry points into innermost DO loops, and tags such points.
G0800	FORMAT LBL SCAN	Scans a label, registers it if necessary, and ensures that it is a FORMAT label if already defined.
G0801	FORMAT LBL TEST	Tests that pointer in W0 indicates format label (vs. jump target label); if not, there is an error.
G0802	LBL SCAN	Scans referenced label, defines on LBL roll if required, produces error messages, leaves pointer in W0.
G0803	REGISTER LBL	Records label on LBL roll if not previously defined; leaves pointer in W0.
G0804	NEXT ZERO LEVEL COMMA NEXT ZERO COMMA OR R PAREN	Scans source input to next comma not in parentheses or to close off a pair of parentheses.
G0805	NEXT ZERO COMMA OR CS	Scans source input until next comma or slash not in parentheses.
G0806	NEXT CLOSING SLASH	Scans source input until second of the next pair of slashes not enclosed in parentheses.
G0807	NEXT ZERO COMMA SLASH OR CRP	Scans source input until next comma or slash not enclosed in parentheses or a closing right parenthesis.
G0808	NEXT ZERO R PAREN	Scans source input until next zero level right parenthesis.
G0809	COMMA TEST	Advances scan arrow and returns ANSWER BOX true if next active character is a comma; if it is a letter, sets up missing comma message, does not advance, and returns true; if it is neither, returns false.
G0810	INTEGER TERM SCAN AND MOVE	Scans integer constant or variable, defines on appropriate roll, puts pointer on POLISH roll.
G0811	INTEGER CONST SCAN AND MOVE	Scans integer constant; defines on FX CONST roll if required; puts pointer on POLISH roll.
G0812	INTEGER VAR SCAN AND MOVE	Scans integer variable; defines on roll if required; puts pointer on POLISH roll.
G0813	INTEGER TEST	Determines whether a pointed to variable or constant is an integer.
G0814	SIGNED INTEGER SCAN	Scans and converts signed integer constant; defines on FX CONST roll if required.
G0815	INTEGER SCAN	Scans and converts an unsigned integer constant and register on FX CONST roll if required.
G0816	DP CONST MAKER	Builds a double-precision constant from source input.
G0817	DP ADJUST CONST	Used in converting floating point numbers; adjusts for E or D field.

Label	Routine Name	Comments
G0818	CONVERT TO FLOAT	Converts integer constant to floating point.
G0820	CLEAR TWO AND EXIT TRUE	Remove the specified number of groups from the WORK roll, set ANSWER BOX to true, and return.
G0821	CLEAR ONE AND EXIT TRUE	
G0823	EXIT TRUE EXIT TRUE ML	Sets ANSWER BOX to true and returns.
G0824	CLEAR ONE AND EXIT FALSE	Removes one group from WORK roll, sets ANSWER BOX to true, and returns.
G0825	EXIT FALSE	Sets ANSWER BOX to false and returns.
G0826	CLEAR TWO AND EXIT	Remove specified number of groups from WORK roll and return.
G0827	CLEAR ONE AND EXIT	
G0829	EXIT EXIT ML EXIT ON ROLL	Returns.
G0832	SYNTAX FAIL ML ILLEGAL SYNTAX FAIL SYNTAX FAIL	Records syntax error message and goes to FAIL.
G0833	FAIL	If JPE flag off, restores WORK and EXIT roll addresses from last status control, housekeeps Polish notation through STA XLATE EXIT, and returns with ANSWER BOX set to false; if the flag is on, values are restored for JPE and exit is to the location following last JPE POP instruction.
G0834	STATUS CONTROL	Saves addresses of WORK and EXIT roll bottoms.
G0835	DIGIT CONV SCAN	Converts integer from decimal to binary, and leaves in DATA area.
G0836	CONV ONE DIGIT	Converts decimal digit to binary, and leaves in DATA area.
G0838	PRINT A CARD	Controls printing of source listing and error messages.
G0839	TEST FOR ERROR MESSAGE	Determines whether error messages are to be printed; if so, prints dollar sign markers.
G0840	PRINT MESSAGES	Prints line of error messages.
G0841	TEST AND ZERO PRINT BUFFER	Clears output area for printer.
G0842	INIT READ A CARD	Scans source input for assignment statement (flag 1) or Logical IF with assignment for consequence (flag 2).
G0843	READ A CARD	Puts card onto SOURCE roll and re-enters INIT READ A CARD at proper point.
G0845	SKIP TO NEXT CHAR MASK	Scans input to next source character not of a class of characters specified as input to routine.
G0846	REENTRY SKIP TO NEXT CHAR MASK	Entry point used to continue masking operation on a new card.
G0847	NEXT CHAR NEXT CHARACTER	Advance scan arrow to next active character.
G0848	NEXT CHAR ML NEXT CHARACTER ML	
G0849	BCD TO EBCDIC	Converts CRRNT CHAR from BCD to EBCDIC.
G0850	DIGIT CONV INITIAL	Initializes for the conversion of a number from decimal to binary (resets digit counts, clears DATA area, etc.)
G0851	MAPT1 TO TMP1	Converts value in format of TOP or BOTTOM, a virtual address, to a true address.
G1034	BUILD LOOP DATA GROUP	Constructs group on LOOP DATA roll.
G1035	DATA TERM ANALYSIS	Checks for and sets flag if it finds unary minus in DATA statement.
G1037	CONST REGISTER EXIT	Common exit routine for constant recording routines; leaves pointer to constant in W0.

Label	Routine Name	Comments
G1038	T AND F CONST SCAN	Scans for logical constants T and F in DATA statements.
G1039	EXIT ANSWER	General routine used by all EXITs which set ANSWER BOX to store value in ANSWER BOX and return.
G1040	DEBUG STA XLATE	Translates DEBUG statement.
G1041	AT STA XLATE	Constructs AT roll entry from AT statement.
G1042	TRACE STA XLATE	Constructs Polish notation for TRACE statement.
G1043	DISPLAY STA XLATE	Constructs Polish notation and roll entries for DISPLAY statement.
G1044	IEYSKP SKIP TO NEXT PROGRAM	Calls IEYFORT to skip to end of present source module when roll storage is exhausted.
G1070	PRESS MEMORY	Called by REASSIGN MEMORY to obtain additional core storage from roll space that is no longer in use. If it obtains 32 or more bytes, exit is back to REASSIGN MEMORY. Otherwise, exit is to IEYNOCR in IEYFORT to print NO CORE AVAILABLE message.

ALLOCATE LABEL LIST

The labels enumerated in the following list are used in the flowcharts provided for the illustration of the major routines used by Allocate.

Label	Chart ID	Routine Name
G0359	05	START ALLOCATION
G0451	CA	ALPHA LBL AND L SPROGS
	CA	ALPHA SCALAR ARRAY AND SPROG
G0362	CB	PREP EQUIV AND PRINT ERRORS
G0361	CC	BLOCK DATA PROG ALLOCATION
G0365	CD	PREP DMY DIM AND PRINT ERRORS
G0371	CE	PROCESS DO LOOPS
G0372	CF	PROCESS LBL AND LOCAL SPROGS
G0374	CG	BUILD PROGRAM ESD
G0376	CH	ENTRY NAME ALLOCATION
G0377	CI	COMMON ALLOCATION AND OUTPUT
G0381	CK	EQUIV ALLOCATION PRINT ERRORS
G0437	CL	BASE AND BRANCH TABLE ALLOC
G0397	CM	SCALAR ALLOCATE
G0401	CN	ARRAY ALLOCATE
G0402	CO	PASS 1 GLOBAL SPROG ALLOCATE
G0442	CP	SPROG ARG ALLOCATION
G0443	CQ	PREP NAMELIST
G0444	CR	LITERAL CONST ALLOCATION
G0445	CS	FORMAT ALLOCATION
G0441	CT	EQUIV MAP
G0403	CU	GLOBAL SPROG ALLOCATE
G0405	CV	BUILD NAMELIST TABLE
G0438	CW	BUILD ADDITIONAL BASES
G0545	CX	DEBUG ALLOCATE

SUPPLEMENTARY ALLOCATE LABEL LIST

The routines described in this section are listed by G number labels which are presented in ascending order. These routines are those used in the operation of Allocate which are not shown in the section of flowcharts for the phase.

Label	Routine Name	Comments
G0363	PREPROCESS EQUIV	Checks the data contained on the EQUIVALENCE roll and computes the required addresses.
| G0364	REGISTER ERRORS SYMBOL	Checks the ERROR SYMBOL roll for the presence of the error just detected. All duplicate entries are pruned from the roll and all new entries placed on the roll.
G0366	CHECK DMY DIMENSION	The dummy dimension is checked for definition as a global dummy variable, or in COMMON.
G0367	GLOBAL DMY TEST	Sets a pointer to the dummy array on the ENTRY roll; a pointer to the ARRAY roll is also set for each dummy array.
G0368	DMY DIM TEST AND REG	The DMY DIMENSION roll is rebuilt with the information obtained from the COMMON DATA TEMP, TEMP, and GLOBAL DMY rolls.

Label	Routine Name	Comments
G0369	DMY DIM TEST	The dimension data is checked for having been previously defined on the NAMELIST ITEMS and COMMON DATA rolls.
G0370	DMY CLASSIFY	Classifies a dummy, defining it as scalar if undefined; if it is an array sets call by name tag.
G0373	REGISTER BRANCH TABLE	Places work containing zero on the BRANCH TABLE roll.
G0375	PUNCH REMAINING ESD BUFFER PUNCH REMAINING CARD	Punches a card.
G0378	SEARCH ROLL BY MAGNITUDE	The GENERAL ALLOCATION roll is searched to check if the largest equivalenced area has been allocated.
G0379	PRINT COMMON ERRORS	Sets up for, and prints, COMMON allocation errors.
G0380	PRINT COMMON HEADING	COMMON storage map heading is printed.
G0382	EQUIV ALLOCATION	Builds the EQUIV ALLOCATION roll from the boundary calculated; records the absolute address assigned to the variables.
G0383	FLP AND PROCESS EQUIV	Inverts the contents of the EQUIVALENCE roll.
G0384	PROCESS EQUIV	Constructs complete EQUIVALENCE sets on the the GENERAL ALLOCATION roll using information on the EQUIVALENCE roll.
G0385	INTEGRATE	Assigns locations relative to the first variable listed for all variables in an EQUIVALENCE set if not already allocated.

Label	Routine Name	Comments
G0386	TEST FOR BOUNDARY	Sets and checks the smallest equivalenced area and highest boundary required for allocation of the variables indicated; resets program break according to requirement.
G0387	CSECT EQUIV ALLOCATION	Controls the allocation of EQUIVALENCE sets equal to or greater than 3K bytes into a new control section.
G0388	PRINT CSECT EQUIV MAP	Sets up and formats the printing of the storage map for EQUIVALENCE sets equal to or greater than 3K bytes.
G0389	BUILD COMMON ALL ROLL	Calculates the base and displacement for EQUIVALENCE sets equal to or greater than 3K bytes and registers these sets on the COMMON ALLOCATION roll.
G0391	SEARCH FOR LARGE ARRAYS	Determines the size of arrays not defined as EQUIVALENCE or COMMON. Obtains the arrays that are equal to or greater than 3K bytes.
G0392	BUILD A NEW CSECT	Sets the program name and obtains a new control section for the allocation of arrays and EQUIVALENCE sets.
G0393	PRINT A ARRAY CSECT MAP	Sets the information for the printing of the map for arrays equal to or greater than 3K bytes.
G0394	CONV TEMP3 TO HEX	Converts the contents of the temporary register to hexadecimal.
G0395	GLOBAL DMY ALLOCATE	Assigns storage for global dummy variables; expands the contents of the BASE TABLE roll, as required.
G0396	TEST FOR CALL BY NAME	Determines whether the indicated variable was called by name or called by value.

Label	Routine Name	Comments
G0398	ALLOCATE SCALAR BOUNDARY	Sets up allocation of scalars according to the size of the variable.
G0399	ALLOCATE SCALAR	Formats the allocation of scalars not defined as global dummies in COMMON or in EQUIVALENCE sets. Initializes for the printing of the scalar map and calculates the base and displacement.
G0400	CED SEARCH	Determines if the variable is defined as a global dummy, in COMMON or in an EQUIVALENCE set. If it is, it sets the ANSWER BOX = true.
G0404	ALLOCATE SPROG	Sets the type of the ESD cards that are to be punched and initializes for the allocation of subprogram addresses.
G0406	ADJUST AND OUTPUT NAME	Sets the format for the punching of the NAMELIST name, and adjusts for storage.
G0407	PUNCH NAME LIST AND FIELD	Sets the format for the punching of the address allocated for each NAMELIST according to storage required.
G0408	OUTPUT MODE WORD	Sets the format for the punching of the mode of the NAMELIST variable.
G0409	ADVANCE PROG BREAK AND PUNCH	Increases the item PROGRAM BREAK according to the storage allocation required for the variables indicated.
G0410	PUNCH LITERAL	Obtains the number of bytes and the address of the roll indicated for punching of literal constants.
G0411	MOVE TO PUNCH BUFF	Moves the indicated data to the appropriate punch buffer.
G0412	PUNCH TXT CARD	Punches the indicated TXT card after setting up the address and buffer information.
G0413	PUNCH REMAINING TXT CARD	Punches the remaining card indicated, after the area from which data was being taken has been punched.
G0414 G0415	PUNCH ESD PUNCH LD ESD	Punches the indicated ESD cards for the program area indicated.
G0416	PRINT ERROR LBL ROLL	Prints the contents of this roll which contains the errors noted during operation.
G0417	CONVERT LBL	Converts the label of an erroneous statement to BCD for printing.
G0418	PRINT ERROR SYMBOL	Prints the contents of the ERROR SYMBOL roll.
G0420	PRINT SCALAR OR ARRAY MAP	Prints the indicated map.
G0421 G0422	PRINT INIT MAP TEST AND PRINT MAP	Checks the existence of processing of a storage map. Initiates the printing of the indicated map if one is not already being printed.
G0423	PRINT MAP HEADING	Prints the heading of the indicated storage map for the variables designated.
G0424	PRINT FORMAT MAP	Prints map of FORMAT statements.
G0425	PRINT HEADING MESSAGE	Prints the heading indicated for error messages.
G0426	PRINT MAP PRINT MAP ML	Prints the variables associated with the storage map heading from the rolls indicated.
G0431 G0432	PRINT REMAINING BUFFER PRINT ERROR REMAINING BUFFER	Print the remaining information in the print buffer after the data has been obtained from the indicated storage area.
G0433	ALLOCATE FULL WORD MEMORY	Initializes for the allocation of a full word of storage.
G0434 G0435	ALLOCATE MEMORY ALLOCATE BY TYPE	Allocate storage according to the type of the variable indicated; fullword, halfword, or byte.

Label	Routine Name	Comments
G0436	CALCULATE SIZE AND BOUNDARY	Determines the size and the boundary required for the variable indicated.
G0439	CALCULATE BASE AND DISP	Determines the base table entry and displacement for variable being allocated, constructing a new base table entry if necessary.
G0440	REGISTER BASE	Constructs a new BASE TABLE roll group.
G0446	BUILD FORMATS	The base and displacement for FORMAT statements are calculated and the PROGRAM BREAK increased as required.
G0447	INCREMENT PNTR	Increases the address field of the pointer to the indicated roll so that the pointer points to the next group on the roll.
G0448	ID CLASSIFY	Variables are checked for a previous classification as a global dummy, a scalar, an array, global sprog, used library function, or a local sprog.
G0449	REGISTER SCALAR	Builds new group onto the SCALAR roll.
G0450	MODE SET	Sets the mode of the variable to fixed or floating, explicit or implicit, or not used.
G0455	CLEAR THREE AND EXIT TRUE	Prunes three groups from the WORK roll, and exits with a true answer in ANSWER BOX.
G0456	CLEAR TWO AND EXIT TRUE	Prunes two groups from the WORK roll, and exits with a true answer in ANSWER BOX.
G0457	CLEAR ONE AND EXIT TRUE	Prunes one group from the WORK roll, and exits with a true answer in ANSWER BOX.
G0458	EXIT TRUE EXIT TRUE ML	Set ANSWER BOX to true and exit.
G0460	CLEAR TWO AND EXIT FALSE	Prunes two groups from the WORK roll, and exits with a false answer in ANSWER BOX.
G0461	CLEAR ONE AND EXIT FALSE	Prunes one group from the WORK roll, and exits with a false answer in ANSWER BOX.
G0462	EXIT FALSE	Sets ANSWER BOX to false, and exits.
G0464	CLEAR FOUR AND EXIT	Prunes four groups from the WORK roll, and exits.
G0465	CLEAR THREE AND EXIT	Prunes three groups from the WORK roll, and exits.
G0466	CLEAR TWO AND EXIT	Prunes two groups from the WORK roll, and exits.
G0467	CLEAR ONE AND EXIT	Prunes one group from the WORK roll, and exits.
G0468	EXIT	Obtains return address from the EXIT roll, and transfers to that address.

UNIFY LABEL LIST

The labels enumerated in the following list are used in the flowcharts provided for the illustration of the major routines used by Unify.

Label	Chart ID	Routine Name
G0111	07	START UNIFY
G0145	DA	ARRAY REF ROLL ALLOTMENT
G0113	DB	CONVERT TO ADR CONST
G0112	DC	CONVERT TO INST FORMAT
G0115	DD	DO NEST UNIFY

SUPPLEMENTARY UNIFY LABEL LIST

The routines described in this section are listed by G number labels which are presented in ascending order. These routines are those used in the operation of Unify which are not shown in the section of flowcharts for the phase.

Label	Routine Name	Comments
G0114	CALL GEN	Transfers to the Gen phase of the compiler.
G0116	NOTE ARRAY ALLOCATION DATA	Processes SCRIPT roll block to reflect storage allocation.
G0117	LEVEL ONE UNIFY	Sets variables for the processing of a single loop or the outer loop of a nest of loops.
G0118	DO LOOP UNIFY	Controls the processing of script data associated with current innermost loop.
G0119	SWEEP SCRIPT EXP NOTE	Compares the area code and the outer coefficient of all other entries on the NEST SCRIPT roll to the bottom entry on the roll.
G0120	ZERO COEF UNIFY	Sweeps the script entries for the innermost loop, determining whether the outer coefficient is zero and that the inner coefficients are also the same. Depending upon the condition, the loops are re-registered on the LOOP SCRIPT roll.
G0121	NOTE SCRIPT EXP	Establishes the nature of the script entries as standard or nonstandard.
G0122	ESTABLISH STD SCRIPT EXP	Forms the LOOP CONTROL and REG roll entries for each STD SCRIPT pointer found in W0, also registering the STD SCRIPT LOOP CONTROL rung.
G0123	NOTE HI FREQ STD	Checks the frequency used for a particular standard script expression, and sets the frequency count.
G0124	SCRIPT EXP UNIFY	Controls the processing of innermost LOOP SCRIPT roll entries with matching area code and outer coefficients; also links each NONSTD roll entry with each STD roll entry, comparing the induction coefficients.
G0126	STANDARD EXPS UNIFY	Processes STD SCRIPT roll when NONSTD roll entries have all been processed or have never existed. Moves entries to next outermost loop.
G0127	CONVERT NONSTD SCRIPT TO STD	Picks a NONSTD roll entry with a minimum displacement and processes it as if it were a standard script.
G0128	SIGN ALLOC DISPLACEMENT	Utility routine to spread the sign of negative displacements.
G0129	DELTA GE 4087 UNIFY	Processes paired STD or NONSTD roll entries with DELTA greater than 4087 bytes. Generates second register and LOOP CONTROL entries.
G0130	DELTA LE 4087 UNIFY	Processes paired STD or NONSTD roll entries with DELTA less than 4087 bytes. DELTA is placed in each ARRAY REF entry in the chain.
G0131	ESTABLISH REG STRUCTURE	Controls formation of LOOP CONTROL and REG roll groups for SCRIPT pointer in W0.
G0132	EST. REG GROUP	Forms REG roll entry for SCRIPT pointer in W0.
G0133	ESTABLISH LOOP CONTROL	Entry to establish loop control which sets up stamps for impending LOOP CONTROL group.
G0134	EST. LOOP CONTROL	Forms LOOP CONTROL group for SCRIPT entry in W1.
G0135	FORM OUTER SCRIPT	Processes paired STD or NONSTD roll entries with best match in inner coefficients. Forms SCRIPT entry for next outermost loop with coefficient differences in coefficient slots.
G0136	NOTE SECOND REG THREAD	Runs the ARRAY REF thread, removing each link to provide for the second register.

Label	Routine Name	Comments
G0137	UPDATE FREQS	Sums the frequencies of the STD or NONSTD pair to indicate increased usage.
G0138	REG SCRIPT EXP	Registers the STD or NONSTD in W0 on the STD or NONSTD roll.
G0139	PRUNE SCRIPT REL TO PNTR	Utility routine to remove SCRIPT groups.
G0140	NOTE ARRAY REF DELTA	Adjusts the information indicated from the SCRIPT allocation according to the displacement to the associated ARRAY REF roll entries.
G0141	REALIZE REGISTERS SWEEP	Sweeps the REG roll, assigning available registers to the registers and temps, according to the frequency of use of the registers in the REG roll.
G0142	NOTE HI FREQ REG	Utility routine which notes the REG roll group indicating the highest frequency of use.
G0143	ASSIGN TEMPS FOR REGS	Places next temp into the ARRAY REF run and adjusts the LOOP CONTROL stamps to reflect temp usage.
G0144	CONVERT REG TO USAGE	Performs the actual transfer of REG or TEMP roll entries into the ARRAY REF threads.

GEN LABEL LIST

The labels contained in the following list are illustrated in the flowcharts provided with the description of the Gen phase of the compiler.

Label	Chart ID	Routine Name
G0491	08	START GEN
G0499	EA	ENTRY CODE GEN
G0504	EB	PROLOGUE GEN
G0508	EC	EPILOGUE GEN
G0712	ED	GET POLISH
G0493	EF	LBL PROCESS
G0515	EG	STA GEN
G0496	EH	STA GEN FINISH

SUPPLEMENTARY GEN LABEL LIST

The routines described in this section are listed by G number labels which are presented in ascending order. These routines are those used in the operation of Gen but not shown in the section pertaining to the phase.

Label	Routine Name	Comments
G0494	CLINCH	Clears the base register table.
G0497	ZERO THE ACS	Clears the accumulators to be used.
G0498	MOVE ZEROS TO T AND C	Fills the indicated number of groups on the TEMP AND CONST roll with zeros.
G0500	INSERT PROG NAME IN CODE	Puts name of source module on CODE roll.
G0501	MAIN PROGRAM ENTRY	Builds instructions for the entry into the main program.
G0502	PRO AND EPI ADCON GEN	Determines the address constant for prologues and epilogues for the instruction that is created.
G0503	ADCON MAKER GEN	Builds ADCON roll group and places adcon instruction on CODE roll.
G0505	LOAD DMYS GEN	Builds the code to load the dummy arguments specified in a subprogram.

Label	Routine Name	Comments
G0506	BUILD DMY ARRAY DIM	Determines the dummy array dimensions specified in the arguments for the subprogram.
G0507	CALCULATE DMY DIM	Calculates the dummy array dimensions specified as arguments to a subprogram, and builds the appropriate instructions.
G0509	RESTORE DMY GEN	Restores the dummy arguments for value transfer at the end of a subprogram.
G0510	TEST CALL BY NAME	Determines whether the arguments to a subprogram were designated as call by name values.
G0511	BUILD A MOVE DMY GROUP	These routines build the instructions that transmit the indicated values transferred by the dummy arguments to subprogram.
G0512	BUILD A STORE DMY ADD	
G0513	INCREMENT DMY PNTR	
G0514	BUILD A LOAD TWO	
G0516	ASSIGNMENT STA GEN	Controls the construction of the code for an assignment statement.
G0517	AFDS STA GEN	Controls and constructs the instructions for an arithmetic function definition statement.
G0518	AFDS INIT	Initializes the construction of the code for an arithmetic function definition statement by constructing the label and jump instructions.
G0519	ASSIGN STA GEN	Constructs the object code for an ASSIGN statement.
G0520	IF STA GEN	Constructs the object code for an IF statement.
G0521	LOGICAL IF STA GEN	Constructs the object code for a Logical IF statement.
G0522	BUILD JUMP INST	Constructs a branch instruction, with input indicating type and branch point.
G0523	GO TO STA GEN	These routines control and construct the object code required to execute the indicated type of GO TO statement.
G0524	ASSIGN GO TO STA GEN	
G0525	GO TO JUMP GEN	
G0526	CGOTO STA GEN	These routines construct the object code for a GO TO statement that is the subprogram return.
G0527	CGOTO FOR CALL RETURN GEN	
G0528	CONTINUE STA GEN	Returns.
G0529	BLOCK DATA GEN	Sets up the rolls and data used in the construction of the object code for the BLOCK DATA statement.
G0530	STA INIT	Stores the statement number and leaves statement drives in W0.
G0531	DATA STA GEN	Determines the use and mode of the data variables and constructs the object code based on this information.
G0532	ALIGN DATA	Adjusts the data for instruction format.
G0533	INIT FOR VAR	Obtains the base, size, displacement, and area code of the indicated variable and adjusts the instruction format for the variable according to the information obtained.
G0534	MOVE DATA	Sets up the beginning of the data for card format.
G0535	MOVE TO CARD IMAGE	Obtains the location of the indicated data for transfer to instruction format.

Label	Routine Name	Comments
G0536	MOVE TO CARD REPEAT	Controls the insertion of the data into the card format and the punching of the appropriate TXT card.
G0537	PUNCH A TXT CARD	Write a TXT card from data whose location is provided.
G0538	PUNCH A TXT CARD ML	
G0539	PUNCH TXT ENTRY2	
G0542	CALCULATE VAR SIZE	Determines size of a variable from TAG field of pointer in W0.
G0543	END STA GEN	Builds code for AT if required and branches to TERMINATE PHASE.
G0547	BSREF STA GEN	Controls the construction of the object code for a BACKSPACE, REWIND, or END FILE statement.
G0548	STOP PAUSE STA GEN	Constructs the object code for a STOP or PAUSE statement.
G0549	LOAD IBCOM	Builds an instruction for a call to the IBCOM routine.
G0550	RETURN STA GEN	Builds the object code for a RETURN statement.
G0551	ENTRY STA GEN SPROG STA GEN	Constructs the label instruction for an ENTRY statement or the entry into a subprogram.
G0552	DEFINE FILE STA GEN	Constructs the object code instructions for the DEFINE FILE statement.
G0553	GRNTEE A TEMP	Ensures that the constant from DEFINE FILE is registered on the TEMP AND CONST roll.
G0554	ILLEGAL AFDS STA GEN	Generates an error link for a statement function which was invalid.
G0555	ILLEGAL STA GEN ENTRY	Constructs a no-operation instruction and an error link for the statement in error.

Label	Routine Name	Comments
G0556	IO STA GEN	Determines the type of input/output statement that is indicated and transfers to the routines that process that particular type of statement.
G0557	INIT IO LINK GEN	Initiates and sets data for the generation of the input/output linkage.
G0558	UNIT IO ARG	Determines the logical unit number of the input/output device.
G0559	DIRECT IO ARG	Sets up controls for the construction of the object code for direct-access input/output statements.
G0560	FORMAT IO ARG	Sets up data pertaining to the FORMAT for the construction of the object code of an input/output statement under format control.
G0561	IO INITIAL ENTRY GEN	Sets up code for the call to IBCOM to control execution of the indicated input/output statement.
G0562	BUILD UNIT ARG	Constructs argument passed for unit number in input/output linkages.
G0563	BUILD A LINK ARG	Constructs the object code for the arguments designated in the input/output statements.
G0564	BUILD FORMAT ARG	Constructs the object code for the designated format control of an input/output statement.
G0565	GRNTEE IO LINK ADD	Constructs the object code for input/output linkage.
G0566	IOL DO CLOSE GEN	Generates object code for closing of implied DO in I/O list.
G0567	IO LIST GEN RUN	Determines whether I/O list is DO implied.

Label	Routine Name	Comments
G0568	IOL DO OPEN GEN	Sets up the data for the generation of instructions for input/output DO loop.
G0569	IOL ARRAY GEN	Generates linkage for secondary array entry to IBCOM.
G0570	IO LIST PNTR GEN IOL PNTR GEN	Determines the type of the I/O list, and controls the construction of the object code for the list.
G0571	IO LIST ARRAY PNTR GEN	Sets up the data and determines the type of array list.
G0572	BUILD ELEMENTS ARG	Builds an argument for input/output linkage for a single element in an I/O list.
G0573	IO LIST DMY ARRAY	Builds the object code for a dummy array I/O list.
G0574	GLOBAL DMY TEST	Determines whether the variable in question has been defined in usage as a global dummy.
G0575	IO STA END IO STA END GEN	Generates call for end of I/O list.
G0576	BUILD IO LINK	Controls construction of the object code to terminate an input/output operation.
G0577	LOAD ADDRESS IBCOM	Inserts the absolute call to the system input/output routine, IBCOM.
G0578	INIT IBCOM PNTR AND ENTRY	Initializes for processing of input/output statements by storing code word and IBCOM pointer from POLISH roll.
G0579	CALCULATE LENGTH AND TYPE	Determines the length and type of variables designated in input/output statements.
G0580	DO STA GEN	Determines the nature of the DO statement, sets up the data for the code of the statement.
G0581	LOOPS OPEN GEN	Obtains the DO control data and controls the construction of the appropriate instructions.
G0582	INIZ LOOP GEN	Determines the nature of the indicated DO loop after determining whether a loop exists.
G0583	INIZ GIVEN COEFF GEN	Constructs the object code for the initialization of the indicated induction variable coefficient.
G0584	DO CLOSE SBR	Constructs the object code for the close of a DO loop after setting up controls for the increment and terminal values of the loop iteration.
G0585	FIND COEFF INSTANCE	Determines the existence of the indicated nature of a loop through comparison of the designated traits and coefficient.
G0586	NOTE TEMP REQ	Determines whether a register has been assigned for the script expression in question or whether a temporary storage is required.
G0587	INITIALIZE BY LOAD GEN	Generates the load of registers to be used throughout a DO loop.
G0588	GRNTEE TEMP STORED GEN	Builds a store instruction for the temporary storage used by the script expression.
G0589	GRNTEE SOURCE REG LOADED	Determines the area and location for the register to be used by the script expression, and generates the load instruction for the indicated temporary storage.
G0590	INCR GIVEN COEFF GEN	Determines the nature and use of the loop increment and builds the appropriate instructions for the execution of the increment.

Label	Routine Name	Comments
G0607	CALL STA GEN	Calls the routines which build the object code for the CALL statement.
G0608	FLP AND PREP VAR	Flips POLISH roll and moves first variable to WORK roll.
G0609	EXP GEN BY MODE	Controls the determining of the mode of the indicated expression.
G0610	EXP GEN AND GRNTEE AC	Generates code for expression on bottom of POLISH roll and ensures that result is in a register.
G0611	GRNTEE EXP	Guarantees that the mode of the expression is positive.
G0612	EXP GEN	Obtains the expression for GEN processing.
G0613	GEN RUN	Determines the operation mode of the entity in question.
G0614	NOT GEN UNARY MINUS GEN	Inverts sign indicator for variable on bottom of WORK roll.
G0615	DIV GEN	Controls production of object code for divide operation.
G0616	INTEGER DIV GEN	Generates code for integer divide.
G0617	SUB GEN	Generates code for subtract operation.
G0618	ADD GEN	Generates code for add operation.
G0619	MPY GEN	Controls production of object code for multiply operation.
G0620	INTEGER MPY GEN	Generates code for integer multiply.
G0621	INTEGER MPY DIV END	Common end for multiply and divide generation routines; records register usage.
G0622	SUM OR PROD GRNTEE	Guarantees that one of the two elements on WORK roll is in a register and that mode of operator is correct.
G0623	DRIVER GEN	If an array driver, goes to SCRIPT PREP; if not, exits false indicating end of an expression.
G0624	AND GEN	Generates code for an AND operation.
G0625	AND FINISH GEN	Actually builds an AND operation on CODE roll.
G0626	OR GEN	Generates code for an OR operation.
G0627	OR FINISH GEN	Actually builds an OR operation on CODE roll.
G0628	PREPARE FOR LOGICAL GEN	Sets up the data for the statement containing a logical operation.
G0629	EQ GEN	Generates code for an EQ relational operation.
G0630	NE GEN	Generates code for an NE relational operation.
G0631	LT GEN	Generates code for an LT relational operation.
G0632	GT GEN	Generates code for a GT relational operation.
G0633	GE GEN	Generates code for a GE relational operation.
G0634	LE GEN	Generates code for an LE relational operation.
G0635	RELATIONAL GEN	Builds the object code instructions based on the relational condition specified in the logical operation.
G0636	PREPARE FOR RELATIONAL	Converts and adjusts data for construction of the object code of a relational comparison.
G0637	POWER GEN	Builds exponentiation linkage on the CODE roll.
G0638	POWER AND COMPLEX MPY DIV GEN	Sets up the data for operations involving multiplication or division of exponentiated or complex variables.

Label	Routine Name	Comments
G0639	INTEGER POWER GEN	Builds the appropriate load and multiply instructions for exponentiation depending on the mode of the operation.
G0640	SPROG GEN	Determines the nature of the operand of a CALL statement or of a subprogram.
G0641	SPROG GEN SUB	Generates the code for a subprogram call including argument calculations.
G0642	SPROG END GEN	Constructs the object code for the return or close of a subprogram.
G0643	SPROG ARG SEQ GEN	Controls the interpretation of the sequence of arguments designated to a subprogram.
G0644	REG SPROG ARG	Controls the register assignment to subprogram arguments as they are encountered in sequence.
G0645	GRNTEE ADR GEN	Guarantees that the subprogram arguments are assigned and builds the indicated load and store instructions.
G0646	TEST CONST ARG	Determines mode of a constant subprogram argument.
G0647	TEST AND STORE REGS	Tests to determine if any register used as an accumulator contains data; if so, generates code to store the contents in a temporary location.
G0648	GRNTEE AC GEN	Stores the contents of W0 in an accumulator if not already designated.
G0649	GRNTEE NEW AC GEN	These routines determine the accumulator to be used in an indicated operation depending upon the mode of the variable in question.
G0650	PICK A NEW AC	
G0651	PICK FL AC	
G0652	PICK A COMPLEX AC	
G0653	CLEAR A PAIR	These routines determine and clear a pair of fixed or floating accumulators depending on the type of the register in W0. These routines are used in integer, multiply, divide, and complex operations.
G0654	PICK A PAIR	
G0655	PICK A PAIR END	
G0656	TEST FOR BEST PAIR	Determines the two optimal accumulators to be used for the operation indicated.
G0657	GRNTEE POSITIVE GEN	Sets the mode of the indicates accumulator to positive if not already set, and generates appropriate code.
G0658	COMP FX CONST	Set the mode of the indicated constant.
G0659	COMP FL CONST	
G0660	COMP DP CONST	
G0661	COMP COMPLEX CONST	Sets the mode of the indicated constant.
G0662	CORRECT FOR SIGN DATA 1	Complements the value in DATA1.
G0663	INCLINE FUNCTION GEN	Sets up table for the generation of code for in-line functions.
G0664	CONVERSION FUNCTION GEN	Generates code to perform an in-line mode conversion.
G0665	ABS FUNCTION GEN	These routines generate the object code instructions for the inline function indicated by the name of the routine.
G0666	MOD FUNCTION GEN	
G0667	INT FUNCTION GEN	
G0668	AIMAG FUNCTION GEN	
G0669	CMPLEX FUNCTION GEN	
G0670	TWO ARG INLINE COMMON	
G0671	CONJG FUNCTION GEN	

Label	Routine Name	Comments
G0672	SIGN FUNCT GEN	(see Label G0665)
G0673	DIM FUNCT GEN	
G0674	GRNTEE BOTH MODES	Sets the mode of the data in W0 and W1 to positive if not already set.
G0675	GRNTEE MODE W1	Determines the mode of the variable in W1 and transfers to the appropriate conversion routine depending on the mode of W0.
G0676	LOGICAL CONVERSION	Places the logical variable contained in W0 into an accumulator.
G0677	FX CONVERSION	Places the variables contained in W0 and W1 in an accumulator if the mode is I*2; otherwise, a conversion to floating point is made.
G0678	FL CONVERSION	Tests the contents of W0 and W1 for floating variables or constants. If the contents are not floating variables or constants, it determines the nature of the data, registers the variable or constant, and assigns an accumulator for the operation.
G0679	CONVERT TO COMPLEX END	Generates code to clear the imaginary register and loads the real register in real to complex conversion.
G0680	TEST A FL CONST	Exits false if pointer in W0 is not to a floating constant; otherwise, it loads the constant into central area and exits true.
G0681	DP CONVERSION	Determines the nature of the double-precision variable or constant indicated, converts into the indicated format, assigns an accumulator, depending on the mode of the variable.
G0682	TEST DP CONST	Exits false if pointer in W0 is not to a double-precision constant; otherwise, loads constant into central area and exits true.
G0683	COMPLEX CONVERSION	Determines the mode and nature of the two components of the complex variable or constant.
G0684	DP COMPLEX CONVERSION	Determines the mode and registers the indicated double-precision complex variable or constant.
G0685	COMPLEX AC TEST	Sets up FL AC roll for proper pointers to a value converted to complex.
G0686	AC END AND CONV RETEST	Used during conversion, to set up AC roll, and to determine whether conversion is complete.
G0687	CONVERT RETEST	Sets up WORK roll so that GRNTEE MODE W1 can determine whether a conversion is complete.
G0688	REGISTER WORK CONST	Records constant in W0 as an integer constant.
G0689	REGISTER FX CONST	Register the constant from DATA area on the indicated roll if not already defined; constant is compiler generated.
G0690	REGISTER FL CONST	
G0691	REGISTER DP CONST	
G0692	REGISTER COMPLEX CONST	
G0693	REGISTER DO COMPLEX CONST	
G0695	FLOAT A FX	Converts a floating constant or generates code to convert a floating variable to fixed mode.
G0696	FIX A FL	Converts a fixed mode constant or generates code to convert a fixed variable to floating mode.
G0697	FLOAT AND FIX COMMON	Common exit for routines which write code to float or fix variables.
G0708	TEST AC AC TEST	Determines whether the mode of the indicated accumulator is fixed or floating.

Label	Routine Name	Comments
G0709	AC END	Determines whether one or two accumulators are being used.
G0710	GRNTEE AC ZERO	Assures that the accumulator being used in the operation is register zero.
G0711	SPOIL STO REG	Clears appropriate entry on AC roll for a register which has been stored.
G0713	CLEAR THREE AND EXIT TRUE	Remove indicated number of groups from WORK roll, set ANSWER BOX to true, and return.
G0714	CLEAR TWO AND EXIT TRUE	
G0715	CLEAR ONE AND EXIT TRUE	
G0716	EXIT TRUE EXIT TRUE ML	Sets ANSWER BOX to true and returns.
G0718	CLEAR THREE AND EXIT FALSE	Remove indicated number of groups from WORK roll, set ANSWER BOX to false, and return.
G0719	CLEAR TWO AND EXIT FALSE	
G0720	CLEAR ONE AND EXIT FALSE	
G0721	EXIT FALSE EXIT FALSE ML	Sets ANSWER BOX to false and returns.
G0723	CLEAR THREE EXIT CLEAR THREE AND EXIT	Remove indicated number of groups from WORK roll and return.
G0724	CLEAR TWO EXIT CLEAR TWO AND EXIT	
G0725	CLEAR ONE EXIT CLEAR ONE AND EXIT	
G0727	EXIT EXIT ML	Returns.
G0728	EXIT ANSWER ML	Sets ANSWER BOX and exits for EXIT routines which set ANSWER BOX.

Label	Routine Name	Comments
G0730	ADCON MADE LBL MAKER	Builds ADCON roll and returns a pointer to the start of a group on the roll.
G0731	CHECK JUMP LBL	Determines whether pointer in W0 refers to a jump target label.
G0732	MADE LBL MAKER	Creates entry on BRANCH TABLE roll for made label, and returns pointer to group created.
G0733	SCRIPT PREP	Sets up the data for the calculation of the indicated script expression.
G0734	CALCULATE SCRIPT	Determines the mode and operation of the variables contained in the script expression.
G0735	TEST END SCRIPT	Determines the end of the script expression.
G0736	CALCULATE OFFSET AND SIZE	Determines the size of each element contained within an expression, and the displacement pertaining to each array.
G0737	GRNTEE REG 9	Place the index values for arrays in register 9 if not already set.
G0738	TEST AND STORE REG 9	
G0739	BUILD A SHIFT 9	Builds a shift register 9 instruction for subscripting; shift length is determined by array element size.
G0744	BID INIT	Initializes data for the contsruction of the instruction designated by the BID, BIN, or BIM POP instructions.
G0745	BIM INIT	
G0746	BIM BID INIT	
G0747		
G0748	EXIT FULL	Used on entry to BIN when BIN fills the EXIT roll.
	BID BIDPOP	This is the assembler language routine which constructs the instruction designated by the BIDPOP instruction.

Label	Routine Name	Comments
G0750	BIN BINPOP	This is the assembler language routine which constructs the instruction designated by the BINPOP instruction.
G0751	NOTE A CSECT	This routine obtains the Control section in which the current instruction being generated is to be placed.
G0752	BIM BIMPOP	This is the assembler language routine which constructs the instruction designated by the BIMPOP instruction.
G0753	RX FORMAT	General routine used to build all RX type instructions.
G0754	RR FORMAT	This routine implements the RR format designation for the instruction being generated.
G0755	ADDRESS MAKER	Used to build all base, displacement, and index type addresses.
G0756	BUILD A BASE REG	Determines the base location within a particular control section at which the object code instructions begin.
G0757	SCALAR OPERAND ARRAY OPERAND GLOBAL SPROG OPERAND USED FUNCTION LIB OPERAND NAMES LIST OPERAND FORMAT LBL OPERAND GLOBAL DMY OPERAND	Builds address for the specified type of operand.
G0758	DMY LBL COMMON	Generates address for FOMAT references.
G0759	LBL OPERAND LOCAL SPROG OPERAND	Builds address for references to labels and statement functions.

Label	Routine Name	Oments
G0760	SPROG ARG OPERAND	Builds address for reference to subprogram argument list.
G0761	BRANCH TABLE OPERAND	Builds address for references to made labels.
G0762	BRANCH TABLE COMMON	Used by LBL and BRANCH TABLE OPERAND routines to contstruct address.
G0763	BRANCH SPROG COMMON	Used by LBL, BRANCH TABLE and SPROG ARG OPERAND to construct address.
G0764	T AND C OPERAND	Constructs address for references to temporary storage or constants.
G0765	T AND C COMMON	Used for T AND C OPERAND and pointers to constant rolls.
G0766	T AND C B COMMON	Common exit for all branch and temporary and constant operand routines.
G0767	LOCAL DMY OPERAND	Determines the base location for the indicated operand and builds the code data from this information.
G0768	FX CONST OPERAND	Determines the size of the fixed constant operand and constructs the instruction depending upon this information.
G0769	FX FL CONST SEARCH AND REG FL CONST OPERAND	Moves single-precision constant pointed to TEMP AND CONST roll if not already on roll.
G0770	FX FL CONST COMMON	Performs part of move of constant to TEMP AND CONST roll.
G0771	SEARCH AND REG SP CONST SEARCH AND REG FX CONST SEARCH AND REG FL CONST	Searches TEMP AND CONST roll, registers constant if not already there, and returns pointer to TEMP AND CONST roll group.

Label	Routine Name	Comments
G0772	REG SP CONST	Registers single-precision constant on TEMP AND CONST roll.
G0773	DP FL CONST OPERAND COMPLEX CONST OPERAND	Construct address for references to double-precision real and single-precision complex constants.
G0774	SEARCH AND REG DP CONST SEARCH AND REG COMPLEX CONST	Ensures that a double-precision real or single-precision complex constant is on the TEMP AND XONST roll and returns a pointer to it.
G0775	REG DP CONST	Registers a new double-precision constant on the TEMP AND CONST roll.
G0776	DP COMPLEX CONST OPERAND	Constructs address for reference to a double-precision complex constant.
G0777	SEARCH AND REG DP COMPLEX CONST	Ensures that a double-precision complex constant is on the TEMP AND CONST roll and returns a pointer to it.
G0778	REG DP COMPLEX CONST	Registers a new double-precision complex constant on the TEMP AND CONST roll.
G0779	TEST DOUBLE WORD BOUNDARY	Determines if the address designated to the variable or constant in W0 begins on a doubleword boundary.
G0780	ARRAY REF OPERAND	Handles array reference pointers to obtain scripted arrays addresses.
G0781	LOAD REG FROM TEMP	Generates a load of a base register from a temporary storage location.
G0782	ARRAY PLEX OPERAND	Handles building addresses when array plex is the indicated operand.
G0783	SRCH AND ST X9 FROM ARRAY PLEX	Stores register 9 in a temporary register if needed for generation of array plex addresses.

Label	Routine Name	Comments
G0784	STORE IN TEMP	Generates code to store that register in a temporary location if W0 is a pointer to a register.
G0785	STORE AND RETURN TEMP	Uses a temporary location in checking temporary pointers for the indicated constants.
G0786	SEARCH TEMP ROLL	Beginning with a pointer to the TEMP PNTR roll in W0, searches for an available temporary already defined. Returns true, with pointer to TEMP AND CONST roll if found; otherwise, returns false.
G0787	OPERAND RUN	Selects processing routine for present operand from pointer.
G0930	SPOIL STO VAR SPOIL STORE VAR	Determines whether pointed to variable is being used in subscript which is now contained in register 8 or 9; if so, spoils that register.
G0931	SPOIL STORE VAR NON READ IO	Determines whether a stored variable which has not appeared in a READ should be stored.
G0932	CLEAR ONE AND SPOIL CEAD	Determines if pointed to variable is COMMON, EQUIVALENCE, alterable, or dummy; if so, spoils any register containing a subscript which uses any CEAD variable; and prunes one group from WORK.
G0933	SPOIL CEAD	Same as CLEAR ONE AND SPOIL CEAD except it does not prune WORK roll.
G0934	TEST A CEAD	Tests to determine if variable pointed to by W0 is COMMON, EQUIVALENCE, alterable, or dummy.
G0935	NO ARG SPROG END GEN	Entry point for generating a subprogram call without arguments.

Label	Routine Name	Comments
G0937	SIMPLE SCRIPT PREP	Builds ARRAY PLEX roll for subscripts handled in registers 8 and 9.
G0938	CLEAR 3 EXIT BIN	Exits from BIN, BIM and BID POP subroutines which remove the indicated number of groups from WORK.
G0939	CLEAR 1 EXIT BIN	
G0940	EXIT BIN	Exits from BIN, BIM, and BID POP subroutines.
G0941	SUBCHK GEN	Builds code for SUBCHK entry if required.
G0942	SIMPLE SCRIPT OPERAND	Generates the code to compute a subscript value to be held in register 8 or 9.
G0943	TEST FOR HIT	Determines whether register 8 or 9 already contains the present subscript.
G0944	LOAD SIMPLE X REG	Generates code to set up register 8 or 9.
G0945	PICK A NEW SIMPLE X REG	Determines whether register 8 or 9 will be used for subscript which must be loaded.
G0946	CALC ELEM SIZE AND SHIFT	Calculates array element size and the length of shift necessary to multiply by that value.
G0947	AT STA GEN	Generates the object code for an AT statement.
G0948	TRACE ON STA GEN	Generates DEBUG linkage for a TRACE ON statement.
G0949	TRACE OFF STA GEN	Generates DEBUG linkage for a TRACE OFF statement.
G0950	DEBUG INITIAL LINKAGE GEN	Generates initial linkage to DEBUG.
G0951	DEBUG VAR ADR GEN	Generates address for INIT or SUBCHK variable.
G0952	DEBUG ELEMENTS GEN	Generates number of elements for DEBUG linkage.

Label	Routine Names	Comments
G0953	BIN VARIABLE NAME	Puts name of variable on CODE roll.
G0954	RETURN SCALAR OR ARRAY PNTR	Returns pointer to a SCALAR or ARRAY roll group from less direct reference.
G0955	DEBUG INIT GEN	Generates DEBUG linkage for INIT variables.
G0956	DEBUG SHORT LIST INIT GEN	Generates DEBUG linkage for INIT of a full array.
G0957	DEBUG DMY INIT GEN	Generates DEBUG linkage for INIT of a dummy variable.
G0958	DISPLAY STA GEN	Generates DEBUG linkage for a DISPLAY statement.
G0959	DEBUG INIT ARG GEN	Generates DEBUG calls after a CALL statement.

EXIT LABEL LIST

The labels enumerated in the following list are used in the flowcharts provided for the illustration of the major routines used by Exit.

Label	Chart ID	Routine Name
G0381	09	EXIT PASS
G0382	FA	PUNCH TEMP AND CONST ROLL
G0383	FB	PUNCH ADR CONST ROLL
G0384	FC	PUNCH CODE ROLL
G0399	FD	PUNCH BASE ROLL
G0400	FE	PUNCH BRANCH ROLL
G0402	FF	PUNCH SPROG ARG ROLL
G0403	FG	PUNCH GLOBAL SPROG ROLL
G0404	FH	PUNCH USED LIBRARY ROLL
G0405	FI	PUNCH ADCON ROLL
G0416	FJ	PUNCH RLD ROLL
G0424	FK	PUNCH END CARD
G0564	FL	PUNCH NAMELIST MPY DATA

SUPPLEMENTARY EXIT LABEL LIST

The routines described in this section are listed by G number labels which are presented in ascending order. These routines are those used in the operation of Exit which are not shown in the section of flowcharts for the phase.

Label	Routine Name	Comments
G0385	SWEEP CODE ROLL SWEEP CODE ROLL ML	Determines the nature of a word on the CODE roll and processes it according to type.
G0386	PUNCH INST PUNCH INST ML	Determines the type of instruction to be punched (one, two, or three halfwords).
G0388	PUNCH TWO HALFWORDS	Sets up a two halfword instruction format.
G0389	PUNCH ONE HALFWORD	Sets up a one halfword instruction format.
G0390	PUNCH THREE HALFWORDS	Sets up a three halfword instruction format.
G0391	PUNCH CODE	Punches the indicated instruction in the indicated format.
G0392	ABS PUNCH	Sets up for the punching of object module absolute constants.
G0393	RELOC CONST PUNCH	Sets the format for the punching of a relocatable absolute constant.
G0394	ABS CONST PUNCH ABS CONST PUNCH ML	Punches the indicated absolute constants in the object module.
G0396	DEFINE LBL	Defines indicated label on BRANCH TABLE roll.
G0397	ADCON PUNCH	Punches the address constant indicated in W0.
G0398	POC DATA PUNCH	Sets up the information needed for the listing and punching of code contained on the CODE roll.
G0401	SWEEP BASE BRANCH ROLL	Initializes for the punching of the groups contained on the BASE and BRANCH TABLE rolls.
G0406	HALF WORD W0 TO TXT CARD	A halfword instruction format is set up for the contents of W0.
G0407	W0 TO TXT CARD W0 TO TXT CARD ML	Transfers the contents of W0 to the output area to be punched.

Label	Routine Name	Comments
G0409	MOVE CODE TO TXT CARD	Transfers the indicated code to the output area to be punched.
G0410 G0411	INITIALIZE TXT CARD INITIALIZE TXT CARD ML	Initialize the format for the punching of the TXT cards.
G0412	PUNCH PARTIAL TEXT CARD	Punches any part of a TXT card.
G0413	PUNCH A CARD ML	Punches a complete TXT card.
G0414	PUNCH AN ESD CARD	Sets the format for the punching of an ESD card.
G0417	DEPOSIT LAST ESD NO. ON RLD CARD	Obtains and deposits the last ESD number on the indicated RLD card for punching.
G0418	DB SECOND RLD WORD WITH CONT	Sets the format of a card with a continuation to a second card.
G0419	DB SECOND RLD WORD WITH NO CONT	Turns off the continuation indicator for the punching of the RLD card.
G0420	DB SECOND RLD WORD	Places the second word into the RLD format in the output area.
G0421	DEPOSIT WORD ON RLD CARD	Places the indicated word into the appropriate location in the RLD format.
G0422	PUNCH AN RLD CARD	Punches the indicated RLD card.
G0423	TERMINATE RLD PUNCHING	Determines whether the RLD card is full and sets controls accordingly.
G0425	LIST CODE	Sets up the format for the object module listing, and determines the instruction format for each indicated instruction to be printed.
G0426	RS OR SI FORMAT	Determines whether the indicated instruction is RS or SI format.

Label	Routine Name	Comments
G0427	RS FORMAT	Sets up the RS format for the indicated instruction.
G0428	SI FORMAT	Sets up the SI format for the indicated instruction.
G0429	RX FORMAT	Sets up the RX format for the indicated instruction.
G0430	RR FORMAT	Sets up the RR format for the indicated instruction.
G0431	SS FORMAT	Sets up the SS format for the indicated instruction.
G0432	ADCON LIST	Sets up the format (DC format) for the address constants in the object module that are to be listed.
G0433	DC LIST	Lists DC constants.
G0434	PRINT ADCON LBL	Sets controls for the printing of the indicated address constant.
G0435	PRINT A MADE LBL	Sets controls for the printing of the indicated label which has been created by the compiler.
G0436	MADE LBL ADCON LBL COMMON	Inserts the indicated label into the print output area.
G0437	PRINT A LBL	Prints the indicated label on the object module listing.
G0438	PRINT BCD OPERAND	Inserts the indicated operand into the appropriate position of the object listing in the output area.
G0439	PRINT A LINE PRINT A LINE PLUS ONE ML	Print the indicated line once a full line has been set up in the output area.
G0440	PRINT A LINE ML	

Label	Routine Name	Comments
G0443	PRINT HEADING PRINT HEADING ML	Prints this indicated heading that is to appear on the object module listing.
G0444	PRINT TOTAL PROG REQMTS MESS	Sets up this indicated message in the print output area.
G0445	PRINT CSECT MEMORY REQMTS MESS	Sets up the indicated message in the print output area.
G0446	PRINT CSECT TOTAL MESSAGE ML	Sets up this indicated message in the print output area.
G0447	PRINT CSECT MESSAGE	Sets up this indicated message in the print output area.
G0448	CONV AND PRINT D2(B2) ML	Converts the indicated general register designation for the RX, RS, and RR formats.
G0449	CONV AND PRINT D1B1 ML	Converts the indicated address and general register designation for the SI and SS formats.
G0450	CONV AND PRINT D2 ML CONV AND PRINT D1 ML	Converts the indicated address and general register designations to instruction format.
G0452	CONV AND PRINT B1 ML CONV AND PRINT B2 ML	Converts the indicated address and general register designations to instruction format.
G0453	CONV AND PRINT R2 ML CONV AND PRINT X2 ML	Converts the indicated address and general register designations to instruction format.
G0454	CONV AND PRINT 12 ML	Converts the indicated address and general register designations to instruction format.
G0455	CONV AND PRINT R1 ML CONV AND PRINT L1 ML	Converts the indicated address and general register designations to instruction format.
G0456	CONV W0 AND PRINT CONVERT W0 AND PRINT	Converts the contents of W0 to decimal and inserts into print output area.

Label	Routine Name	Comments
G0458	CONV AND PRINT PLUS ONE ML	Converts a number to decimal and places in print buffer.
G0459	PRINT A COMMA ML	Places a comma into print output area.
G0460	PRINT A LEFT PAREN ML	Places a left parenthesis into the print output area.
G0461	PRINT A RIGHT PAREN ML	Places right parenthesis into the print output area.
G0462	PRINT A CHAR ML	Places the indicated character into the print output area.
G0464	CLEAR ONE EXIT CLEAR ONE AND EXIT	Prunes one word from the WORK roll and exits.

Label	Routine Name	Comments
G0465	EXIT EXIT ML EXIT ANSWER ML	Obtains the last entry on the EXIT roll and transfers to the indicated location.
G0566	RLD ALIGN SWEEP TE	Sorts RLD entries so that all RLDs in one CSECT appear together.
G0567	RLD ALIGN TEST SWEEP TEST	Determines whether present RLD is in the CSECT now being constructed.
G0569	GET ADR FROM PNTR ML	Gets location on DATA VAR roll from pointer in W0.

This appendix describes the logic of some of the object-time library subprograms that may be referenced by the FORTRAN load module. Included at the end of this appendix are flowcharts that describe the logic of the subprograms. (G is the first character in the chart identification for each flowchart associated with a library subprogram.)

Each object module compiled from a FORTRAN source module must first be processed by the linkage editor prior to execution on the IBM System/360. The linkage editor must combine certain FORTRAN library subprograms with the object module to form an executable load module. The library subprograms exist as separate load modules on the FORTRAN system library (SYS1.FORTLIB). Each library subprogram that is externally referenced by the object module is included in the load module by the linkage editor. Among the library subprograms to which such references may be made are:

- IHCFCOMH (object-time input/output source statement processor) -- entry name IBCOM#. If the extended error message facility is specified, this module is replaced by IHCECOMH.
- IHCFIOSH (object-time sequential access input/output data management interface) -- entry name FIOCS#. If the extended error message faciltiy is specified, this module is replaced by IHCEFIOS.
- IHCNAMEL (object-time namelist routines) -- entry names FRDNL# and FWRNL#.
- IHCDIOSE (object-time direct access input/output data management interface) -- entry name DIOCS#. If the extended error message facility is specified, this module is replaced by IHCEDIOS.
- IHCIBERH (object-time source statement error processor) -- entry name IBERH#.
- IHCFCVTH (object-time conversion routine) -- entry name ADCON#.
- IHCDBUG (object-time debug facility support routine) -- entry name DEBUG#.
- IHCTRCH (object-time terminal error message and diagnostic traceback routine) -- entry name IHCTRCH. If the extended error message facility is specified, this module is replaced by IHCETRCH.
- IHCADJST -- processing boundary misalignment.
- IHCFINTH (object-time program interrupt processor). If the extended error message facility is specified, this module is replaced by IHCEFNTH.
- IHCERRM (object-time error message processor. The module monitors all execution-time errors).

Module names used in the following discussions are those in effect when the extended error message facility has not been specified. However, the descriptions apply also with the extended error message facility, unless otherwise stated.

IHCFCOMH receives input/output requests from the FORTRAN load module via compiler-generated calling sequences. IHCFCOMH, in turn, submits these requests to the appropriate data management interface (IHCFIOSH or IHCDIOSE).

IHCFIOSH receives sequential access input/output requests from IHCFCOMH and, in turn, submits those requests to the appropriate BSAM (basic sequential access method) routines for execution.

IHCDIOSE receives direct access input/output requests from IHCFCOMH and, in turn, submits those requests to the appropriate BDAM (basic direct access method) routines for execution.

If source statement errors are detected during compilation, the compiler generates a calling sequence to the IHCIBERH subprogram. IHCIBERH processes object-time errors resulting from improperly coded source statements. IHCFCVTH contains the various object time conversion routines required by IHCFCOMH and IHCNAMEL. IHCTRCH processes terminal object-time error messages and produces a diagnostic traceback for IHCFCOMH. IHCADJST processes object-time specification exceptions if the boundary alignment option is specified by the user during system generation.

IHCFCOMH

IHCFCOMH performs object-time implementation of the following FORTRAN source statements.

- READ and WRITE (for sequential input/output)
- READ, FIND, and WRITE (for direct access input/output)
- BACKSPACE, REWIND, and ENDFILE (sequential input/output device manipulation)
- STOP and PAUSE (write-to-operator)

In addition, IHCFCOMH: (1) initializes arithmetic-type program interruptions, and (2) terminates load module execuion.

All linkages from the load module to IHCFCOMH are compiler generated. Each time one of the above-mentioned source statements is encountered during compilation, the appropriate calling sequence to IHCFCOMH is generated and is included as part of the object module. At object-time, these calling sequences are executed, and control is passed to IHCFCOMH to perform the specified operation.

Note: IHCFCOMH itself does not perform the actual reading from or writing onto data sets. It submits requests for such operations to the appropriate input/output data management interface (IHCFIOSH or IHCDIOSE). The input/output interface, in turn, interprets and submits the requests to the appropriate access method (BSAM or BDAM) routines for execution. Figure 16 illustrates the relationship between IHCFCOMH and the input/output data management interfaces.

Charts G0, G1, and G2 illustrate the overall logic and the relationship between the routines of IHCFCOMH. Table 16, the IHCFCOMH routine directory, lists the routines used in IHCFCOMH and their functions.

The routines of IHCFCOMH are divided into the following categories:

- Read/write routines
- Input/output device manipulation routines
- Write-to-operator routines
- Utility routines

The read/write routines implement both the sequential input/output statements (READ and WRITE) and the direct access input/output statements (READ, FIND, and WRITE). (The direct access FIND statement is treated as a READ statement without format and list.)

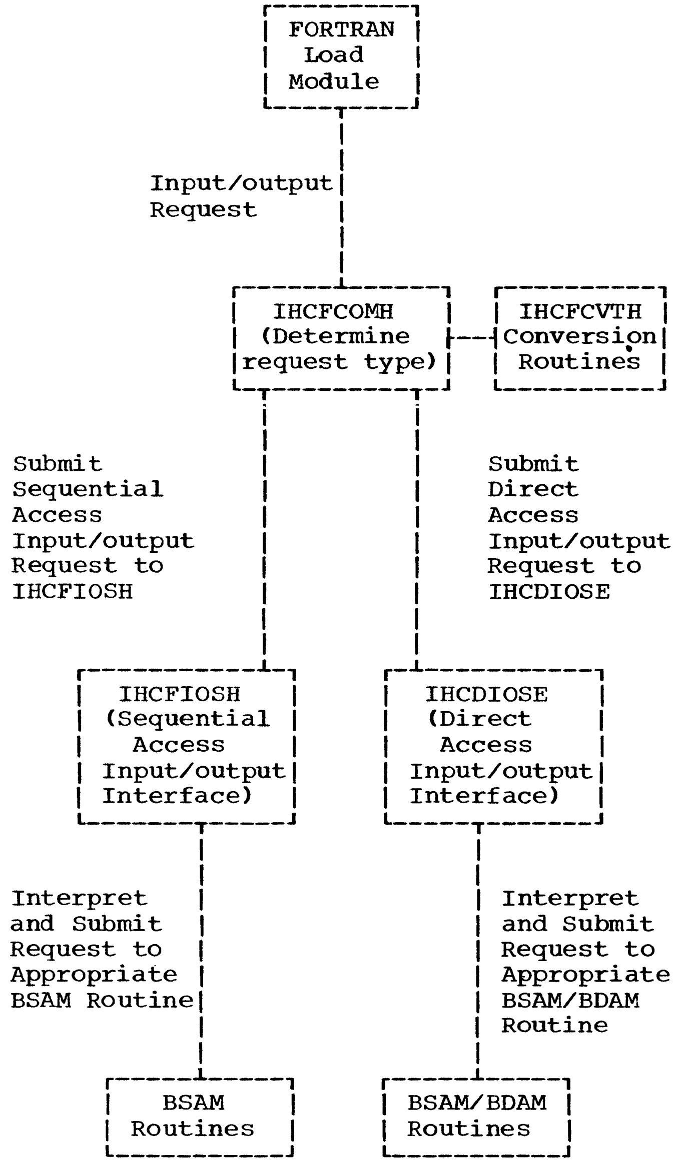


Figure 16. Relationship Between IHCFCOMH and Input/Output Data Management Interfaces

The input/output device manipulation routines implement the BACKSPACE, REWIND, and END FILE source statements for sequential data sets. These statements are ignored for direct access data sets.

The write-to-operator routines implement the STOP and PAUSE source statements.

The utility routines: (1) process errors detected by FORTRAN library subprograms, (2) process arithmetic-type program interrupts, and (3) terminate load module execution.

READ/WRITE ROUTINES

The READ/WRITE routines of IHCFCOMH implement the various types of READ/WRITE statements of the FORTRAN IV language. For simplicity, the discussion of these routines is divided into two parts:

- READ/WRITE statements not using NAMELIST
- READ/WRITE statements using NAMELIST

READ/WRITE Statements Not Using NAMELIST

For the implementation of both sequential and direct access READ and WRITE statements, the READ/WRITE routines of IHCFCOMH consist of the following three sections:

1. An opening section, which initializes data sets for reading and writing.
2. An I/O list section, which transfers data from an input buffer to the I/O list items or from the I/O list items to an output buffer.
3. A closing section, which terminates the input/output operation.

Within the discussion of each section, a read/write operation is treated in one of two ways:

- As a read/write requiring a format.
- As a read/write not requiring a format.

Note: In the following discussion, the term "read operation" implies both the sequential access READ statement and the direct access READ and FIND statements. The term "write operation" implies both the sequential access WRITE statement and the direct access WRITE statement.

OPENING SECTION: The compiler generates a calling sequence to one of four entry points in the opening section of IHCFCOMH each time it encounters a READ or WRITE statement in the FORTRAN source module. These entry points correspond to the operations of read or write, requiring or not requiring a format.

Read/Write Requiring a Format: If the operation is a read requiring a format, the opening section passes control to the appropriate input/output data management interface to initialize the unit number specified in the READ statement for reading. (The unit number is passed, as an argument, to the opening section via the calling sequence.) The input/output interface: (1) opens the data control block (via the OPEN macro instruction) for the specified data set if it was not previously opened, and (2) reads a record (via the READ macro instruction) containing data for the I/O list items into an input/output buffer that was obtained when the data control block was opened. The input/output interface then returns control to the opening section of IHCFCOMH. The address of the buffer and the length of the record read are passed to IHCFCOMH by the input/output interface. These values are saved for the I/O list section of IHCFCOMH. The opening section then passes control to a portion of IHCFCOMH that scans the FORMAT statement specified in the READ statement. (The address of the FORMAT statement is passed, as an argument, to the opening section via the calling sequence.) The first format code (either a control or conversion type) is then obtained.

For control type codes (e.g., an H format code or a group count), an I/O list item is not required. Control passes to the routine associated with the control code under consideration to perform the indicated operation. Control then returns to the scan portion, and the next format code is obtained. This process is repeated until either the end of the FORMAT statement or the first conversion code is encountered.

For conversion type codes (e.g., an I format code), an I/O list item is required. Upon the first encounter of a conversion code in the scan of the FORMAT statement, the opening section completes its processing of a read requiring a format and returns control to the next sequential instruction within the load module.

The action taken by IHCFCOMH when the various format codes are encountered is illustrated in Table 9.

●Table 9. IHCFCOMH FORMAT Code Processing

FORMAT Code	Description	Type	Corresponding Action Upon Code by IHCFCOMH
	beginning of statement	control	Save location for possible repetition of the format codes; clear counters.
n(	group count	control	Save n and location of left parenthesis for possible repetition of the format codes in the group.
n	field count	control	Save n for repetition of format code that follows.
nP	scaling factor	control	Save n for use by F, E, and D conversions.
Tn	column reset	control	Reset current position within record to _n_th column or byte.
nX	skip or blank	control	Skip n characters of an input record, or insert n blanks in an output record.
'text' or nH	literal data	control	Move n characters from an input record to the FORMAT statement, or n characters from the the FORMAT statement to an output record.
Fw.d Ew.d Dw.d Iw Aw Gw.d Lw.d Zw	F-conversion E-conversion D-conversion I-conversion A-conversion G-conversion L-conversion Z-conversion	conversion conversion conversion conversion conversion conversion conversion conversion	Exit to the load module to return control to entries FIOLF or FIOAF in IHCFCVTH. Using information passed to the I/O list section, the address and length of the current list item are obtained and passed to the proper conversion routine together with the current position in the input/output buffer, the scale factor, and the values of W and d). Upon return form the conversion routine the current field count is tested. If it is greater than 1, another exit is made to the load module to obtain the address of the next list item.
)	group end	control	Test group count. If it is greater than 1, repeat format codes in group: otherwise, continue to process FORMAT statement from current position.
/	record end	control	Input or output one record via input/output interface and READ/WRITE macro instruction.
	end of statement	control	If no I/O list items remain to be transmitted, return control to load module to link to the the closing section; if list items remain, input or output one record using input/outout interface and READ/WRITE macro instruction. Repeat format codes from last parenthesis.

If the operation is a write requiring a format, the opening section passes control to the input/output interface to initialize the unit number specified in the WRITE statement for writing. (The unit number is passed, as an argument, to the opening section via the calling sequence.) The input/output interface opens the data control block (via the OPEN macro instruction) for the specified data set if it was not previously opened. The input/output interface then returns control to the opening section of IHCFCOMH. The address of an input/output buffer that was obtained when the data control block was opened is saved for the I/O list section of IHCFCOMH. Subsequent opening section processing, starting with the scan of the FORMAT statement, is the same as that described for a read requiring a format.

Read/Write Not Requiring a Format: If the operation is a read or write not requiring a format, the opening section processing except for the scan of the FORMAT statement is the same as that described for a read or write requiring a format. (For a read or write not requiring a format, there is no FORMAT statement.)

I/O LIST SECTION: The compiler generates a calling sequence to one of four entry points in the I/O list section of IHCFCOMH each time it encounters an I/O list item associated with the READ or WRITE statement under consideration. These entry points correspond to a variable or an array list item for a read and write, requiring or not requiring a format. The I/O list section performs the actual transfer of data from: (1) an input buffer to the list items if a READ statement is being implemented, or (2) the list items to an output buffer if a WRITE statement is being implemented. In the case of a read or write requiring a format, the data must be converted before it is transferred.

Read/Write Requiring a Format: In processing a list item for a read requiring a format, the I/O list section passes control to the conversion routine associated with the conversion code for the list item. (The appropriate conversion routine is determined by the portion of IHCFCOMH that scans the FORMAT statement associated with the READ statement. The selection of the conversion routine depends on the conversion code of the list item being processed.) The selected conversion routine obtains data from an input buffer and converts the data to the form dictated by the conversion code. The converted data is then moved into the main storage address assigned to the list item.

In general, after a conversion routine has processed a list item, the I/O list section determines whether or not that routine can be applied to the next list item or array element (if an array is being processed). The I/O list section examines a field count that indicates the number of times a particular conversion code is to be applied to successive list items or successive elements of an array.

If the conversion code is to be repeate and if the previous list item was a vari able, the I/O list section returns contro to the load module. The load module agai branches to the I/O list section an passes, as an argument, the main storag address assigned to the next list item.

The conversion routine that processec the previous list item is then given control. This procedure is repeated until either the field count is exhausted or the input data for the READ statement is exhausted.

If the conversion code is to be repeated and if an array is being processed, the I/O list section computes the main storage address of the next element in the array. The conversion routine that processed the previous element is then given control. This procedure is repeated until either all the array elements associated with a specific conversion code are processed or the input data for the READ statement is exhausted.

If the conversion code is not to be repeated, control is passed to the scan portion of IHCFCOMH to continue the scan of the FORMAT statement. If the scan portion determines that a group of conversion codes is to be repeated, the conversion routines corresponding to those codes are applied to the next portion of the input data. This procedure is repeated until either the group count is exhausted or the input data for the READ statement is exhausted.

If a group of conversion codes is not to be repeated and if the end of the FORMAT statement is not encountered, the next format code is obtained. For a control type code, control is passed to the associated control routine to perform the indicated operation. For a conversion type code, control is returned to the load module if the previous list item was a variable. The load module again branches to the I/O list section and passes, as an argument, the main storage address assigned to the next list item. Control is then passed to the conversion routine associated with the new conversion code. The conversion routine then processes the data for this list item. If the data that was just converted was placed into an element of an array and if the entire array has not been filled, the I/O list section computes the main storage address of the next element in the array and passes control to the conversion routine associated with the new conversion code. The conversion routine then processes the data for this array element. Subsequent I/O list processing for a READ requiring a format proceeds at the point where the field count is examined.

If the scan portion encounters the end of the FORMAT statement and if all the list items are satisfied, control returns to the next sequential instruction within the load module. This instruction (part of the calling sequence to IHCFCOMH) branches to the closing section. If all the list items are not satisfied, control is passed to the input/output interface to read (via the READ macro instruction) the next input record. The conversion codes starting from the last left parenthesis are then repeated for the remaining list items.

If the operation is a write requiring a format, the I/O list section processing is similar to that for a read requiring a format. The main difference is that the conversion routines obtain data from the main storage addresses assigned to the list items rather than from an input buffer. The converted data is then transferred to an output buffer. If all the list items have not been converted and transferred prior to the encounter of the end-of-the FORMAT statement, control is passed to the input/output interface. The input/output interface writes (via the WRITE macro instruction) the contents of the current output buffer onto the output data set. The conversion codes starting from the last left parenthesis are then repeated for the remaining list items.

Read/Write Not Requiring a Format: In processing a list item for a read not requiring a format, the I/O list section must know the main storage address assigned to the list item and the size of the list item. Their values are passed, as arguments, via the calling sequence to the I/O list section. The list item may be either a variable or an array. In either case, the number of bytes specified by the size of the list item is moved from the input buffer to the main storage address assigned to the list item. The I/O list section then returns control to the load module. The load module again branches to the I/O list section and passes, as arguments, the main storage address assigned to the next list item and the size of the list item. The I/O list section moves the number of bytes specified by the size of the list item into the main storage address assigned to this list item. This procedure is repeated either until all the list items are satisfied or until the input data is exhausted. Control is then returned to the load module.

If the operation is a write not requiring a format, the I/O list section processing is similar to that described for a read not requiring a format. The main difference is that the data is obtained from the main storage addresses assigned to the list items and is then moved to an output buffer. In addition, the segment length (i.e., the number of bytes in the record segment) and a code indicating the position of this segment relative to other segments, if any, of the logical record are inserted in the segment control word.

CLOSING SECTION: The compiler generates a calling sequence to one of two entry points in the closing section of IHCFCOMH each time it encounters the end of a READ or WRITE statement in the FORTRAN source module. The entry points correspond to the operations of read and write, requiring or not requiring a format.

Read/Write Requiring a Format: If the operation is a read requiring a format, the closing section simply returns control to the load module to continue load module execution. If the operation is a write requiring a format, the closing section branches to the input/output interface. The input/output interface writes (via the WRITE macro instruction) the contents of the current input/output buffer (the final record) onto the output data set. The input/output interface then returns control to the closing section. The closing section, in turn, returns control to the load module to continue load module execution.

Read/Write Not Requiring a Format: If the operation is a read not requiring a format, the closing section branches to the input/output interface. The input/output interface reads (via the READ macro instruction) successive records until the end of the logical record being read is encountered. (A FORTRAN logical record consists of all the records necessary to contain the I/O list items for a WRITE statement not requiring a format.) When the input/output interface recognizes the end-of-logical-record indicator, control is returned to the closing section. The closing section, in turn, returns control to the load module to continue load module execution.

If the operation is a write not requiring a format, the closing section inserts: (1) the segment length (i.e., the number of bytes in the record segment) and, a code indicating that this segment is either the last or the only segment of the logical record into the segment control word of the input/output buffer to be written, and (2) an end-of-logical-record indicator into the last record of the input/output buffer being written. The closing section then branches to the input/output interface. The input/output interface writes (via the WRITE macro instruction) the contents of this input/output buffer onto the output data set. The input/output interface then returns control to the closing section. The closing section, in turn, returns control to the load module to continue load module execution.

Examples of IHCFCOMH READ/WRITE Statement Processing

The following examples illustrate the opening section, I/O list section, and closing section processing performed by IHCFCOMH for sequential access READ and WRITE statements, requiring or not requiring a format.

Note: IHCFCOMH processing for the direct access READ, FIND, and WRITE statements is essentially the same as that described for the sequential access READ and WRITE statements. The main difference is that for direct access statements, IHCFCOMH branches to the direct access input/output interface (IHCDIOSE) instead of to the sequential access input/output interface (IHCFIOSH).

READ REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following READ statement and FORMAT statement is illustrated in Table 10.

```
READ (1,2) A,B,C
2 FORMAT (3F12.6)
```

Table 10. IHCFCOMH Processing for a READ Requiring a Format

Opening Section	1. Receives control from load module and branches to IHCFIOSH to initialize data set for reading. 2. Passes control to scan portion of IHCFCOMH. 3. Returns control to load module.
I/O List Section	1. Receives control from load module, converts input data for A using IHCFCVTH, and moves converted data to A. 2. Returns control to load module. 3. Receives control from load module, converts input data for B, and moves converted data to B. 4. Returns control to load module. 5. Receives control from load module, converts input data for C, and moves converted data to C. 6. Returns control to load module.
Closing Section	1. Receives control from load module and closes out input/output operation. 2. Returns control to load module to continue load module execution.

WRITE REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following WRITE statement and FORMAT statement is illustrated in Table 11.

```
WRITE (3,2) (D(I),I=1,3)
2 FORMAT (3F12.6)
```

Table 11. IHCFCOMH Processing for a WRITE Requiring a Format

Opening Section	1. Receives control from load module and branches to IHCFIOSH to initialize data set for writing. 2. Passes control to scan portion of IHCFCOMH. 3. Returns control to load module.
I/O List Section	1. Receives control from load module, converts D(1), and moves D(1) to output buffer. 2. Returns control to load module. 3. Receives control from load module, converts D(2), and moves D(2) to output buffer. 4. Returns control to load module. 5. Receives control from load module, converts D(3), and moves D(3) to output buffer. 6. Returns control to load module.
Closing Section	1. Receives control from load module and branches to IHCFIOSH to write contents of output buffer. 2. Returns control to load module to continue load module execution.

READ NOT REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following READ statement is illustrated in Table 12.

```
READ (5) X,Y,Z
```

Table 12. IHCFCOMH Processing for a READ Not Requiring a Format

Opening Section	1. Receives control from load module and branches to IHCFIOSH to initialize data set for reading. 2. Returns control to load module.
I/O List Section	1. Receives control from load module and moves input data to X. 2. Returns control to load module. 3. Receives control from load module and moves input data to Y. 4. Returns control to load module. 5. Receives control from load module and moves input data to Z. 6. Returns control to load module.
Closing Section	1. Receives control from load module and branches to IHCFIOSH to read successive records until the end-of-logical-record indicator is encountered. 2. Returns control to load module to continue load module execution.

WRITE NOT REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following WRITE statement is illustrated in Table 13.

```
WRITE (6) (W(J),J=1,10)
```

Table 13. IHCFCOMH Processing for a WRITE Not Requiring a Format

Opening Section	1. Receives control from load module and branches to IHCFIOSH to initialize data for writing. 2. Returns control to load module.
I/O List Section	1. Receives control from load module and moves W(1) to output buffer. 2. Returns control to load module. 3. Receives control from load module and moves W(2) to output buffer. 4. Returns control to load module. . . . 5. Receives control from load module and moves W(10) to output buffer. 6. Returns control to load module.
Closing Section	1. Receives control from load module and branches to IHCFIOSH to write contents of output buffer. 2. Returns control to load module to continue load module execution.

READ/WRITE Statement Using NAMELIST

Included in the calling sequence to IHCNAMEL[1] generated by the compiler when it detects a READ or WRITE using a NAMELIST is a pointer to the object-time namelist dictionary associated with the READ or WRITE. This dictionary contains the names and addresses of the variables and arrays into which data is to be read or from which data is to be written. The dictionary also contains the information needed to select the conversion routine that is to convert the data to be placed into the variables or arrays, or to be taken from the variables and arrays.

READ USING NAMELIST: The data set containing the data to be input to the variables or arrays is initialized and successive records are read until the one containing the namelist name corresponding to that in the namelist dictionary is encountered. The next record is then read and processed.

The record is scanned and the first name is obtained. The name is compared to the variable and array names in the namelist dictionary. If the name does not agree, an error is signaled and load module execution is terminated. If the name is in the dictionary, processing of the matched variable or array is initiated.

Each initialization constant assigned to the variable or an array element is obtained from the input record. (One constant is required for a variable. A number of constants equal to the number of elements in the array is required for an array. A constant may be repeated for successive array elements if appropriately specified in the input record.) The appropriate conversion routine is selected according to the type of the variable or array element. Control is then passed to the conversion routine to convert the constant and to enter it into its associated variable or array element.

The process is repeated for the second and subsequent names in the input record. When an entire record has been processed, the next record is read and processed.

Processing is terminated upon recognition of the &END record. Control is then returned to the calling routine within the load module.

----

[1]IHCNAMEL is included in the load module only if reads and writes using NAMELISTs appear in the compiled program. Calls are made directly to FRDNL# (for READ) or to FWRNL# (for WRITE).

WRITE USING NAMELIST: The data set upon which the variables and arrays are to be written is initialized. The namelist name is obtained from the namelist dictionary associated with the WRITE, moved to an input/output buffer, and written. The processing of the variables and arrays is then initiated.

The first variable or array name in the dictionary is moved to an input/output buffer followed by an equal sign. The appropriate conversion routine is selected according to the type of the variable or array elements. Control is then passed to the conversion routine to convert the contents of the variable or the first array element and to enter it into the input/output buffer. A comma is inserted into the buffer following the converted quantity. If an array is being processed, the contents of its second and subsequent elements are converted, using the same conversion routine, and placed into the input/output buffer, separated by commas. When all of the array elements have been processed or if the item processed was a variable, the next name in the dictionary is obtained. The process is repeated for this and subsequent variable or array names.

If, at any time, the record length is exhausted, the current record is written and processing resumes in the normal fashion.

When the last variable or array has been processed, the contents of the current record are written, the characters &END are moved to the buffer and written, and control is returned to the calling routine within the load module.

Input/Output Device Manipulation Routines

The input/output device manipulation routines of IHCFCOMH implement the BACKSPACE, REWIND, and END FILE source statements. These routines receive control from within the load module via calling sequences that are generated by the compiler when these statements are encountered.

Note: Backspace, rewind, and end file requests are honored only for sequential data sets, and are ignored for direct access data sets. However, these statements are device independent and can be used for sequential data sets on either sequential or direct access devices.

The implementation of REWIND, BACKSPACE, and END FILE statements is straightforward. The input/output device manipulation routines submit the appropriate control request to IHCFIOSH, the input/output interface module. After the request is executed, control is returned to the calling routine within the load module.

Write-to-Operator Routines

The write-to-operator routines of IHCFCOMH implement the STOP and PAUSE source statements. These routines receive control from within the load module via calling sequences generated by the compiler upon recognition of the STOP and PAUSE statements.

STOP: A write-to-operator (WTO) macro instruction is issued to display the message associated with the STOP statement on the console. Load module execution is then terminated by passing control to the program termination routine of IHCFCOMH.

PAUSE: A write-to-operator-with-reply (WTOR) macro instruction is issued to display the message associated with the PAUSE statement on the console and to enable the operator's reply to be transmitted. A WAIT macro instruction is then issued to determine when the operator's reply has been transmitted. After the reply has been received, control is returned to the calling routine within the load module.

Utility Routines

The utility routines of IHCFCOMH perform the following functions:

- Process arithmetic-type program interruptions
- Process specification interruptions
- Terminate load module execution

PROCESSING OF ERROR MESSAGES: The error message processing routine (IHCERRM) receives control from various FORTRAN library subprograms when they detect terminal object-time errors.

Error message processing consists of initializing the data set upon which the message is to be written and writing the message and a diagnostic traceback. After the traceback is completed for error message IHC218I, control is passed to the statement designated in the ERR parameter of a FORTRAN READ statement, if that parameter was specified. In all other cases, control is passed to the termination routine, a routine that will terminate the job. Program interrupts will cause a message to be printed, but execution will continue. When the extended error message facility has been specified, execution may continue after the detection of an error.

PROCESSING OF ARITHMETIC INTERRUPTIONS: The interrupt routine (IBFINT) of IHCFCOMH initially receives control from within the load module via a compiler-generated calling sequence. The call is placed at the start of the executable coding of the load module so that the interrupt routine can set up the program interrupt mask. Subsequent entries into the interrupt routine are made through specification or arithmetic-type interruptions.

The interrupt routine sets up the program interrupt mask by means of a SPIE macro instruction. This instruction specifies the type of interruptions that are to cause control to be passed to the interrupt routine, and the location within the routine to which control is to be passed if the specified interruptions occur. After the mask has been set, control is returned to the calling routine within the load module.

In processing an interruption, the first step taken by the interrupt routine is to determine its type.

A. Arithmetic Interruptions: If exponential overflow or underflow has occurred, the appropriate indicators, which are referred to by OVERFL (a library subprogram), are set. If any type of divide check caused the interruption, the indicator referred to by DVCHK (also a library subprogram) is set.

Regardless of the type of interruption that caused control to be given to the interrupt routine, the old program PSW is written out for diagnostic purposes.

After the interruption has been processed, control is returned to the interrupted routine at the point of interruption.

B. Specification Interruptions: If an interrupt is caused by a specification exception and the boundary alignment option was specified by the user during system generation, the boundary adjustment routine (IHCADJST) is loaded from the link library (SYS1.LINKLIB).

This routine determines whether or not the interrupt was caused by an instruction that referred to improperly aligned data. If not, the routine causes abnormal termination of the load module. If so, the routine:

1. Causes message IHC210I, which contains the main program, PSW, to be generated.

2. Moves the misaligned data to a properly aligned boundary.

3. Re-executes the instruction that refers to the data.

If no interruption occurs when the instruction is re-executed, the data is moved back to its original location. If there is a new condition code, it is placed in the PSW of the Program Interruption Element (PIE). The boundary adjustment routine then returns control to the control program, which loads the PSW of the PIE to effect a return to the interrupted program.

If a divide check, exponential overflow or underflow interruption occurs when the instruction is re-executed, the interruption will be handled as described under "Arithmetic Interruptions."

If a data, protection, or addressing interruption occurs when the instruction is re-executed, the boundary adjustment routine generates the message IHC210I. The PSW information in this message gives the cause of the interruption and the location of the instruction in the main program that caused the interruption. Then, since processing cannot continue, the routine issues a SPIE macro instruction to remove specification interrupts from those interruptions handled by this routine and re-executes the instruction. This causes abnormal termination of the load module because of the original specification error.

PROGRAM TERMINATION: The load module termination routine (IBEXIT) of IHCFCOMH receives control from various library subprograms (e.g., DUMP and EXIT) and from other IHCFCOMH routines (e.g., the routine that processes the STOP statement).

This routine terminates execution of the load module by the following means:

- Calling the appropriate input/output interface(s) to check (via the CHECK macro instruction) outstanding write requests.

- Issuing a SPIE macro instruction with no parameters indicating that the FORTRAN object module no longer desires to give special treatment to program interruptions and does not want maskable interruptions to occur.

- Returning to the operating system supervisor.

CONVERSION ROUTINES (IHCFCVTH)

The conversion routines either convert data to be placed into I/O list items or convert data to be taken from I/O list items.

These routines receive control either from the I/O list section of IHCFCOMH during its processing of list items for READ/WRITE statements requiring a format, from the routines that process a READ/WRITE statements using a NAMELIST, or from the DUMP and PDUMP subprograms.

Each conversion routine is associated with a conversion type format code and/or a type. If an I/O list item for a READ/WRITE statement requiring a format is being processed, the conversion routine is selected according to the conversion type format code which is to be applied to the list item. If a list item for a READ/WRITE using a NAMELIST is being processed, the conversion routine is selected according to the type of the list item.

If a READ statement is being implemented, the conversion routine obtains data from the input/output buffer, converts it according to its associated conversion type format code or type, and enters the converted data into the list item. The process is reversed if a WRITE statement is being implemented.

For the DUMP and PDUMP subprograms, the format code parameter passed to them determines the selection of the output conversion routine to be used to place the output in the desired form.

See Table 17, at the end of this appendix, for a complete directory of the subroutines used by IHCFCVTH.

IHCFIOSH

IHCFIOSH, the object-time FORTRAN sequential access input/output data management interface, receives input/output requests from IHCFCOMH and submits them to the appropriate BSAM (basic sequential access method) routines and/or open and close routines for execution.

When the extended error message facility has been specified at system generation time, IHCFIOSH will include programming to allow execution to continue after an error occurs.

Chart G3 illustrates the overall logic and the relationship among the routines of IHCFIOSH. Table 18, the IHCFIOSH routine directory, lists the routines used in IHCFIOSH and their functions.

BLOCKS AND TABLE USED

IHCFIOSH uses the following blocks and table during its processing of sequential access input/output requests: (1) unit blocks, and (2) unit assignment table. The unit blocks are used to indicate input/output activity for each unit number (i.e., data set reference number) and to indicate the type of operation requested. In addition, the unit blocks contain skeletons of the data event control blocks (DECB) and the data control blocks (DCB) that are required for input/output operations. The unit assignment table is used as an index to the unit blocks.

ABYTE	BBYTE	CBYTE	DBYTE	4 bytes	Housekeeping Section
Address of Buffer 1				4 bytes	
Address of Buffer 2				4 bytes	
Current buffer pointer*				4 bytes	
Record offset (RECPTR)*				4 bytes	
Address of last DECB				4 bytes	
Mask for alternating buffers				4 bytes	
DECB1 skeleton section				20 bytes	
Not used			LIVECNT1	4 bytes	
DECB2 skeleton section				20 bytes	
Work space		Not used	LIVECNT2	4 bytes	
DCB skeleton section				88 bytes	

*Used only for variable length and/or blocked records.

• Figure 17. Format of a Unit Block for a Sequential Access Data Set

Unit Blocks

The first reference to each unit number (data set reference number) by an input/output operation within the FORTRAN load module causes IHCFIOSH to construct a unit block for each unit number. The main storage for the unit blocks is obtained by IHCFIOSH via the GETMAIN macro instruction. The addresses of the unit blocks are placed in the unit assignment table as the unit blocks are constructed. All subsequent references to the unit numbers are then made through the unit assignment table. Figure 17 illustrates the format of a unit block for a unit that is defined as a sequential access data set.

Each unit block is divided into four sections: a housekeeping section, two DECB skeleton sections, and a DCB skeleton section.

HOUSEKEEPING SECTION: The housekeeping section is maintained by IHCFIOSH. The information contained in it is used to indicate data set type, to keep track of input/output buffer locations, and to keep track of addresses internal to the input/output buffers to enable the processing of blocked records. The fields of this section are:

ABYTE

This field, containing the data set type passed to IHCFIOSH by IHCFCOMH, can be set to one of the following:

F0 -- Input data set requiring a format

FF -- Output data set requiring a format

00 -- Input data set not requiring a format

0F -- Output data set not requiring a format

BBYTE
This field contains bits that are set and examined by IHCFIOSH during its processing. The bits and their meanings, when on, are as follows:

0 -- Exit to IHCFCOMH on input/output error

1 -- Input/output error occurred

2 -- Current buffer indicator

3 -- Not used

4 -- End-of-current buffer indicator

5 -- Blocked data set indicator

6 -- Variable record format switch

7 -- Not used

CBYTE
This field also contains bits that are set and examined by IHCFIOSH. The bits and their meanings, when on, are as follows:

0 -- Data control block opened

1 -- Data control block not TCLOSEd

2 -- Data control block not previously opened

3 -- Buffer pool attached

4 -- Data set not previously rewound

5 -- Not used

6 -- Concatenation occurring -- reissue READ

7 -- Data set is DUMMY

DBYTE
This field contains bits that are set and examined by IHCFIOSH during the processing of an input/output operation involving a backspace request. The bits and their meanings, when on, are as follows:

0 -- A physical BACKSPACE has occurred

1 -- Previous operation was BACKSPACE

2 -- Not used

3 -- End of file routine should retain buffers

4 -- Not used

5 -- Not used

6 -- END FILE followed by BACKSPACE

7 -- Not used

Addresses of Buffer 1 and of Buffer 2
These fields contain pointers to the two input/output buffers obtained during the opening of the data control block for this data set.

Current Buffer Pointer
This field contains a pointer to the input/output buffer currently being used.

Record Offset (RECPTR)
This field contains a pointer to the current logical record within the current buffer.

Address of Last DECB
This field contains a pointer to the DECB last used.

Mask for Alternating Buffers
This field contains the bits which enable an Exclusive Or operation to alternate the current buffer pointer.

DECB SKELETON SECTIONS (DECB1 AND DECB2): The DECB (data event control block) skeleton sections are blocks of main storage within the unit block. They have the same format as the DECB constructed by the control program for an L format of an S-type READ or WRITE macro instruction (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro-Instructions, Form C28-6647). The various fields of the DECB skeleton are filled in by IHCFIOSH; the completed block is referred to when IHCFIOSH issues a read/write request to BSAM. The read/write field is filled in at open time. For each input/output operation, IHCFIOSH supplies IHCFCOMH with: (1) an indication of the type of operation (read or write), and (2) the length of and a pointer to the input/output buffer to be used for the operation.

LIVECNT1 and LIVECNT2
These fields indicate whether any input/output operation performed for the data set is unchecked. (A value of 1 indicates that a previous read or write has not been checked; a value of 0 indicates that all previous read and write operations for the data set have been checked.)

Work Space
This field is used to align the logical record length of a variable record segment on a fullword boundary.

DCB SKELETON SECTION: The DCB (data control block) skeleton section is a block of main storage within the unit block. It is of the same form as the DCB constructed by the control program for a DCB macro instruction under BSAM (see the Supervisor and Data Management Macro-Instructions publication, Form C28-6647). The various fields of the DCB skeleton are filled in by the control program when the DCB for the data set is opened (see the publication IBM System/360 Operating System: Concepts and Facilities). Standard default values may also be inserted in the DCB skeleton by IHCFIOSH. See "Unit Assignment Table" for a discussion of when default values are inserted into the DCB skeleton.

Unit Assignment Table

The unit assignment table (IHCUATBL) resides on the FORTRAN system library (SYS1.FORTLIB). Its size depends on the maximum number of units that can be referred to during execution of any FORTRAN load module. This number ($\leq$ 99) is specified by the user during the system generation process via the FORTLIB macro instruction.

The unit assignment table is designed to be used by both IHCFIOSH and IHCDIOSE. It is included once, by the linkage editor, in the FORTRAN load module as a result of an external reference to it within IHCFIOSH and/or IHCDIOSE.

The unit assignment table contains a 16-byte entry for each of the unit numbers that can be referred to by the user. These entries differ in format depending on whether the unit has been defined as a sequential access or a direct access data set.

Figure 18 illustrates the format of the unit assignment table.

Unit number (DSRN) being used for current operation		[5] n x 16		4 bytes
ERRMSG DSRN[1]	READ DSRN[2]	PRINT DSRN[3]	PUNCH DSRN[4]	4 bytes
UBLOCK01 field				4 bytes
DSRN01 default values				8 bytes
LIST01 field				4 bytes
. . .				. . .
UBLOCKn field[6]				4 bytes
DSRNn default values[7]				8 bytes
LISTn field[8]				4 bytes

[1]Unit number (DSRN) of error output device.

[2]Unit number (DSRN) of input device for a read of the form: READ b,list.

[3]Unit number (DSRN) of output device for a print operation of the form: PRINT b,list.

[4]Unit number (DSRN) of output device for a punch operation of the form: PUNCH b,list.

[5]n is the maximum number of units that can be referred to by the FORTRAN load module. The size of the unit table is equal to (8 + n x 16) bytes.

[6]The UBLOCKn field contains either a pointer to the unit block constructed for unit number n if the unit is being used at object time, or a value of 1 if the unit is not being used.

[7]The default values for the various unit numbers are specified by the user and are assembled into the unit assignment table entries during the system generation process. The default values are used only by IHCFIOSH; they are ignored by IHCDIOSE.

[8]If the unit is defined as a direct access data set, the LISTn field contains a pointer to the parameter list that defines the direct access data set. Otherwise, this field contains a value of 1.

Figure 18. Unit Assignment Table Format

Because IHCFIOSH deals only with sequential access data sets, the remainder of the discussion on the unit assignment table is devoted to unit assignment table entries for sequential access data sets. If IHCFIOSH encounters a reference to a direct access data set, it is considered as an error, and control is passed to the load module termination routine of IHCFCOMH.

The pointers to the unit blocks created for sequential data sets are inserted into the unit assignment table entries by IHCFIOSH when the unit blocks are constructed.

Note: Default values are standard values that IHCFIOSH inserts into the appropriate fields (e.g., BUFNO) of the DCB skeleton section of the unit blocks if the user does either of the following:

1. Causes the load module to be executed via a cataloged procedure, or

2. Fails, in stating his own procedure for execution, to include in the DCB parameter of his DD statements those subparameters (e.g., BUFNO) that he is permitted to include (see the publication IBM System/360 Operating System: FORTRAN IV (G) Programmer's Guide).

Control is returned to IHCFIOSH during data control block opening so that it can determine whether or not the user has included the subparameters in the DCB parameter of his DD statements. IHCFIOSH examines the DCB skeleton fields corresponding to user-permitted subparameters and, upon encountering a null field (indicating that the user has not specified the subparameter), inserts the standard value (i.e., the default value) for the subparameter into the DCB skeleton. (If the user has included these subparameters in his DD statement, the control program routine performing data control block opening inserts the subparameter values, before transferring control to IHCFIOSH, into the DCB skeleton fields reserved for those values.)

BUFFERING

All input/output operations are double buffered. (The double buffering scheme can be overridden by the user if he specifies in a DD statement: BUFNO=1.) This implies that during data control block opening, two buffers will be obtained. The addresses of these buffers are given alternately to IHCFCOMH as pointers to:

- Buffers to be filled (in the case of output)

- Information that has been read in and is to be processed (in the case of input)

COMMUNICATION WITH THE CONTROL PROGRAM

In requesting services of the control program, IHCFIOSH uses L and E forms of S-type macro instructions (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro-Instructions, Form C28-6647).

OPERATION

The processing of IHCFIOSH is divided into five sections: initialization, read, write, device manipulation, and closing. When called by IHCFCOMH, a section of IHCFIOSH performs its function and then returns control to IHCFCOMH.

Initialization

The initialization action taken by IHCFIOSH depends upon the nature of the previous input/output operation requested for the data set. The previous operation possibilities are:

- No previous operation

- Previous operation read or write

- Previous operation backspace

- Previous operation write end-of-data set

- Previous operation rewind

NO PREVIOUS OPERATION: If no previous operation has been performed on the unit specified in the input/output request, the initialization section generates a unit block for the unit number. The data set to be created is then opened (if the current operation is not rewind or backspace) via the OPEN macro instruction. The addresses of the input/output buffers, which are obtained during the opening process and placed into the DCB skeleton, are placed into the appropriate fields of the housekeeping section of the unit block. The DECB skeleton is then set to reflect the nature of the operation (read or write), the format of the records to be read or written, and the address of the input/output buffer to be used in the operation.

If the requested operation is a write, a pointer to the buffer position, at which IHCFCOMH is to place the record to be written, and the block size or logical record length (to accommodate blocked logical records) are placed into registers, and control is returned to IHCFCOMH.

If the requested operation is a read, a record is read, via a READ macro instruction, into the input/output buffer, and the operation is checked for completion via the CHECK macro instruction. A pointer to the location of the record within the buffer, along with the number of bytes read or the logical record length, are placed into registers, and control is returned to IHCFCOMH.

Note: During the opening process, control is returned to the IHCDCBXE routine in IHCFIOSH. This routine determines if the data set being opened is a 1403 printer. If it is, the RECFM field in the DCB for the data set is altered to machine carriage control (FM). In addition, a pointer to the unit block generated for the printer, and the physical address of the printer are placed into a control block area (CTLBLK) for the printer within IHCFIOSH. CTLBLK also contains a third print buffer. This buffer is used in conjunction with the two buffers already obtained for the printer.

Figure 19 illustrates the format of CTLBLK.

CTLBLK	a(BUF 3)		4 bytes
	a(unit block)		4 bytes
	a(printer)	record length	4 bytes
	[1] FT00		4 bytes
	[1] F001		4 bytes
BUF3	third print buffer		144 bytes
	[1]Used in the task input/output table (TIOT) search.		

Figure 19. CTLBLK Format

PREVIOUS OPERATION READ OR WRITE: If the previous operation performed on the unit specified in the present input/output request was either a read or write, the initialization section determines the nature of the present input/output request. If it is a write, a pointer to the buffer position, at which IHCFCOMH is to place the record to be written, and the block size or logical record length are placed into registers, and control is returned to IHCFCOMH.

If the operation to be performed is a read, a pointer to the buffer location of the record to be processed, along with the number of bytes read or logical record length, are placed into registers, and control is returned to IHCFCOMH.

PREVIOUS OPERATION BACKSPACE: If the previous operation performed on the unit specified in the present input/output request was a backspace, the initialization section determines the type of the present operation (read or write) and modifies the DECB skeleton, if necessary, to reflect the operation type. (If the operation type is the same as that of the operation that preceded the backspace request, the DECB skeleton need not be modified.) Subsequent processing steps are the same as those described for "No Previous Operation," starting at the point after the DECB skeleton is set to reflect operation type.

PREVIOUS OPERATION WRITE END-OF-DATA SET: If the previous operation performed on the unit specified in the present input/output request was a write end-of-data set, a new data set using the same unit number is to be created. In this case, the initialization section closes the data set. Then, in order to establish a correspondence between the new data set and the DD statement describing that data set, IHCFIOSH increments the unit sequence number of the ddname. (The ddname is placed into the appropriate field of the DCB skeleton prior to the opening of the initial data set associated with the unit number.) During the opening of the data set, the ddname will be used to merge with the appropriate DD statement. The data set is then opened. Subsequent processing steps are the same as those described for "No Previous Operation," starting at the point after the data set is opened.

PREVIOUS OPERATION REWIND: If the previous operation performed on the unit specified in the present input/output request was a rewind, the ddname is initialized (set to FTxxF001) in order to establish a correspondence between the initial data set associated with the unit number and the DD statement describing that data set. The data set is then opened. Subsequent processing steps are the same as those described for "No Previous Operation," starting at the point after the data set is opened.

Read

The read section of IHCFIOSH performs two functions: (1) reads physical records into the buffers obtained during data set opening, and (2) makes the contents of these buffers available to IHCFCOMH for processing.

If the records being processed are blocked, the read section does not read a physical record each time it is given control. IHCFIOSH only reads a physical record when all of the logical records of the blocked record under consideration have been processed by IHCFCOMH. However, if the records being processed are either unblocked or of U-format, the read section of IHCFIOSH issues a READ macro instruction each time it receives control.

The reading of records by this section is overlapped. That is, while the contents of one buffer are being processed, a physical record is being read into the other buffer. When the contents of one buffer have been processed, the read into the other buffer is checked for completion. Upon completion of the read operation, processing of that buffer's contents is initiated. In addition, a read into the second buffer is initiated.

Each time the read section is given control it makes the next record available to IHCFCOMH for processing. (In the case of blocked records, the record presented to IHCFCOMH is logical.) The read section of IHCFIOSH places: (1) a pointer to the record's location in the current input/output buffer, and (2) the number of bytes read or logical record length into registers, and then returns control to IHCFCOMH.

Write

The write section of IHCFIOSH performs two functions: (1) writes physical records, and (2) provides IHCFCOMH with buffer space in which to place the records to be written.

If the records being written are blocked, the write section does not write a physical record each time it is given control. IHCFIOSH only writes a physical record when all of the logical records that comprise the blocked record under consideration have been placed into the input/output buffer by IHCFCOMH. However, if the records being written are either unblocked or of U-format, the write section of IHCFIOSH issues a WRITE macro instruction each time it receives control.

The writing of records by this section is overlapped. That is, while IHCFCOMH is filling one buffer, the contents of the other buffer are being written. When an entire buffer has been filled, the write from the other buffer is checked for completion. Upon completion of the write operation, IHCFCOMH starts placing records into that buffer. In addition, a write from the second buffer is initiated.

Each time control is transferred to the write section, it provides IHCFCOMH with buffer space in which to place the record to be written. IHCFIOSH places: (1) a pointer to the location within the current buffer at which IHCFCOMH is to place the record, and (2) the block size or logical record length into registers, and then returns control to IHCFCOMH.

Note: The write section checks to see if the data set being written on is a 1403 printer. If it is, the carriage control character is changed to machine code, and three buffers, instead of the normal two, are used when writing on the printer.

ERROR PROCESSING WITHOUT EXTENDED ERROR MESSAGE FACILITY: An error number is put into a parameter list and register 13 is set up to point to a save area in IBCOM. The user's save area is linked to this save area. The error monitor is then called to print a message on the object error unit.

ERROR PROCESSING WITH EXTENDED ERROR MESSAGE FACILITY: A common subroutine is called to prepare for a call to the error monitor. The common subroutine:

1. Converts the data set reference and puts it into the last four bytes of the message.
2. Links save areas as described when no error message facility has been specified.
3. Calls the error monitor (IHCERRM).

The error monitor may return to continue execution.

For error conditions 214 and 217, 218, 219, 220, and 231 if user corrective action is taken, and for error 214 if the operation was input, the remainder of the I/O list is ignored upon return from the common subroutine. For error condition 214 under any other condition, the record format is changed to V and execution continues.

For any error condition except 214 and 217, upon return from the error monitor, IHCFIOSH returns an indication that an error has occurred to the caller.

In the case of an end-of-data set, IHCFIOSH simply passes control to the end-of-data set routine of IHCFCOMH.

Chart G4 illustrates the execution-time input/output recovery procedure for any input/output errors detected by the input/output supervisor.

Device Manipulation

The device manipulation section of IHCFIOSH processes backspace, rewind, and write end-of-data set requests.

BACKSPACE: IHCFIOSH processes the backspace request by issuing the appropriate number of BSP (physical backspace) macro instructions (0, 1, 2, or 3) and adjusting the RECPTR in the unit block to point to the preceding logical record. The number of BSP's issued will depend on the number of buffers used, the previous input/output operation, and the position of RECPTR prior to the backspace.

For unformatted records, the processing of a backspace request also includes examining the SDW (segment descriptor word) of each record segment in order to locate the first segment of a spanned record (i.e., a logical record which causes more than one physical input/output operation to be performed). Control is then returned to IHCFCOMH.

REWIND: IHCFIOSH processes the rewind request by issuing a CLOSE macro instruction, using the REREAD option. This option has the same effect as a rewind. Control is then returned to IHCFCOMH.

WRITE END-OF-DATA SET: IHCFIOSH processes this request by issuing a CLOSE macro instruction, type=T. It then frees the input/output buffers by issuing a FREEPOOL macro instruction, and returns control to IHCFCOMH.

Closing

The closing section of IHCFIOSH examines the entries in the unit assignment table to determine which data control blocks are open. In addition, this section ensures that all write operations for a data set are completed before the data control block for that data set is closed. This is done by issuing a CHECK macro instruction for all double-buffered output data sets. Control is then returned to IHCFCOMH.

Note: If a 1403 printer is being used, a write from the last print buffer is issued to ensure that the last line of output is written.

IHCDIOSE

IHCDIOSE, the object-time FORTRAN direct access input/output data management interface, receives input/output requests from IHCFCOMH and submits them to the appropriate BDAM (basic direct access method) routines and/or open and close routines for execution. (For the first input/output request involving a nonexistent data set, the appropriate BSAM routines must be executed prior to linking to the BDAM routines. The BSAM routines format and create a new data set consisting of blank records.)

IHCDIOSE receives control from: (1) the initialization section of the FORTRAN load module if a DEFINE FILE statement is included in the source module, and (2) IHCFCOMH whenever a READ, WRITE, or FIND direct access statement is encountered in the load module.

Charts G5 and G6 illustrate the overall logic and the relationship among the routines of IHCDIOSE. Table 19, the IHCDIOSE routine directory, lists the routines used in IHCDIOSE and their functions.

BLOCKS AND TABLE USED

IHCDIOSE uses the following blocks and table during its processing of direct access input/output requests: (1) unit blocks, and (2) unit assignment table. The unit blocks are used to indicate input/output activity for each unit number (i.e., data set reference number) and to indicate the type of operation requested. In addition, each unit block contains skeletons of the data event control blocks (DECB) and the data control block (DCB) that are required for input/output operations. The unit assignment table is used as an index to the unit blocks.

Unit Blocks

The first reference to each unit number (i.e., data set reference number) by a direct access input/output operation within the FORTRAN load module causes IHCDIOSE to construct a unit block for each of the referenced unit numbers. The main storage for the unit blocks is obtained by IHCDIOSE via the GETMAIN macro instruction. The addresses of the unit blocks are inserted into the corresponding unit assignment table entries as the unit blocks are constructed. Subsequent references to the unit numbers are then made through the unit assignment table.

Figure 20 illustrates the format of a unit block for a unit that has been defined as a direct access data set.

IOTYPE	STATUSU	Not used	Not used	4 bytes
RECNUM				4 bytes
STATUSA	CURBUF			4 bytes
BLKREFA				4 bytes
STATUSA	CURBUF			4 bytes
BLKREFB				4 bytes
DECBA				28 bytes
DECBB				28 bytes
DCB				104 bytes

Figure 20. Format of a Unit Block for a Direct Access Data Set

The meanings of the various unit block fields are outlined below.

IOTYPE
This field, containing the data set type passed to IHCDIOSE by IHCFCOMH, can be set to one of the following:

F0 -- Input data set requiring a format

FF -- Output data set requiring a format

00 -- Input data set not requiring a format

0F -- Output data set not requiring a format

STATUSU
This field specifies the status of the associated unit number. The bits and their meanings, when on, are as follows;

0 -- Data control block for data set is open for BSAM

1 -- Error occurred

2 -- Two buffers are being used

3 -- Data control block for data set is open for BDAM

4-5 -- 10 - U-form specified in DEFINE FILE statement

01 - E-form specified in DEFINE FILE statement

11 - L-form specified in DEFINE FILE statement

6-7 -- Not used

Note: IHCDIOSE references only bits 1, 2, and 3.

RECNUM
This field contains the number of records in the data set as specified in the parameter list for the data set in a DEFINE FILE statement. It is filled in by the file initialization section after the data control block for the data set is opened.

STATUSA
This field specifies the status of the buffer currently being used. The bits and their meanings, when on, are as follows:

0 -- READ macro-instruction has been issued

1 -- WRITE macro instruction has been issued

2 -- CHECK macro instruction has been issued

3-7 -- Not used

CURBUF
This field contains the address of the DECB skeleton currently being used. It is initialized to contain the address of the DECBA skeleton by the file initialization section of IHCDIOSE after the data control block for the data set is opened.

BLKREFA
This field contains an integer that indicates either the relative position within the data set of the record to be read, or the relative position within the data set at which the record is to be written. It is filled in by either the read or write section of IHCDIOSE prior to any reading or writing. In addition, the address of this field is inserted into the DECBA skeleton by the file initialization section of IHCDIOSE after the data control block for the data set is opened.

STATUSB
This field specifies the status of the next buffer to be used if two buffers are obtained for this data set during data control block opening. The bits and their meanings are the same as described for the STATUSA field. However, if only one buffer is obtained during data control block opening, this field is not used.

NXTBUF
This field contains the address of the DECB skeleton to be used next if two buffers are obtained during data control block opening. It is initialized to contain the address of the DECBB skeleton by the file initialization section of IHCDIOSE after the data control block for the data set is opened. However, if only one buffer is obtained during data control block opening, this field is not used.

BLKREFB
The contents of this field are the same as described for the BLKREFA field. It is filled in either by the read or the write section of IHCDIOSE prior to any reading or writing. In addition, the address of this field is inserted into the DECBB skeleton by the file initialization section of IHCDIOSE after the data control block for the data set is opened. However, if only one buffer is obtained during data control block opening, this field is not used.

DECBA Skeleton
This field contains the DECB (data event control block) skeleton to be used when reading into or writing from the current buffer. It is of the same form as the DECB constructed by the control program for an L form of an S-type READ or WRITE macro instruction under BDAM (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro-Instructions.

The various fields of the DECBA skeleton are filled in by the file initialization section of IHCDIOSE after the data control block for the data set is opened. The completed DECB is referred to when IHCDIOSE issues a read or a write request to BDAM. For each input/output operation, IHCDIOSE supplies IHCFCOMH with the address of and the size of the buffer to be used for the operation.

DECBB Skeleton
The DECBB skeleton is used when reading into or writing from the next buffer. Its contents are the same as those described for the DECBA skele-

ton. The DECBB skeleton is completed in the same manner as described for the DECBA skeleton. However, if only one buffer is obtained during data control block opening, this field is not used.

DCB Skeleton
: This field contains the DCB (data control block) skeleton for the associated data set. It is of the same form as the DCB constructed by the control program for a DCB macro instruction under BDAM (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro-Instructions).

The various fields of the DCB skeleton are filled in by the control program when the DCB for the data set is opened (see the publication IBM System/360 Operating System: Concepts and Facilities).

Unit Assignment Table

The unit assignment table (IHCUATBL) resides on the FORTRAN system library (SYS1.FORTLIB). Its size depends on the maximum number of units that can be referred to during execution of any FORTRAN load module. This number (≤99) is specified by the user during the system generation process via the FORTLIB macro instruction.

The unit assignment table is designed to be used by both IHCFIOSH and IHCDIOSE. It is included once, by the linkage editor, in the FORTRAN load module as a result of an external reference to it within IHCFIOSH and/or IHCDIOSE.

The unit assignment table contains a 16-byte entry for each of the unit numbers that can be referred to by either IHCDIOSE or IHCFIOSH. These entries differ in format depending on whether the unit has been defined as a direct access or as a sequential access data set. Because IHCDIOSE deals only with direct access data sets, only the entry for a direct access unit is shown here. (See the IHCFIOSH section "Table and Blocks Used", for the format of the unit assignment table as a whole.) If IHCDIOSE encounters a reference to a sequential access data set, it is considered as an error, and control is passed to the load module termination routine of IHCFCOMH.

Figure 21 illustrates the unit assignment table entry format for a direct access data set.

Pointer to unit block xx (UBLOCKxx)	4 bytes
Default values for DSRNxx (applies only to sequential access data sets -- not used by IHCDIOSE)	8 bytes
Pointer to parameter list xx (LISTxx)	4 bytes
UBLOCKxx is the unit block generated for unit number xx. DSRNxx is the unit number for the direct access data set (xx≤99). LISTxx is the parameter list that defines the direct access data set associated with unit number xx.	

Figure 21. Unit Assignment Table Entry for a Direct Access Data Set

The pointers to the unit blocks are inserted into the unit assignment table entries by IHCDIOSE when the unit blocks are constructed.

The pointers to the parameter lists are inserted into the unit assignment table entries by IHCDIOSE when it receives control from the initialization section of the FORTRAN load module being executed.

BUFFERING

All direct access input/output operations are double buffered. (The double buffering scheme may be overridden by the user if he specifies in his DD statements: BUFNO=1.) This implies that during data control block opening, two buffers will be obtained for each data set. The addresses of these buffers are given alternately to IHCFCOMH as pointers to:

- Buffers to be filled in the case of output
- Data that has been read in and is to be processed in the case of input

Each buffer has its own DECB. This increases input/output efficiency by overlapping of input/output operations.

COMMUNICATION WITH THE CONTROL PROGRAM

In requesting services of the control program BSAM and BDAM routines, IHCDIOSE uses L and E forms of S-type macro instructions (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro-Instructions).

OPERATION

The processing of IHCDIOSE is divided into five sections: file definition, file initialization, read, write, and termination. When a section receives control, it performs its functions and then returns control to the caller (either the FORTRAN load module or IHCFCOMH).

File Definition Section

The file definition section is entered from the FORTRAN load module, via a compiler-generated calling sequence, if a DEFINE FILE statement is included in the FORTRAN source module. The file definition section performs the following functions:

- Checks for the redefinition of each direct access unit number.
- Enters the address of each direct access unit number's parameter list into the appropriate unit assignment table entry.
- Establishes addressability for IHCDIOSE within IHCFCOMH.

Each direct access unit number appearing in a DEFINE FILE statement is checked to determine whether it has been defined previously. If it has been defined previously, the current definition is ignored. If it has not been defined previously, the address of its parameter list (i.e., the definition of the unit number) is inserted into the proper entry in the unit assignment table. The next unit number, if any, is then obtained.

When the last unit number has been processed in the above manner, the file definition section stores the address of IHCDIOSE into the FDIOCS field within IHCFCOMH. This enables IHCFCOMH to link to IHCDIOSE when IHCFCOMH encounters a direct access input/output statement. Control is then returned to the FORTRAN load module to continue normal processing.

File Initialization Section

The file initialization section receives control from IHCFCOMH whenever input or output is requested for a direct access data set. The processing performed by the initialization section depends on whether an input/output operation was previously requested for the data set.

NO PREVIOUS OPERATION: If no operation was previously requested for the data set specified in the current input/output request, the file initialization section first constructs a unit block for the data set. (The GETMAIN macro instruction is used to obtain the main storage for the unit block.) The address of the unit block is inserted into the appropriate entry in the unit assignment table.

The file initialization section then reads the JFCB (job file control block) via the RDJFCB macro instruction. The value in the BUFNO field of the JFCB is inserted into the DCB skeleton in the unit block. This value indicates the number of buffers that are obtained for this data set when its data control block is opened. If the BUFNO field is null (i.e., if the user did not include the BUFNO subparameter in the DD statement for this data set), or other than 1 or 2, the file initialization section inserts a value of 2 into the DCB skeleton.

The file initialization section next examines the JFCBIND2 field in the JFCB to determine if the data set specified in the current input/output request exists. If the JFCBIND2 field indicates that the specified data set does not exist, and if the current request is a write, a new data set is created. (If the current request is a read, an error is indicated and control is returned to IHCFCOMH which may terminate load module execution. If the current request is a find, the request is ignored, and control is returned to IHCFCOMH.) If the JFCBIND2 field indicates that the specified data set already exists, a new data set is not created for BDAM use.

If the specified data set is already opened when the file initialization section is entered, the following checks are made: (1) If the data set is already opened for BDAM, the appropriate branch is taken to perform a READ or WRITE operation. (2) If the specified data set has been opened for BSAM, the data set is then closed, since an input/output error must have occurred during the formatting of the data set. The data set is then reopened to provide a fresh start.

The file initialization section processing for a data set to be created, and for a data set that already exists is discussed in the following paragraphs.

Data Set to be Created: The data control block for the new data set is first opened for the BSAM, load mode, WRITE macro instruction. The BSAM WRITE macro instruction is used to create a new data set according to the format specified in the parameter list for the data set in a DEFINE FILE statement. The data control block is then closed. Subsequent file initialization section processing after creating the new data set is the same as that described for a data set that already exists (see the section "Data Set Already Exists").

Data Set Already Exists: The data control block for the data set is opened for direct access processing by the BDAM routines. After the data control block is opened, the file initialization section fills in various fields in the unit block:

- The number of records in the data set is inserted into the RECNUM field.
- The address of the DECB skeletons (DECBA and DECBB) are inserted into the CURBUF and the NXTBUF fields, respectively.
- The addresses of the input/output buffers obtained during data control block opening are inserted into the appropriate DECB skeletons.
- The address of the BLKREFA and the BLKREFB fields in the unit block are inserted into the appropriate DECB skeletons.

Note: If the user specifies BUFNO=1 in the DD statement for this data set, only one input/output buffer is obtained during data control block opening. In this case, the NXTBUF field, the BLKREFB field, and the DECBB skeleton are not used.

Subsequent file initialization section processing for the case of no previous operation depends upon the nature of the input/output request (FIND, READ, or WRITE). This processing is the same as that described for the case of a previous operation (see the section "Previous Operation").

PREVIOUS OPERATION: If an operation was previously requested for the data set specified in the current input/output request, the file initialization section processing depends upon the nature of the current input/output request.

If the current request is either a find or a read, control is passed to the read section.

If the current request is a write, control is passed to the secondary entry in the write section.

Read Section

The read section of IHCDIOSE processes read and find requests. The read section may be entered either from the file initialization section of IHCDIOSE, or from IHCFCOMH. In either case, the processing performed is the same. In processing read and find requests, the read section performs the following functions:

- Reads physical records into the buffer(s) obtained during data control block opening.
- Makes the contents of these buffers available to IHCFCOMH for processing.
- Updates the associated variable that is defined in the DEFINE FILE statement for the data set.

Upon receiving control, the read section first checks to see whether the record to be found or read is already in an input/output buffer. Subsequent read section processing depends upon whether the record is in the buffer.

RECORD IN BUFFER: If a record is in the buffer, the read section determines whether the current request is a find or a read.

If the current request is a find, the associated variable for the data set is updated so that it points to the relative position within the direct access data set of the record that is in the buffer. Control is then returned to IHCFCOMH.

If the current request is a read, the read operation that read the record into the buffer is checked for completion. The read section then places the address of the buffer and the size of the buffer into registers for use by IHCFCOMH. The associated variable for the data set is updated so that it points to the relative position within the direct access data set of the record following the record just read. Control is then returned to IHCFCOMH.

RECORD NOT IN BUFFER: If a record is not in the buffer, the read section first obtains the address of the buffer to be used for the current request. The relative record number of the record to be read is then inserted into the appropriate BLKREF field in the unit block (i.e., BLKREFA or BLKREFB). The proper record is then read from the specified data set into the buffer. Subsequent read section processing for the case of a record not in the buffer is the same as that described for a record in the buffer (see the section "Record In Buffer").

Note 1: Record retrieval can proceed concurrently with CPU processing only if the user alternates FIND statements with READ statements in his program.

Note 2: If an input/output error occurs during reading, the control program returns control to the synchronous exit routine (SYNADR) within IHCDIOSE. The SYNADR routine sets a switch to indicate that an input/output error has occurred, and then returns control to the control program. The control program completes its processing and returns control to IHCDIOSE. IHCDIOSE interrogates the switch, finds it to be set, and passes control to the input/output error routine of IHCFCOMH (see "Error Processing").

Write Section

The write section of IHCDIOSE processes write requests. The write section may be entered either from the file initialization section of IHCDIOSE or from IHCFCOMH. The processing performed by the write section depends upon where it is entered from.

PROCESSING IF ENTERED FROM FILE INITIALIZATION SECTION: If the write section is entered from the file initialization section of IHCDIOSE, no writing is performed. The write section only provides IHCFCOMH with buffer space in which to place the record to be written. The relative record number of the record to be written is inserted into the appropriate BLKREF field (i.e., BLKREFA or BLKREFB). (The record is written the next time the write section is entered.) For a formatted write, the buffer is filled with blanks. For an unformatted write, the buffer is filled with zeros. The write section then places the address of the buffer and the size of the buffer into registers for use by IHCFCOMH. Control is then returned to IHCFCOMH.

PROCESSING IF ENTERED FROM IHCFCOMH: Each time the write section is entered from IHCFCOMH, it writes the contents of the buffer onto the specified data set. Subsequent write section processing for entrances from IHCFCOMH is the same as that described for entrances from the file initialization section of IHCDIOSE (see the section "Processing if Entered from File Initialization Section"). In addition, the associated variable is modified prior to returning to IHCFCOMH. The associated variable for the data set is updated so that it points to the relative position within the direct access data set of the record following the record just written.

Note 1: The writing of physical records by this section is overlapped. That is, while IHCFCOMH is filling buffer A, buffer B is being written onto the output data set. When buffer A has been filled, the write from buffer B is checked for completion. Upon completion of the write operation, IHCFCOMH starts placing data into buffer B. In addition, a write from buffer A is initiated.

Note 2: If an input/output error occurs during writing, the control program returns control to the synchronous exit routine (SYNADR) within IHCDIOSE. The SYNADR routine sets a switch to indicate that an input/output error has occurred, and then returns control to the control program. The control program completes its processing and returns control to IHCDIOSE. IHCDIOSE interrogates the switch, finds it to be set, and passes control to the input/output error routine of IHCFCOMH (see "Error Processing").

Error Processing

The way in which errors are processed is dependent upon whether or not the extended error message facility was specified at system generation time.

WITHOUT EXTENDED ERROR MESSAGE FACILITY: An error number is put into a parameter list and register 13 is set up to point to a save area in IBCOM. The user's save area is linked to this save area. The error monitor is then called.

WITH EXTENDED ERROR MESSAGE FACILITY: A 2-part common subroutine is called to prepare for a call to the error monitor. The first part of the subroutine links save areas as described when no error message facility has been specified. It is used only when an error occurs in the portion of IHCDIOSE which was called directly from the problem program - i.e., for error conditions 234 and 235. The second part of the common subroutine is used for those errors as well as for errors detected in that portion of IHCDIOSE called from IHCFCOMH - i.e., error conditions 231 through 233, and 236 through 237. It puts the data set reference number into the last four bytes of the error message and links to the error monitor.

For error condition 232, the number of the record requested is placed in the parameter list before calling the common subroutine. For error conditions 218 and 237, the DCB address is placed in the parameter list.

Termination Section

The termination section of IHCDIOSE receives control from the load module termination routine of IHCFCOMH. The function of this section is to terminate any pending input/output operations involving direct access data sets. The unit blocks associated with the direct access data sets are examined by IHCDIOSE to determine if any input/output is pending. CHECK macro instructions are issued for all pending input/output operations to ensure their completion.

The data control blocks for the direct access data sets are closed, and the main storage occupied by the unit blocks is freed via the FREEMAIN macro instruction. Control is then returned to the load moduletermination routine of IHCFCOMH to complete the termination process.

IHCIBERH

IHCIBERH, a member of the FORTRAN system library (SYS1.FORTLIB), processes object-time source statement errors. IHCIBERH is entered when an internal sequence number (ISN) cannot be executed because of a source statement error.

The ISN of the invalid source statement is obtained (from information in the calling sequence) and is then converted to decimal form. IHCIBERH then links to IHCFCOMH to implement the writing of the following error message:

```
IHC230I - SOURCE ERROR AT ISN xxxx -
          EXECUTION FAILED AT SUBROUTINE  -
          (xxxx)
```

After the error message is written on the user-designated error output data set, IHCIBERH passes control to the IBEXIT routine of IHCFCOMH to terminate execution.

Chart G7 illustrates the overall logic of IHCIBERH.

IHCDBUG

IHCDBUG performs the object-time operations of the debug facility statements. All linkages from the load module to IHCDBUG are compiler generated.

Items and Buffer

The following items in IHCDBUG are initialized to zero at load time:

DSRN -- the data set reference number

TRACFLAG -- trace flag

IOFLAG -- input/output in progress flag

DATATYPE -- variable type bits

Whenever information is assembled for output, it is placed in a 70-byte area called DBUFFER. The first character of this area is permanently set to blank for single spacing.

Operation

The first portion of IHCDBUG, called by entry name DEBUG#, is a transfer table; this table is referred to by the code generated for the debug facility statements, and branches to the 13 sections of IHCDBUG. These sections are discussed individually.

TRACE ENTRY: If TRACFLAG is off, this routine exits. Otherwise, the characters 'TRACE' are moved to DBUFFER + 1, the subroutine OUTINT converts the statement label to EBCDIC and places it in DBUFFER, and a branch is made to OUTBUFFR.

SUBTRACE ENTRY: The characters 'SUBTRACE' and the name of the program or subprogram are moved to DBUFFER and a branch is made to OUTBUFFR.

SUBTRACE RETURN ENTRY: The characters 'SUBTRACE *RETURN*' are moved to DBUFFER and a branch is made to OUTBUFFR.

UNIT ENTRY: The unit number argument is placed in DSRN and the routine exits.

INIT SCALAR ENTRY: The data type is saved, the location of the scalar is computed, subroutine OUTNAME places the name of the scalar in DBUFFER, and a branch is made to OUTITEM.

INIT ARRAY ELEMENT ENTRY: This routine saves the data type, computes the location of the array element, and (via the subroutine OUTNAME) places the name of the array in DBUFFER. It then computes the element number as follows:

element number = ((element location - first array location) / element size) + 1

and places a left parenthesis, the element number (converted to EBCDIC by subroutine OUTINT), and a right parenthesis in DBUFFER following the array name. A branch is then made to OUTITEM.

INIT FULL ARRAY ENTRY: If IOFLAG is on, the character X'FF' is placed in DBUFFER, followed by the address of the argument list, and a branch is made to OUTBUFFR. Otherwise, a call to the INIT ARRAY ELEMENT entry is constructed, and the routine loops through that call until all elements of the array have been processed.

SUBSCRIPT CHECK ENTRY: The location of the array element is computed; if it is less than or equal to the maximum array location, the routine exits. If the array element location is outside the bounds of the array, the element number is computed and the characters 'SUBCHK' are placed in DBUFFER. The subroutine OUTNAME then places the name of the array in DBUFFER, OUTINT supplies the EBCDIC code for the element number (which is enclosed in parentheses), and a branch is made to OUTBUFFR.

TRACE ON ENTRY: TRACFLAG is turned on (set to nonzero) and the routine exits.

TRACE OFF ENTRY: TRACFLAG is turned off (set to zero) and the routine exits.

DISPLAY ENTRY: If IOFLAG is on, the characters 'DISPLAY DURING INPUT/OUTPUT SKIPPED' are moved to DBUFFER and a branch is made to OUTBUFFR. Otherwise, a calling sequence for the NAMELIST write routine is constructed. If DSRN is equal to zero, the unit number for SYSOUT (in IHCUATBL + 6) is used as the unit passed to the NAMELIST write routine. On return from the NAMELIST write, this routine exits.

START INPUT/OUTPUT ENTRY: The BYTECNT is set to 252 to indicate that the current area is full, the IOFLAG is set to X'80' to indicate that input/output is in progress, the CURBYTLC is set to the address of SAVESTRT (where the location of the first main block will be), and the routine exits. (See the discussion of ALLOCHAR.)

END INPUT/OUTPUT ENTRY: The IOFLAG is saved in TEMPFLAG and IOFLAG is reset to zero so that this section may make debug calls that result in output to a device. If no information was saved during the input/output, this routine exits.

The subroutine FREECHAR is used to extract one character at a time from the save area. If an X'FF' is encountered (indicating the output of a full array), the next three bytes give the address of the call to INIT FULL ARRAY entry. A call to the DEBUG INIT FULL ARRAY entry is then constructed and executed. If X'FF' is not encountered, characters are placed in DBUFFER until an X'15' is found, indicating the end of a line. When this code is found, the subroutine OUTPUT is used to write out the line.

If no main storage or insufficient main storage was available for saving information during the input/output, the characters 'SOME DEBUG OUTPUT MISSING' are placed in DBUFFER after all saved information (if any) has been written out. The subroutine OUTPUT is then used to write out the message, and this routine returns to the caller.

Subroutines

The following subroutines are used by the routines in IHCDBUG.

OUTITEM: First, the characters ' = ' are moved to DBUFFER. Four bytes of data are then moved to a work area on a doubleword boundary to avoid any boundary alignment errors when registers are loaded for logical or integer conversion. A branch on type then takes place. For fixed point, the routine OUTINT converts the value to EBCDIC and places it in DBUFFER. A branch to OUTBUFFR then takes place.

For floating values, subroutine OUTFLOAT places the value in DBUFFER. A branch to OUTBUFFR then takes place.

For complex values, two calls to OUTFLOAT are made -- first with the real part, then with the imaginary part. A left parenthesis is placed in DBUFFER before the first call, a comma after the first call, and a right parenthesis after the second call. A branch to OUTBUFFR then takes place.

For logical values, a T is placed in DBUFFER if the value was nonzero; otherwise, an F is placed in DBUFFER. A branch to OUTBUFFR then takes place.

OUTNAME: This is a closed subroutine. Up to six characters of the name are placed in DBUFFER. However, the first blank in the name causes the routine to exit.

OUTINT: This is a closed subroutine. If the value (passed in R2) is equal to zero, the character '0' is placed in DBUFFER and the routine exits. If it is less than zero, a minus sign is placed in DBUFFER. The value is then converted to EBCDIC and placed in DBUFFER with leading zeros suppressed. The routine then exits.

OUTFLOAT: This subroutine calls the library module IHCFCVTH to put the floating-point number out under G conversion with a format of Gx+7.x, where:

x = 7 for single precision
x = 16 for double precision

OUTBUFFR: If IOFLAG is not set, the routine calls the subroutine OUTPUT and then exits. Otherwise, IOFLAG is set to indicate that debug output during input/output occurred. Then, a call is made to ALLOCHAR for each character in DBUFFER, and finally, a call to ALLOCHAR with X'15' indicating the end of the line. The routine then exits.

ALLOCHAR: This is a closed subroutine. If BYTECNT is equal to 252 bytes, indicating the current block is full, a new block of 256 bytes is obtained by a GETMAIN macro instruction. If no storage was available, an X'07', indicating the end of core storage, is placed in the last available byte position, IOFLAG is set to full, and the routine exits. Otherwise, the address of the new block is placed in the last three bytes of the previous block, preceded by X'37' indicating end of block with new block to follow. CURBYTLC is then set to the address of the new block and BYTECNT is set to zero. The character passed as an argument is then placed in the byte pointed to by CURBYTLC, one is added to both CURBYTLC and BYTECNT, and the routine exits.

FREECHAR: This is a closed subroutine. If the current character extracted is X'37', the next three bytes are placed in CURBYTLC and the current block is freed. If the current character is X'07' the block is freed and a branch is made to the end input/output exit. Otherwise, the current character is passed to the calling routine and CURBYTLC is incremented by one.

OUTPUT: This is a closed subroutine. If DSRN is zero, the SYSOUT unit number is obtained from IHCUATBL + 6. A call is then made to FIOCS# output initialize, DBUFFER is transferred to the FIOCS# buffer, and a call is made to FIOCS# output. The routine then exits.

IHCTRCH

IHCTRCH, a member of the FORTRAN system library (SYS1.FORTLIB), processes terminal errors detected by FORTRAN library subroutines at object time. IHCTRCH is entered when an error is detected in order to print a traceback map. After this is accomplished, the job is terminated unless the extended error message facility has been requested.

IHCTRCH issues the following message:

```
IHCxxxI
TRACEBACK FOLLOWS ROUTINE ISN REG.  14
REG.  15 REG.  0 REG.  1
```

where xxx is the error code (in decimal form) that it obtains from the calling sequence.

If the error occurred in IHCFCOMH, IHCFCVTH, IHCNAMEL, IHCDIOSE, or IHCFIOSH, IHCTRCH sets up an area which can be processed as a standard save area for the first traceback line.

For each traceback line, IHCTRCH gets the name of the called routine, the internal statement number, if any, of the call within the calling routine, and the contents of registers 14, 15, 0, and 1 in hexadecimal.

After printing each line, IHCTRCH checks to determine whether or not the called routine was the main FORTRAN routine. If it was, the entry point is printed in hexadecimal and a branch is made to IBEXIT. If it was not, a traceback loop-check routine is entered, which builds and checks a table of save area addresses. If the table is full or if a loop is detected, IHCTRCH prints TRACEBACK TERMINATED and then prints the main FORTRAN routine entry point and branches to IBEXIT.

IHCTRCH uses IHCFCVTH to convert to printable hexadecimal format, and it uses IHCFIOSH for printing.

Further information about traceback, including an example of output, is contained in the publication IBM System/360 Operating System, FORTRAN IV (G) Programmer's Guide, Form C28-6639.

IHCFINTH

The module IHCFINTH processes asynchronous program interrupts. Every FORTRAN main program notifies the system's first level interrupt handler (via a SPIE macro instruction) to transfer to the entry point ARITH# in module IHCFINTH in the event of a program interrupt.

FORTRAN requests interrupt service for the program interrupts listed below. All others cause job termination by the system. (For a description of program interrupts, see the publication IBM System/360: Principles of Operation, Form A22-6821.)

Code	Description
9	Fixed-point divide
11	Decimal divide
12	Exponent overflow
13	Exponent underflow
15	Floating-point divide

Codes 8 and 14 are masked so that no interrupt occurs.

If boundary alignment adjustments were requested when the system was created, then interrupt 6 specification is also requested. The processing for specification interrupts is handled by the module IHCADJST, however.

The services performed by the interrupt processing routine IHCFINTH are as follows:

1. A message is printed that identifies the interrupt.

2. Switches are set for exponent overflow, exponent underflow and, divide check for the FORTRAN subprograms CALL OVERFL(J) and CALL DVCHK(J).

3. Result registers are altered for exponent overflow and underflow as follows:

   Overflow -- maximum floating-point number.

   Underflow -- zero.

   In addition, if the operation was an add or subtract and exponent underflow occurred, then the condition code is set to 0.

When the extended error message facility has been requested, then the module IHCFINTH has the ability to accept a user exit and to control the printing of messages and the number of occurrences of the various interrupts. The user exit may provide an alternate value to be placed in the result register for underflow and overflow before execution continues.

IHCERRM

IHCERRM is the execution error monitor. Each FORTRAN library program that detects an error calls the IHCERRM module for error message service. The service available is dependent upon which of two options, basic message facility or extended error message facility, was selected at system generation.

When the basic message facility is requested, each error causes job termination and a traceback map is produced. The messages printed on the object error unit will contain a description of the error situation if the error was detected by the mathematical library. For other error situations, only an error code is printed. For a full description of these error codes, see the FORTRAN (G) Programmer's Guide publication.

When the extended error message facility is present, the error monitor is directed by the option table to perform one or more of the following actions:

- Print a message
- Terminate the job
- Call a user-written routine for corrective action. Upon return from the user-written routine, the return is made to the caller of the error monitor.
- Return to the caller of the error monitor an indication that standard corrective action is required. The routine that called the error monitor has the programming to provide the standard corrective action.

To enable dynamic control of error occurrences and printing suppression, routines can be called from the FORTRAN source language.

Because error message printing can be suppressed, a summary of error occurrences is given before return is made to the system.

The FORTRAN library provides the error message facility through the following services:

1. Each module that detects an error calls the error monitor. The module can accept a return from the error monitor and supply a standard corrective action.

2. An error monitor is supplied.

3. Routines are supplied to change the option table.

4. An option table is supplied.

5. The exit code of the FORTRAN library provides for the printing of an error summary.

The following is a description of the error monitor:

On initial entry, the error monitor will set a switch. If entered again before the switch is set to off, a recursive situation is detected and the job is terminated.

The error monitor then retrieves the error entry from the option table and makes the following actions and tests them in the order listed:

1. Updates the current count of errors encountered.

2. Does the current count of errors exceed the number of allowable errors, indicating that the job should be terminated?

3. Does the current count of messages printed exceed the number of messages to be printed, indicating that message printing is to be suppressed?

4. Should a traceback map be printed?

5. Is a user exit specified? If it is, the exit routine, which must return to the error monitor, is called.

6. Return to the caller of the error monitor after turning off the switch that indicates that the error monitor has been entered.

Charts G8 and G9 show the overall logic of the error monitor.

Alter Option Table Routine (IHCFOPT)

IHCFOPT, the routine that allows the user to alter the option table thereby achieving dynamic control over error occurrence, has three entry points: ERRSTR, ERRSAV, and ERRSET.

The option table consists of an entry for each error number and a preface of eight bytes. An option table entry for an error number is described in Table 14.

If the extended error message facility has not been specified at system generation time, the option table is reduced to the preface alone. The option table preface is described in Table 15.

• Table 14. Description of Option Table Entry

Field	Length	Default Settings[1]	Description
1	1 byte	10[2]	Contains a count. When the count in this field matches field 3, the job is terminated. The maximum count is 255. A count of zero means unlimited number of occurrences.[3] Any count greater than 255 supplied by ERRSET will set this field to zero.
2	1 byte	5[4]	A count of the number of messages to be printed; message printing is suppressed after the count is exceeded. A count of zero means no messages are to be printed.
3	1 byte	0	Count of number of errors that have occurred, where 0 means no errors have occurred.
4	1 byte		Eight option bits defined as follows:
	bit 0	0	Control character indicator 0 = none, 1 = single space
	1	1	Table entry modifiable 0 = no, 1 = yes (See Note 5)
	2	0	Extension of count of errors that have occurred
	3	(See Note 6)	Buffer contents to be printed 0 = no, 1 = yes
	4	0	Unused (reserved)
	5	0	Unlimited number of messages allowed 0 = no, 1 = yes
	6	1	Traceback required 0 = no, 1 = yes
	7	0	Unused (reserved)
5	4 bytes	1	Address of user's exit routine. If the value of entry is odd, standard corrective action is indicated.

[1]The default values shown apply to all error numbers unless excepted by a footnote.
[2]Errors 208, 210, and 215 are set as unlimited, and errors 217 and 230 are set to 1.
[3]When the user sets the count of allowed errors as unlimited, the FORTRAN job may loop endlessly unless the operator intervenes.
[4]Error 210 is set to 10, and errors 217 and 230 are set to 1.
[5]The entry for error 230 is not modifiable.
[6]This entry is set to 0 except for error numbers 212, 215, 218, 221, 222, 223, 224, and 225.

• Table 15. Description of Option Table Preface

Field	Length (in bytes)	Default	Description
1	4	95	Contains the count of the number of entries in the option table.
2	1	1=bit 1	Boundary alignment switch 1 = ALIGN 0 = NOALIGN Bit 1 of this byte contains the switch.
3	1	0	Error message handling selected FF = no, 00 = yes For no error message facility, the default will be FF.
4	1	0	For no error message facility, boundary align count is kept here. Default is then 10.
5	1	0	Not used (reserved).

To obtain an entry from the option table, the source program calls IHCFOPT through its entry name ERRSAV. When the requested entry is located in the option table, it is placed in the address passed in the call to ERRSAV. If the requested entry is not in the option table, a message is printed.

To store an entry in the option table, the source program calls IHCFOPT through its entry name ERRSTR. If the requested entry exists in the option table, it is checked to see whether or not that entry can be modified. If it can be modified, the entry passed to ERRSAV is placed in the option table to replace the previous entry. If the existing entry is unmodifiable, a message stating this is printed.

To change individual fields in the option table, the source program calls IHCFOPT through its entry name ERRSET. If the requested entry exists in the option table, each field of the entry for which an alteration is requested is checked to see whether or not it contains a value of zero. (The IRANGE field error IHC215I is an exception.) If it does, that field will not be altered. If it does not, the field is replaced with the new field passed in the call to ERRSET. As parameters are processed, a check is made for an early end to the parameter list.

Charts G10, G11, and G12 show the overall logic of the routine to alter the option table.

• Chart G0. IHCFCOMH OVERALL LOGIC AND UTILITY ROUTINES

```
       ****A3*********
       *    LOAD     *     THE LOAD MODULE ENTERS
       *   MODULE    *     IHCFCOMH VIA A COMPILER-
       *             *     GENERATED CALLING SEQUENCE.
       ***************
              |
              v
      *****B3**********
      *               *
      *   DETERMINE   *
      * REQUEST TYPE  *
      *               *
      *               *
      *****************
              |
              v
```

REQUEST TYPE	CHART	MAJOR PROCESSING ROUTINES WITHIN IHCFCOMH/IHCNAMEL	EXTERNAL SUBROUTINES CALLED
SEQUENTIAL AND DIRECT ACCESS READ/WRITE REQUIRING A FORMAT	G1A2	FRDWF, FWRWF, FIOLF, FIOAF, FENDF	IHCFIOSH (SEQUENTIAL), OR IHCDIOSE (DIRECT ACCESS), ALSO FOLLOWING ROUTINES WITHIN IHCFCVTH - FCVII, FCVIO, FCVEI, FCVEO, FCVDI, FCVDO, FVCLI, FCVLO, FCVZI, FCVZO, FCVFI, FCVFO, FCVAI, FCVAO, FCVGI, FCVGO
SEQUENTIAL AND DIRECT ACCESS READ/WRITE NOT REQUIRING A FORMAT	G1F2	FRDNF, FWRNF, FIOLN, FIOAN, FENDN	IHCFIOSH (SEQUENTIAL) IHCFIOSE (DIRECT ACCESS)
READ USING NAMELIST	G2E1	FRNDL	IHCFIOSH, FCVEI, FCVDI, FCVAI, FCVLI, FCVGI, FCVCI, FCVFI, FCVII
WRITE USING NAMELIST	G2E5	FWDNL	FCVGO, FCVCO, FCVIO, FCVFO
DEVICE MANIPULATION	G2B3	FBKSP, FRWND, FEOFM	IHCFIOSH
WRITE TO OPERATOR	G2G3	FSTOP, FPAUS	NONE

```
IBEXIT              ERRMON              EXCEPT              IBFINT              FERROR

 ****F1*********     ****F2*********     ****F3*********     ****F4*********     ****F5*********
 * FROM FSTOP OR *   *  FROM MODULE  *   *     FROM      *   *     FROM      *   *     FROM      *
 *    IBFERR     *   *DETECTING ERROR*   * IHCFIOSH OR   *   * LOAD MODULE   *   *   IHCFCOMH    *
 *               *   *               *   *   IHCDIOSE    *   *               *   *               *
 ***************     ***************     ***************     ***************     ***************
        |                   |                   |                   |                   |
        v                   v                   v                   v                   v
*****G1**********   *****G2**********   *****G3**********   *****G4**********   *****G5**********
*    IBEXIT     *   *    IHCERRM    *   * EXCEPT/FERROR *   *    IBFINT     *   *    FERROR     *
*-*-*-*-*-*-*-*-*   *-*-*-*-*-*-*-*-*   *-*-*-*-*-*-*-*-*   *-*-*-*-*-*-*-*-*   *-*-*-*-*-*-*-*-*
*  CLOSE DATA   *   *     CALL      *   *   DETERMINE   *   *    PROCESS    *   * DETERMINE IF  *
*SETS (TERMINATE*   * ERROR MONITOR *   * IF PARAMETER  *   *  ARITHMETIC   *   * ERR PARAMETER *
*  EXECUTION)   *   *               *   *   SPECIFIED   *   * INTERRUPTION  *   *   SPECIFIED   *
*****************   *****************   *****************   *****************   *****************
        |                   |                   |                   |                   |
        v                   v                   v                   v                   v
 ****H1*********     ****H2*********     ****H3*********     ****H4*********     ****H5*********
 *      TO       *   *      TO       *   *      TO       *   *      TO       *   *      TO       *
 *   OPERATING   *   *    IBEXIT     *   * LOAD MODULE   *   * LOAD MODULE   *   * LOAD MODULE   *
 *    SYSTEM     *   *               *   * IF SPECIFIED  *   *               *   * IF SPECIFIED  *
 ***************     ***************     ***************     ***************     ***************
                     IF EXECUTION        IF PARAMETER NOT                        IF PARAMETER NOT
                     IS TO CONTINUE,     SPECIFIED, EXIT IS                      SPECIFIED, EXIT IS
                     RETURN TO CALLER    TO ERRMON                               TO ERRMON
```

Chart G1. IMPLEMENTATION OF READ/WRITE/FIND

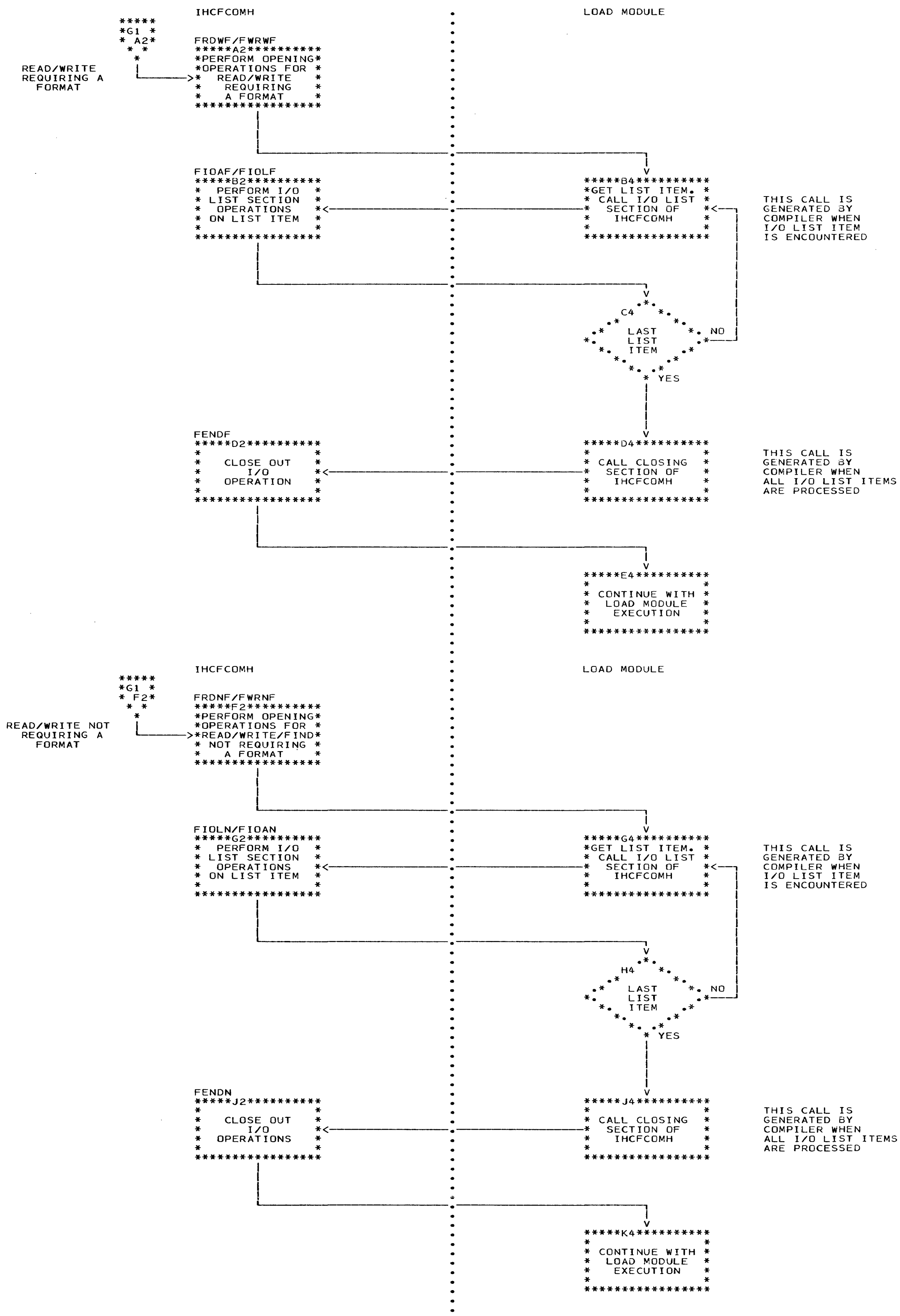


Chart G2. DEVICE MANIPULATION AND WRITE-TO-OPERATOR ROUTINES

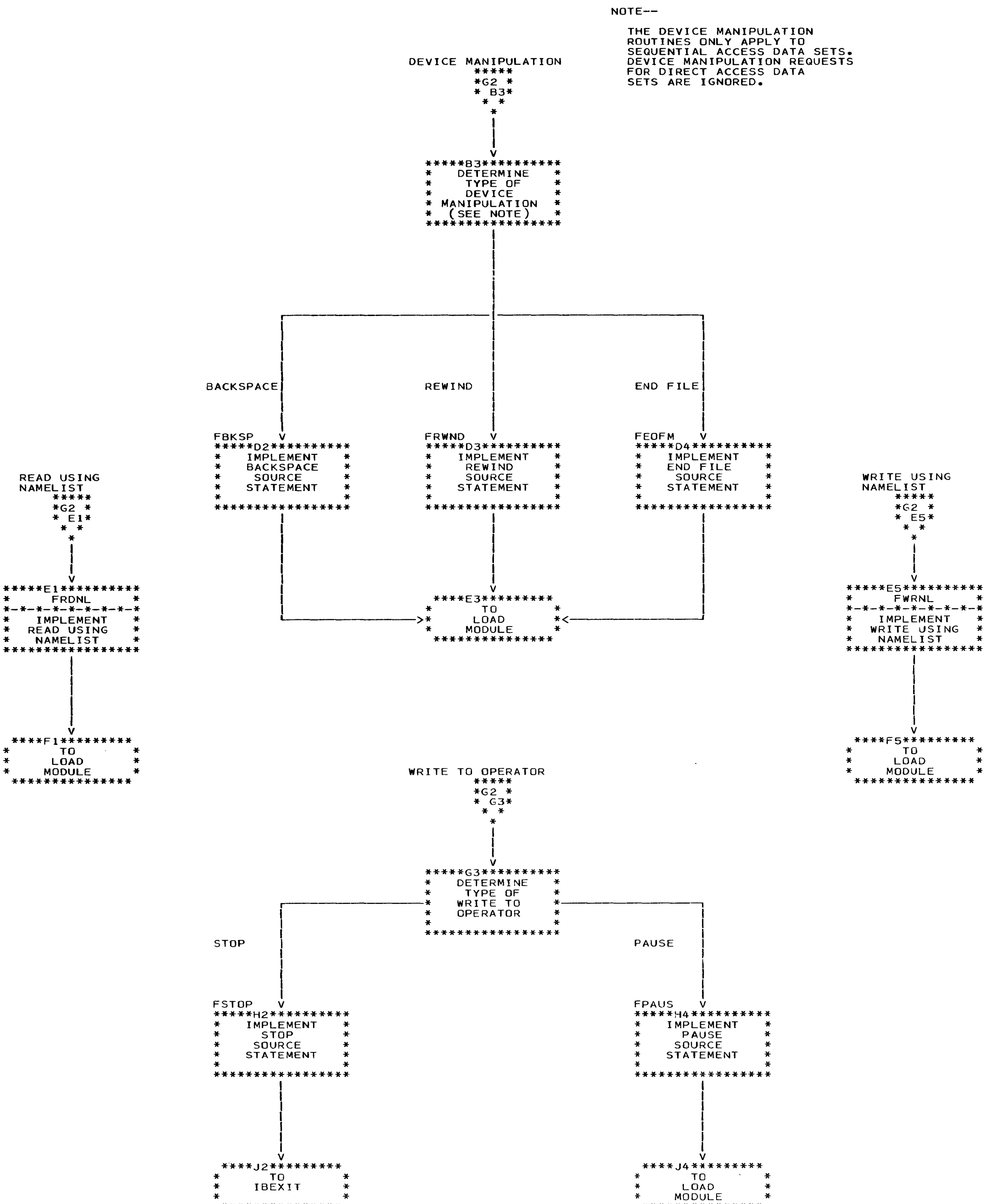


Table 16. IHCFCOMH Subroutine Directory

Subroutine	Function
EXCEPT	Checks for presence of END= parameter, and passes control to the load module if present.
FENDF	Closing section for a READ or WRITE requiring a format.
FENDN	Closing section for a READ or WRITE not requiring a format.
FEOFM	Implements the END FILE source statement.
FERROR	Checks for the presence of the ERR= parameter, and passes control to the load module if present.
FIOAF	I/O list section for list array of a READ or WRITE requiring a format.
FIOAN	I/O list section for list array of a READ or WRITE not requiring a format.
FIOLF	I/O list section for a list variable of a READ or WRITE requiring a format.
FIOLN	I/O list section for a list variable of a READ or WRITE not requiring a format.
FPAUS	Implements the PAUSE source statement.
FRDNF	Opening section of a READ not requiring a format.
FRDWF	Opening section of a READ requiring a format.
FRWND	Implements the REWIND source statement.
FSTOP	Implements the STOP source statement.
FWRNF	Opening section for WRITE not requiring a format.
FWRWF	Opening section for WRITE requiring a format.
IBEXIT	Closes all data sets and terminates execution.
IBFERR	Calls IHCTRCH to process terminal object-time errors.
IBFINT	Processes program interruptions.
FBKSP	Implements the BACKSPACE source statement.

Table 17. IHCFCVTH Subroutine Directory

Subroutine	Function
FCVAI	Reads alphameric data.
FCVAO	Writes alphameric data.
FCVCI	Reads complex data.
FCVCO	Writes complex data.
FCVDI	Reads double-precision data with an external exponent.
FCVDO	Writes double-precision data with an external exponent.
FCVEI	Reads real data with an external exponent.
FCVEO	Writes real data with an external exponent.
FCVFI	Reads real data without an external exponent.
FCVFO	Writes real data without an external exponent.
FCVGI	Reads general type data.
FCVGO	Writes general type data.
FCVII	Reads integer data.
FCVIO	Writes integer data.
FCVLI	Reads logical data.
FCVLO	Writes logical data.
FCVZI	Reads hexadecimal data.
FCVZO	Writes hexadecimal data.

Chart G3. IHCFIOSH OVERALL LOGIC

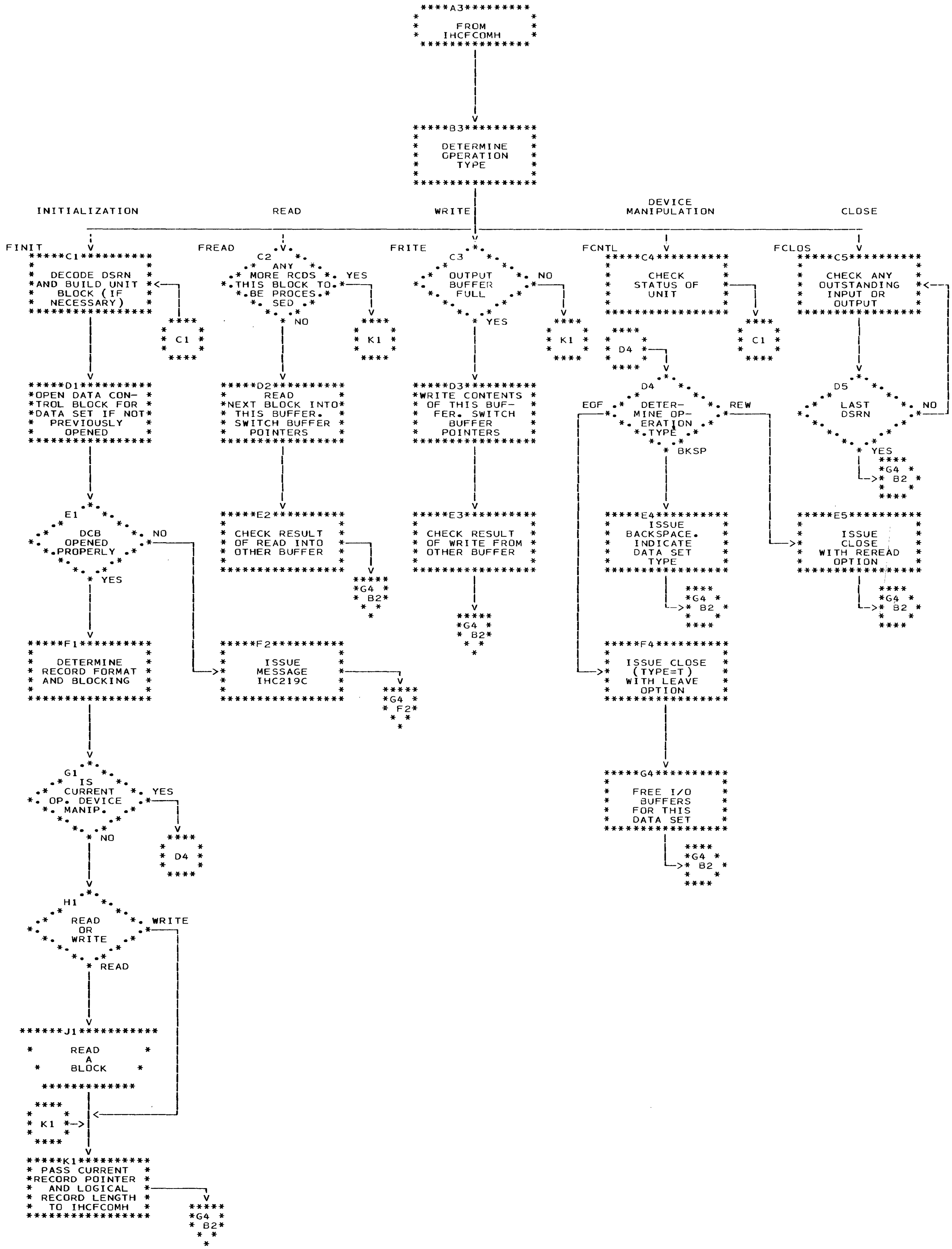


Chart G4. **EXECUTION-TIME INPUT/OUTPUT RECOVERY PROCEDURE**

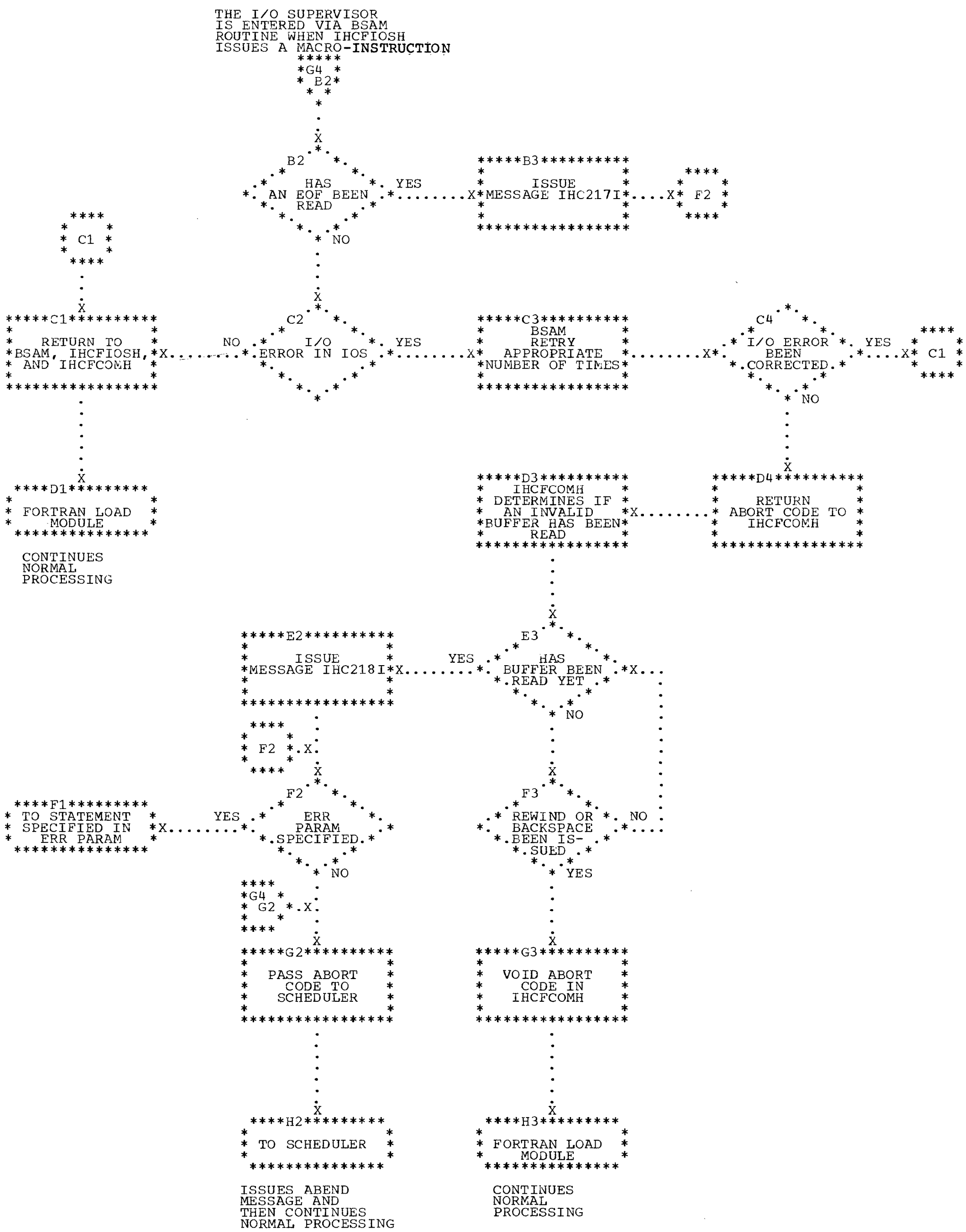


Chart G5. IHCDIOSE OVERALL LOGIC - FILE DEFINITION SECTION

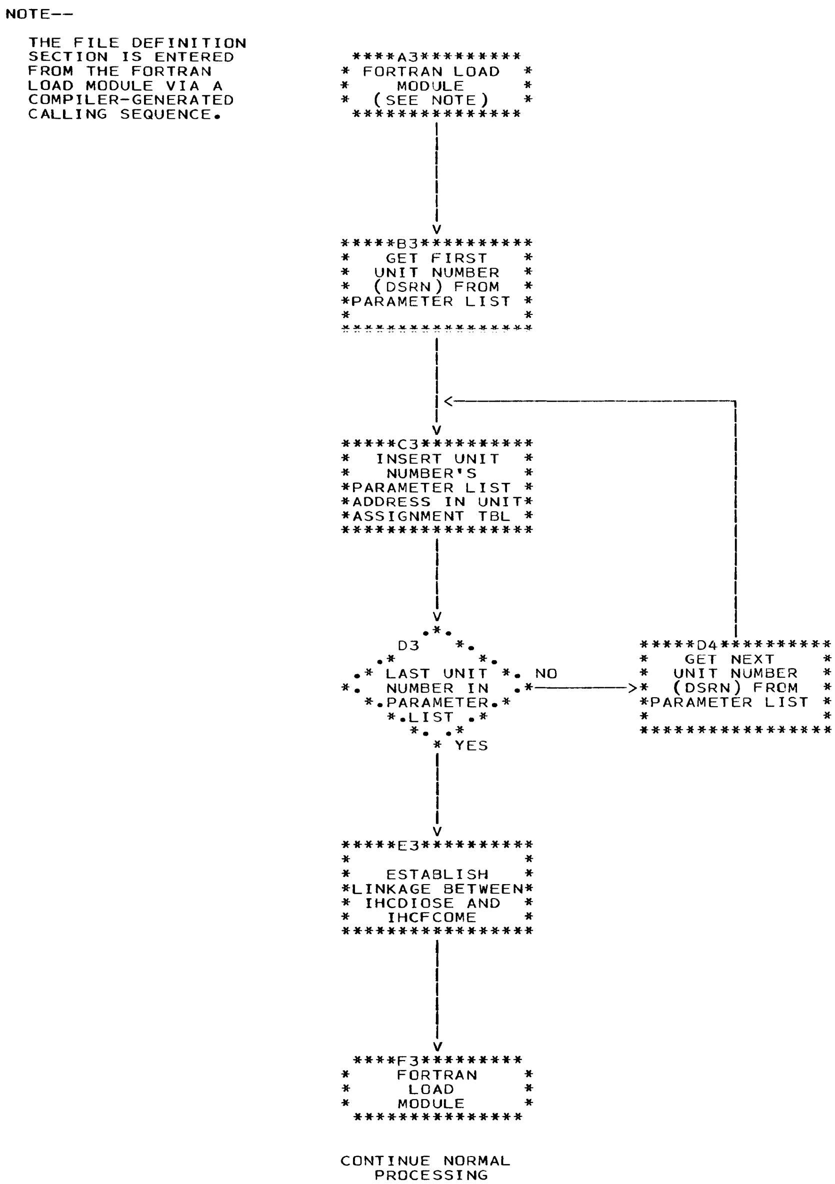


Chart G6. IHCDIOSE OVERALL LOGIC - FILE INITIALIZATION, READ, WRITE, AND TERMINATION SECTIONS

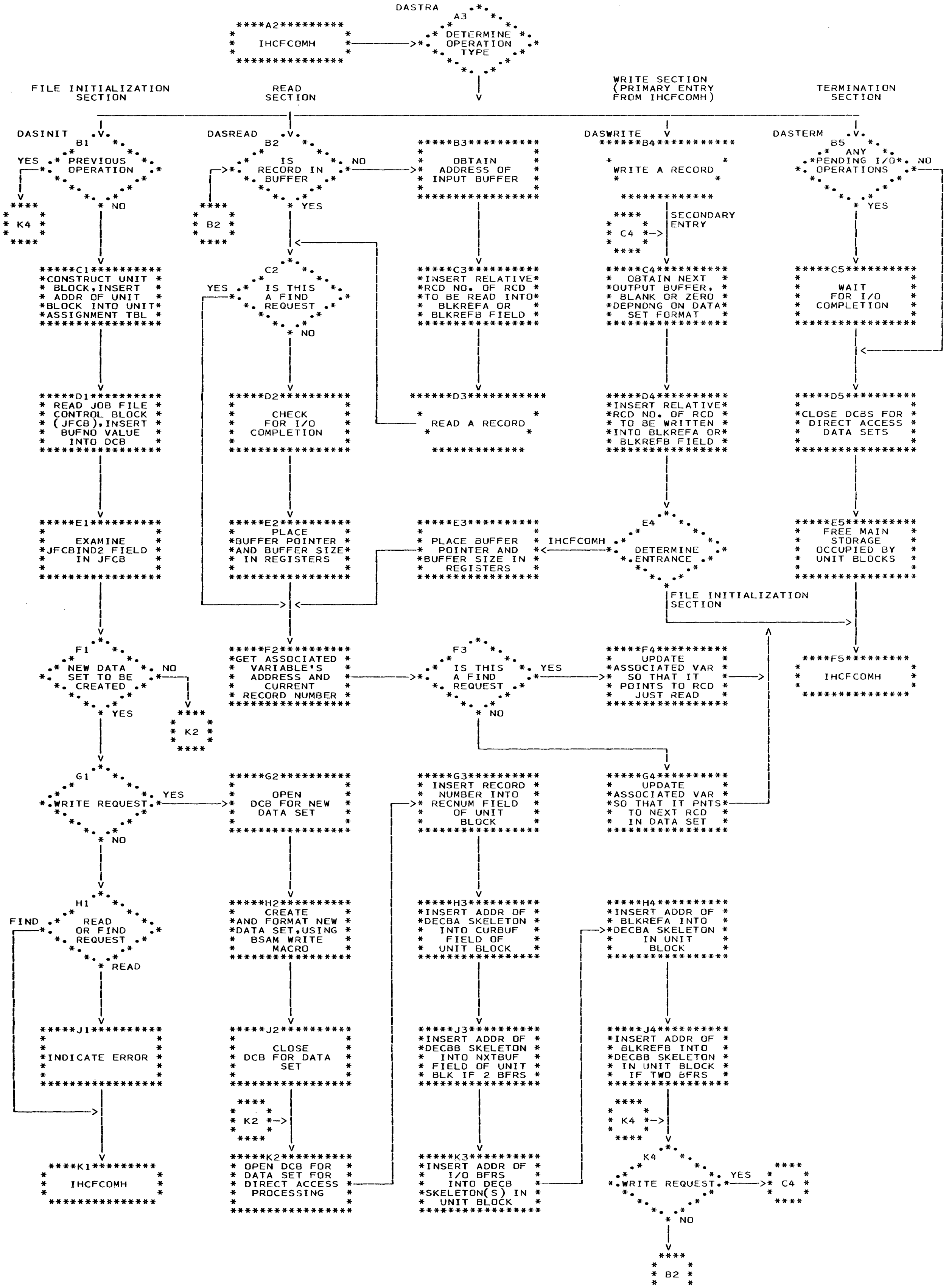


Table 18. IHCFIOSH Routine Directory

Routine	Function
FCLOS	CHECKS double-buffered output data sets.
FCNTL	Services device manipulation requests.
FINIT	Initializes unit and data set.
FREAD	Services read requests.
FRITE	Services write requests.

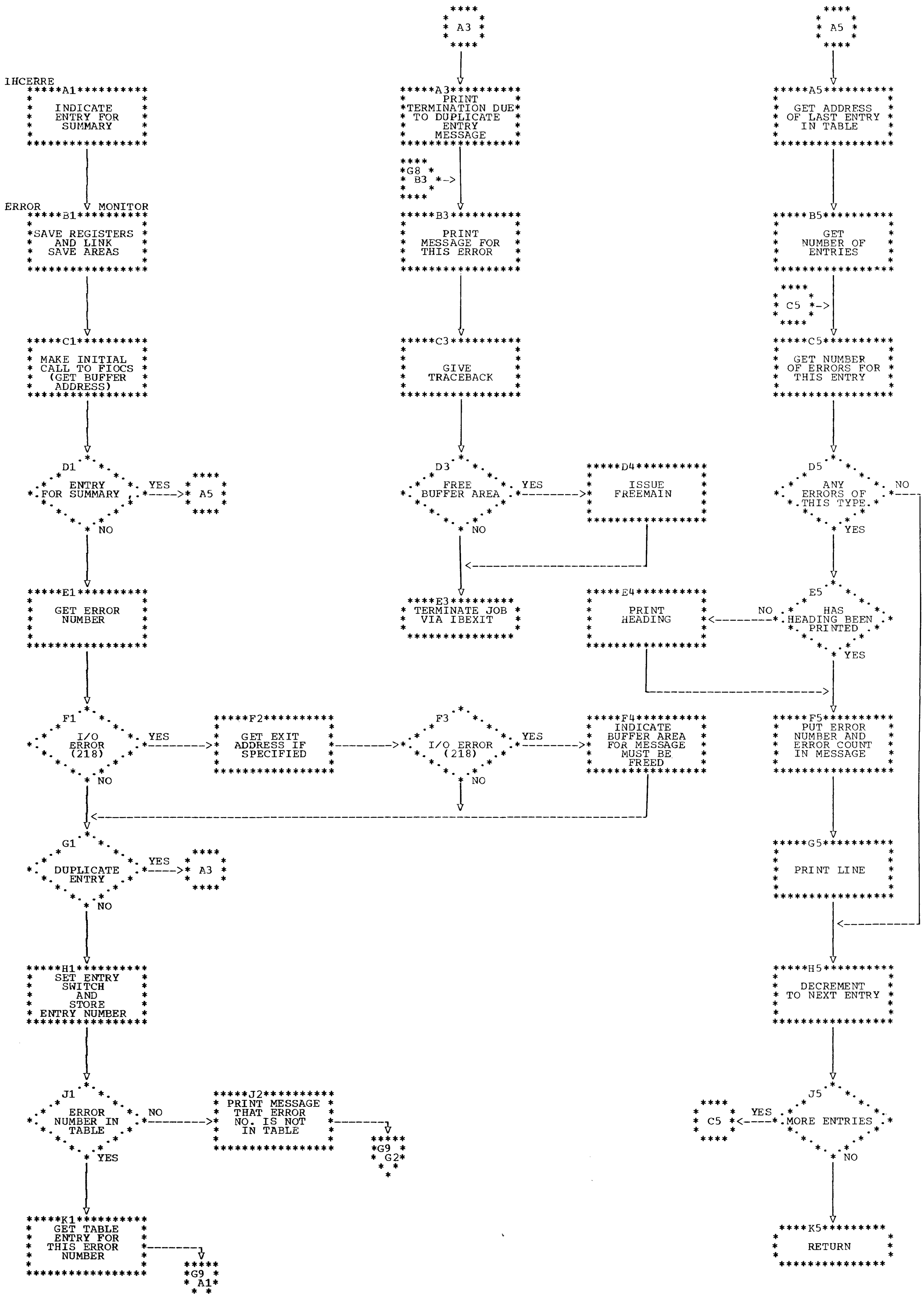
Table 19. IHCDIOSE Routine Directory

Routine	Function
DASDEF	Processes DEFINE FILE statements: enters address of parameter lists into unit assignment table, checks for redefinition of direct access unit numbers, and establishes addressability for IHCDIOSE within IHCFCOMH.
DASINIT	Constructs unit blocks for non-opened direct access data sets, creates and formats new direct access data sets, and opens data control blocks for direct access data sets.
DASREAD	Reads physical records, passes buffer pointers and buffer size to IHCFCOMH, and updates the associated variable.
DASTERM	Checks pending input/output operations, closes direct access data sets, and frees main storage occupied by unit blocks.
DASTRA	Determines operation type and transfers control to appropriate routine.
DASWRITE	Writes physical records, provides IHCFCOMH with buffer space, and updates the associated variable.

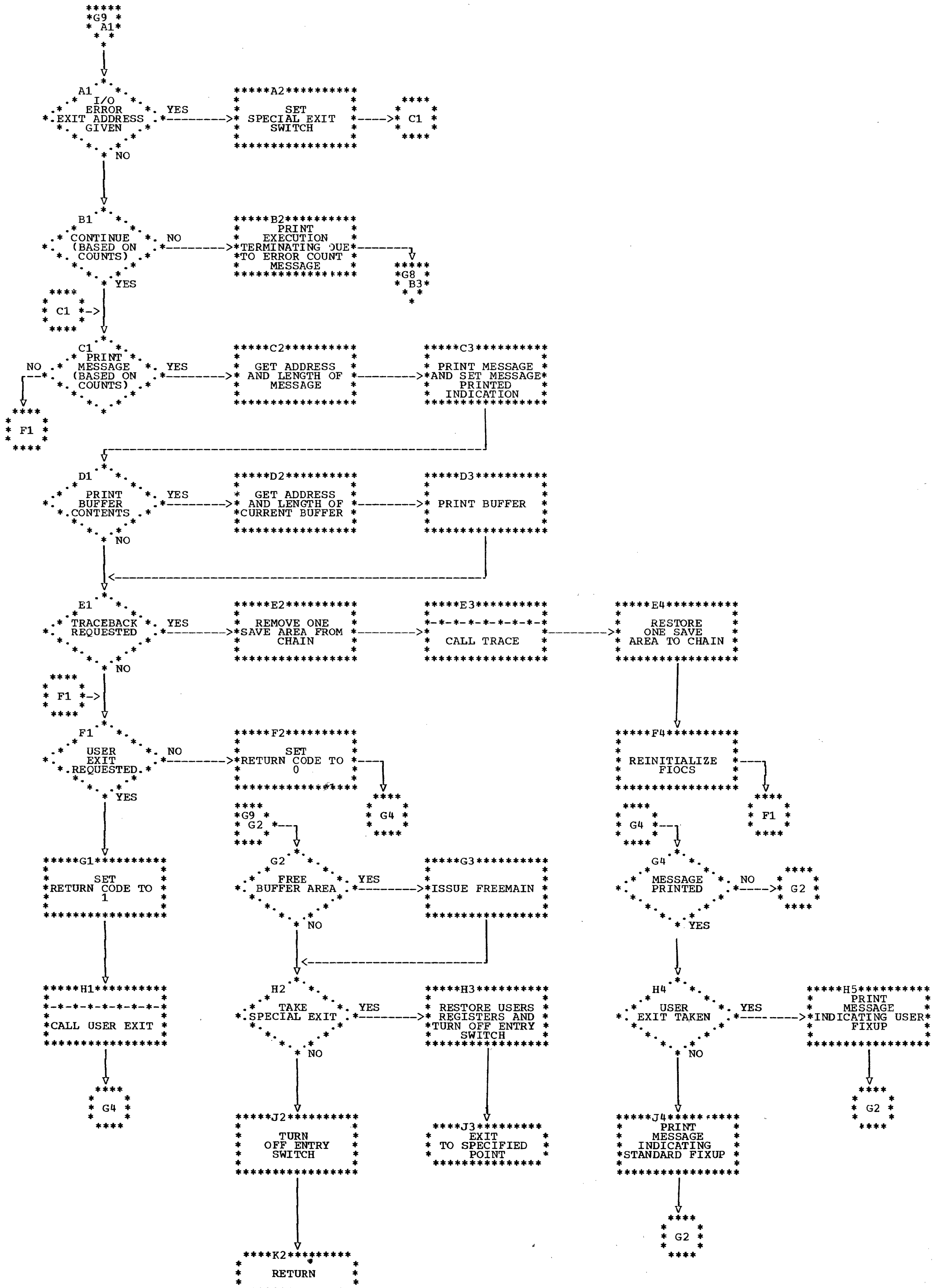
Chart G7. IHCIBERH OVERALL LOGIC

```
                              IHCIBERH IS
                              ENTERED VIA
                              CALLING SE-
                              QUENCES GEN-
      ****A3*********         ERATED AT
     *    FORTRAN    *        COMPILE-TIME.
     *     LOAD      *
     *    MODULE     *
      ***************
              |
              |
              |
              |
              |
              V
     *****B3**********
     *               *
     *OBTAIN INTERNAL*
     *SEQUENCE NUMBER*
     *     (ISN)     *
     *               *
     *****************
              |
              |
              |
              |
              |
              V
     *****C3**********
     *               *
     *  CONVERT ISN  *
     *  TO DECIMAL   *
     *    FORMAT     *
     *               *
     *****************
              |
              |
              |
              |
              |
              V
     *****D3**********
     *   BRANCH TO   *
     *  IHCFCOMH TO  *
     *  HANDLE THE   *
     *  WRITING OF   *
     * ERROR MESSAGE *
     *****************
              |
              |
              |
              |
              |
              V
      ****E3*********
     *  IBEXIT RTN   *
     *      OF       *
     *   IHCFCOMH    *
      ***************
```

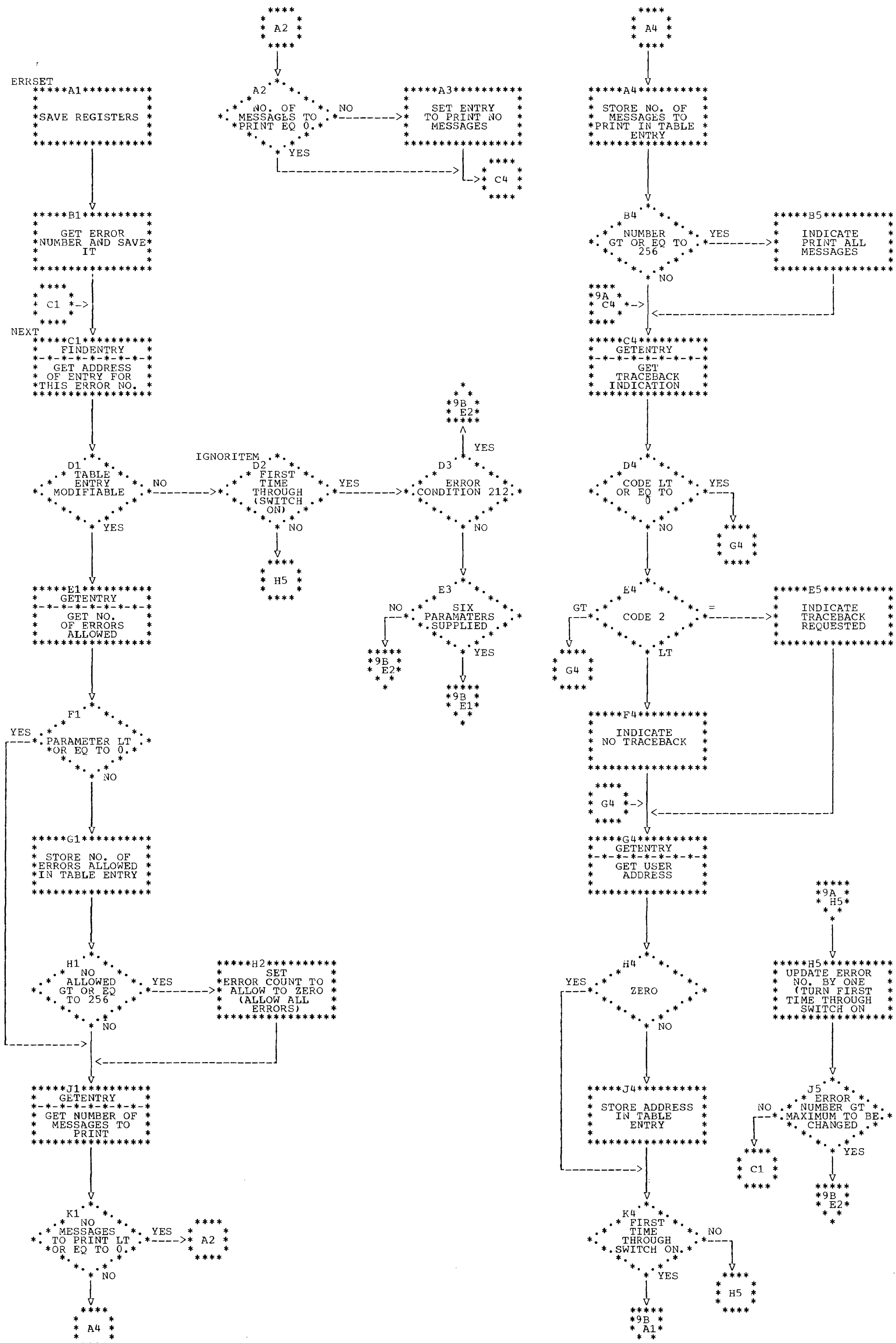
• Chart G8. ERROR MONITOR OVERALL LOGIC (Part 1 of 2)



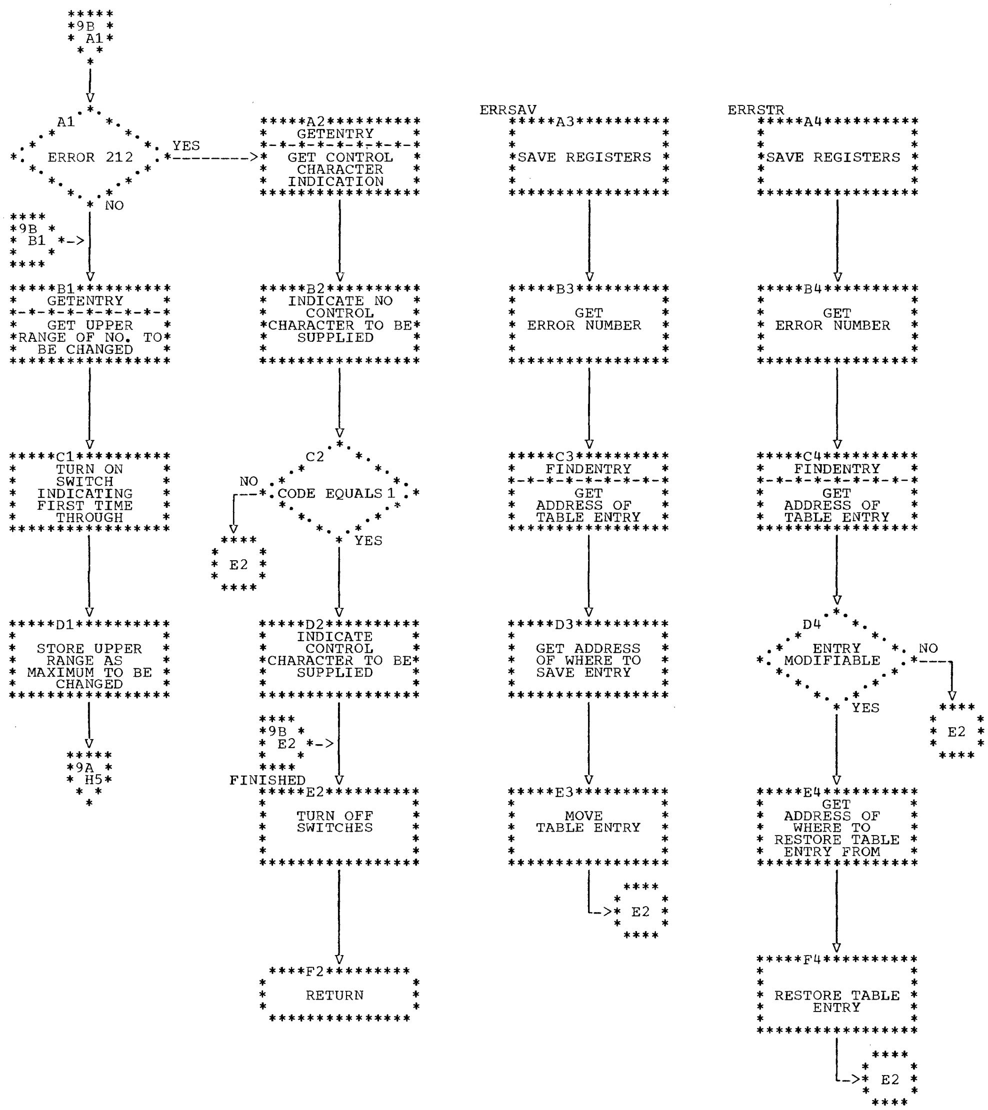
• Chart G9. ERROR MONITOR OVERALL LOGIC (Part 2 of 2)



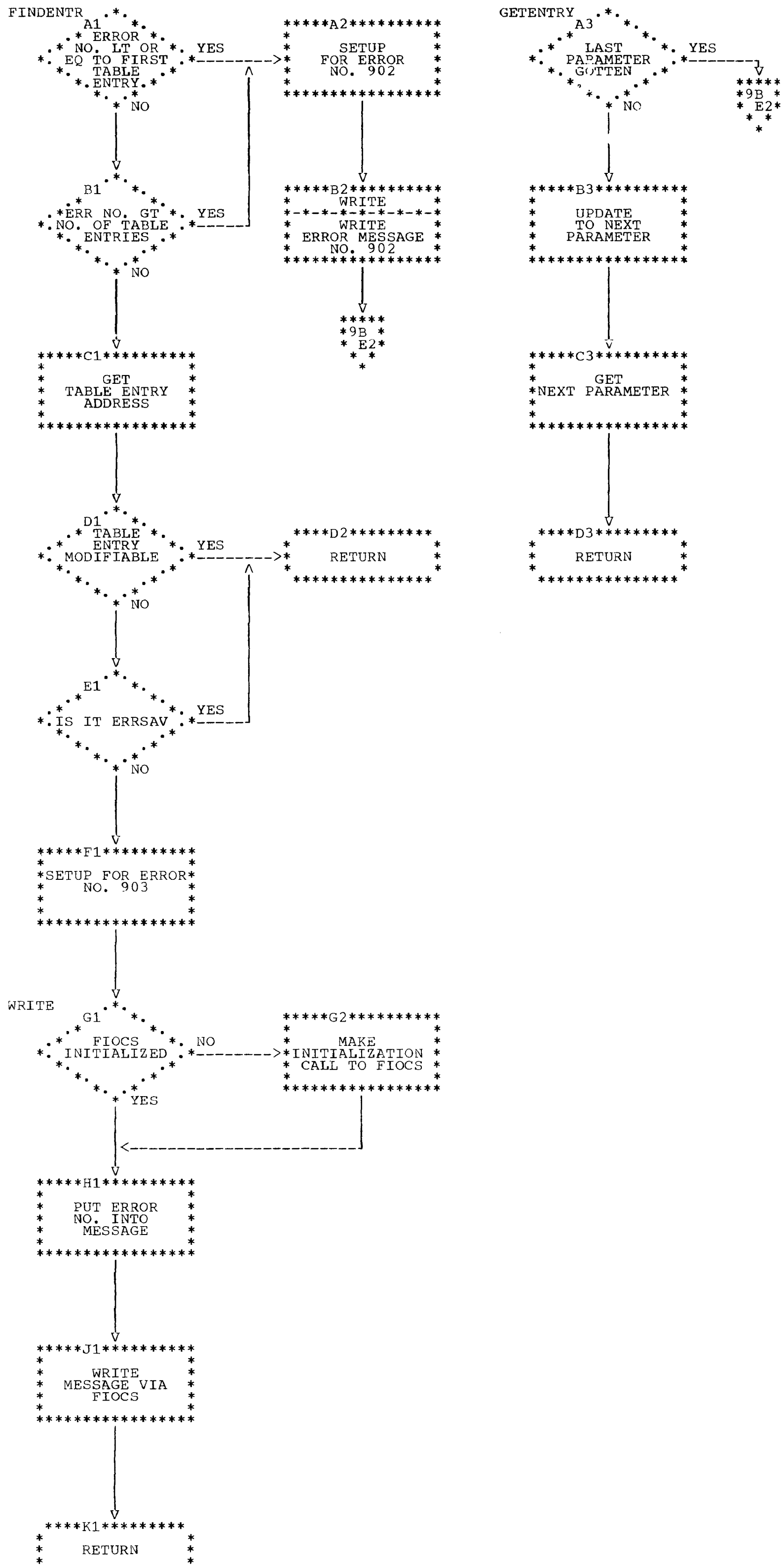
• Chart G10. ALTER OPTION TABLE ROUTINE OVERALL LOGIC (Part 1 of 3)



• Chart G11. ALTER OPTION TABLE ROUTINE OVERALL LOGIC (Part 2 of 3)



• Chart G12. ALTER OPTION TABLE ROUTINE OVERALL LOGIC (Part 3 of 3)



active character: A significant character in the interpretation of a source statement. Always non-blank except during parsing of literal or IBM card code information.

ADDR: Contains the address portion of the current POP instruction.

ADDRESS (field): A 2-byte item that is part of the pointer (indicating an address on a roll) and a driver (indicating the forcing strength of an operation).

ANSWER BOX: An item used to hold a true or false answer for those POP instructions which use or return an answer in their execution.

BASE: A status variable maintained for each roll used by the compiler which contains the beginning address of that roll minus 4.

Base Table: A list of absolute addresses from which the object module loads a general register prior to accessing data.

BOTTOM: A status variable maintained for each roll which holds the address of the last word on the roll containing information.

Branch Table: A list containing the address of each branch target label and statement function used in the source module.

branch target label: A label which is the target of a branch instruction or statement.

Central Items: Another name for SYMBOL 1-3 and DATA 0-5.

compiler phase: A program consisting of several routines written in machine language and/or POP language; each phase performs a well-defined function in the transformation of the source module to the object module.

compiler routines: The routines that comprise each phase of the compiler and which may be written in machine language and/or POP language.

CONSTR: Contains the beginning address of the data referred to by the compiler routines.

control driver: A driver in Polish notation to indicate types of statements and other control functions.

CRRNT CHAR: Contains the character (from the input statement) that is currently being inspected.

CRRNT CHAR CNT: Contains the column number of the contents of CRRNT CHAR; also called the 'scan arrow'.

DATA 0, 1, 2, 3, 4, 5: Halfword variables (except DATA 5, which is two words long) used to hold constants used in the source module and other data.

error listing: The display of messages indicating error conditions detected in the processing of the source module.

EXIT roll: A special roll used by the compiler for maintaining exit addresses from compiler routines when a POP subroutine jump instruction is executed.

EXTADR: Contains the address of the current "bottom" of the EXIT roll.

forcing strength: A value contained in the driver which indicates the order of the indicated operation (e.g., multiplication and division operations precede addition and subtraction).

global dummy variable: A dummy argument to a SUBROUTINE or FUNCTION subprogram.

global label: A label used to define a program block. These labels may be referred to from any point in the program.

group: The logical collection of information maintained on rolls; an entry on a roll.

group size: The number of bytes of information constituting the group on a roll.

Group Stats: Information maintained for each roll used by the compiler; pertains to comparative search operations.

heading: Initializing instructions required prior to the execution of the body of the object module.

IEYALL: The system name for the compiler phase Allocate.

IEYEXT: The system name for the compiler phase Exit.

IEYFORT: The system name for the compiler Invocation phase.

IEYGEN: The system name for the compiler phase Gen.

IEYPAR: The system name for the compiler phase Parse.

IEYROL: The system name for that area of the compiler which holds the WORK and EXIT rolls and the roll controls and group stats.

IEYUNF: The system name for the compiler phase Unify.

indirect addressing: A method of obtaining information held at one location by referring to another location which contains the address of the value in question.

INDIRECT BOX: Used to contain the address needed in the indirect addressing operation performed by the POP instructions.

INSTR: Contains the "operation code" portion of the current POP instruction.

item: Synonymous with variable.

jump: Synonymous with branch.

keep: Indicates the moving of information contained on a roll to another storage location and retaining the original information on the roll.

LAST CHAR CNT: This item contains the column number of the last active character, i.e., the active character preceding the one currently being inspected.

local dummy variable: A dummy argument to a statement function.

local label: A label defined within a program block which may be referred to only within that block.

MPAC 1, MPAC 2: Two fullword items used by the compiler in double-precision arithmetic operations.

NAMELIST Table: A table which holds the name, address, etc., for each variable listed in a single NAMELIST list in the source module.

operation driver: A 1-word variable which is an element of Polish notation and indicates arithmetic and logical operations designated in source module statements.

OPERATOR (field): A 1-byte item that is part of the pointer and driver indicating the roll used (pointer) or type of operation to be performed (driver).

optimization: The reduction and reorganization of object code for the increased efficiency of the object module.

PGB2: Contains the beginning address of the global jump table.

plex: A variable length group on a roll; the first word holds the number of words exclusive of itself.

pointer: This item is one element of Polish notation used to indicate references to variables or constants; indicates location of additional information on a roll.

Polish notation: An intermediate language into which the source module is translated during processing and generation of the object module.

POPADR: Holds the address of the POP instruction presently being executed.

POP instruction: A component part of the POP language defined as a macro.

POP interpreter: A program written in machine language for the purpose of executing the POP subroutines; labeled POP SETUP.

POP jump table: A table used by the POP interpreter in transferring control to the POP subroutines. Holds addresses of these routines.

POPPGB: Contains the beginning address of the machine language code for the POP instructions and the POP jump table.

POPs, POP language: A macro language in which most of the compiler is written.

POP subroutines: The subroutines used by the POP interpreter to perform the operations of each POP instruction.

program text: The object code produced for the object module.

prune, pruning: A method of removing information from a roll, thereby making it inaccessible in subsequent operations.

quote: A sequence of characters preceded by a character count; used for comparisons with the input data.

QUOTE BASE: The initial address of the first quote (Parse).

recursion: A method of call and recall employed by the routines and subroutines of the compiler whereby routine X may call routine Y which, in turn, calls routine X.

releasing rolls: The method of making information reserved on a roll available for use by the compiler.

reserve mark: The 1-word value placed on a roll as a result of a reserve operation.

reserving rolls: A method of roll manipulation whereby information contained on a roll remains unaltered regardless of other operations involving the roll.

RETURN: Contains the return addresses for the POP subroutines.

roll: A type of table used by the compiler whose location and size are changed dynamically.

ROLLBR: Contains the beginning address of the base table.

roll control: A term applied collectively to those items used in roll maintenance and manipulation.

roll number: A number assigned to each roll in the compiler for the purpose of internal reference.

roll status items: Those variables maintained for each roll which contain the statistics needed in roll manipulation.

roll storage area: An area of the compiler in main storage that is allocated to the rolls.

rung: A word of a multiword group on a roll.

RUNTIME operations: Several routines which support object code produced by the compiler.

Save Area: An area of the object module used in linking to and from subprograms.

scalar variables: Nonsubscripted variables.

scan arrow: An item which refers to the position of the source statement character currently being scanned.

source module listing: The display of the statements constituting the source module.

storage allocation: The assignment of main storage to variables used in the source module.

storage map: The logical organization of a program or module and its components as they are maintained in main storage. (This map may also be displayed on an output device.)

SYMBOL 1,2,3: Halfword variables used to hold variable names used in the source module and other data.

TAG (field): A 1-byte item that is part of the pointer (indicating mode and size of the object pointed to) and driver (indicating mode of operation).

temporary storage: An area of main storage used by the compiler to temporarily maintain information for subsequent use.

terminal errors: Errors internal to the compiler causing termination of compilation of the source module.

TOP: A status variable maintained for each roll which indicates the new BASE of the roll when reserved information is contained on the roll.

traits: The TAG field (uppermost byte) of a word on a roll.

translation: The conversion from one type of language to another.

WORK roll: A special roll used by the compiler for maintaining values temporarily during processing.

WRKADR: The address maintained for the WORK roll that indicates the last word into which information has been stored; the "bottom" of the roll.

W0,W1,W2,....: Acronyms used to refer to the last groups of the WORK roll.

active characters
   definition 259
   description 26
ACTIVE END STA XLATE routine 14,39
active statements 36,39
ADCON roll 57,145
ADDR register
   definition 259
   description 29
address computation instructions 134,135
   cross-reference list 139
address constants 17,20,52,56,57
ADDRESS field
   definition 258
   description 29-30
addressing
   indirect 136,259
   relative 29,138
ADR CONST roll
   description 159
   in Exit 56
   in Unify 52
AFTER POLISH roll
   description 23,161
   in Gen 53,54
   in Parse 37-40,42
Allocate label lists 193-196
Allocate phase (IEYALL)
   cards produced 51
   definition 258
   detailed description 44-51
   general description 12
   location in storage 17
   rolls used by 44
   subprogram list 51
ALLOCATION FAIL routine 42
allocation of main storage 28
ALPHA LBL AND L SPROG routine 14,45
ALPHA SCALAR ARRAY AND SPROG routine 14,45
ALTER OPTION TABLE routine 240
ANSWER BOX variable
   definition 258
   description 26
   in Parse 38
AREA CODE variable 45,55,57,146
arithmetic and logical instructions
   130,131,139
array
   description 18
   dummy 47,48
   in Allocate 48,49
   listing of 21
   position in object module 17
   roll 26,47,146
ARRAY ALLOCATE routine 14,45,47
ARRAY DIMENSION roll 150
ARRAY PLEX roll 158
ARRAY REF roll 52,159
ARRAY REF ROLL ALLOTMENT 14,52
ARRAY REF ROLL ALLOTMENT routine 52
ARRAY roll
   assigning storage for 47
   description 146
   group stats for 25
artificial drivers 40
ASSIGNMENT STA GEN routine 54
AT roll 54,159

base addresses 28
BASE AND BRANCH TABLE ALLOC routine
   14,45,47
BASE, BOTTOM, and TOP tables 23,28
base table
   assigning storage for 47
   definition 259
   description 17
   position in object module 17
   use in Allocate 48
   use in Exit 57
BASE TABLE roll
   description 146
   in Allocate 45-48
   in Exit 56
BASE variable 23
   definition 259
BCD roll 45
BLOCK DATA PROG ALLOCATION routine 14,46
BLOCK DATA subprogram
   allocation for 46
   Parse processing of 39
BOTTOM variable 23
   definition 259
branch table
   assigning storage for 47
   description 18
   position in object module 17
   use in Allocate 47
   use in Exit 56
BRANCH TABLE roll
   description 150
   in Allocate 47
   in Exit 56
branch target label 12,18
BUILD ADDITIONAL BASES routine 14,45,49
BUILD NAMELIST TABLE routine 14,45,48
BUILD PROGRAM ESD routine 14,45,46
BYTE SCALAR roll 47,151

CALCULATE BASE AND DISP routine 14,45
CALL LBL roll 149
central items
   DATA 24,192,259
   definition 259
   description 24
   SYMBOL 24,191,259
CGOTO STA XLATE routine 38

character scanning 26-27
code producing instructions 134
CODE roll
description 160
in Exit 56
in Gen 53,54
location 22
COMMON ALLOCATION AND OUTPUT routine 14,45,47
COMMON ALLOCATION roll 47,156
COMMON AREA roll 155
COMMON data 12
COMMON DATA roll 152
COMMON DATA TEMP roll 155
COMMON NAME roll 152
COMMON NAME TEMP roll 156
COMMON statements
allocation for 45
COMMON variables
allocation of storage for 45
listing of 21
compiler
arrangement 28-29
assembly and operation of 136
code produced by 175-183
data structures 22
design of 9
flags used 27
general register usage 28
initialization of 33
limitations of 9
machine configuration for 9
messages 27
organization of 10,14
output from 16
purpose of 9
receiving control 33
relationship to system 19
rolls used in 140-162
storage configuration 15
termination of 33,35
COMPLEX CONST roll 143
CONSTR register
definition 259
description 28
control block area (CTLBLK) 227
control driver
definition 259
description 31
formats of 185-211
CONVERT TO ADR CONST routine 14,52
CONVERT TO INST FORMAT routine 14,52
CRRNT CHAR CNT variable
definition 259
description 26
in Parse 38
CRRNT CHAR variable
definition 259
description 26
in Parse 38

data items 24,192,259
DATA SAVE roll 145
data sets
SYSIN 15,33
SYSLIN 15,33
SYSPRINT 15,33
SYSPUNCH 15,33
DATA statements
allocation for 45
DATA VAR roll 56,154
DDNAMES routine 35
DEBUG ALLOCATE routine 14,45,49
decision making instructions 131,132
DECK option 51
DIMENSION statement
allocation for 46
variables specified on 29
DISPLAY statement
NAMELIST table for 18,19
DMY DIMENSION roll 14,46,147
DO loops
in Allocate 46
in Gen 55
in Parse 39
in Unify 12,51,52,53
DO LOOPS OPEN roll
description 144
in Allocation 46
in Parse 39
DO LOOP UNIFY routine 53
DO NEST UNIFY 14,53
DO STA XLATE routine 38
DP COMPLEX CONST roll 143
DP CONST roll
description 143
general 25
drivers
ADDRESS field 30
artificial 40
control 31,185-211,259
definition of 30
EOE 40,41
formats of 185-211
operation 30,260
OPERATOR field 30
plus and below phony 40,41
TAG field 30
dummy array 46,47
dummy dimension 46

END card 13
omission of 39
produced by Exit 57
END STA GEN routine 54,55
ENTRY CODE GEN routine 14,53,54
ENTRY NAME ALLOCATION routine 14,45,46
ENTRY NAMES roll 54,147
ENTRY roll 46
EOE driver 40,41
EPILOGUE GEN routine 14,53,54
epilogues 12,53,54
EQUIV ALLOCATION PRINT ERRORS routine 14,45,47
EQUIV MAP routine 14,45,48
EQUIVALENCE (EQUIV) ALLOCATION roll 47,48,156
EQUIVALENCE (EQUIV) HOLD roll 145
EQUIVALENCE (EQUIV) roll 46,47,151
EQUIVALENCE (EQUIV) TEMP roll 145
EQUIVALENCE OFFSET roll 45,152
EQUIVALENCE statements 12,45
EQUIVALENCE variables
allocation of storage for 45
description 18

listing of 21
map of 48
position in object module 17
EREXITPR routine 34
ERROR CHAR roll 144
ERROR LBL roll 148
ERROR MESSAGE roll 144
error messages 21
error recording 42
ERROR roll 42,148
errors
detection of 42
recording of 21,42
ERROR SYMBOL roll 149
ERROR TEMP roll 144
ESD cards
general 12
produced by allocate 44,47,51
Exit label list 208-211
EXIT PASS routine 14,55
Exit phase (IEYEXT)
definition 259
detailed description 55-58
general description 13
location in storage 15
rolls used by 55
exit roll
definition 259
description 24,161
general 10
in IEYROL 53
in Parse 38
location in storage 15
EXPLICIT roll 149
EXTADR register
definition 259
description 29
extended error message facility 229,235

FL AC roll 153
FL CONST roll 143
flags 27
forcing strength
definition 259
description 30,31
in Parse 40
table 31
FORMAT ALLOCATION routine 14,45,48
FORMAT roll 48,157
FORMAT statements
description 20
in Allocate 12,44,48
listing of 21
position in object module 17
FORTRAN error routine (IHCIBERH) 42,212
FULL WORD SCALAR roll 47,155
FUNCTION subprogram 46,49
FX AC roll 151
FX CONST roll 143

Gen label list 198-208
Gen phase (IEYGEN)
definition 259
detailed description 53-55
general description 12
location in storage 15
rolls used by 53
GEN PROCESS routine 14,53
GENERAL ALLOCATION roll 160
general register usage
used by compiler 28-29
used by object module 20
GET POLISH routine 14,53,54
global area 136
GLOBAL DMY roll 47,49,148
global jump table 28,137,138
global jumps 137,138
global label 136,137,259
GLOBAL SPROG ALLOCATE routine 14,45,48
GLOBAL SPROG roll
description 142
general 42
in Allocate 48
in Exit 56
GO TO STA GEN routine 55
GO TO statements, processing of 54,55
group
definition 259
description 24,25
group stats
definition 25,259
description 26
location in storage 15
sizes 25
group stats table 26

HALF WORD SCALAR roll 47,152
heading
position in object module 17
HEADOPT routine 35
HEX CONST roll 154

IBEXIT routine 223
IBFINT routine 222
IEYALL (see Allocate phase)
IEYEXT (see Exit phase)
IEYFINAL routine 35
IEYFORT (see Invocation phase)
IEYGEN (see Gen phase)
IEYJUN subroutine 138
IEYMOR routine 34
IEYPAR (see Parse phase)
IEYPCH routine 34
IEYPRNT routine 33
IEYREAD routine 34
IEYRETN routine 35
IEYROL (see roll module)
IEYUNF (see Unify phase)
IF statement 37,38,39
IHCDBUG 212,236
IHCDIOSE 212,229-236
routine directory 251
IHCERRM 212,229,239
IHCFCOMH 213-223
subroutine directory 246
IHCFCVTH 212,223
subroutine directory 246
IHCFINTH 212,239
IHCFIOSH 212,224
routine directory 251
IHCFOPT 240
IHCIBERH 42,212
IHCNAMEL 212,221

IMPLICIT roll 153
indirect addressing 135,260
indirect addressing instruction 135
IND VAR roll
description 141
in Parse 37
INIT roll 49,145
Invocation phase (IEYFORT)
definition 260
detailed description 33-36
general description 12
location in storage 15

jump instructions 132,133

keep
definition 260
general 23

label lists
Allocate 193-196
Exit 208-211
Gen 198-208
Parse 185-193
Unify 196-198
labeled statement references 12
labels
branch target 12,18
detailed description 135,136
global 135,136
local 135,136
mode 17,54
LAST CHAR CNT variable
definition 259
description 26
in Parse 38
LAST SOURCE CHAR variable 38
LBL FIELD XLATE routine 14,37,38
LBL process routine 14,53,54
LBL roll 45,46,54,153
LEVEL ONE UNIFY routine 53
LIB roll 140
LITERAL CONST ALLOCATION routine 14,45,47
literal constants
description 20
in Allocate 12,44,45
position in object module 17
LITERAL CONST roll 143
LITERAL TEMP (TEMP LITERAL) roll 155
LOAD and DECK options 33
LOCAL DMY roll 148
local label 136,137,259
LOCAL SPROG roll 45,46,149
LOGICAL IF STA XLATE routine 38
LOOP CONTROL roll 52,156
LOOP DATA roll
description 157
in Parse 38
in Unify 53
LOOP SCRIPT roll 142

made labels 17,54
map
of scalars 47
storage 21,44,50,260
MAP option 51

messages
description 27
location in storage 15
printing of (IEYPRNT) 33
produced by Allocate 48,49
produced by Invocation 35,36
produced by Parse 43,44
minimum system configuration 9
MOVE ZEROS TO T AND C routine 14
MPAC1 and MPAC2 variables
definition 259
description 26
multiple precision arithmetic 26

NAMELIST ALLOCATION roll 48,49,155
NAMELIST ITEMS roll 149,150
NAMELIST MPY DATA roll 57,160
NAMELIST name
roll 48
table for 19
NAMELIST NAMES roll 48,149
NAMELIST tables
definition 259
description 19
in Allocate 12,44,47
in Exit 57
listing of 20,48
position in object module 20
NEST SCRIPT roll
description 141
in Unify 53
NONSTD SCRIPT roll 141

object module
configuration of 17
description of 17
general register usage 20
listing of 20,21,54,57
writing of 49
object-time library subprograms 212-240
operation driver
definition 259
description 30
formats of 185-211
OPERATOR field
definition 259
description 30-32
optimization 52,53,259
option table 241
ORDER AND PUNCH RLD ROLL routine 14,55,57

Parse phase (IEYPAR)
definition 260
detailed description 36-42
general description 12
location in storage 15
rolls used by 37
PASS 1 GLOBAL SPROG ALLOCATE routine 14,45,48
phases
Allocate 12,15,44-51
components of 14
Exit 13,15,55-57
Gen 12,15,53-55
Invocation 12,15,33-35
Parse 12,15,36-44
Unify 12,15,51-53

plex
  definition 260
  description 25
plus and below phony driver 40,41
pointer
  ADDRESS field 29
  definition 260
  description 29
  OPERATOR field 29
  TAG field 29
Polish notation
  arithmetic and logical assignment statement 164
  arithmetic expressions 39
  arithmetic IF statement 165
  array references 163
  ASSIGN statement 164
  assigned GO TO statement 164
  BACKSPACE statement 171
  BLOCK DATA statement 166
  CALL statement 172
  computed GO TO statement 165
  CONTINUE statement 165
  DATA statement 166
  debug statements 172-173
  DEFINE FILE statement 170
  definition of 259
  direct-access statements 170
  DO statement 165
  END FILE statement 171
  END statement 166
  ENTRY statement 164
  Explicit specification statements 166
  FIND statement 170
  formats 163-173
  FUNCTION statement 171
  general 10
  in Gen 12,53,54
  in Parse 13,36,39
  input/output lists 167-168
  labeled statements 163
  logical IF statement 164
  PAUSE and STOP statements 165
  PRINT statement 169
  PUNCH statement 169
  READ statement 167,168,169
  RETURN statement 164
  REWIND statement 171
  statement function 171
  SUBROUTINE statement 171
  unconditional GO TO statement 165
  WRITE statement 168,169,170
POP instructions
  ADD 130
  AFS 130
  AND 130
  APH 127
  ARK 127
  ARP 127
  ASK 127
  ASP 127
  BID 134
  BIM 134
  BIN 134
  BOP 127
  CAR 128
  CLA 128
  CNT 128
  CPO 128
  cross reference list 139
  CRP 128
  CSA 131
  CSF 133
  definition 259
  detailed description 127-135
  DIM 130
  DIV 130
  EAD 128
  EAW 128
  ECW 128
  EOP 128
  ETA 128
  FET 128
  FLP 128
  FRK 128
  FRP 128
  FTH 128
  general description 10
  IAD 129
  IND 135
  IOP 129
  IOR 130
  ITA 129
  ITM 129
  JAF 133
  JAT 133
  JOW 133
  JPE 133
  JRD 133
  JSB 133
  JUN 133
  LCE 129
  LCF 129
  LCT 129
  LGA 131
  LGP 129
  LLS 130
  LRS 131
  LSS 129
  MOA 131
  MOC 129
  MON 129
  MPY 131
  NOG 129
  NOZ 129
  PGO 130
  PGP 130
  PLD 130
  PNG 130
  POC 130
  POW 134
  PSP 131
  PST 130
  QSA 131
  QSF 133
  REL 134
  RSV 134
  SAD 131
  SBP 131
  SBS 131
  SCE 132
  SCK 132
  SFP 132
  SLE 132
  SNE 132
  SNZ 132

SOP 132
SPM 132
SPT 132
SRA 132
SRD 132
STA 132
STM 133
SUB 131
SWT 130
TLY 131
W0P 135
W1P 135
W2P 135
W3P 135
W4P 135
XIT 133
ZER 130
POP interpreter
definition 260
description 136
general 10
POP jump table (POPTABLE)
definition 260
description 28,137
location in storage 15
POP language
cross-reference list 139
definition 260
detailed description 127-138
general description 10
notation used 127
POP SETUP routine 137
POP subroutines
assembler references to 137
definition 260
general 10
location in storage 15
POPADR register
definition 260
description 29
POPPGB register
definition 260
description 29
POPXIT register
description 29
PREP DMY DIMAND PRINT ERRORS routine 14,45
PREP EQUIV AND PRINT ERRORS routine 14,45
PREP NAMELIST routine 14,45,48
PRESS MEMORY 21,22,193
PRINT A LINE routine 14
PRINT AND READ SOURCE routine 14,37
PRINT HEADING routine 14
PRINT TOTAL PROG REQMTS MESS routine 14
printmsg table 35-36
PRNTHEAD routine 34
PRNTMSG routine 34
PROCESS DO LOOPS routine 14,45,46
PROCESS LBL AND LOCAL SPROGS routine 14,45,46
PROCESS POLISH routine 14,39
production of object code
branches 175
computed GO TO statement 175
DEFINE FILE statement 179
direct-access READ and WRITE statements 179
DO loops 175
DO statement 175
FIND statements 179
FORMAT statements 180,181
formatted arrays 177
formatted list items 177
functions 176
input/output 177
PAUSE statement 179
READ and WRITE statements 177
statement functions 176
STOP statement 179
subroutines 176
unformatted arrays 178
unformatted READ and WRITE statements 178
PROGRAM BREAK variable 45,46,47,48,49
PROGRAM SCRIPT roll
description 158
in Parse 39
in Unify 52
program text
definition 260
description 20
position in object module 17
prologue 12,53,54
PROLOGUE GEN routine 14,53,54
pruning
definition 260
description 23
pseudo instructions 10,127
PUNCH ADCON ROLL routine 14,55,57
PUNCH ADR CONST ROLL routine 14,55,56
PUNCH BASE ROLL routine 14,55,56
PUNCH BRANCH ROLL routine 14,55,56
PUNCH CODE ROLL routine 14,55,56
PUNCH END CARD routine 14,55,57
PUNCH GLOBAL SPROG ROLL routine 14,55,57
PUNCH NAMELIST MPY DATA routine 55,57
PUNCH PARTIAL TXT CARD routine 55,56
PUNCH SPROG ARG ROLL routine 14,55,56
PUNCH TEMP AND CONST ROLL routine 14,55,56
PUNCH USED LIBRARY ROLL routine 14,55,57

quick link output 136
quote
definition 260
description 27
location in storage 15
QBASE 27
quote base (QBASE)
definition 260
description 27

REASSIGN MEMORY 185
recursion
definition 261
in compiler 10
REG roll 146
REGISTER IBCOM routine 14,37
register usage
by compiler 28
by object module 20
relative addressing 29,137
releasing rolls
definition 261
in Allocate 45
in Invocation 35

reserve mark
   definition 261
   description 23
RETURN register
   definition 261
   description 29
RETURN statement
   Polish notation for 37
RLD cards 13,56
RLD roll 55,56,57,156
ROLL ADR table
   in IEYROL 53
   in Invocation 35
   location in storage 15
   use in allocating storage 22,35
   use in finding address of variable 30
   use in releasing storage 35
roll control instructions 133
roll controls
   general 21
roll module (IEYROL)
   definition 261
   detailed description 53
   general description 13
   location in storage 15
roll statistics
   BASE, BOTTOM, TOP 22
   location in storage 15
roll storage area
   definition 261
   general description 21
ROLLBR register
   definition 261
   description 29
rolls
   ADCON 57,145
   ADR CONST 52,56,159
   AFTER POLISH 23,37-40,42,53,54,161
   allocating storage for 21,22,34
   ARRAY 26,47,146
   ARRAY DIMENSION 150
   ARRAY PLEX 158
   ARRAY REF 52,159
   AT 54,159
   BASE TABLE 45-48,56,146
   BCD 45
   BRANCH TABLE 47,56,150
   BYTE SCALAR 47,151
   CALL LBL 149
   CODE 22,53,54,56,160
   COMMON ALLOCATION 47,156
   COMMON AREA 155
   COMMON DATA 152
   COMMON DATA TEMP 155
   COMMON NAME 152
   COMMON NAME TEMP 156
   COMPLEX CONST 143
   DATA SAVE 145
   DATA VAR 56,154
   definition of 261
   detailed description 140-162
   DMY DIMENSION 14,46,147
   DO LOOPS OPEN 39,46,144
   DP COMPLEX CONST 143
   DP CONST 25,143
   ENTRY 46
   ENTRY NAMES 54,147
   EQUIV ALLOCATION 43,47,48,156
   EQUIVALENCE (EQUIV) 46,47,151
   EQUIVALENCE (EQUIV) HOLD 145
   EQUIVALENCE (EQUIV) TEMP 145
   EQUIVALENCE OFFSET 45,152
   ERROR 42,148
   ERROR CHAR 144
   ERROR LBL 148
   ERROR MESSAGE 144
   ERROR SYMBOL 149
   ERROR TEMP 144
   EXIT 10,15,24,38,53,161,259
   EXPLICIT 149
   FL AC 153
   FL CONST 143
   FORMAT 48,157
   formats 140-162
   FULL WORD SCALAR 47,155
   FX AC 151
   FX CONST 143
   GENERAL ALLOCATION 160
   general description 10,21
   GLOBAL DMY 47,49,148
   GLOBAL SPROG 42,48,56,142
   HALF WORD SCALAR 47,152
   HEX CONST 154
   IMPLICIT 153
   IND VAR 37,141
   INIT 49,145
   LBL 45,46,54,153
   LIB 140
   LITERAL CONST 143
   LITERAL TEMP 155
   LOCAL DMY 148
   LOCAL SPROG 45,46,149
   location in storage 15
   LOOP CONTROL 52,156
   LOOP DATA 38,53,157
   LOOP SCRIPT 142
   NAMELIST ALLOCATION 48,49,155
   NAMELIST ITEMS 149,150
   NAMELIST MPY DATA 57,160
   NAMELIST NAMES 48,149
   NEST SCRIPT 53,141
   NONSTD SCRIPT 141
   POLISH 36-42,53,54
   PROGRAM SCRIPT 39,52,158
   pruning of 23
   REG 146
   releasing of 35,45,260
   reserving of 23,261
   RLD 55,56,57,156
   SCALAR 47,48,154
   SCRIPT 36,37,52,53,157
   size limitations 22
   SOURCE 37,38,140
   special 24
   SPROG ARG 56,147
   STD SCRIPT 144
   SUBCHK 49,160
   TEMP 144
   TEMP AND CONST 45,55,57,144
   TEMP DATA NAME 150
   TEMP NAME 36,143
   TEMP PNTR 153
   TEMP POLISH 151
   used by Allocate 44
   used by Exit 55
   used by Gen 53

used by Parse 36
used by Unify 52
USED LIB FUNCTION 48,55,152
WORK 10,15,24,38-41,53,54,161,261
rungs
definition 261
description 24

save area
assigning storage for 47
definition 261
position in object module 17
SCALAR ALLOCATE routine 14,45,47
SCALAR roll 47,48,154
SCALAR routine 14
scalar variable
definition 261
listing of 21
position in object module 17
scan arrow
definition 261
description 26
scan control variables 26,27
SCRIPT roll
description 157
in Parse 36,37
in Unify 52,53
source module listing
definition 261
description 20,42
format of 42
SOURCE option 36
SOURCE roll
description 140
in Parse 37,38
special rolls 24
specification statements 35
SPROG ARG ALLOCATION routine 14,45,48
SPROG ARG roll 56,147
STA FINAL routine 14,37,39
STA GEN FINISH routine 14,54,55
STA GEN routine 14,54,55
STA INIT routine 14,38
STA LBL BOX 54
STA RUN TABLE 54
STA XLATE EXIT routine 38
STA XLATE routine 14,37,38,39
START ALLOCATION routine 14
START COMPILER routine 14,37
START GEN routine 14,53
START UNIFY routine 14,52
STATEMENT PROCESS routine 14,37,39
status variable 23
STD SCRIPT roll 144
STOP statement
Polish notation for 37
storage map
compiler 14
definition 261
description 21
object module 17
produced by Allocate 44,50
SUBCHK roll 49,160
subprogram addresses
position in object module 17
subprogram argument lists
position in object module 17,51
SUBSCRIPTS FAIL routine 42
SYMBOL item 24,261
syntax error 42
SYNTAX FAIL routine 38,42
system names 11

tables
base 17,47,56,259
BASE, BOTTOM, and TOP 23,28
branch 18,46,56
global jump 28,137
group stats 25,26
NAMELIST 12,18,19,20,44,48,49,57,260
POP jump 15,28,136,260
printmsg 35
ROLL ADR 15,22,28,34,53
STA RUN 54
unit assignment 225,232
TAG field
definition 261
description 29-31
TEMP AND CONST roll
description 144
in Allocate 45
in Exit 55,57
TEMP DATA NAME roll 150
TEMP NAME roll
description 143
in Parse 38
TEMP PNTR roll 153
TEMP POLISH roll 151
TEMP roll 144
temporary storage and constants
description 20
position in object module 17
TERMINATE PHASE routine 54,55
termination of compiler 33,35
TIMEDAT routine 35
TOP variable 23
definition 261
TRACE option 54
transmissive instructions 127-130
TXT cards
general 12
produced by Allocate 44,49,51
produced by Exit 55,56,57,58
type statements
allocation for 46

Unify label list 196-198
Unify phase (IEYUNF)
definition 260
detailed description 51-53
general description 12
location in storage 15
rolls used by 52
unit assignment table (IHCUATBL) 225,232
unit blocks 224,230
USED LIB FUNCTION roll
description 152
in allocation 48
in Exit 55

variables
  ANSWER BOX 26,38,259
  AREA CODE 45,56,57,146
  BASE 23,259
  BOTTOM 23,259
  COMMON 21,46
  CRRNT CHAR 26,38,259
  CRRNT CHAR CNT 26,38,259
  EQUIVALENCE 18,21,44,45,48
  LAST CHAR CNT 26,38,260
  LAST SOURCE CHAR 38
  MPAC1 and MPAC2 26,260
  PROGRAM BREAK 45,46,47,48
  scalar 18,21,261
  scan control 26,27
  status 23
  TOP 23,261

WORK roll
  definition 261
  description 24,161
  general 10
  in Exit 57
  in Gen 54
  in IEYROL 53
  in Parse 38,40,41
  location in storage 15
WRKADR register
  definition 261
  description 29

Y28-6638-1

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 S360-25

Re: Form No. Y28-6638-1

This Newsletter No. Y28-6826

Date November 15, 1968

Previous Newsletter Nos. None

IBM System/360 Operating System
FORTRAN IV (G) Compiler
Program Logic Manual

This Technical Newsletter, a part of Release 17 of the IBM System/360 Operating System, provides replacement pages for IBM System/360 Operating System: FORTRAN IV (G) Compiler Program Logic Manual, Form Y28-6638-1. These replacement pages remain in effect for subsequent releases unless specifically altered. Pages to be replaced and/or added are listed below.

Pages

Cover, preface	
19-20	145-146.1
27-32	153-154.1
35-38.1	157-158
39-40	177-178
45-46	185-186
53-54	191-192
67-68 (67.1 added)	221-224.1
69-70.1	225-226 (225.1 added)
71-72	229-230.1
77-78	253-254

Changes to the text, and small changes to illustrations, are indicated by a vertical line to the left of the change; changed or added illustrations are denoted by the symbol • to the left of the caption.

Summary of Amendments

New information about innermost DO loops with a possible extended range has been added. The information includes descriptions of two new routines, XTEND LABEL and EXTND TARGET LABEL, and changes to existing routines and flowcharts involved in phase 1 processing. There is also additional information about the NAMELIST table entries and the GET POLISH routine, and a description of the improvements made by the FORTRAN object-time library in the processing of BACKSPACE statements.

File this cover letter at the back of the publication to provide a record of changes.

IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, N.Y. 10020

PRINTED IN U.S.A.

Restricted Distribution