Description of

/360 WATFOR©

A Fortran-IV Compiler

Faculty of Mathematics

University of Waterloo

Waterloo, Ontario

Canada

Department of Applied Analysis

&

Computer Science

Description of

/360 WATFOR©

A Fortran-IV Compiler

P. H. Cress
P. H. Dirksen
M. S. Doyle
L. K. Kesselhut
W. C. Kindree
D. S. Meek
W. R. Milne
E. L. Schmidt
S. J. Ward
C. L. Williams

- Department of Applied Analysis
  and Computer Science
- Computing Centre
  University of Waterloo.

April, 1968.

CSTR-1000

Acknowledgements

This documentation is another result of the effort that has gone into the production of the first version of the /360 WATFOR Compiler. Many persons other than the authors of the compiler and this documentation have contributed in some way but are too numerous to mention by name. However, we must thank Miss Ann Cross for her patience with us while carefully typing the complete manuscript.

This project has been sponsored in part by the following grants:

IBM Fellowship for 1966

NRC Grant No. A-2244

Page

Chapter 1 Introduction

1.1 The WATFOR Project 1
1.2 The Project Personnel 4
1.3 This Manual 5

Chapter 2 General Structure

2.1 Introduction 6
2.2 Logic Flow of WATFOR 8
2.3 Memory Layout 9
2.4 Symbol Table Entries for /360 WATFOR 11
2.5 Work Areas 18

2.5.1. ISN Coding 18
2.5.2. Handling of Undefined Variables at Execution Time 19
2.5.3. Addressing of Simple Variables in COMMON or EQUIVALENCE 20

2.6 Relocation Principle 22

2.6.1. General 22
2.6.2. Symbol Table Pointers 22
2.6.3. Temporary Pointers 23
2.6.4. Relocator Codes 24
2.6.5. Addressing of Complex Values 24

2.7 Intermediate Text Created by SCAN routine (STACK) 26
2.8 Conventions 28

2.8.1. Register 28
2.8.2. Prefix 30
2.8.3. Programming 31

Chapter 3 Communication Regions

3.1.1. Introduction 32
3.1.2. Constants and Switches in STARTA and COMMR 35

3.2 Entry and Return Routines (CENT,CRET) 38
3.3 Symbol Table Lookup Routine 39

Page

3.3.1. Introduction 39
3.3.2. Statement Number Lookup 40
3.3.3. Symbol Lookup 42
3.3.4. Library List Lookup 44
3.3.5. Constant Collector and Lookup 45
3.3.6. Constant Lookup 47

3.4 Constant Collection 49
3.5 Object Code Output Routine (COUT) 56
3.6 Addressing Routines (CUSING,CBALR11) 57
3.7 Input Routine (CREAD) 57

3.7.1. Constants and Switches 57
3.7.2. Calling Sequence 58
3.7.3. Description of routine 58

3.8 Output Initialization Routine (CPOINT) 60
3.9 Memory Overflow. Check Routine (CGETSYM) 60
3.10 Keyword Elimination Routine - CSETSTAK 60
3.11 Alignment Routines (CNOP's and CDS0's) 63
3.12 Entry and Return Routines (XENT,XRET) 64
3.13 Error Message Editor 67
3.14 Execution-Time Subscripting Routine (XA1,XAN) 72
3.15 Array Utility Routine (X1STELT) 76
3.16 Exponential Routines 76
3.17 Complex Multiply and Divide Routines 77
3.18 Fix and Float Routines 78
3.19 Undefined Variable Checking Routine (XROUT*n) 78
3.20 Execution Input/Output (XIOINIT, XARRAY, XSIMPELT, XSUBSELT) 79
3.21 Pre-Execution DATA Statement Processor - XDATA, PDATA 81
3.22 Compiler Output Routine (XPRINT) 86
3.23 Execution-Time Interrupt Handling Routine (XRUPT) 87
3.24 Run Time Operator Message Routines (XSTOPN,XPAUSE) 89

Chapter 4 Compile-Time Statement Processors

4.1 SCAN 90

4.1.1. Introduction 90
4.1.2. Scanning a Statement 91
4.1.3. Transformation of Statement 93
4.1.4. Determining the Statement Type (SFIND) 98
4.1.5. Error Checking and Miscellaneous 100

			Page
4.2	LINKR		103
	4.2.1.	Introduction	103
	4.2.2.	FUNCTION Statement Processor	103
	4.2.3.	SUBROUTINE Statement Processor	104
	4.2.4.	Argument List Processor	104
	4.2.5.	Type FUNCTION Statement Processors	104
	4.2.6.	ENTRY Statement Processor	106
	4.2.7.	RETURN Statement Processor	107
	4.2.8.	END Statement Processor	108
	4.2.9.	END Statement Simulator	108
	4.2.10.	Main Entry Point Processor	108
	4.2.11.	BLOCK DATA Statement Processor	108
	4.2.12.	Arithmetic Statement Function (ASF)	109
	4.2.13.	ASF Return Processor	110
	4.2.14.	Programme Segment Initialization	111
4.3	SPECS		112
	4.3.1.	Introduction	112
	4.3.2.	Tables and Switches used in SPECS	113
	4.3.3.	Linking Operations in SPECS	115
	4.3.4.	Object Code	116
	4.3.5.	Detailed Description of Processor	116
	4.3.6.	EQUIVALENCE	120
	4.3.7.	COMMON Processor	128
	4.3.8.	IMPLICIT Processor	131
	4.3.9.	EXTERNAL Processor	132
	4.3.10.	Service Routines	133
4.4	ARITH		136
	4.4.1.	General Description	136
	4.4.2.	Object Code Generated	138
	4.4.3.	Syntax-Checking and Lookup Routine	141
	4.4.4.	Code-Generating Phase	142
	4.4.5.	The Routine To Output Code for an Operator	147
	4.4.6.	The Pieces of Dummy Object Code	150
	4.4.7.	Outputting Routines	151
	4.4.8.	Routines to Output Subscript, Indirect Addressing and/or Undefined Variable Checking Code for Operands in ARITH	152
	4.4.9.	Routine to Prepare An Argument or Subscript in a List	153
	4.4.10.	Routine to Output a Function CALL	154
	4.4.11.	Contents of the Register Table	155

			Page
	4.4.12.	Routine to 'Detach' Subscript List From Stack	155
	4.4.13.	Routine to Assign Temporary Storage	156
	4.4.14.	Routine to Output Subscript Coding	157
	4.4.15.	The End of Expression Routine	158
	4.4.16.	Error End-of-Statement Routine	158
	4.4.17.	Entry Points and Termination Routines for Various Types of Statements	158
4.5	DODO		163
	4.5.1.	Introduction	163
	4.5.2.	Compile GO TO	163
	4.5.3.	Assigned GO TO	165
	4.5.4.	Arithmetic IF Statement	166
	4.5.5.	ASSIGN Statement	166
	4.5.6.	DO Statement	167
	4.5.7.	End of DO loop coding	174
	4.5.8.	Common Routines	175
	4.5.9.	Switches	175
4.6	INOUT		176
	4.6.1.	General Organization	176
	4.6.2.	CONTINUE	178
	4.6.3.	STOP	178
	4.6.4.	PAUSE	179
	4.6.5.	BACKSPACE, ENDFILE, REWIND	180
	4.6.6.	PUNCH, PRINT, WRITE, READ	181
	4.6.7.	DATA	186
	4.6.8.	INOUT Utility Routines	192
4.7	FORMAT		203
	4.7.1.	Introduction	203
	4.7.2.	Compile-Time Entry (FORMAT)	203
	4.7.3.	Execution-Time Entry (FORMATEX)	204
	4.7.4.	Specification Converter (FORSCAN)	204
4.8	Relocator		208
	4.8.1.	Introduction	208
	4.8.2.	Phase One	208
	4.8.3.	Phase Two	216
	4.8.4.	Relocation Phase Three	220

		Page
Chapter 5	EXECUTION-TIME ROUTINES	
5.1	FUNCTION	227
5.1.1.	The Mathematical Functions	227
5.1.2.	Macros Used by FUNCTION	229
5.1.3.	The 'IN-LINE' Functions	232
5.2	EXECUTION FORMAT	239
5.3	FRIOSCAN	240
5.3.1.	Introduction	240
5.3.2.	Initialization	242
5.3.3.	Scan of Input Field	243
5.3.4.	Routines to Process the Input Data	243
5.4	FORMCONV	246
5.4.1.	Introduction	246
5.4.2.	Formatted Output	248
5.4.3.	Formatted Input	254
5.4.4.	Free Output	257
5.4.5.	Free Input	258
5.4.6.	Common Subroutines and Utility Routines	259
5.4.7.	Important Variables and Switches	267
5.4.8.	Non-Formatted (Binary) I/O (INBINI, OUTBINI, INBIN, OUTBIN)	269
5.5	FIOCS	270
5.5.1.	Error Messages	270
5.5.2.	Switches Added	270
5.5.3.	Line and Page Control (PRITE)	271
5.5.4.	Entry Coding Added	271
Chapter 6	MAIN	273
6.1	Introduction	273
6.2	Initial Batch Entry	276
6.3	End of Batch	276
6.4	Start of Compilation	276
6.5	End of Execution	279
6.6	Library Programme Processor	280

		Page
Chapter 7	Debugging the Compiler	281
Chapter 8	Improvements and Further Development	285
Appendix A	Subprogramme Linkage Conventions in WATFOR	287

Figures

		Page
2.2.1.	Logic Flow of WATFOR	8
2.3.1.	Memory Layout	10
2.5.3.	Addressing Equivalenced Variables in Common Blocks	21
2.6.3.	Addressing of Complex Variables	25
3.3.1.	Routine to Generate Pseudo Statement Number	41
3.3.2.	Symbol Lookup	43
3.3.3.	Library List Lookup	44
3.3.4.	Logic of Constant Lookup Routine	48
3.4.1.	Constant Collector	54
3.10.1.	Stack Showing Keyword Elimination	61
3.10.2.	Logic of CSETSTAK	62
3.12.1.	Description to Type Code in Argument List	65
3.14.1.	Subscripting Routine For Multiply Subscripted Variables (XAN)	73
3.14.2.	Subscripting Routine For Singly Subscripted Variables (XA1)	74
3.14.3.	Object Code and Dot Routine for a Subscripted Variable	75
3.21.1.	Flowchart for XDATA and PDATA	84
4.1.1.	General Flow of SCAN Routine	92
4.2.1.	Object Code for FUNCTION and SUBROUTINE	105
4.3.1.	Linkage in Symbol Table to Dimension List	119
4.3.3.	Examples of Equivalence Processing	125
4.4.1.	General Flow Diagram for ARITH	137
4.4.2.	Use of Registers in Object Code of ARITH	140
4.4.3.	Flow Diagram of Syntax-Checking and Lookup Phase of ARITH	144
4.4.4.	Table of Routines for Operator Pairs	146
4.4.5.	Routine to Output Code for An Operator	148
4.4.6.	Example of Compilation of a Complete Statement	161
4.4.7.	Routines Used in Code-Generating Scan	162
4.5.1.	GO TO statements	163
4.5.2.	DO Statement	167
4.6.1.	Processing CONTINUE	178
4.6.2.	Processing PUNCH, PRINT, WRITE, READ	182
4.6.3.	Processing DATA Statement	190
4.6.4.	I/O List Compiler	197.1
4.6.5.	Stack in Stages and Object Code as it appears for a PRINT Statement with an implied DO-LOOP	198
4.6.6.	END/ERR Processors	201
4.8.1.	Equivalence List Processor in RELOC-1	210
4.8.2.	Common Block Processor in RELOC-1	213
4.8.3.	Combined Common-Equivalence List	214

		Page
4.8.4.	Equivalence List Processor in RELOC-3	222
4.8.5.	Named Common Block Processor in RELOC-3	224
4.8.6.	Common Block List Processor in RELOC-3	225
5.1.1.	Function Dependency Table	228
5.3.1.	Flowchart of FRIOSCAN	241
6.1.	Linkage between COMMR, STARTA and MAIN	274
6.2.	General Flow of MAIN	275

Tables

		Page
4.1.1.	Transformed Operator Table	91
4.3.1.	CSECTS in SPECS	112
4.3.2.	Entry Points in SPECS	113
5.3.1.	Use of Registers in FRIOSCAN	242
5.3.2.	Switches and counters in FRIOSCAN	243
5.3.3.	Format of Incoming Data	244

CHAPTER 1

Introduction

1.1 The WATFOR Project

The /360 WATFOR compiler was not the first in-core compiler ever written nor, probably for that matter, will it be the last. One of the purposes of this manual is to attempt to make life simpler for those of you who in the future might be faced with the task of producing such a compiler and want to see how somebody else once did it.[1]

A few words might be said at this point about another in-core compiler with which some of you might be familiar. In the summer of 1965 four undergraduate students at the University of Waterloo developed a fast Fortran compiler[2] for the IBM 7040 computer which the University then had. This was the result of the first WATFOR compiler project and its principle aims were fast compilation with good debugging facilities for the non-professional programmer. The project was an enormous success and proved at least the following points: a fast, useful load-and-go compiler could be produced in a reasonable time by a few relatively non-professional programmers.

Sometime during early 1966 the University decided to order an IBM 360 computer to replace its 7040 and as the number of non-professional programmers the Computing Centre was obliged to give service to was increasing dramatically and as it was obvious from the success of 7040 WATFOR that a replacement for it would be needed for the new machine, another team was formed to produce another WATFOR compiler. This team first started meeting in early May of 1966 and consisted of three full-time employees of the University (Mrs. Betty Schmidt, Paul Dirksen, Paul Cress) and three undergraduate students (Lothar K. 'Ned' Kesselhut, Bill Kindree, Dereck Meek). The team held a series of planning sessions and on May 27 a combined meeting was held with other members of the Computing Centre. The purpose of this meeting was to reveal the objectives of the /360 WATFOR Project that had been decided, in part, in the earlier planning meetings and to seek further suggestions and criticism.

---

1. We were planning to carve this manual in stone but felt the mailing costs would be prohibitive.
2. See Shantz, P.W. et. al. "WATFOR - The University of Waterloo Fortran-IV Compiler" CACM v10, no. 1, January 1967.

The objectives agreed upon at that time are listed here:

1. To produce an in-core load-and-go compiler.
2. Its compilation speed to be as fast as our coding skill would produce.
3. The compiler was to produce at least as good error diagnostics as its forerunner 7040 WATFOR (which are excellent) at both compile and execution time.
4. We were to implement as much of full /360 Fortran-IV as we felt we could in a reasonable amount of time.
   (Since the only manual we had available to us at the time which described /360 Fortran-IV was "IBM System/360 FORTRAN IV Language", form C28-6515-0, we decided, for what we hoped would be compatability reasons, that our compiler would implement the language therein described, occasionally taking educated guesses at points which were vague or missing).
5. A software check for undefined variables would be provided. (This was done in 7040 WATFOR using a hardware feature.)
6. Under the limitation imposed by 5, produce as good execution speed as possible from well designed object code.
7. Format-free READ and PRINT statements (e.g. READ,A,B) were to be implemented since they are invaluable to learning programmers.
8. 'FORMAT(' when used as the first seven characters of a statement was to be the only 'reserved' character sequence.
9. The compiler was to work on a minimum 128K machine with standard instruction set and floating point feature.
10. We would use some routines from IBM's run-time Fortran library (e.g. I/O interface, function approximation routines) to save ourselves some work.
11. The compiler was to accept source decks punched on cards or stored on a direct access library.
12. Infrequently used machine language run-time routines were to be stored on direct access device and loaded into memory as required by a compiled programme (cf. LOAD feature of O.S.)
13. The compiler itself and the object code generated by it were to be as independent of machine position as possible (i.e. no absolute adcons were to be stored by the programmes). All adcons used or constructed by the compiler would be relative to some standard value which would be kept in an index register. Hence the origin of our term 'offset addressing' and the frequent label START found throughout the compiler. (This objective was imposed since at the time we had vague plans of producing an operating system with roll-out and scatter roll-in feature.)
14. The compiler was to be written in assembler language and be modularly constructed to facilitate changing.
15. The result of the project was to be a package that would be distributable since we felt other installations could use such a processor as we were designing.
16. The package was to be well documented.
17. If nothing else, we were to gain experience and learn.

We were going to accept the following restrictions from the outset:

1. No user written assembly language subprogrammes would be accepted by the compiler i.e. FORTRAN programmes only.
2. No 'object decks' would be produced.
3. No overlay facility would be provided.

The objectives listed above were established in the face of the following potential impediments:

- at that time none of the project members had ever seen an IBM 360 computer.
- none of the project team had ever written an assembly language programme for the 360 (although we were busy reading the Principles of Operations and Assembler Language manuals).
- we had never even heard the words "DD card" at that time.[3]
- we didn't know if any IBM run-time library existed but guessed it would by the time we would be ready to use it.
- we had no 360 Fortran compiler available to compare results with or try test cases on. (The G compiler had not been revealed by that time and the H compiler was still too big to fit into any machine we would see for quite a while.)

We did however have the benefit of the documentation for 7040 WATFOR and association and discussion with the members of that team.

WATFOR has been in use now for almost a year. In truth it must be said that not all of the objectives listed above have been met by the project. For example we dropped number 12 entirely; in number 13 we have been only partially successful. However, if we may take the liberty of saying so in our own manual, we feel that most objectives have been met sufficiently to call the project successful. We're damn proud of it at any rate.

---

3. Ignorance is bliss. (Anon.)

1.2 The Project Personnel

In September 1966 the three undergraduate students returned to school but continued working on a part-time basis. Two other members of the University staff (Mike Doyle, Rod Milne) began to assist the project about that time by writing certain routines which had been planned but not written.

It might be fitting to mention at this time some of the work that each person did.

Betty Schmidt wrote the arithmetic compiler ARITH, part of the relocator RELOC and converted almost all of IBM's function library to WATFOR's use as well as writing some functions herself.

Lothar Kesselhut wrote the type statement compiler SPECS.

Dereck Meek wrote the control statement compiler DODO.

Bill Kindree wrote LINKR, RELOC-I, II and sections of MAIN.

Rod Milne wrote FORMCONV, the execution time data converter and a Trace Programme which has been invaluable for debugging.

Mike Doyle wrote FRIOSCAN, the execution-time input-field scanning routine.

Paul Dirksen wrote the SCAN routine, FORMAT compiler and converted IBM's FIOCS and parts of IBCOM to our use. He also acted as editor of this manual.

Paul Cress wrote the I/O statement processor INOUT and occassionally acted as project supervisor.

All project members contributed to the design of the compiler and, as well as writing the above mentioned compiler modules, contributed routines and macros too numerous to mention individually e.g. routines in COMMR, STARTA, MAIN. The job of keypunching and debugging routines was also the responsibility of the individual members.

Since February 1967 Mrs. Sandra Ward has ably acted as our chief correspondent, distribution agent and coordinator of updates.[4]

Mrs. Lynn Williams has lately assumed responsibility for

---

4. An autographed picture of this group, suitable for framing,may be obtained by writing Mrs. Ward.

FRIOSCAN and has supplied the documentation of it for this manual since Mike Doyle is busy as Supervisor of Systems for the Computing Centre. All other project members supplied the documentation of their routines.

All persons mentioned above have aided in the continuing maintenance of the compiler by very cooperatively fixing those bugs which seem to have cropped up in their respective routines. They are also involved in preparing Version 1 of the compiler for release later this summer.

1.3 This Manual

One of the purposes of this manual has been mentioned earlier. It is partly for that purpose and partly because this manual may be of some use to persons wishing to add to or modify WATFOR that the manual is very detailed. In fact some of the descriptions presented may become meaningful only if the reader has access to listings of the compiler modules and uses this manual as a guide to the listings. It is not a particularly didactic manual although an attempt has been made to start each chapter with an overview of the module described in that chapter, with chapters 2 and 3 containing an overview of the compiler as a whole.

The reader is advised that a knowledge of both C28-6515 and the WATFOR Implementation Guide is presupposed. This manual pertains to Version 0 Level 5 of WATFOR.

Not the least of the objectives of the WATFOR project, and this is particularly so in an academic environment, is the last listed: experience and learning. We have learned considerably, even about writing manuals.

Finally in keeping with what seems to be a well established tradition for manuals of this sort, every attempt has been made to keep it as dry and humourless as possible.

CHAPTER 2

General Structure

2.1 Introduction

A compiler, of course, has one main purpose; to translate a programme (FORTRAN in our case) to machine language. To accomplish this purpose the WATFOR compiler generates and uses two major tables. The first and most obvious is the object code to execute the user programme. The second, the symbol table, contains information about variables, statement labels and constants used by the programmer. The compiler's functions are basically built around these two tables.

The symbol table contains information about the elements used by the programmer. Since many parts of the compiler use the symbol table it is described in detail in this chapter. The object code is dependent on the particular statement being processed. Hence most of the object code is discussed in Chapter 4 where the statement processors are presented. However, this chapter does present some of the common code generated for each statement.

When writing a compiler many terms are used to make communication between programmers easier and more precise. Some of the common and frequently used terms are introduced now. Most of them will be described in more detail later.

1. Stack

   The FORTRAN source statement is transformed to an internal list (stack) suitable for processing by the various statement processors.

2. Work area

   Since the compiler is completely core resident an area is required to generate object code and symbol table information.

3. Programme Segment

   This is either a main programme or a subprogramme.

4. Local Data Area

   Storage for any constants, statement numbers, address constants, or simple variables in a Programme Segment comprise the local data area for the segment.

5. Array Area

Any variables that are subscripted or appear in a COMMON or an EQUIVALENCE statement are included in the array area.

6. Symbol Table

A number of lists used by the compiler to retain information about statement numbers, variables and constants.

7. Object Code

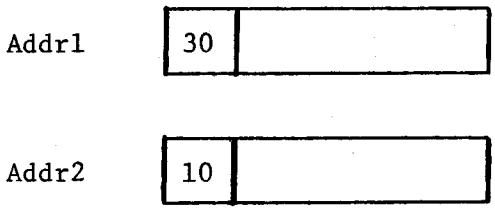
Machine instructions generated by the compiler to execute the user programme.

8. Dot Routine

The manner in which WATFOR handles subscripts requires that each subscripted variable have a routine containing the necessary information to calculate subscripts. This routine is called the dot routine (cf dope vector). (BAL R14,.A would be a way of representing a call to the routine to perform subscripting for the variable A).

9. Linked List

The first half-word of each entry of a table contains a value that is used to point to the next entry of the same type in the table.



e.g. Addr1 + 30 = Addr2
where 30 is called the link.

Most of the major tables use this technique. The advantages include:

1. Ease and speed in handling data of the same type. (Only 1 instruction is required to get the next item in a table.)
2. Allows variable length entries in tables.
3. Creating many different tables is just a sequential operation as long as the proper link is inserted.
4. Implementation on /360 computer is easy.

2.2. Logic Flow of WATFOR

WATFOR is a batch processor of FORTRAN jobs. This allows the installation to run a batch of FORTRAN jobs without any intervention required from the operating system. Hence transition from one job to the next is minimized and this allows speedy processing of the batch. A second and more important reason for WATFOR's speed is that the compiler is load-and-go. No I/O utilities are required or used at compile time and hence we can again gain some valuable time. (The user can of course use I/O utilities in the standard manner, as allowed by the FORTRAN language, at execution time.)

The following diagram shows the basic relationship between the various major steps of WATFOR's compilation and execution process.

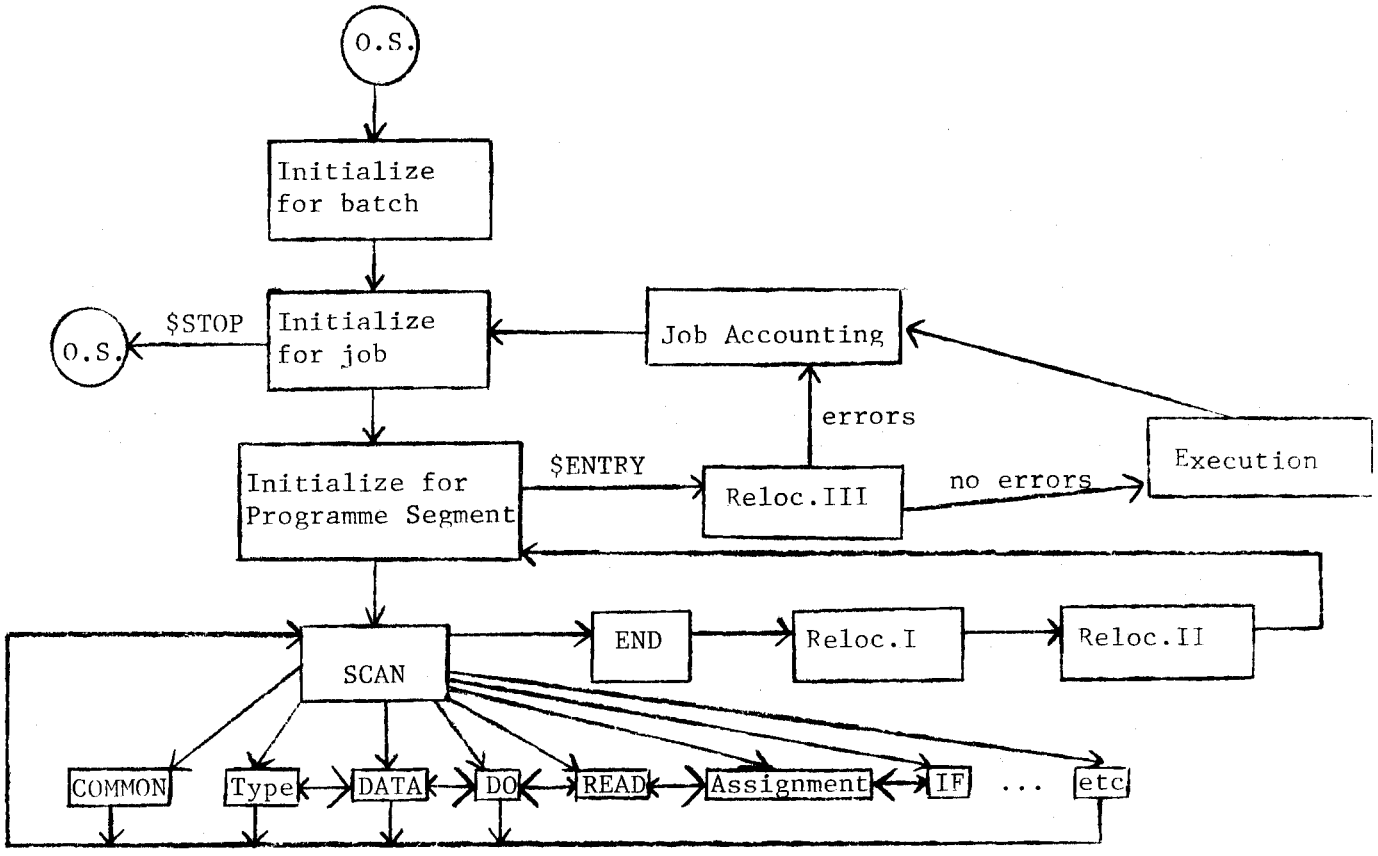


Figure 2.2.1.

SCAN - produces intermediate text and identifies statement
COMMON,DATA, etc. represent routines which compile the various statements. The arrows at this level indicate that some of these routines call one another. The larger arrowhead suggests the call e.g. READ calls DO.

Reloc.I - Relocator Phase I - processes the symbol table and produces programme segment local data areas. Missing statement numbers are detected.

Reloc.II - Relocator Phase II - relocates object code using addresses prepared by Phase I.

Reloc.III - Relocator Phase III - processes Global Symbol list and assigns addresses to equivalenced variables, arrays, common blocks; subprogramme cross references are resolved and missing subprogrammes are detected.

2.3. Memory Layout

The following diagrams show the memory layout for the compiler. The link-editor may re-arrange the order of the various modules. However, the "work area" will always occur following STARTB. The work area consists of two tables at compile time. Object code is produced from low to high addresses and symbol table entries are generated from high to low addresses. This gives us two open ended lists and we need only be concerned if the two lists meet.

Several rules have to be used with regard to the overflow of the work area. If this area does overflow, the programme will not execute under any circumstance. However, in the event of overflow WATFOR will continue compiling the programme, subject to the following rules:

1. If the object code area and symbol table area meet, no more object code is generated.
2. The symbol table, however, still continues to grow downward until it reaches the bottom of the work area. In this case compilation is immediately terminated.

It should be noted that the symbol table is not retained at execution time and hence this space can be used for assigning storage for arrays, common blocks and equivalenced variables.

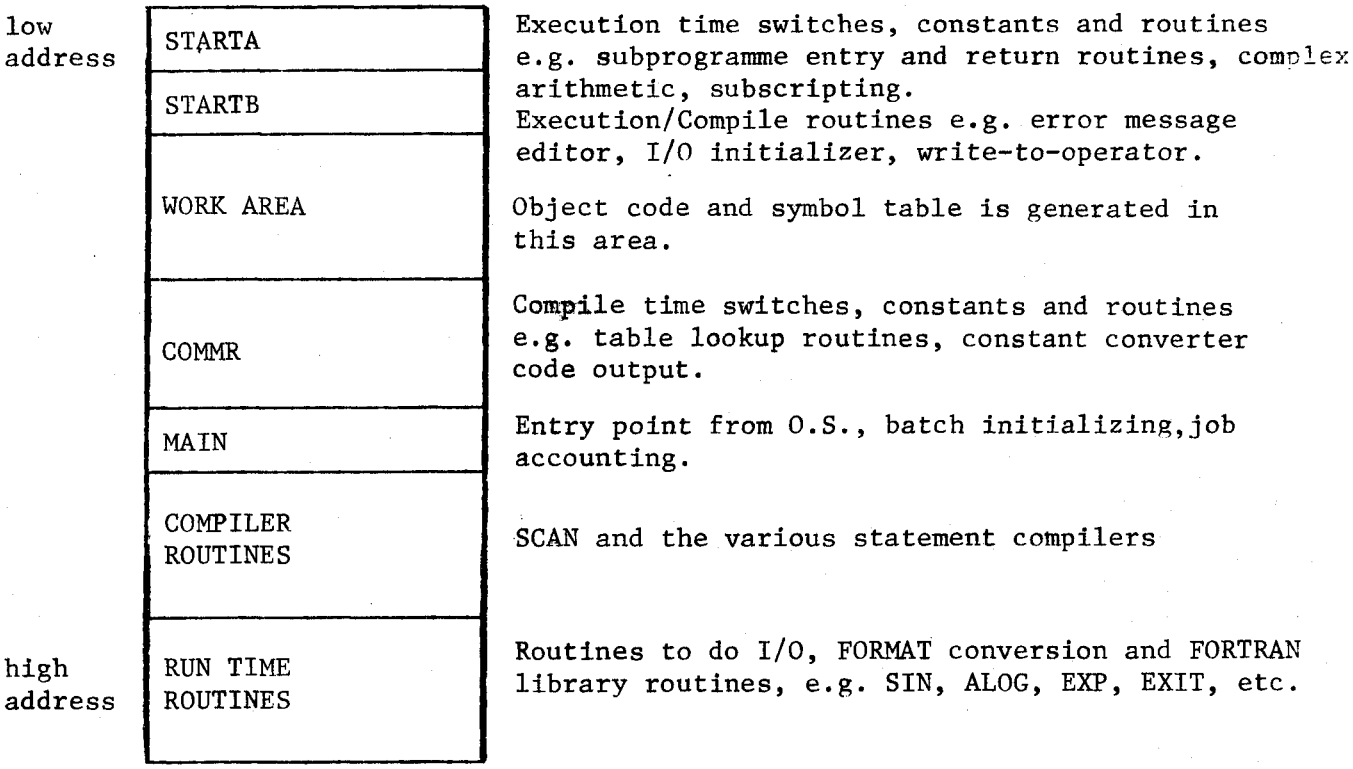

low address
STARTA
STARTB
WORK AREA
COMMR
MAIN
COMPILER ROUTINES
RUN TIME ROUTINES
high address
Execution time switches, constants and routines e.g. subprogramme entry and return routines, complex arithmetic, subscripting.
Execution/Compile routines e.g. error message editor, I/O initializer, write-to-operator.
Object code and symbol table is generated in this area.
Compile time switches, constants and routines e.g. table lookup routines, constant converter code output.
Entry point from O.S., batch initializing, job accounting.
SCAN and the various statement compilers
Routines to do I/O, FORMAT conversion and FORTRAN library routines, e.g. SIN, ALOG, EXP, EXIT, etc.
low
object code for main programme
local data for main
object code for subprogramme A
local data for A
symbol table for A
symbol table for main programme
Compile time layout of work area for
$JOB
Main Programme
END
Subprogramme A
END
$ENTRY
object code for main
local data for main
object code for subprogramme A
local data for A
object code for last subprogramme
local data for last subprogramme
arrays
equivalenced
common blocks
unused
Execution time layout of work area

Figure 2.3.1.

2.4. Symbol Table Entries for /360 WATFOR

The symbol table consists of 6 linked lists which contain information about the various elements used in the programme being translated.

The various lists are:

1. Name List (VLIST) - This list contains all names which are used in a programme segment. They may be of 3 modes:

   (i) Variable names
   (ii) Subprogramme names
   (iii) Common block names

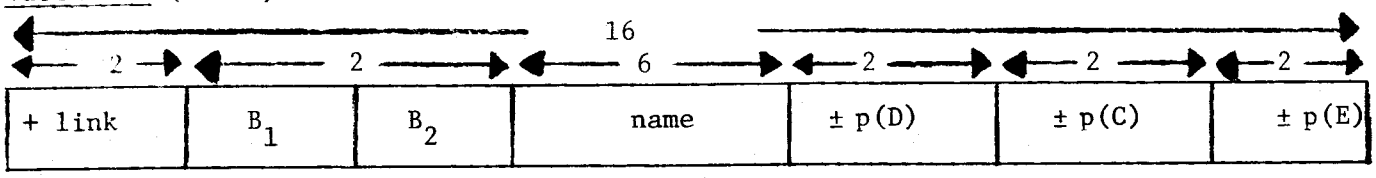
   The various modes of names are put in one list to detect attempted duplicate use.
2. Statement Number List (NLIST) - This list contains all statement numbers mentioned in the segment as well as all pseudo-statement numbers constructed by the DO-compiler for branching back in DO-loops.
3. Constant list (KLIST) - This list contains all integer, real, complex and hexadecimal constants used in a segment.
4. Hollerith Constant list (HLIST) - This list contains references to all hollerith constants not appearing in FORMAT statements. The SCAN routine stores the constants in the object code and prepares the entries in this list.
5. Library List (LLIST) - This list contains 1 entry for each common block name, subprogramme or library function. (i.e. if a name appears in a SUBROUTINE or FUNCTION statement or if a variable is used as a subscripted variable and no "dimension" statement is encountered for the variable, the variable is placed in the LLIST. Entries are placed in this list by Relocator Phase One for each programme-segment when the END statement is encountered).
6. Relocator Phase Three List (GVLIST - global variable) - This list is not a new type of list but rather consists of entries from the VLIST's. It is basically a list of incompleted tasks of the Relocator. It is started at the end of the first programme-segment, and grows at the end of each succeeding programme segment. Processing of the list occurs in Relocator Phase Three at the end of compilation and just prior to execution. This processing includes programme segment linking and assignment of storage for arrays and common or equivalence variables.

Lists 1, 2, 3, 4 are produced on a programme-segment basis and lists 5 and 6 are prepared on a complete programme basis.

The symbol table is created in the work area from high to low addresses. It is required only at compile time and at execution time can be used as space for the array area.

The symbol table as described contains 6 different lists. In designing this table it was decided to use as little space as possible for each type of entry. The constant entry can contain for example from 2 - 5 words depending on the constant type. However, to speed up processing time it was decided to make each type of entry a multiple of full words. The various lists are also consistent in that if a NAME appears in the list it always starts in the second word and the attribute bytes always appear as the second half of the first word. Hence obtaining information from the list is easy and processing the list is relatively fast.

Variables (VLIST)



B_1 & B_2

NI	# SS	Type	L	T	U	P	Do	As	In	C	E

NI - Name Indicator { 11 for arrays; 101 for simple variables

SS - Number of Subscripts: 001 - 111 for 1-7 subscripts

Type:
- 00 - logical
- 01 - integer
- 10 - real
- 11 - complex

L - Length: 0 = standard; 1=optional

T - Type established: 0 = not established; 1=established

U - Usage established: 0 = not established; 1=established
(used to resolve e.g. COMPLEX Z, EXTERNAL Z)

P - Subprogramme Parameter: 0 = not a parameter; 1=parameter

DO - DO-parameter: 0 = not currently a DO-par; 1=currently a DO-parameter

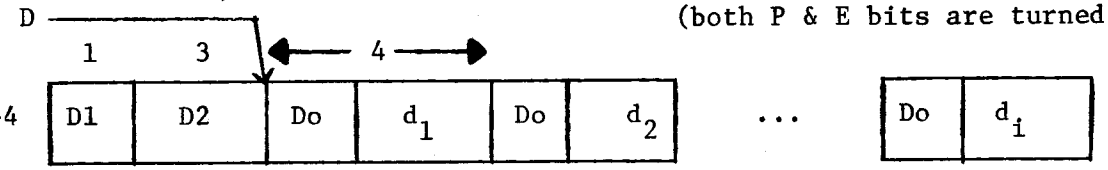
As - ASSIGNED variable: 0 = not an ASSIGNed var; 1=ASSIGNed variable

In - Initialized: 0 = not initialized; 1=initialized in DATA or Type statement

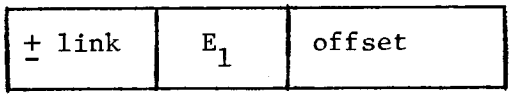
C - COMMONed: 0 = not in common; 1=in common

E - EQUIVALENCed: 0 = not equivalenced; 1=equivalenced
(both P & E bits are turned on for a call-by-name parameter)

DIMENSION Sublist D-4



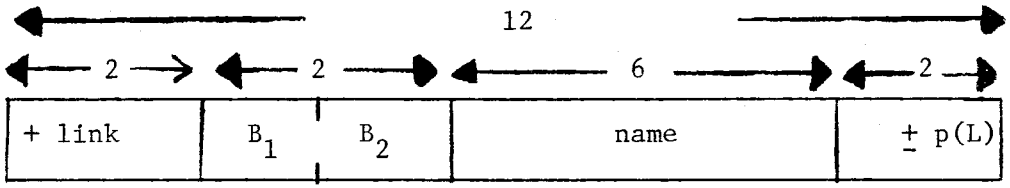
COMMON and EQUIVALENCE Sublists C,E



link points to next element in this equivalence or common list

Do - set to zero
D1 - shifts
D2 - length in bytes
d_i - actual dimensions
E_1 - used by COMMON or EQUIVALENCE

Subprogrammes (VLIST)

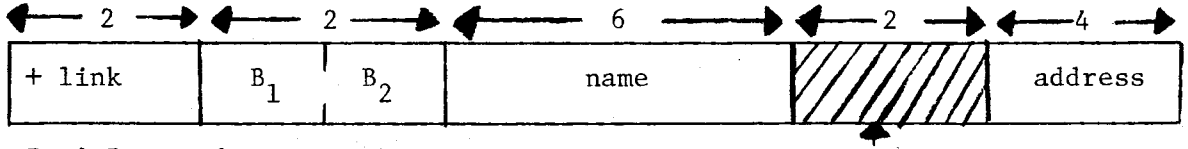


B_1 & B_2

0 0	0 1	F/S	Type	L	T	U_1	P	U_2	C_n	Ex	U_{34}

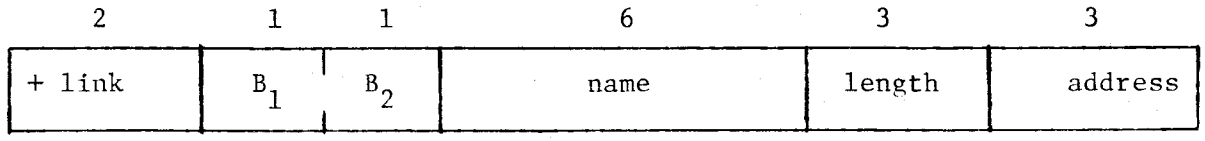
NI - Name indicator = 00 for subprogramme names
F/S - Function/Subroutine 0 = subroutine ; 1=function
Type - 00 - logical
01 - integer
10 - real
11 - complex
L - Length 0 = standard : 1=optional
T - Type established 0 = type not yet established; 1 = type established
U_1 - Usage established 0 = not established; 1=established (NI can/can't be changed)
P - Subprogramme Parameter
Ex - EXTERNAL'led 0 = no; 1=used as subprogramme parameter
Cn - Consistency bit 0 = no; 1=yes (must be externalled in calling programme if P = 1)
U_2 - Usage established 1 =not established; 0=established (F/S can/can't be changed)
U_{34} - Usage 00 = external function; 01=statement function; 10=secondary entry point; 11=primary entry point
p(L) assigned by Relocator 1 and pointer to LLIST

Library Subprogramme List (LLIST)

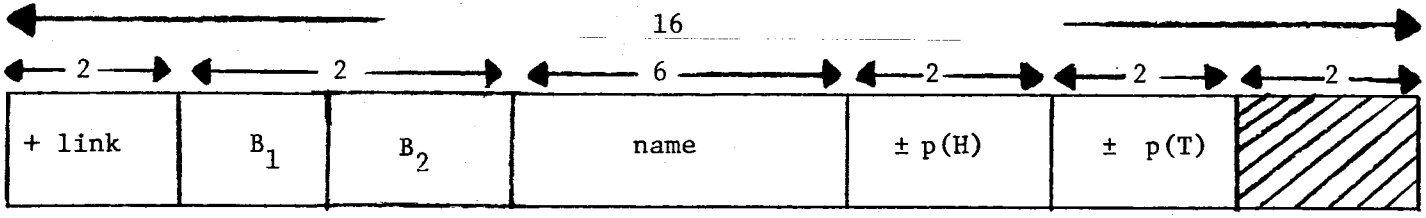


B_1 & B_2 as above

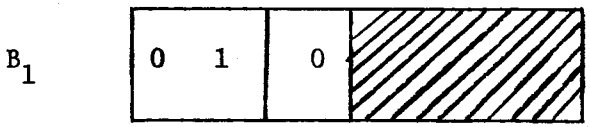
Common Block Name in Library List (LLIST)



Common Blocks (VLIST)

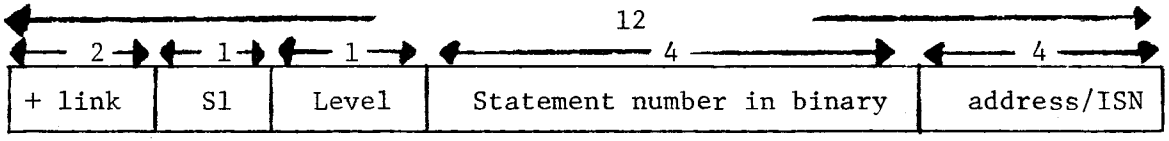


p(H), p(T) are pointers to first and last elements in this block

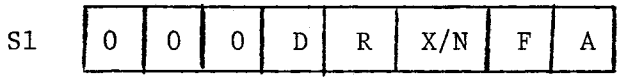


NI - Name indicator = 010 for common block names

Statement Numbers (NLIST)

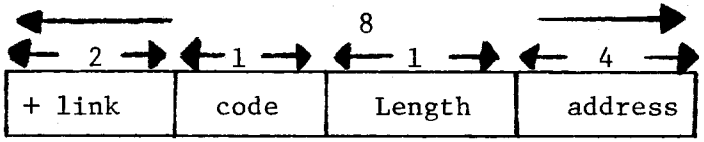


1st ISN reference until address determined



X/N - Executable/Non executable — 0 = non executable, 1 = executable
F - FORMAT — 0 = not a FORMAT, 1 = FORMAT statement
A - Address-assigned — 0 = not assigned, 1 = address assigned
R - Referenced — 0 = statement number not referenced, 1 = referenced in GOTO,DO etc.
D - DO-statement — 0 = doesn't appear in a DO statement, 1 = appears in DO.

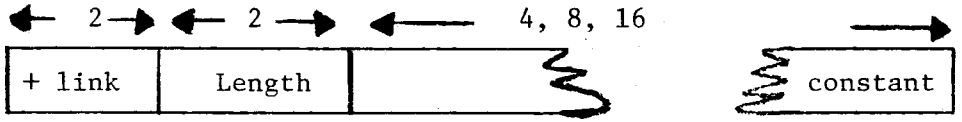
Hollerith Constant List (HLIST)



- this list created by SCAN routine

Length = number of characters -1 (≡3 mod4 as SCAN right pads with blanks to word boundary)
address = location of the stored constant
code = X'80' if constant is in PAUSE or DATA statement
= X'00' otherwise

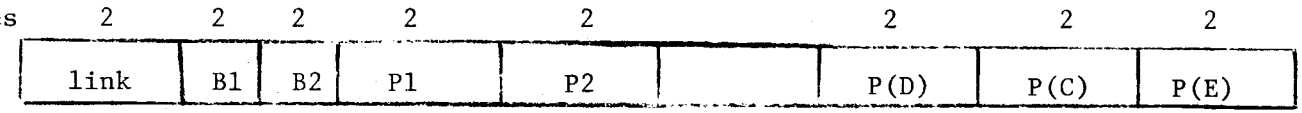
Constant List (KLIST)



Length = number of bytes -1

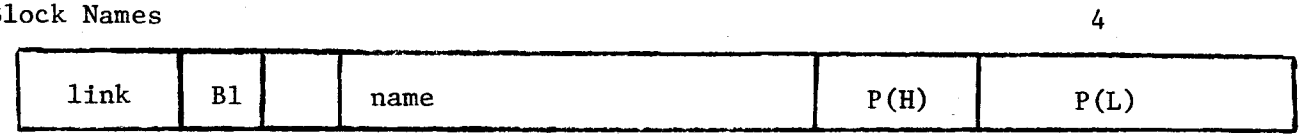
Relocator Phase Three List (GVLIST)

Variables



Subprogrammes

Common Block Names



P1 Pointer to data area

Variable type

simple	-	B/D address
subscripted	-	pointer to dot routine
equivalenced or common or simple	-	pointer to an address constant

P2 Used only if variable is simple and complex - pointer to the second half of a complex variable

P3 Pointer to the address constant in the data area

P(L) Pointer to the entry in the LLIST (offset from START)

2.5. Work Areas

Figure 2.3.1. describes the basic core layout of the work area at compile and execution time. Chapter 4 will describe the object code that each of the statement processors will produce. This section describes some of the more general features of the object code.

Object code is produced by the various processors. This object code is in general, in a coded form. The various phases of the relocator (Phase 1, 2, and 3) change this code from pointers to the symbol table and special codes, to pointers to the data areas. The principle of relocation is discussed in section 2.6.

Certain object code is pertinent to the whole programme and other object code is required for each statement. This code plus some other general facts will be described now.

2.5.1. ISN CODING

The following object code is generated before each executable statement in the object code.

```
BAL     R11,XISNRTN
DC      H'ISN'
```

XISNRTN is a routine in the communications region STARTA which in general returns to the instruction following the half word constant. ISN is the source line number. This coding is used for three purposes.

1. If an error is detected in a statement, register 11 points to the ISN which can be included as part of the error message.
2. Register 11 is used as the programme base register and the BAL R11,XISNRTN guarantees a base register at any statement in the programme. (Unless a statement takes more than 4095 bytes).
3. The routine XISNRTN merely branches back to the object code. If a timer interrupt occurs at execution time, the timer routine changes the branch at XISNRTN to a NOP and then continues processing the FORTRAN statement. (This is done because the I/O routine IHCFIOSH is not serially reusable and hence if an interrupt occurred during an I/O operation all succeeding jobs in the batch would hang.) Hence when we start to execute the next FORTRAN statement control does not transfer back to the object code from XISNRTN but rather issues an error message and terminates the job. XISNRTN is used as a common point through which each executable statement must pass.

2.5.2. Handling of Undefined Variables at Execution Time

The user or installation has the option to allow the compiler to detect undefined variables (variables which have not been assigned a value). Since object code is generated to perform this check, the method used and how this is implemented is now discussed.

The 7040 WATFOR compiler performed undefined variable checking by setting all unassigned variables and arrays to 'bad parity' (i.e. An instruction exists on the 7040 to set a word to bad parity. Any attempt to 'read' this word will cause a machine interrupt and hence undefined variables can be detected.) The /360 computers do not have this facility. However, even if this were possible we decided that using the parity trick was not suitable for a multi-programmed system. Roll-out/roll-in for example would be awkward.

The solution decided upon was to choose a particular quantity and insert it in all unassigned variables. Any routine generating object code to access a particular variable outputs special code (call's to routines in STARTA) to check for the undefined situation (if RUN = FREE or CHECK.) For RUN = NOCHECK no such code is generated and hence we have more efficient code for an experienced programmer or for a bug-free programme.

The particular value chosen is n * X'80' where n is the number of bytes required for the particular variable type. This value is usually not produced under normal calculation. For the four variable types the value is:

floating point	- a very small number	- .4335017E-77
fixed point	- a very large negative number	- 2139062144
alphabetic	- an invalid character	
logical	- an invalid representation of either .TRUE. or .FALSE.	

To date we have found only one way of generating this number easily.

```
e.g.  DATA I/4Hbbbb/
      J = I + I        J is now undefined
```

If problems arise as a result of our choice, the user can always get around these by running under the option RUN = NOCHECK.

In order to give the user as much information as possible when he has an undefined variable, we also print out the variable name. Therefore, the variable name must be available at execution time. (We do not keep the symbol table at execution time.) The variable name is stored immediately preceding the value of variable in the data area for simple variables. For subscripted variables the variable name precedes the dot routine for the particular variable.

e.g. IXXBC = 1

data area

IXXBC⊔⊔⊔	00000001

DO-loop parameters must be positive and hence, if they are undefined, it is easy to determine since the undefined value is negative number.

Not all cases of undefined variables can be detected by a simple comparison against a funny bit pattern. In a certain obscure case, we had to resort to other means to detect incorrect logic. The FORTRAN subroutine following illustrates this case:

```
SUBROUTINE A(/X/)
RETURN
ENTRY B(/Y/)
X = Y
RETURN
END
```

If this subroutine is invoked in the following manner, everything is OK.

```
CALL    A(R)
CALL    B(S)
```

However, if we call it in the reverse order

```
CALL    B(S)
CALL    A(R)
```

we obviously have an error condition, and it is one which cannot be detected by normal means. Subroutine parameters called by name are handled like common or equivalenced simple variables. That is, there is an address constant in the data area pointing to the storage for the variable. In this case, it's the address of the variable which appeared in the argument list of the latest call to subroutine A. Clearly, if we do not call this routine in the correct order, we will store the value R in some random location. This is an impossible situation as we may clobber the compiler. The answer to this dilemma is to force a programme interruption by initializing the address constant to an illegal value. (Specification interrupts could not be used as we might be addressing a logical *1variable.) It was then decided to force a protection or addressing interrupt by storing in the initial value of the address, an address beyond the normal value expected. Hence, we store or load at address 8M bytes unless this address is initialized.

2.5.3. Addressing of Simple Variables Equivalenced or in Common Blocks

The base/displacement addressing scheme of system /360 limits the amount of data storage which can be directly addressed. As a result, simple variables which are in common blocks or are equivalenced must be addressed in a slightly more complex manner. This manner, quite naturally, is to keep in the data area of the subprogramme an address constant pointing to the storage area for the variable. Thus, manipulation of the variable involves first loading its address into a work register (R3) and then doing the required operations. For the sake of simplicity, it was decided to defer assignment of storage space for all such variables until the relocator phase three stage. The reason being that a simple variable might be equivalenced to an array whose storage area is assigned at that time.

e.g.

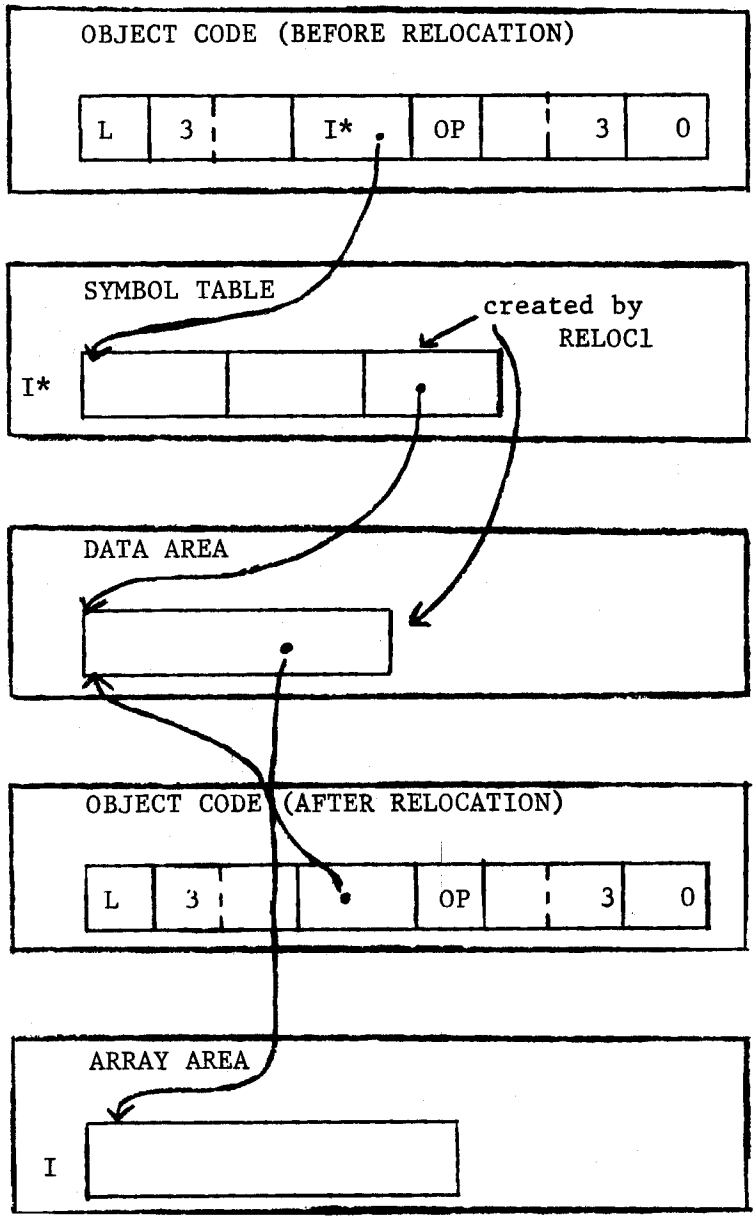


Figure 2.5.3.

2.6. RELOCATION PRINCIPLE

2.6.1. General

At a particular step in the generation of code we have no knowledge of the storage locations that the variables will have assigned to them. However, each variable has an entry in the symbol table. Hence the object code generated contains pointers to the symbol table. The relocator changes these pointers to actual addresses pointing to the data area.

```
e.g.    LE      F0,pointer to symbol table
     → LE      F0,Base/displacement address
```

The major design criterion which influences the form of symbol table pointers is that the relocation phase, (the phase which passes over the code which has been generated by the various statement processors) must be able to decide just what addresses in the code are or are not to be transformed. An examination of the code generated reveals a set of addresses which are not to be relocated. The complimentary set of addresses is used as a basis for the processing of the object code by the relocator.

2.6.2. Symbol Table Pointers

The relocator first determines the opcode of the instruction it is scanning and does one of three classes of things accordingly

1. Ignores the operation (mostly RR instructions).
2. Relocates operand field (mostly RX instructions).
3. Performs special relocation functions.

If an operation falls into class two, the operand address is examined and if it points to a symbol table entry, it replaces the pointer with the base/displacement (B/D) address of the data. Since we are replacing the pointer with a B/D address which occupies a half-word, the pointer can take up no more than a half-word. An examination of B/D address which appear in code not to be relocated reveals the following set of "constant" addresses:

Base register 0 is used in immediate operands
(i.e. LA instruction)
Base register 11 is used for local branching
(i.e. around logical IF statements, entry point code,
Hollerith constants or FORMAT statements)

Base register 12 is used to address routines and data in the execution time communications region (CSECT STARTA)
Base register 13 is used to address the programme segment save area and temporary area.

This leaves base registers 1 through 10 and 14 and 15 unused in the generated object code. Hence we adopt the convention that an address in the object code with a base register number one through ten, is a symbol table pointer. It shall be regarded as a 16 bit positive integer. For each entry in the symbol table, its pointer is calculated by subtracting from its address, a constant value (for each subprogramme) stored in location CSYMBASE. This constant is used later in reconstructing symbol table addresses. The pointers allow us to think of the symbol table as running from addresses X'AFFF' through X'1000' (remember that the symbol table is filled downwards).

2.6.3. Temporary Pointers

Early in the writing of the arithmetic statement compiler, it was decided to separate the fixed and floating temporaries. This was primarily because the registers are separate. Moreover, temporaries would come from separate stacks simplifying the temporary assignment algorithm. Both temporary stacks are addressed by register 13 at execution time. However, the full length of the fixed temporary stack is not known until the END statement. So their addresses cannot both be assigned at statement compile time as an offset from the beginning of register R13. Register R14 (one of the still unused registers) was picked to indicate the floating point temporary stack and will be considered to be aligned on a double-word boundary. Note that the fixed register stack is aligned on a word boundary. At relocation time, if an address refers to the floating point temporary stack, it is transformed into the B/D address by the simple subtraction of a constant (calculated when the storage for the temporary stack area is assigned). Its value then is register R13 plus some displacement.

2.6.4. Relocator Codes

Since the WATFOR compiler is based on a principle of partial machine code, partial interpreted code, the relocator (phase two) must be able to cope with both. As phase two takes its cue from the opcode of the object code being scanned, a section of illegal /360 opcodes (henceforth called relocator codes) was set aside to represent interpreted code. These opcodes start at X'A0'.

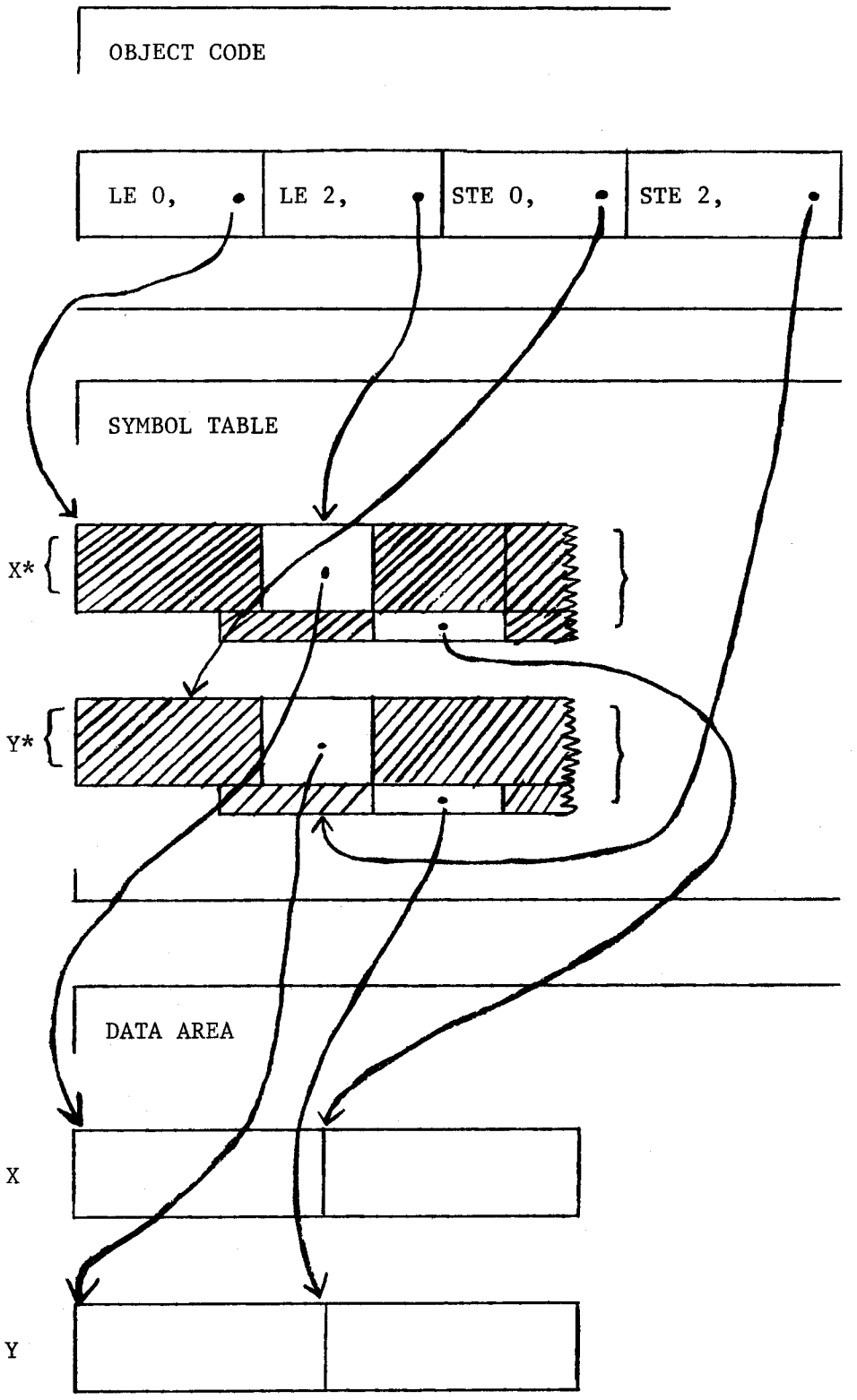
For example, these opcodes are used in argument lists generated by various statement processors: subroutine calls and the corresponding model argument lists, calls to dimensioning routines, and calls to input/output routines. Another example of the use of a special relocator code appears in the code generated for an internal statement number. Here the relocator must perform special tasks upon scanning this code (to be explained later). We also have a set of codes set aside for similar operations but which are not "interpreted" at execution time. These will be explained in more detail later.

2.6.5. Addressing of Complex Values

In the production of object code handling complex simple variables (not in common or equivalenced) and constants by the arithmetic statement processor, we must be able to access at execution time both the real and the imaginary parts of the value. It would be impractical to set up two separate symbol table entries for each variable or constant. An examination of the way that phase two of the relocator works suggested an easy method of solving the above problem. Let us suppose that we have an RX instruction with its operand field pointing to the symbol table. Phase two uses this pointer to find the symbol table entry. The pointer is then replaced by the half-word base displacement address four bytes beyond the start of the symbol table entry. Notice that not only does this address point to the whole variable, but it also points to the real part of the variable. We have only to do a similar operation for the imaginary part. This second symbol table entry would be offset from the first by one half word so that the base displacement addresses would not overlap. Another way of saying this is: The pointer to the real part is the pointer to the variable, the pointer to the imaginary part is the pointer to the variable plus two bytes.

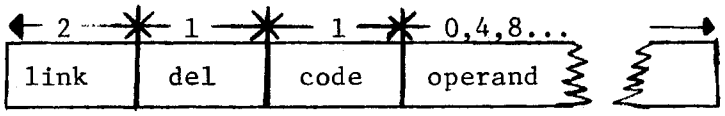
For example: Y = X where both are COMPLEX *8

Figure 2.6.3.



2.7. Intermediate Text Created by SCAN Routine (Stack)

All FORTRAN statements, with the exception of FORMAT statements are transformed into a form that is more useable by the various statement processors. The text is a linked list of operator-operand pairs and is an internal representation of the original source statement. For convenience we will refer to this list as the 'stack'. The entries are link, delimiter (operator), code and operand, the last of which may be omitted. Pictorially we have



Following are two examples of FORTRAN statements and their corresponding stack entries:

X = ABCDEF + Z*3.9

(a)

8	⊣	01	Xbbb
12	=	02	ABCDEFbb
8	+	01	Zbbb
8	*	81	$1^{(3)}_{2}$
8	.	81	$1^{(9)}_{2}$
0	⊢	00	

X = ((A+B**3 - 1234567890))

(b)

8	⊣	01	X b b b
4	=	00	
4	(	00	
8	(	01	A b b b
8	+	01	B b b b
8	**	81	$1^{(3)}_{2}$
12	-	82	$8^{(12345678)}_{2}$ $2^{(90)}_{2}$
4	)	00	
4	)	00	
0	⊢	00	

The stack is located at the end of STARTA. For a description of how the stack is generated see section 4.1.

A detailed description of the various entries follows:

(a) The link, a two byte constant, is added to the address of the current entry to obtain the address of the next entry of the stack.

(b) The operators appear in coded form as shown in table 4.1.1. Since it is possible for two operands to follow each other the use of a null operator is required. The operator requires 1 byte per entry.

e.g. INTEGER *2 A, B, C,

0008	⊣	02	INTEGERb
8	*	81	$(2)_8$
8	Φ	01	Abbb
		etc.	

(c) The code consisting of 1 byte is subdivided into two 4 bit parts. The first part is 0, 2, 4, or 8 corresponding to the operand being a symbol, hollerith constant, logical constant or numeric constant. The second part gives the length in words of the operand field.

(d) The operand field can contain one of five entities:

(i) Empty. Two consecutive operators appear in the source statement.

(ii) Symbol. Any set of alphanumeric characters whose first character is alphabetic is considered a symbol. These are padded on the right with blanks to make operand field a multiple of full words.

(iii) Numeric Constant. Any set of 8 or less consecutive digits is converted to binary and placed in a word in the operand field. The same rule is applied if more digits follow and the result is appended in the operand field (see example (b) above). The five high order bits of each word so formed contain the digit count for the particular constant.
e.g. $12345 - {}_5(12345)_2$

(iv) Hollerith Constant. These constants are removed from the source text, stored in-line in the object area, and a symbol table entry is generated. The operand field contains a pointer to the symbol table entry for the constant.

(v) Logical Constant. The operand field contains a pointer to one of two cells containing the values of .TRUE. and .FALSE. (These cells are located in the communication region STARTA).

Notes:

(a) With the exception of Hollerith constants, all embedded blanks are removed from the source text and do not appear in the stack.
(b) In the case of FORMAT statements the actual FORMAT source statement is used by the processor.

2.8. Conventions

Various conventions were used when writing the compiler. These were done so that changes and additions could be made easily.

2.8.1. Register Conventions

Most routines obey the following conventions for register usage.

(a) Compile-Time

Register	Use	Value
R0	work	relative
R1	work	absolute
R2	work	relative
R3	work	relative
R4	work	relative
R5	ptr to object code	relative
R6	ptr to symbol table	relative
R7	work	relative
R8	work	relative
R9	work	absolute
R10	ptr to compile communications area	absolute
R11	programme base register	absolute
R12	ptr to START area	absolute
R13	ptr to current save area	absolute
R14	return address and work	absolute
R15	work	absolute

At our initial planning of the compiler we wished to have some register scheme that would allow us to relocate both the compiler

and the object code generated. Hence, technically we need change only the absolute registers to take care of the new load point. Obviously certain other things (saveareas etc.) have to be modified. After attempting to obey the above rules several instances arose where programming became awkward and hence inefficient. At this time it was decided to drop this idea. However, in general whenever possible the compiler obeys the above register rules.

(b) Execution Time

Register	Use	Value
R0	arithmetic work register (function result)	relative
R1	arithmetic work register accumulator result	relative
R2	subscripting and work	relative
R3	work and subscripting	relative
R4	subscripting	relative
R5	pointer to data area	absolute
R6	pointer to data area	absolute
R7	pointer to data area	absolute
R8	pointer to data area	absolute
R9	pointer to data area	absolute
R10	pointer to data area	absolute
R11	programme base register	absolute
R12	pointer to START area	absolute
R13	pointer to save area	absolute
R14	linkage and indexing	absolute
R15	work and dimension	absolute
F0	work register and function result	
F2	work register and result for complex	
F4	work register	
F6	always set to zero on entry at execution time (used for conversion to double precision)	

2.8.2. Prefix Conventions

Most major routines of the compiler use a prefix naming convention. (i.e. all labels are prefixed by a different letter.) Several sections do not follow this rule vigorously. This resulted from code being moved from one section of the compiler to another.

The reason for writing the compiler using the prefix convention was to facilitate the possible merging of decks and to minimize errors due to duplication of names. Following is the convention used:

A	Arithmetic Statement Processor
C	Compile time communications region
D	DO-statement processor
G	GOTO statements processor
F	FORMAT statement processor
I	Input/Output statement processor
L	Linkage statement processor
M	MAIN - the supervisory programme
R	Relocator routine
S	SCAN routine
T	Type declaration processor
X	Execution time communications region
$	Macro generated symbols
Z	Dsects and definitions

2.8.3. Programming Conventions

WATFOR uses the rule (O.S.) that register 13 points to a "save-area" for storing registers. Also as already mentioned it was decided to use register 11 as a base register for processor routines. Hence if the save area is placed approximately 4096 bytes from the start of the routine register 13 can also be used as a base register for the processor. This is particularly convenient if a processor has more than 1 entry point and if some of the entry points require common service routines. These routines can be placed following the save area and hence are addressable from any entry point.

It should be noted that the macro CENT and CRET are used to provide addressability and linkage for processor routines (see 3.2.)

```
ENTRY1    CENT    SAVE
          .
          .
          .
          BAL     14,CHECK
          .
          .
          .
ENTRY2    CENT    SAVE
          .
          .
          .
          BAL     14,CHECK
          .
          .
          .
SAVE      DS      18F
CHECK     EQU     *
          .
          .
          .
```

3 Communication Regions

3.1.1. Introduction

When writing a programme consisting of many subroutines, common information is required by many of the routines. FORTRAN, for example, has the COMMON statement to accomplish this purpose. In the case of a compiler it is also necessary to have a number of 'utility' routines to perform various common tasks. These routines are in general small, with well defined inputs and outputs, and hence it becomes very time and space consuming to go through a normal calling and return sequence to perform the required task. Hence, WATFOR has adopted the technique of a communication region (nucleus?) quite commonly used in larger programmes. The /360 addressing scheme permits us to address these areas by means of a general purpose register which covers these regions with the usual restriction that only 4095 bytes may be addressed.

Since the compiler has two main steps, compile and execution, we have set up two corresponding communication regions. The portion of code referred to as COMMR contains compile time routines and areas and general purpose register 10 points to the beginning of COMMR. The control section STARTA contains routines and areas for execution time and is covered by register 12. It became evident, as our planning and coding progressed, that STARTA would be too large to fit in the required 4095 bytes. We also did not want to "steal" another register and hence we set up the "extended" execution communication region which will be called 'STARTB'. Most of the longer and less used routines were placed in this area. However, they were still linked to via the STARTA area.

e.g. Call to a routine in extended start area

```
        BAL     R14,XIOINIT
        -
        -
        -
```

In STARTA

```
XIOINIT     L     R15,=A(XIOBASE)  provide base register
            B     PIOINIT
```

In the extended start area

```
PIOINIT     EQU   *
```

The first instruction appears in the object code generated for an I/O statement. The second set of statements appear in STARTA and they link to a routine in STARTB after setting up a base register.

Now each of the WATFOR's routines can access COMMR, STARTA and STARTB as long as these communication regions are assembled as part of the particular routine requiring them. (The communication regions are stored in our macro library and included by means of the COPY instruction.)

Several problems arise as a result of the method of implementing communication regions. Assembly time for each routine including the regions is considerably longer. However, more important is that if the regions STARTA or COMMR are changed in size (i.e. instructions inserted or deleted) any routine containing these regions, must be reassembled. This means that basically the whole compiler must be reassembled. (It should be noted that Version one of the compiler has reorganized these regions to cut down on assembly time.)

We feel that the increased speed at execution and compile time as well as reduced storage compensates for the extra assembly time required. Finally, since the assembler allows a maximum number (small) of EXTRN's and ENTRY's in a routine, our solution is one of the best feasible.

The reader will find the description of the routines and areas that follow quite detailed and hence a brief summary of the major routines is now presented. This hopefully will allow the reader to skip the details until they are required.

COMMR

CENT	Entry sequence for each of the statement processors. It provides addressibility and sets up the save area linkage.
CRET	Return sequence for each of the statements processors.
CLSTN CLPSTN	Determine if a statement number has occurred or if this is its first occurrence by searching the symbol table. The address of the symbol table entry is returned as output.
CLSYM CLBCOM	Same as above except for variable names.
COLINTGR COLCONST CONTEST	These routines accept as input a constant in the stack and determine the type of constant (real, integer etc.). The constant is then entered in the symbol table.
COUT	Output object code determined by statement processor to the work area. This routine checks if the work area is full.

CNOP's, CDSO's — Several routines to align the object code on a particular boundary by means of NOP's or just spacing over.

CREAD — This routine reads card images at compile time.

STARTA and extended area (STARTB)

XENT's — Entry sequence routines for subprogrammes. This includes passing of arguments and checking argument types.

XRET's — Return sequence routines for subprogrammes. This includes returning of arguments and restoring registers.

XA1, XAN — Subscripting routines. The routines calculate the position of a subscripted variable in a vector or array.

XIEXPI, XEXP's — Exponential routines. If exponentiation is required a call is made to one of these routines.

XCMULT's, XCDIV's — Complex Arithmetic Routines.

XFLOAT's, XFIX's — Fix and Float Routines.

XROUTE's — Undefined variable checking routines. The user can specify if he wishes to have extra code generated to check for undefined variables and these routines are used.

XERRENT's, XTRACEBK, XERRPROC — Error processing routines. These routines are used both at compile and execution time to process and output error messages.

XIOINIT, XARRAY, XSIMPELT, XSUBSELT, PIOINIT — Execution time I/O routines. These routines act as an interface between the object code, the I/O conversion routines (FORMCONV) and the I/O routine (FIOCS).

XDATA, PDATA — Data initialization routines. These routines are used in conjunction with object code generated for DATA statements.

3.1.2. Constants and Switches in STARTA and COMMR

Many of the constants and switches used will be described in the various processor descriptions as well as in descriptions of routines in the communication regions. However, some of these are used by many of the routines and others are used in a more general context. These are described now.

STARTA

XUNDEF	Contains the special bit setting for undefined variable checking. (i.e. n*X'80')
XSAVER	MAIN's save area. MAIN 'calls' the user programmer and if any errors occur we can always recover registers from a fixed addressable place.
XENTRYP	Used as the entry point to the user programme (M/PROG).
XENTRYPD	Links to the object code for the first data statement or XENTRYP if no data statements.
XTRUE	Logical value of .TRUE.
XFALSE	Logical value of .FALSE.
XZERO	An integer constant value zero.
XBEGDATA	Address of the start of the array data area.
XENDDATA	Address of the end of the array data area.
XREADS, XPRINTS	Register save area when I/O is in progress.
XERRSWT	Switch set on if a fatal error has occurred during compile time.
CIHGACRD	I have got a card switch. Turned on if we wish to save the last data card in the buffer.
XEXECSW	Compile/execution switch on if we are in execution, off, if in compile.
CUNDEFSW	Undefined variable switch. Used to generate undefined variable checking object code.
XLENTAB	Type as length table. A table of standard and non standard (REAL*4 as REAL*8), and the associated length for each type.

XCARD1, XCARD — Source card image is stored here along with line number for output purposes.

XINPDSRN, XOUTDSRN, XPUNDSRN — The I/O routine FIOCS requires a data set reference number to identify the particular unit required for I/O These constants are pattern DSRN's for the reader,printer and punch units used for compile time I/O. Each consists of 3 words containing the unit number,the end-of-file return and the error return.

XSSMASK, XEDMASK — The constants are masks used by the error processor and the accounting routine to output constants.

XCONTROL — This contains the control character (&CONTROL) and is used for checking purposes in CREAD.

COMMR

CDONO — Each DO statement is assigned a unique sequential integer value. CDONO contains the current value.

CANXTMP — Two consecutive words containing the maximum number of integer and floating point tempories required for a programme segment.

CPRG — Address of the beginning of object programme for a programme segment. This is initialized by LENDPROG and Relocator phase 2 uses this constant.

CSYMBASE — This contains the "magic" constant for calculation of pointers from symbol table entries to data areas and vice versa. It is also initialized by LENDPROG for each programme segment.

As already mentioned the symbol table consists of a number of linked lists. In order to add new elements to a particular list we require a pointer to the last element in list. Upon addition of a new element this pointer is updated to account for the new entry. For completeness a pointer is also required to the start of each list. The following constants are used as start and end of list pointers for the various lists.

CFBEG, CFEND — Function list pointers.

CKBEG, CKEND — Constant list pointers.

CSBEG, CSEND — Statement number list pointers.

CVBEG, CVEND — Variable list pointers.

CHBEG, CHEND — Hollerith list pointers.

CLBEG, CLEND — Library list pointers.

CGVBEG, CGVEND — Global variable list pointers (Relocator phase three list).

CURSTNO — Value of the statement number of the statement being processed. It is set negative if no statement number is present.

CSTNOLK — Pointer to the symbol table entry for the current statement number determined by SCAN. It is set to zero if no statement number is present.

CENTRYPD — Pointer to the object code of the last data statement encountered.

CSRT1, CSRT2 — Pointers to the first and second level zero right brackets in the stack. (A level zero bracket implies that if another right bracket is encountered before another left bracket is encountered we have unmatched brackets.)

CSN — Current ISN saved here for error messages involving statement numbers.

CMOSWTCH — Memory overflow switch. It is turned on (X'80') if the object code area has met the symbol table area.

CSRSWTCH — Switch to determine the stage of compilation having the following settings.

X'00' - We are at the beginning of a programme segment (no entry code generated as yet).

X'94' - We are in a main programme.

X'91' - We are in a subroutine subprogramme.

X'98' - X'9F' - We are in a function subprogramme.

X'92' - We are in a block data subprogramme.

CIFGOTSW — Each transfer statement processor turns on this switch and it is used to determine the ST-4 error. Also used to determine what kind of code must be generated for ENTRY and END statements.

CMODESWT — Keypunch mode switch. Set X'00' for 026 keypunches, X'06' for 029 keypunches and X'0C' for subprogramme entry. This last value is used to maintain the current mode.

CIMLT — Table used to determine and set the type of variables according to their first letter.

3.2 Entry and Return Routines CENT, CRET

The purpose of these routines is to perform compile time saving and restoring of registers. All the compile time processors use these routines for consistency.

CENT

This routine is invoked at every point in a processor routine. In general the macro CENT is used at these entry points

e.g.
```
ENTRYPTA    CENT    SAVEAREA
```

which generates code as follows:

```
            CNOP    0,4
ENTRYPTA    STM     R14,R11,12(R13)
            BAL     R11,CENT
            USING   *,R11
            DC      A(SAVEAREA-XTART)
            USING   SAVEAREA,R13
```

On entry to CENT register 11 points to an address constant pointing to the processor save area (new save area) while register 13 still points to the calling programme's save area. (Old save area) CENT now sets register 13 to link save areas as specified by O.S. convention. Control now returns to the instruction following the address constant. Note that register 15 is used as a work register by CENT.

CRET

This routine is used when a processor has completed its task and wishes to return. This is done by issuing a CRET macro instruction which generates a B CRET. CRET restores the contents of registers 13, 14 and 11 which were in effect when the programme was called and returns via register 14. In cases where other registers have to be restored this is done by the "calling" programme upon return.

3.3 Symbol Table Lookup Routines (CLSTN, CLPSTN, CLSYM, CLBCOM, CLLIB, COLCONST, COLINTGR, CONTEST, CONLOOK)

3.3.1. Introduction

There are basically four routines for creating symbol table list entries for quantities appearing in a programme and all four have roughly the same logic. For example, they distinguish between new and old symbols and return two pointers to the calling programme. (Exception - the constant lookup does not distinguish between first and subsequent appearances of the same constant.)

All routines are called via the LOOKUP macro and (usually) assume that R1 points to the stack entry where the quantity to be looked up may be found.

For example, execution of the statement

```
LOOKUP CLSYM,NEW,OLD
```

would call the symbol lookup routine CLSYM which assumes that there is a name at 4(R1) in the stack. Return is to label NEW if this is the first appearance of this symbol in the programme or to OLD if it has appeared before. The purpose of this is to allow error checking on the symbol. Upon return R15 contains the address of the symbol table entry for the name looked up and R3 will contain (R15-START) i.e. a value which is frequently used in forming object code involving this symbol.

The following table lists by function with entry points the routines which will be described below.

Statement number lookup	- CLSTN - CLPSTN
Symbol lookup	- CLSYM - CLBCOM
Library list lookup	- CLLIB
Constant collector and lookup	- COLCONST - COLINTGR - CONTEST - CONLOOK

3.3.2. Statement Number Lookup

This routine creates symbol table entries in the statement number list for statement numbers (CLSTN) and pseudo-statement numbers (CLPSTN) used in the source programme being compiled. A pseudo-statement number is a statement label created by the DO-compiler for the first executable statement following the DO-statement. This is used for branching back to repeat the loop at execution time. Pseudo-statement numbers are distinguishable from real statement numbers since the former are integers greater than $10^5 > 99999$

```
e.g.          DO1  I=1,5
          p   X = I
              -
              -
              -
          1   CONTINUE
```

The DO-compiler constructs a pseudo-statement number 'p' (> 99999) for the statement X=I to insure that an adcon is created for branching to this statement for loop replication.

The logic of this routine is outlined by Figure 3.3.1. The symbol 'n' is used in the chart and following description to stand for the statement number being processed.

The number to be looked up is in the stack at 4(R1). At entry point CLSTN a check is performed to verify that the number is less than or equal to 99999 using the digit count bits set up by SCAN for numeric stack entries. An entry is created by subtracting 12 from R6, the symtab bottom pointer, and if this does not overflow the work area, n is stored in it. If this is the first statement number of the programme segment, the statement number list head pointer CSBEG is set by storing R6 in it and new number processing is performed before returning to the caller. (A switch is set by the programme segment initializer to indicate that the list is empty; this switch is reset on the first entry to CLSTN.)

If this is not the first call to CLSTN, the new entry for n is linked into the previous end of list and the list is searched for n starting at the head. If n is found at the end of the list it is new; if not, old. For an old number, the entry created at the end of the list is freed by adding 12 to R6. The routine returns to 4(R14) with registers R3, R15 set as described above.

For a new number, the list end pointer CSBEG is updated to current value of R6 and the test bit, DO-level and ISN fields are initialized in the new entry. Return is to 0(R14) with R3, R15 set. (The DO-level and ISN fields are used for checking for branches into DO-loops and in error messages which involve n. See descriptions of DCSTN1, DCSTN2, Section 4.5.6.)

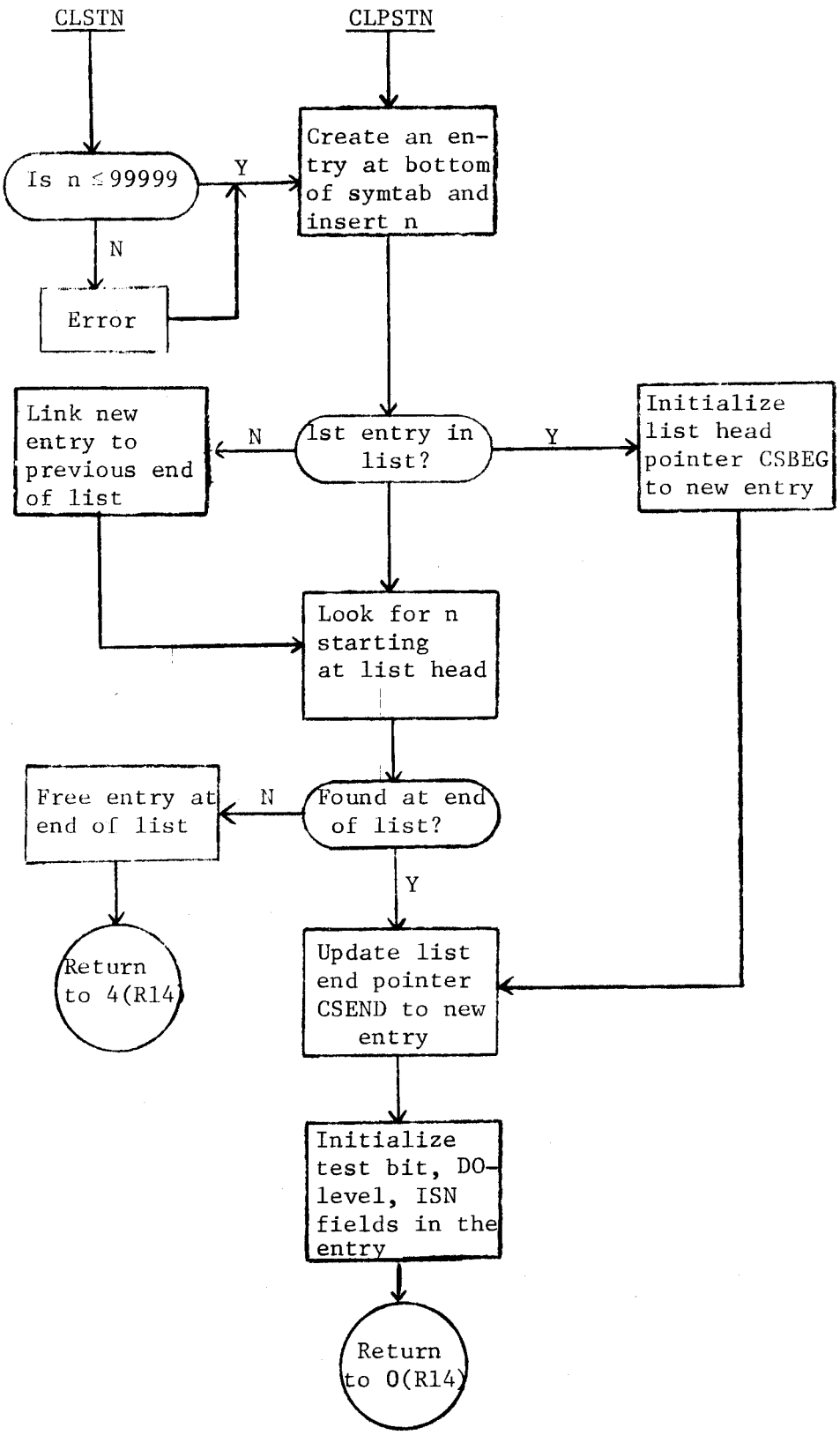

CLSTN
CLPSTN
Is n ≤ 99999
Y
N
Error
Create an entry at bottom of symtab and insert n
Link new entry to previous end of list
N
1st entry in list?
Y
Initialize list head pointer CSBEG to new entry
Look for n starting at list head
Free entry at end of list
N
Found at end of list?
Y
Return to 4(R14)
Update list end pointer CSEND to new entry
Initialize test bit, DO-level, ISN fields in the entry
Return to 0(R14)

Figure 3.3.1.

3.3.3. Symbol Lookup

The routine CLSYM creates entries in the symbol list for all variable, subprogramme, and common block names appearing in a programme segment. The blank common block is handled by WATFOR as a named common block with name '//bbbb' and the entry point CLBCOM is provided for looking up this name in the symbol list.

The logic of CLSYM is virtually the same as CLSTN so the Figure 3.3.2. should suffice with elucidation of the following differences. Entry point CLSYM checks for a name of 6 or fewer characters using the stack operand field length count set by SCAN. Longer names are truncated to the leading six with a warning message. Symbol table entries are 16 bytes. New symbol processing is as follows:

- dimension, common, equivalence pointer fields and the B2 bit field are zeroed out.
- the type of the name is set by its first letter using the first letter/type table CIMLT.
- the mode of the name is set as simple variable.
- mode and type indicator bits are stored as the B1 field of the entry

  i.e. a new symbol is assumed to be a simple variable of type determined by first letter. This assumption might be overruled by the calling programme by modifying the B1 bits.

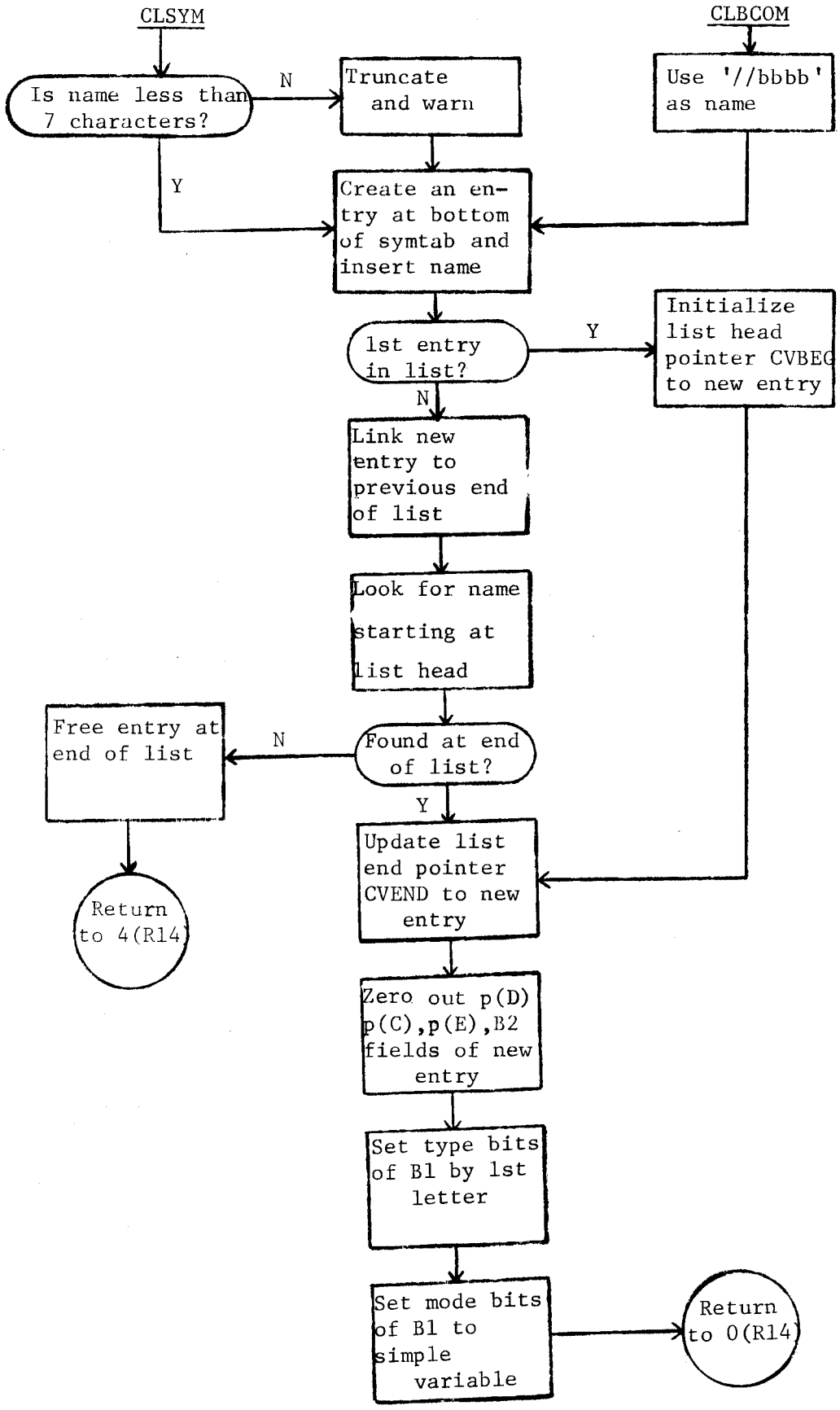

CLSYM
CLBCOM
Is name less than 7 characters?
N
Truncate and warn
Use '//bbbb' as name
Y
Create an entry at bottom of symtab and insert name
1st entry in list?
Y
Initialize list head pointer CVBEG to new entry
N
Link new entry to previous end of list
Look for name starting at list head
Free entry at end of list
N
Found at end of list?
Y
Update list end pointer CVEND to new entry
Return to 4(R14)
Zero out p(D) p(C),p(E),B2 fields of new entry
Set type bits of B1 by 1st letter
Set mode bits of B1 to simple variable
Return to 0(R14)

Figure 3.3.2.

3.3.4. Library List Lookup

The routine CLLIB is used only by Relocator Phase I to enter subprogramme names encountered in the symbol list into the global library list for processing by Relocator Phase III. Thus, the input to CLLIB is not a pointer to the stack but a pointer, in R8, to the symbol list entry to be processed. The logic is the same as CLSYM but simplified since there is no need to check the name length and no field initialization is done for new entries in the library list. See Figure 3.3.3.

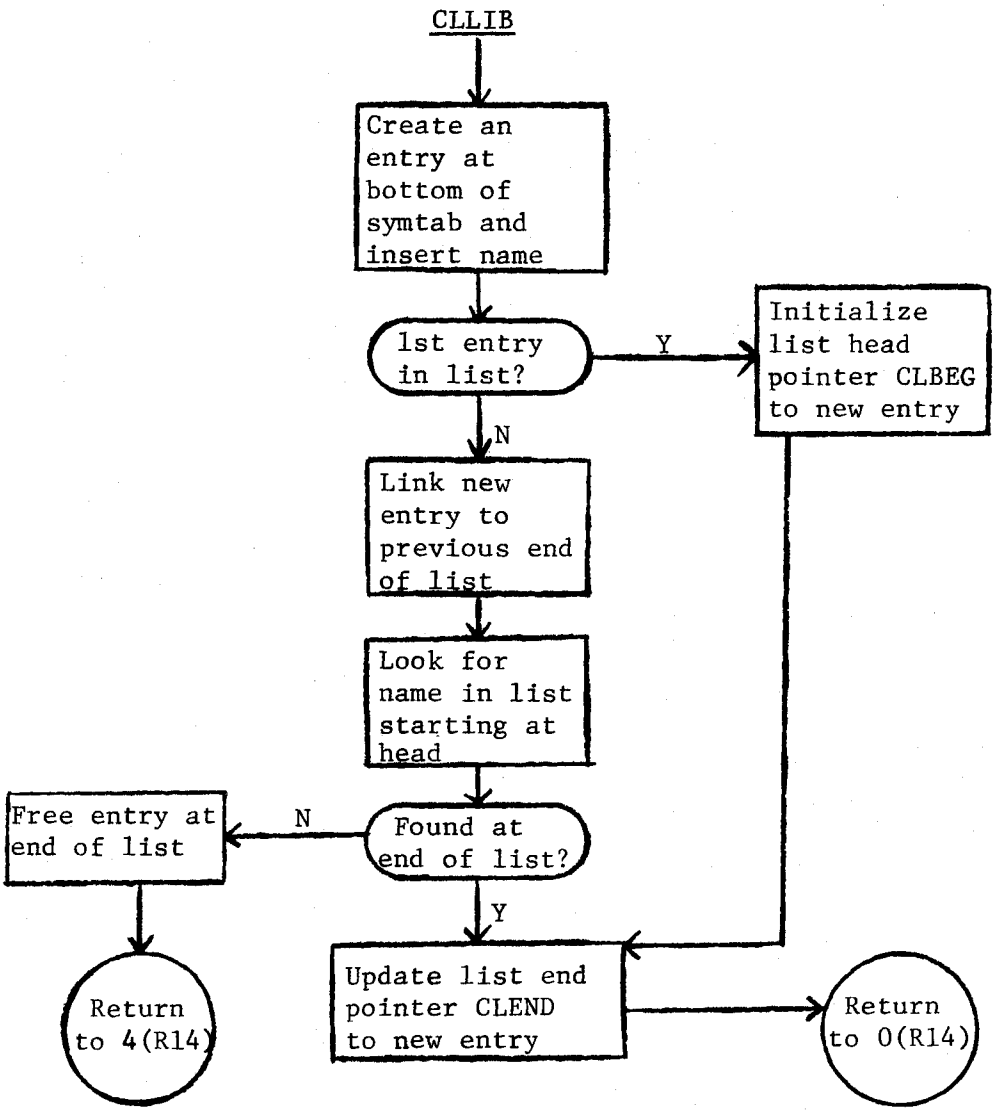


Figure 3.3.3.

3.3.5. Constant Collector and Lookup - COLINTGR, CONTEST, COLCONST, CONLOOK

Since the SCAN routine was designed, basically, to split incoming source statements into operator-operand pairs, no attempt was made to have it recognize FORTRAN constants. As mentioned before the SCAN routine merely converts numeric strings to binary fields with digit count in the stack for later processing. Hence a special routine is necessary to collect together the source components of constants separated by SCAN (e.g. sign, integer, fractional,and exponent parts) into /360 internal form and to place them into the symbol table constant list.

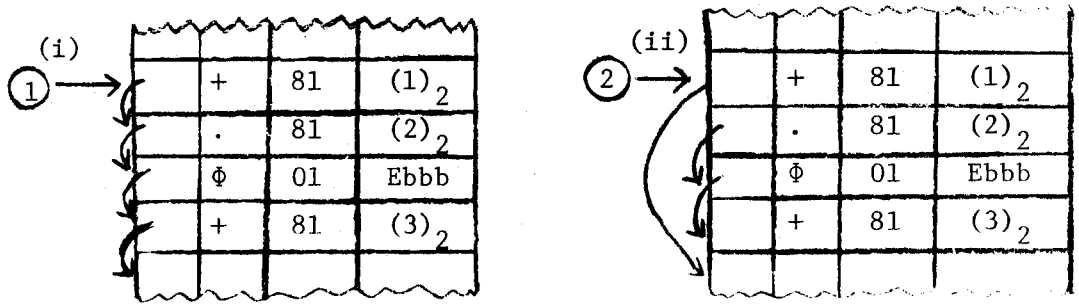
There is essentially one routine which does this conversion with three entry points provided to control its processing. The entry points are:

COLCONST to collect any integer, real or complex constant
COLINTGR to collect an integer constant
CONTEST to test for and collect any integer, real or complex constant. (An error return must be provided in the call to CONTEST should no constant be found in the stack.)

The input to each entry point is a pointer to the stack in R1 where collection of the constant is to begin. Outputs are relative and absolute pointers to the symtab entry for the constant in R3, R15 respectively and a type code in R0. As well the constant collector modifies the stack link to point around any entries it used in collecting the constant.

For example the constant +1.2E+3 would be transformed into (i)

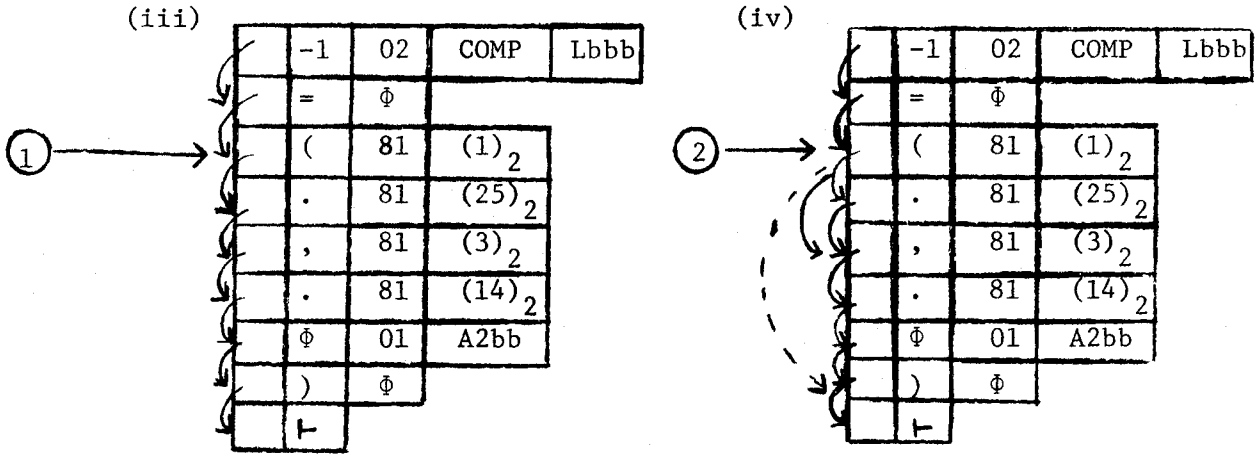
in the following figure; a call to COLCONST with pointer (1), would combine the components of the constant, store it in internal form in the symtab and return with the pointer (2) and stack as shown below in (ii) (with symtab pointers in R3, R15 and type code in R0 as well)



The constant collector makes checks on the constant being processed for violations of the language rules e.g. maximum 2 digit exponent, maximum 16 digit constant, maximum integer $2^{31}-1$ etc but does not make 'guesses' at the type of constant being collected. For example if the source statement were

COMPL = (1.25, 3.14A2)

and the entry point COLCONST was called with pointer 1 in (iii) below, the result would be as in (iv) with the type reported in R0 as 'REAL*4' for the constant 1.25. That is the constant collector makes no attempt to guess this is a mispunched complex constant and the calling processor (ARITH here) would eventually report a syntax error.[1]



Had the statement been COMPL = (1.25, 3.14E2) the type returned by COLCONST would have been 'COMPLEX*8' with the link (broken line) in (iv). The constant collector is described in more detail in section 3.3.6.

The constant lookup routine really consists of two main routines, the constant collector as just described and the routine CONLOOK which creates the symtab entries. Although CONLOOK may be called separately, the constant collector merely 'falls through' into the lookup part once the constant has been converted to internal format. The logic of the routine CONLOOK is very similar to that of CLSYM, etc, except no distinction is made between first or second appearances of the same constant (or any other constant which happens to have the same bit configuration) since no error checking is necessary for constants.

---

1 The constant collector was written before we had access to the version C28-6515-3 in which constants of the form 1E2 were allowed. Hence use of these in a source statement gives rise to syntax error messages.

Thus a call to the constant collector can look like BAL R14,COLCONST or LOOKUP COLCONST. (Before calling CONTEST, the calling routine must load R0 with an S-type adcon of an error return).

3.3.6. Constant Lookup CONLOOK

The inputs to this routine are:

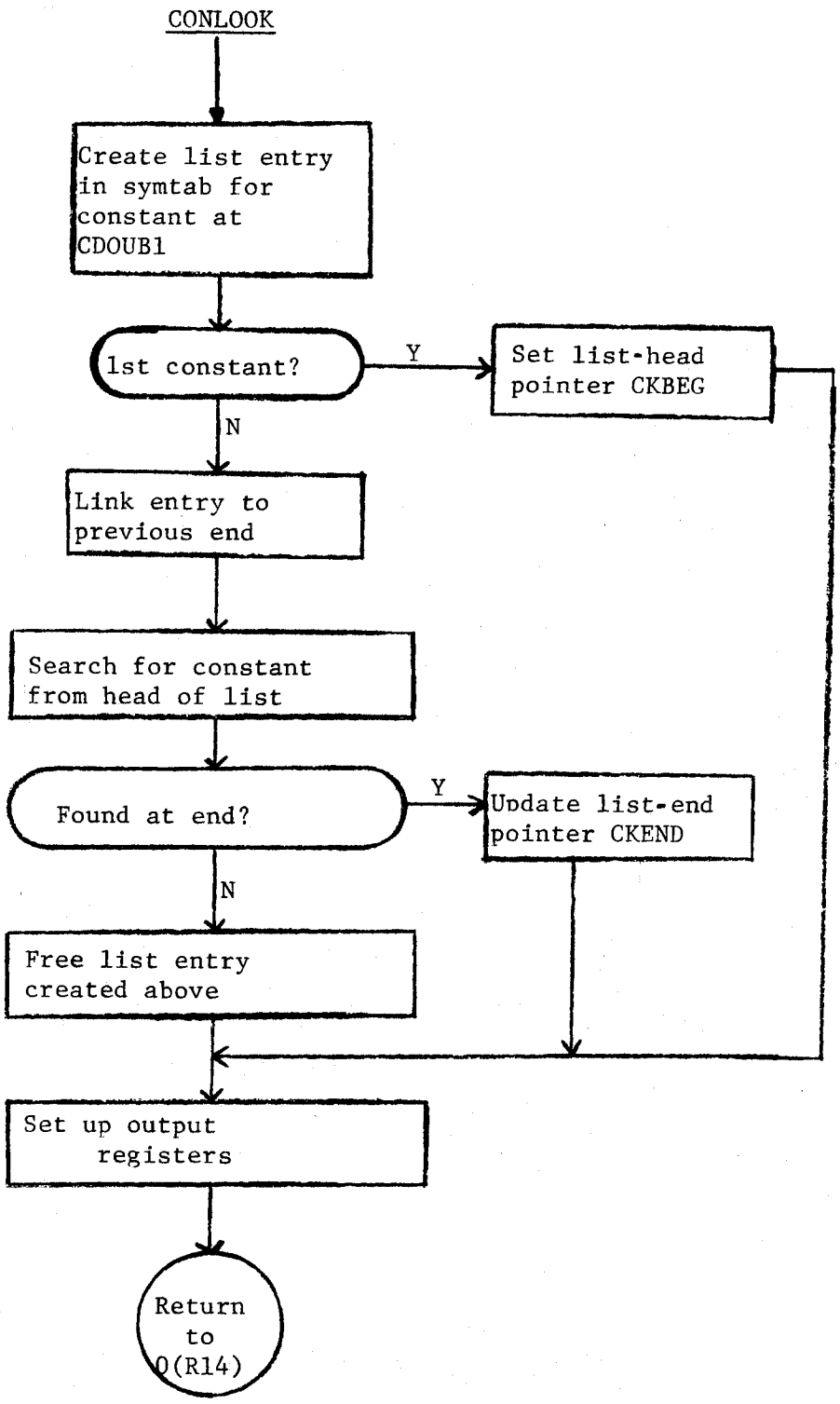
- a 4,8 or 16 byte constant in internal /360 form left justified in field CDOUB1
- a type code for the constant in byte field CTYPE
- length-1 in bytes of constant in R0.

The logic of CONLOOK is very similar to that of CLSYM etc. so will only be sketched here. (See Figure 3.3.4.). Processing starts by creating a list entry of length ℓ + 4 at the bottom of the symtab and (ℓ is length of constant) by subtracting ℓ + 4 from R6. The constant is moved into this from CDOUB1 and its length-1 is stored from R0 (See symtab entry for constant, section 2.4.). If this is the first constant in the list for this programme segment, the current value of R6 is set as list-head pointer CKBEG, the first constant switch is reset, list-end pointer CKEND is set to R6, and final processing is performed.

If this is not the first constant in the programme segment, the new entry just created is linked to the previous end of the list via CKEND. A search is made for the constant from the head of the list. This search compares all constants in the list not shorter than the constant being looked up. This ensures that, to some extent, constants which have the same initial bit pattern will share the same storage at run time. For example 1.D0 and 1.E0 will have the same run time address providing the former appears in the programme segment before the latter.

However, because of RELOC-1's method of constructing primary and secondary pointers for relocating complex quantities, an 8 byte complex constant may not share the same storage as the first 8 bytes of a 16 byte complex constant. A special check is made for this in the list search. For example, two list entries will be created for (1.D0,3.14D0) and (1.E0,0.E0) even though the first 8 bytes of each constant are the same. When the list search terminates, a test is made to see if the constant was found at the end of the list. If so, the list end pointer CKEND is updated to current value of R6; if not, the newly created list entry is returned for further use.

The final processing for a constant consists of setting up the outputs
- absolute pointer to symtab entry in R15
- relative pointer in R3 (R15 - START)
- type code loaded into R0 from CTYPE.


CONLOOK
Create list entry in symtab for constant at CDOUB1
1st constant?
Y
Set list-head pointer CKBEG
N
Link entry to previous end
Search for constant from head of list
Found at end?
Y
Update list-end pointer CKEND
N
Free list entry created above
Set up output registers
Return to 0(R14)

Return is to 0(R14) with no distinction between first or second appearances of the same constant.

3.4 Constant Collection COLCONST, COLINTGR, CONTEST

The objective of this phase of the constant processor is to combine the various stack components into the internal form of the constant, store it left-justified in a field called CDOUBl, store its type code in a byte field CTYPE and leave its length-1 in bytes in R0. These three values are then used by the lookup phase to create the symtab entry.

To attain the object just outlined, the routine uses several internal switches which will be named as they appear in the following discussion. It might be mentioned that some of these switches are implemented as B/NOP condition code masks in branch-on-condition instructions to save time and space at the expense of a highly non-re-entrant routine.

The entry point COLINTGR, before branching to the register save point, sets the switch CINTSW so that only an integer constant is collected.

The entry point CONTEST stores the error return address in R0 and sets a switch at CONERR to enable this return before joining COLCONST, bypassing its first switch setting operation.

COLCONST sets the switch last mentioned to disable error returns, sets COMPSW off and sets CINTSW to collect integer, real or complex constants. A test is then performed to see if the stack delimiter is '('. If it is, the switch COMPSW is set on to indicate that the stack may contain a complex constant.

Registers R2-R6 are then saved in order to be used as work registers by the routine, switch C1STCON is set on to indicate that, should this be a complex constant, the real part is presently being collected and pointer register R5 is set to the input stack pointer R1. Thus R5 is initialized to the start of a constant and is used to move down the stack for purposes of identifying the components of a constant.

The main processing now proceeds at label COMP. Pointer register R6 is set from R5. (R6 thus points to the stack entry that a constant starts in; it will be moved later to point to the start of an imaginary part if a complex constant is being collected.) Register R0 is zeroed to be used as a digit count register and switch CSIGN is set to assume a positive constant will result.

At this point a test is made to determine what sort of a stack configuration heads the constant to be collected. There are four possibilities which the test distinguishes and these are:

(i)	θn	e.g. +123
(ii)	θ.n	+.123
(iii)	θsn	=.123
(iv)	θs.n	(+.123

Here, θ is any delimiter (often '(' for start of complex constant), s is unary '+' or '-' and n should be a numeric operand and, in each case, register R5 points to the delimiter θ. For cases (iii), (iv) the constant collector will produce properly signed constants by resetting, and later testing, CSIGN should be '-'. (In cases (ii), (iv) the pointer R5 is advanced to the stack entry which should contain '.n' and control is passed to a point (CDECENT) which performs checking for these cases.

At label CDECENT, real constants with no apparent integer part are processed by

- checking if there is a '.' and numeric fractional part in the stack. (Error exit if not.)
- creating an assumed integer part of zero in F0
- continuing from label COLFLT1 for real constants with a fractional part.)

For cases (i), (iii) a test is made on the stack code to insure a numeric operand is present and an error exit is taken if not. If so, the routine COLINT is called to convert 'n' to binary in registers R2-R3.

COLINT works as follows:

- call is BAL R15,COLINT and return is to 0(R15)
- inputs are:
  - R5 pointing to stack entry containing 'n'
  - R0 containing total digit, count (times 8) for this constant.
- outputs are:
  - a binary integer (maximum value 10^{16}-1) in registers R2-R3
  - number of decimal digits (times 8) of n in R4 (used in processing of real constants).
  - incremented total digit count (times 8) in R0 (used for error checking).

The conversion algorithm used is essentially this.

(For the typical stack entry

R5 → | link | θ | 8i | $d_1(n_1)_2$ | ... | $d_i(n_i)_2$ |):

where d_1, d_i are subscripts e.g. $_{d_i}(n_i)_2$.

- if i = 1; R2 ← 0, R3 ← n_1, R4 ← $8d_1$, R0 ← R0 + R4
- if i ≥ 2; R2R3 ← $10^{d_2}n_1 + n_2$ where d_2 is the number of decimal digits in n_2.

  R4 ← $8(d_1 + d_2)$ where $d_1 = 8$

  R0 ← R0 + R4 (or R0 ← 8 × 16 + 1 if i > 2 to indicate a constant of more than 16 digits).]

On return from COLINT, stack pointer R5 is advanced to see if a '.' follows the integer 'n'. If so the constant being processed is real and control transfers to process a real constant (label COLFLT). If not or if the switch CINTSW is set to collect an integer constant only, the following processing occurs:

- the integer in R2-R3 is tested to insure its not greater than $2^{31}-1$ (warn and continue with value reduced modulo 2^{32} if not).
- the field CTYPE is set to indicate an integer constant.
- if CSIGN is set for a minus sign, the integer in R3 is 2's complemented.
- R0 is set to length-1 of constant (i.e. 3) for input to CONLOOK.
- control transfers to check if we are currently processing a complex constant (label C1STCON).

Processing of a real constant continues at label COLFLT by first floating into F0 the integer just collected in R2-R3. (Floating is done by biasing the integer with exponent of X'4E' and normalizing by adding to zero.) Then a test is made to see if a numeric operand follows the decimal point. If not processing is continued at label CNOFRAC.

(At label CNOFRAC, real constants with no fractional part are processed by

- saving the digit count in R0
- checking for a possible exponent and branching to CHKEXP1 to process it
- branching to COUT1 to finish processing otherwise.)

If so, at COLFLT1, the fractional part is converted by calling COLINT and then floated into F2. The total digit count in R0 is saved and the integer and fractional parts of the real constant combined into F0 using the fraction's digit count in R4; thus $[n.m]_f$ = F0 ← $n_f \oplus m_f \oslash 10^d$ for a constant of the form 'n.m' where d is the number of decimal digits in m, n_f, m_f are n,m as floating point numbers and $\oplus$, $\oslash$ are floating point operations.

The stack pointer R5 is then advanced to check if the next delimiter and stack code are Φ, 01 respectively, signifying a possible exponent. If not, control passes to label COUT1 where final processing of real constants occurs. If so, at label CHKEXP1, a search is made in the stack for an exponent of the forms βd, βdd, $\beta \pm d$, $\beta \pm dd$ (β is E or D). If no such exponent is found control passes to COUT1. The results of a successful search are:

- absolute value of exponent x as binary integer in R4
- switch CEXPSW set to indicate sign of exponent
- switch CEXPTYPE set to indicate E or D exponent
- R5 pointing to stack entry past exponent

The exponent x is then accounted for as follows: (assuming constant is of form $n.m\beta x$). $|x|$ is split into 2 parts b,ℓ where $0 \leq \ell \leq 15$
i.e. $x = b + \ell$
and $[n.m\beta x]_f = F0 \leftarrow [n.m]_f \circledast 10^b \circledast 10^\ell$
where $\circledast$ is floating multiply or divide if x is positive or negative respectively. (The quantity $[n.m]_f$ is in F0 from processing above.) This method was used since only the table 10^i, i = 1, 2, 3, ..., 15, 16, 32, 48, ..., 96 (i.e. 21 constants) need be stored instead of the i = 1, 2, ..., 99 if x were not split as above.

Control is then transferred to CEXPDONE below.

At label COUT1, the type of those constants having no exponent is determined from the total digit count, (REAL*4 if less than 8, REAL*8 otherwise) and the switch CEXPTYPE is set to reflect this choice.

Then at CEXPDONE, the following processing occurs:

- warn if constant had more than 16 digits
- take into account a unary minus, by testing CSIGN
- depending on setting of CEXPTYPE do:

  either
  - warn if constant had more than 7 digits and E exponent; assume D exponent
  - round to single precision (6 hex digits) in F0
  - set field CTYPE to indicate REAL*4 constant
  - set R0 to length-1 of constant (i.e. 3) as input to CONLOOK
  - transfer to check if this is part of a complex constant (label C1STCON)

  or
  - set CTYPE to REAL*8 code for D exponent
  - set R0 to 7 for CONLOOK
  - transfer to C1STCON

At label C1STCON, checking is performed to determine if a complex constant is being processed. If this is the first constant, i.e. possibly the real part of a complex constant, the following processing occurs - the constant is stored at CDOUB1

- if the constant is integer or the switch COMPSW is not set to indicate this is not a real part of a complex constant, transfer to the stack link fix-up label CONEXIT
- if COMPSW is set so that this might be part of a complex constant and if a ',' follows next in the stack, the type (CTYPE) and length (R0) of this constant are saved, the switch C1STCON is reset to indicate a second constant (i.e. imaginary part) will be processed, an error switch CONERR is reset and control transfers to label COMP to process the imaginary part.

If this is the second constant processed (i.e. supposedly the imaginary part of a complex constant) then:

- if this constant was integer or it is not followed by ')' in stack an exit is taken which ignores this constant and uses the first constant as real. e.g. for (1.23,4) or (1.23,4.+ result of call to COLCONST is real constant 1.23 with 4 or 4. ignored.
- check if lengths of real and imaginary parts are the same and store the imaginary part at CDOUB1+4 if the constants are REAL*4 or CDOUB1+8 if REAL*8 or mixed (warn in last case).
- set R0 to 7 or 15 depending on the length of the constant.
- set CTYPE to COMPLEX*8 or COMPLEX*16 depending on the type.
- overlay the opening '(' and closing ')' of the complex constant in the stack with Φ for ARITH's use.
- proceed to CONEXIT.

At CONEXIT, the stack link at input pointer R1 is adjusted to link around all components of the constant, i.e. to the entry pointed to by R5, and registers R2-R6 are restored before proceeding to the lookup phase.

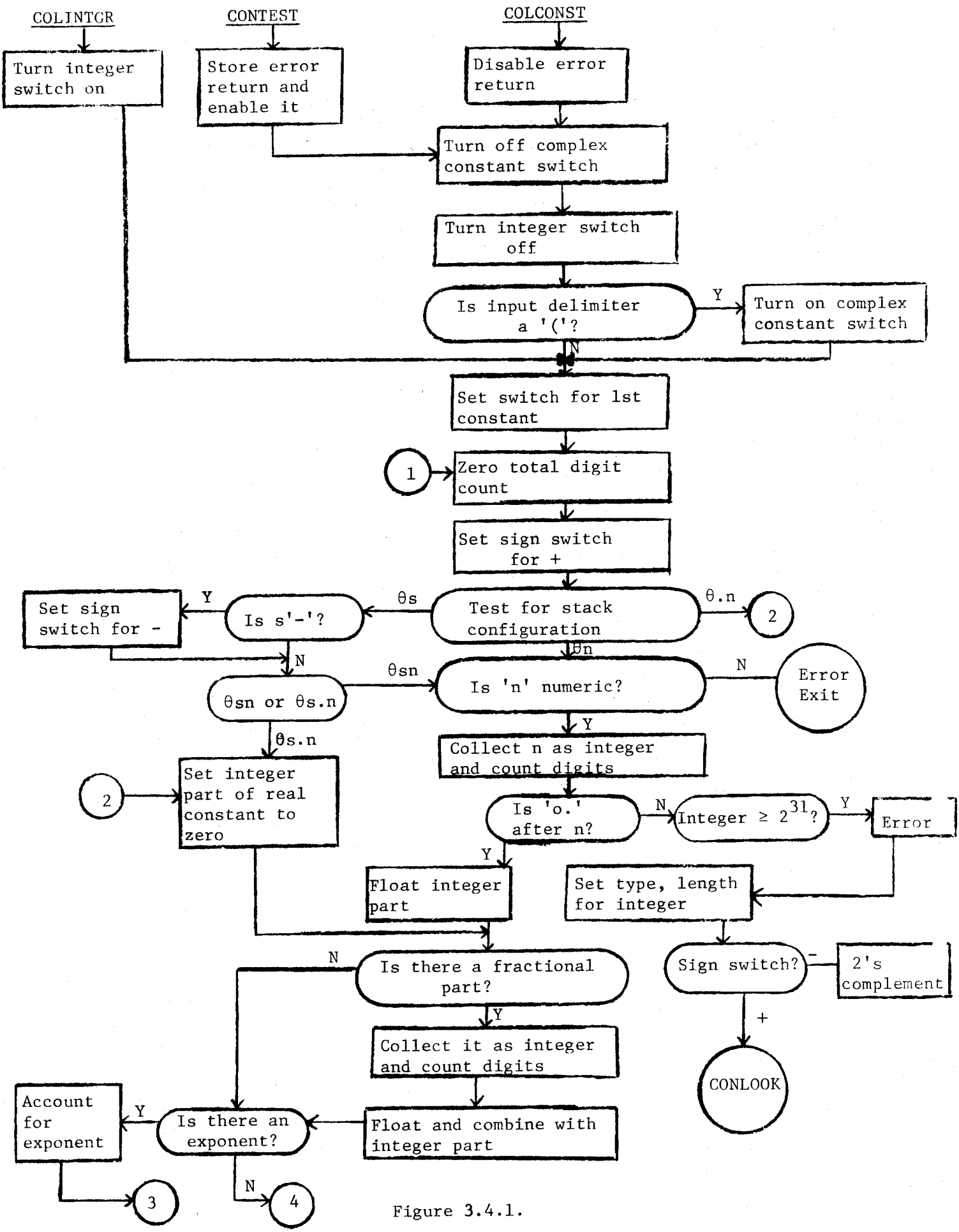

COLINTGR
CONTEST
COLCONST
Turn integer switch on
Store error return and enable it
Disable error return
Turn off complex constant switch
Turn integer switch off
Is input delimiter a '('?
Y
Turn on complex constant switch
N
Set switch for lst constant
1
Zero total digit count
Set sign switch for +
Set sign switch for -
Y
Is s'-'?
θs
Test for stack configuration
θ.n
2
N
θsn or θs.n
θsn
Is 'n' numeric?
N
Error Exit
Y
θs.n
Collect n as integer and count digits
2
Set integer part of real constant to zero
Is 'o.' after n?
N
Integer ≥ 2^31?
Y
Error
Y
Float integer part
Set type, length for integer
N
Is there a fractional part?
Sign switch?
-
2's complement
Y
+
Collect it as integer and count digits
CONLOOK
Account for exponent
Y
Is there an exponent?
Float and combine with integer part
3
N
4

Figure 3.4.1.

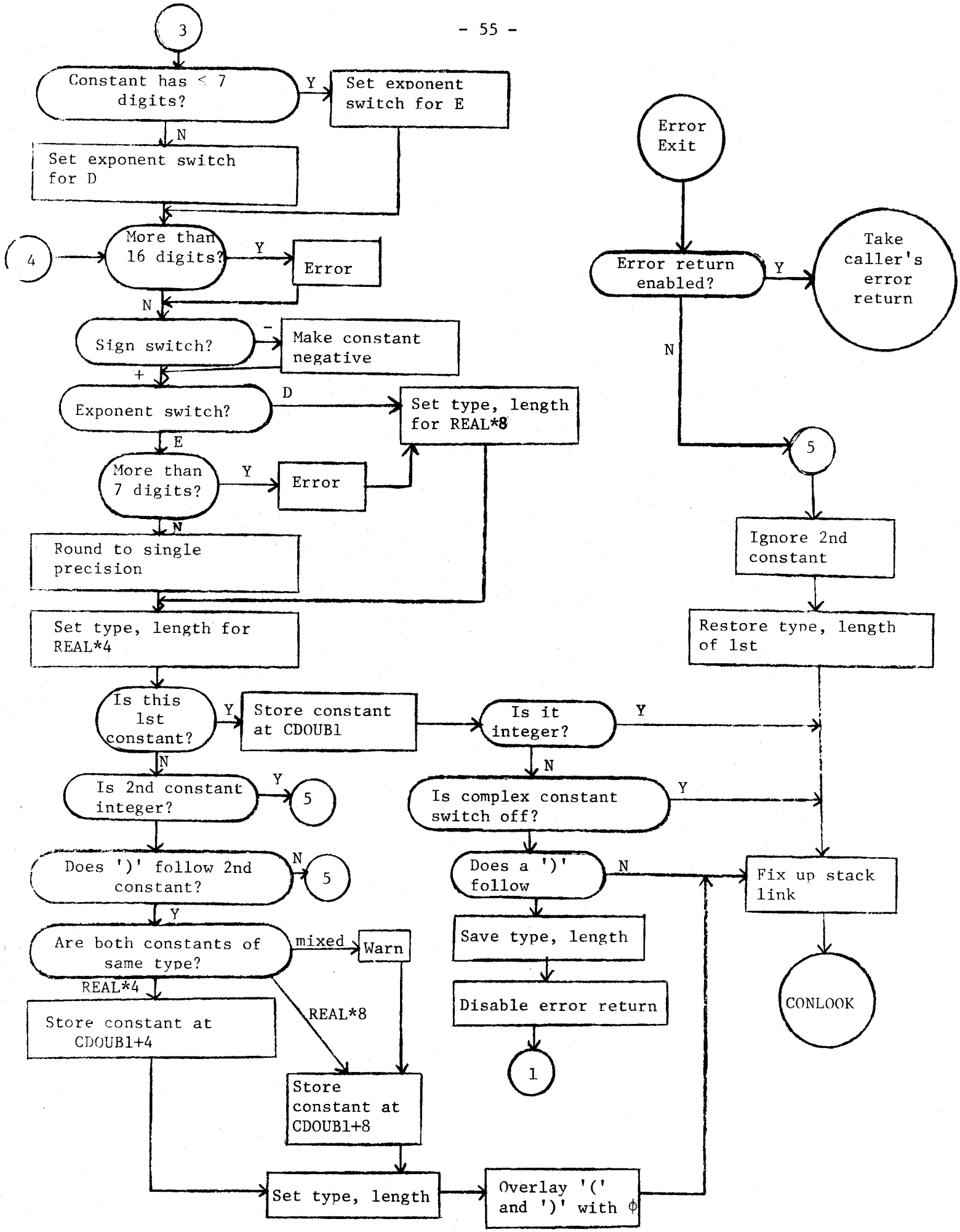


Figure 3.4.1. (Continued)

3.5 Object Code Output Routine COUT

This routine outputs object code, generated by the statement processors, to the work area. COUT adjusts the pointer (register 5) to the top of the object code accordingly as code is produced while monitoring the possibility of a work area overflow. (Check R5 against R6.)

The OUTPUT macro instruction can be used to provide linkage and data to COUT and is described below:

OUTPUT n, a, r

n - is the number of bytes
a - the address of the code to be output
r - return address (if blank fall through)

The macro sets up register 1 with the address of the code, register 2 with the number of bytes and register 14 with the return address. Any or all of the operands may be omitted as long as they are set by the user of the macro. Register 5 and 6 are used as pointers to the object code area and symbol table area respectively and register 3 is used as a work register.

If a memory overflow has occurred no object code will be placed in the work area.

3.6 Addressing Routines CUSING, CBALR11

As previously described register 11 is used as a base register for object code generated. Since we are generating a machine language programme it is necessary to simulate the (BALR-USING) instruction of the assembler. It should be noted that the ISN coding generated usually sets up register 11 for each statement (section 2.5.1.). The routine CBALR11 is used to issue a BALR 11,0 if this is required. The routine CUSING is used to simulate a USING *,11. This is accomplished by storing register 5 in CBAR11. Hence any displacements are calculated by subtracting CBAR11 from register 5.

3.7 Input Routine (CREAD)

The routine CREAD located in COMMR is used if a card image is required by the compiler.

3.7.1. Constants and Switches

XCARD1, XCARD — The card image input/output area. XCARD contains the actual FORTRAN source statement. XCARD1 contains the control character for printing and the line number to be included as part of the source listing. This area is included in STARTA since it is required at both compile and execution time.

CNEWJOB, CDATA, CBTCHEND — These three constants contain the BCD characters that installation decides to use for control card words (JOB, ENTRY, STOP). They are defined using the SETC symbols in the deck OPTIONS.

COPJOB, COPENTRY, COPSTOP — Three instructions (CLC) that check XCARD+1 for JOB, ENTRY or STOP. When a check is required we execute (EX) the appropriate instruction.

CFFCHAR — Column 73 of the source input statement is saved here. (This column is replaced with an end of card delimiter).

CSUBRDS — This switch tells the read routine if input is coming from "input unit" or the library unit. (X'00' - input, X'80' - library).

CBUFF, CBUFFE, CPOINT — These constants are used if the input is coming from the library device. CBUFF points to the beginning of the buffer, CBUFFE points to the end of the buffer and CPOINT points to the end of the current record read from the buffer.

3.7.2. Calling Sequence

```
BAL     R14,CREAD
B       LABEL1      Control card return
B       LABEL2      Continuation card return
B       LABEL3      New statement card return
B       LABEL4      Comment card return
```

3.7.3. Description of CREAD

It is possible under certain circumstances that the card image desired has been read by a previous read (e.g. The user may have read the next user's $JOB card as data). The switch CIHGACRD is turned on when this or a similar situation occurs. This switch is tested on each entry to CREAD and if on, we already have the required card in XCARD and hence no 'read' need be issued. On exit from the routine CREAD the switch is turned off.

If we issue a read, CREAD places the card image at location XCARD and replaces column 73 of the card with the hexadecimal character X'FF', as an end of card delimiter, after saving the character initially present in CFFCHAR. The type of card is determined and control returns to one of 4 possible locations as described above.

Since we are relatively certain that the next I/O operation will be an output of the source line and any associated error messages, CREAD issues a call to the routine CPRINT to initialize FIOCS for the impending output operation.

The source input can be obtained from the "input unit" or from the library 'WATLIB'. A switch CSUBRDS is used to determine this.

"Input unit"

WATFOR is a batch processor and only the $ENTRY card separates the source programme from the data. Also, the next user's $JOB card can immediately follow the last data card. It was decided to use the same data set (FT05F001) for input at both compile and execution time. The routine FIOCS is used to process the reading of source cards.

e.g.

```
L       1,=V(FIOCS)
LA      2,XINPDSRN
BALR    R0,R1
DC      X'00'           INITIALIZE
DC      X'F0'           BCD input
```

where XINPDSRN is 3 consecutive words (in STARTA) containing the unit number (5), the end of file return and the error return.

The address of the card image is returned in register 2 and the number of characters in register 3. The card is now moved to location XCARD.

2 Library (WATLIB)

At the beginning of the batch the WATLIB data set is opened. The blocksize is obtained from the DCB and a GETMAIN is issued for the required buffer space. This coding appears in the deck MAIN at location MAINP. (See Chapter 6.) When a particular subprogramme is required from WATLIB the routine MLIBR in MAIN issues a FIND of the particular member followed by the instructions.

```
MVC     CPOINT(4),CBUFFE
MVI     CSUBRDS,ZSWON
```

These instructions set up the buffer pointer for deblocking and set the switch CSUBRDS so that input will come from WATLIB. After the subprogramme has been compiled the instruction

```
MVI     CSUBRDS,ZSWOFF
```

in MLIBR resets the CREAD routine to accept input from the 'input' unit.

A $ENTRY card must be placed at the end of each data set member (i.e. there can be more than 1 subprogramme per data set member).

3.8 CPRINT

This routine initializes the routine FIOCS for an output operation on the output unit (printer) (See section 5.5) for a description of FIOCS). Following the call to FIOCS the buffer address returned by FIOCS is returned in register 2 and is saved in location XBUFFER. The calling sequence used is

```
BAL     R9,CPRINT
```

3.9 Memory Overflow Check Routine - CGETSYM

Routine CGETSYM is used to check that a new entry to be added at the bottom of the symtab will not overwrite the compiler.

Before calling, the required symbol list entry length is subtracted from R6.

```
For example     SH      R6,=H'12'
                BAL     R14,CGETSYM
```

requests 12 bytes of symtab entry if available. CGETSYM returns on R14 if this is available or issues an MO-2 message and terminates compilation if R6 is less than the lower boundary of the work area set by &MEMSIZE.

3.10 Keyword Elimination Routine - CSETSTAK

The purpose of this routine is to deconcatenate[1] FORTRAN keywords left in the stack by SCAN and to set up the stack for easy processing by the calling routine.

CSETSTAK is called by either form of the SETSTAK macro

```
e.g.            SETSTACK    'DIMENSION'
        or      SETSTACK
```

In the latter case, R1 is assumed to point to the character following the keyword; the former case sets up R1.

Inputs to CSETSTAK are R1 as just described and R9 pointing to the stack entry to be processed. Outputs are R9 pointing to the transformed stack and R0 continuing the value of any concatenated constant. e.g. STOP123

---

1 disconcatenate? decatenate? excatenate? getridof?

See Figure 3.10.2. for logic of CSETSTAK.

Several examples may serve to clarify the processing. (Stack links not shown.)

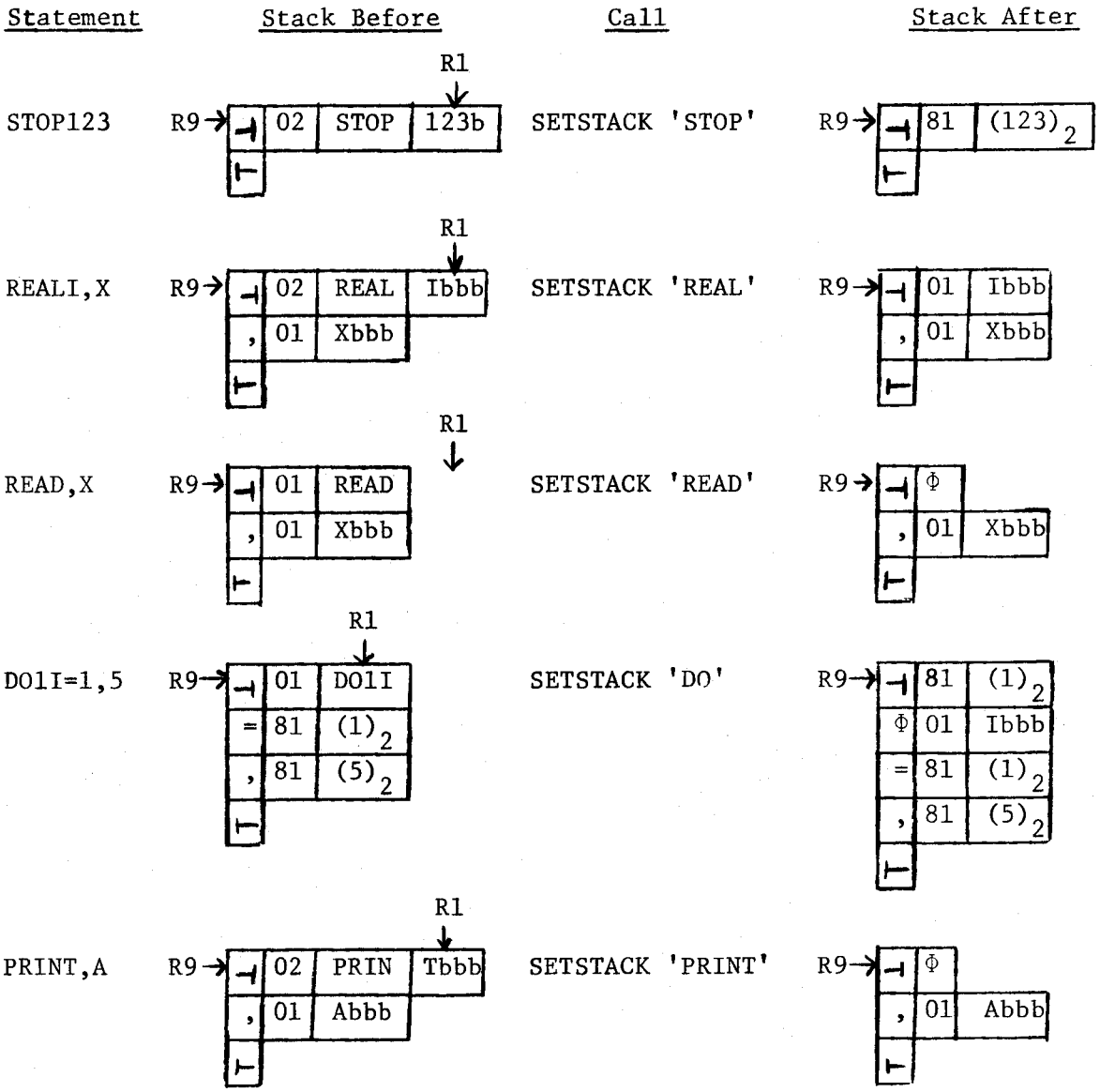


Figure 3.10.1.

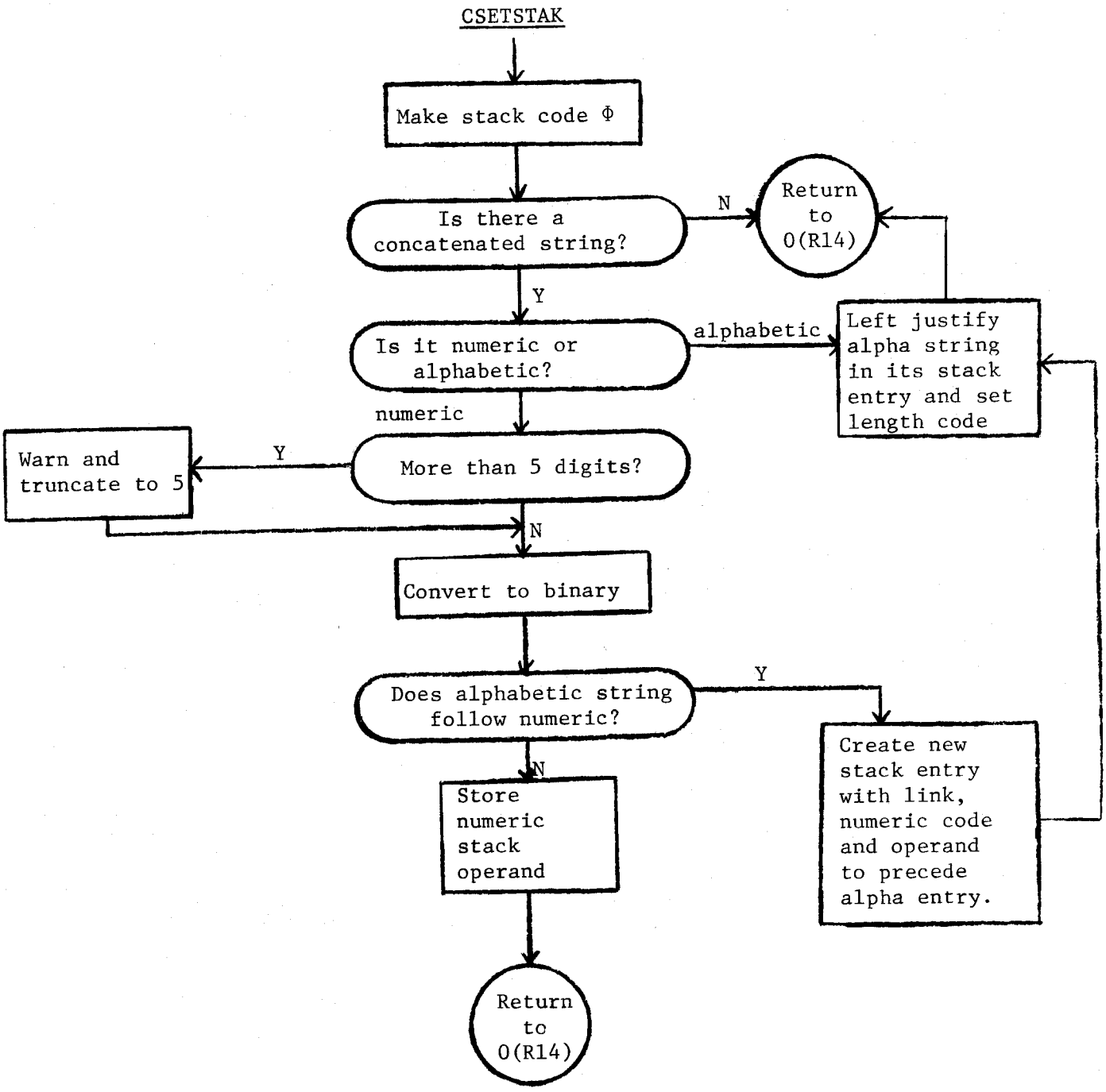

CSETSTAK
Make stack code Φ
Is there a concatenated string?
N
Return to 0(R14)
Y
Is it numeric or alphabetic?
alphabetic
Left justify alpha string in its stack entry and set length code
numeric
Warn and truncate to 5
Y
More than 5 digits?
N
Convert to binary
Does alphabetic string follow numeric?
Y
Create new stack entry with link, numeric code and operand to precede alpha entry.
N
Store numeric stack operand
Return to 0(R14)

Figure 3.10.2.

3.11 Alignment Routines (CNOP's and CDSO's)

The purpose of these routines is to align object code pointers to specific word or half word boundaries and they fall into the following two classes:

(a) CNOP04, CNOP24, CNOP08, CNOP28, CNOP48, CNOP68

These routines are used to generate NOP or NOPR instructions so that argument lists will be aligned properly. They merely simulate the assembler's language instructions

```
CNOP    0,4
CNOP    2,4
  .
  .
  .
CNOP    6,8
```

The relocator recognizes this form of output and merely bypasses it as a valid code (i.e., no relocation necessary).

(b) CDSOD, CDSOF, CDSOH

These routines merely move the object code pointer (register 5) to a double, full or half word boundary respectively. They simulate the assembler instructions.

```
DS      0D
DS      0F
DS      0H
```

The relocator cannot automatically pass over code skipped by these routines.

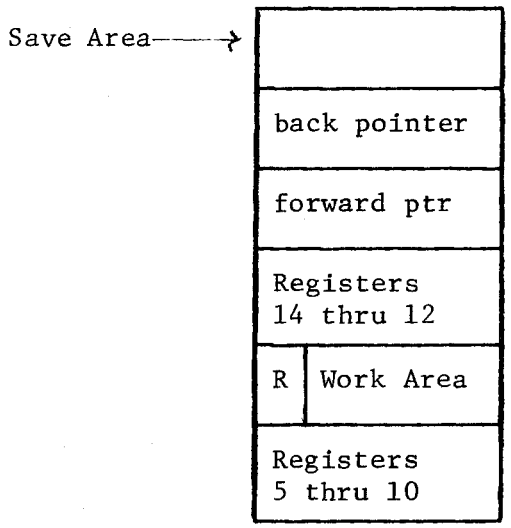
EXECUTION TIME ROUTINES

3.12 Entry and Return Routines XENT, XRET

These routines correspond to CENT and CRET in COMMR as they provide linkage and data for user programmes and subprogrammes at execution time. In order to describe these routines the entry sequence for a subprogramme and a diagram of a typical save area of a subprogramme are presented.

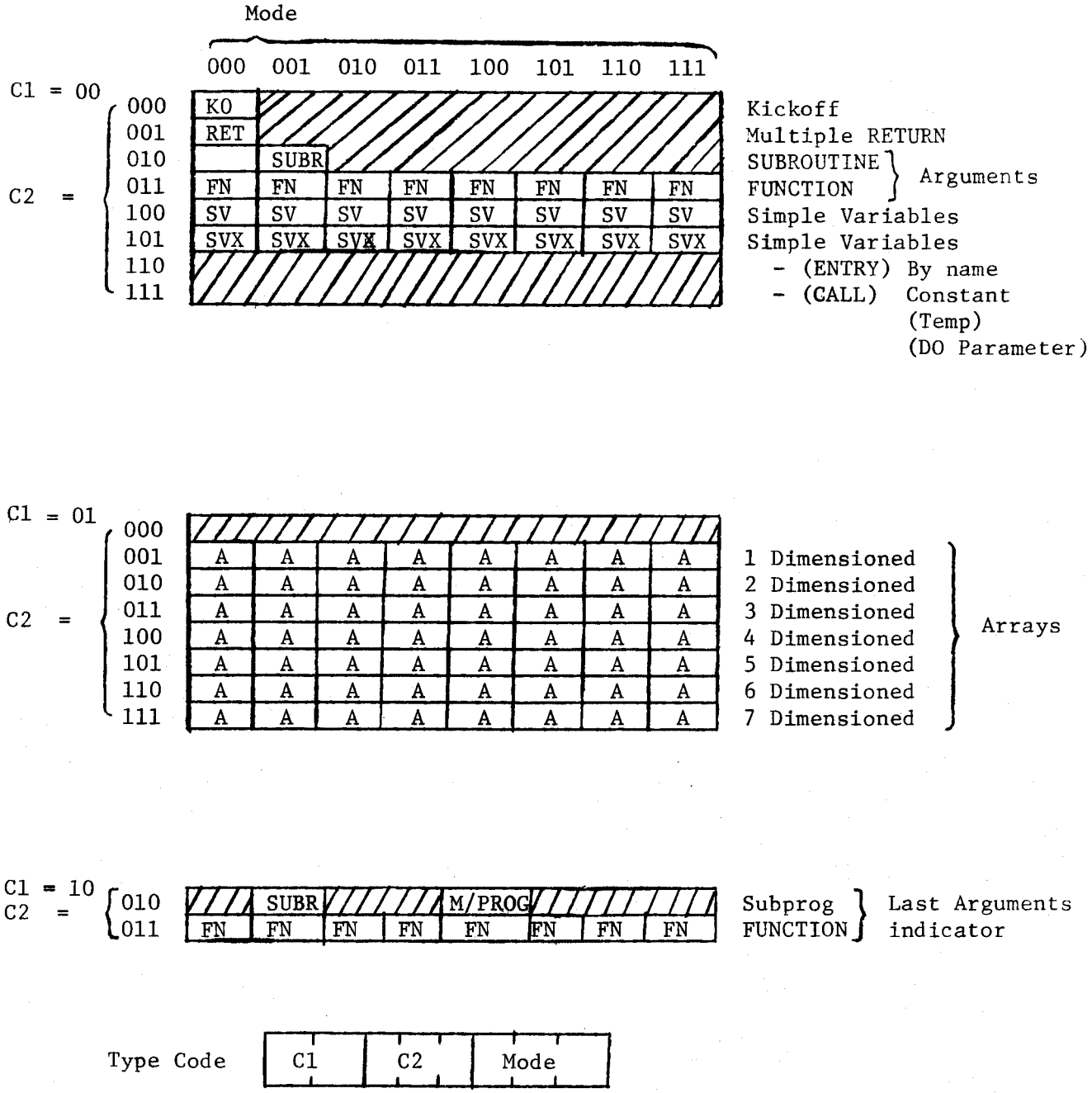
```
FUNCTION NAME (A,B)

        CNOP    0,4
        STM     14,12,12(13)
        BAL     R11,XENT
        DC      A(SAVEAREA)
        DC      CL6'NAME',H'0'
        DC      AL1(type),AL3(A)
        DC      AL1(type),AL3(B)
        DC      AL1(type),AL3(NAME)
```

where type is described in Figure 3.12.1.



R Recursion bit. Used as pointers to the data area for the subprogramme.

The routine XENT links the save areas in the customary manner and loads registers 5 through 10 so that the data area for the subprogrmame can be accessed. If the recursion bit is on (X'80') an error message is issued indicating the appropriate error, otherwise it is turned on. Since register 11 was used for the call to XENT addressability has been provided for the subprogramme. This register also points to the model argument list which is now compared against the calling argument list. (Register 14 points to the calling argument list.) Tests are made for the same number of arguments as well as consistency of the mode of arguments. During this scanning process argument or argument



Action Table for Type Codes in

Subprogramme Arguments Lists

Figure 3.12.1.

addresses are passed down to the subprogramme according to their type.

Arithmetic statement functions use the same entry sequence with the exception that registers 5 through 10 are not altered. The routine XENTASF takes care of this problem.

Library functions use the entry XENTSPEC which does not turn on the recursive bit.

The routine XRET performs the return sequence. Its actions are basically an inverse set of operations of XENT. Registers are restored from the save area, register 11 (the base register) is restored; registers 5 through 10 are restored and the recursion bit is turned off (X'00'). In the case of a FUNCTION subprogramme the return value is loaded into the appropriate register. Simple variable arguments called by value are now passed back. Error checking is provided at this time, checking if the user has attempted to modify a constant, a DO loop parameter or a temporary (i.e. the user placed an expression in the calling sequence and is now trying to change its value). An SR-5 error message is issued and the job is terminated if the above error is detected.

A routine XRETASF exists for the same purpose as XENTASF if we are processing an Arithmetic Statement Function.

XRET is now ready to return and uses one of the following routines depending if the user has said RETURN or RETURN I.

XRETI	normal return (register 3 contains the value of n for RETURN n)
XRETIF, XRETIH	special return where register 3 contains a pointer to I where I is a full or half-word.

3.13 Error Message Editor - XERRENT1, XERRENT2, XERRENT3, XERRENT4, XERRPROC

These routines are used to service a call by the ERROR macro to produce an error message and possibly take some resulting action. The entry points XERRENT1,2,3,4 really serve as branch points to the central routine XERRPROC and four are necessary to handle the various forms of the ERROR macro that result from the presence or absence of some of its parameters. (See description of ERROR macro below).

Typical expansions of some ERROR macros follow:

```
ERROR   (BOOT,SX,0),,SAVE       STM     R0,R15,XHELPS
                                BAL     R14,XERRENT1
                                DC      AL1(c),CL3'SX0',AL2(c)
                                LM      R0,R15,XHELPS

ERROR   ,,SAVE                  STM     R0,R15,XHELPS
                                BAL     R14,XERRENT2
                                LM      R0,R15,XHELPS

ERROR   (BOOT,SX,0)             BAL     R14,XERRENT3
                                DC      AL1(c),CL3'SX0'AL2(c)

ERROR                           BAL     R14,XERRENT4
```

In cases 1 and 3, 'c' are bit fields which describe the error action; in cases 2 and 4 R1 is assumed to point to such a code field.

Thus, the four entry points merely save and restore registers R0-R15 if not done in-line and/or set R1 to point to the error action code before loading a base register for XERRPROC and calling it.

The error code field consists of from 4 to 6 bytes packed as follows:

```
BL.1'i',BL.1'j',AL.6'x',CL3'aaa',AL.4(ℓ),AL.4(r),AL1(m)
```

The first 4 bytes are always present and i and j are 1 or 0 depending on whether the 5th and 6th bytes respectively are present or absent.

Here
- x is a code for some action XERRPROC is to take e.g. produce a traceback.
- aaa is the error code itself e.g. SX0
- ℓ is a code for some information to be inserted in the message e.g. a variable name
- r is a register by which information ℓ may be retrieved
- m is the address of a message in a table of standard messages CMESTAB which is to be edited into the error output.

The purpose of the central routine XERRPROC is to interpret this error code field, produce an edited error message and possibly take some resulting action.

The processing of XERRPROC is roughly as follows:

- initialize some registers
- blank out the error message print line
- move the error code aaa into the print line
- if a message is specified by m, move it into the print line
- if additional information is specified by ℓ and r, retrieve it and edit it into the print line and put severity in print line
- perform the action implied by x
- print the error message line
- return via R14

In the calls to the ERROR macro itself, x and ℓ are specified by keywords, e.g. ERROR (BOOT,SX,0,DELR9), but these are translated in the expansion of the macro to numeric codes, e.g. the error code field set up for the above is BL.1'1',BL.1'0',AL.6(2),CL3'SXO',AL.4(1),AL.4(9).

The numeric codes are used by XERRPROC to index into jump tables to reference the proper instructions to handle the processing for 'BOOT' and 'DEL'.

There are 9 possibilities for ℓ, 8 for x and these are outlined briefly next with keyword, numeric equivalent and action taken.

For parameter ℓ we have:

SYM	0	- edit stack operand into print line.
DEL	1	- edit stack delimiter, characters 'BEFORE' and next stack operand into print line.
STN	2	- edit statement number from symtab.
ISN	3	- edit 'IN LINE' followed by the current compile or execution time ISN: for compile time this is found at XISN; for execution at 0(R11).
REL	4	- edit a variable name retrievable from the symtab using a pointer stored in the stack by ARITH's syntax scan.
NAM	5	- edit the 6 characters pointed to by 4(r).
CSN	6	- edit statement number from symtab entry and an ISN saved by the caller at field CSN.
USN	7	- same as CSN except ISN is also in the symtab entry for statement number.
CHR	8	- edit into the message text the single character pointed to.

Possibilities for x are:

LANG	0	- flag the message with *EXTENSION*
WARN	1	- flag with **WARNING**

BOOT	2	- flag with ***ERROR***, output ISN, B XBOOT to object code, set error switch to prevent execution unless RUN=FREE is specified.
ZERO	3	- same as BOOT but output DC F'0'
NOEX	4	- flag with ***ERROR*** and set error switch to inhibit execution entirely.
NOAC	5	- same as BOOT but no object code output.
TRAZ	6	- flag with ***ERROR*** and set R14 to return to the traceback routine XTRACEBK
SUBS	7	- flag with ***ERROR*** and set R14 to return to the out-of-range subscript message routine XSUBS1

Codes 0-5 are used at compile time while 6 and 7 are used only at execution time.

XSUBS1 does the following:

If RUN ≠ CHECK control is transferred to XTRACEBK (see below) without issuing a further message. Otherwise the number of the subscript is calculated from the programme's value of R4 and this number is inserted in the print line.

The value of the subscript is found from the programme's value of R3 and is inserted in the print line.

The name of the array is found using the programme's value of R14, and is inserted in the print line.

The message:

'SUBSCRIPT n OF x HAS THE VALUE n'

is printed. Control is then passed to XTRACEBK.

Execution Time Traceback Routine: (XLIBBK)

The execution-time traceback routine is used when an error condition has caused a programme to be terminated. It's purpose is to print out a traceback of the line number and routine name of the statement being executed, the line number and routine name of the statement which called the routine in execution, etc., back to the main programme.

e.g.

```
          .
          .
          .
     4    CALL   A
          .
          .
          .
    10    SUBROUTINE   A
          .
          .
    13    X = B(4.)
          .
          .
          .
    18    FUNCTION B(X)
          .
          .
    22    X = X /0.0
          .
          .
          .
```

Statement 22 would cause error message KO-2 to be issued. The traceback routine would then issue the following lines:

```
PROGRAMME WAS EXECUTING LINE 22 IN ROUTINE B WHEN TERMINATION OCCURRED
PROGRAMME WAS EXECUTING LINE 13 IN ROUTINE A WHEN TERMINATION OCCURRED
PROGRAMME WAS EXECUTING LINE 4 IN ROUTINE M/PROG WHEN TERMINATION OCCURRED
```

The information is obtained by using R11 and R13.

The termination may have occurred while executing a library function of level 1 or 2 (section 5.1.1.). If this is the case, R11 points to a full-word address A(SAVEAREA-START). The library functions are positioned in memory so that these offset addresses are always less than X'00010000'. Thus, the first two bytes of the word are always zero if termination occurred during such a routine. If termination occurred during a level-0 library routine or during execution of a FORTRAN statement, R11 points to a half-word ISN which is always non-zero.

XLIBBK can thus distinguish between library routines and FORTRAN routines.

If termination occurred during a library routine, XLIBBK finds the name of the routine at 4(R11) and gives the line number as 0. It then gets the previous values of R13 and R11 from the save area and tests to see if the calling programme was a library function.

If termination occurred during a FORTRAN statement or when a FORTRAN statement is encountered after tracing back through library functions, the line number is found at 0(R11). The address of the entry point used is found after the save area, at 72(R13). The name of the routine is located at this address plus 4. The line is printed and then the routine name is compared with the main programme name. If equal, control is transferred to XSTOP. If not equal, the previous values of R13 and R11 are restored from the save area and another line is printed.

Eventually the traceback chain must end at some statement in the main programme.

XRETRACE, XRETR1 and XTRACEBK set up registers R11 and R13 in order to enter XLIBBK. A switch XDNTRZSW is also tested to determine if a traceback is required. Most error conditions branch to XTRACEBK in order to reach XLIBBK. Some error conditions, (e.g. those detected during the routine XENT which passes arguments from one routine to another) branch to XRETRACE or XRETR1. XRETRACE 'backs up' R13 and R11 one level before falling through to XTRACEBK. XRETR1 'backs up' R11 and then falls through to XTRACEBK.

3.14 Execution-Time Subscripting Routine (XA1,XAN)

When a subscripted variable occurs, the object code links via R14 to the 'dot routine' for the array concerned (See section 4.3 and figure 3.14.3.). The dot routine links via R15 to the execution-time subscripting routine (XAN or XA1).

The subscripting routine uses the subscript list at R14 and the dot routine list at R15 to calculate the address of the subscripted variable concerned.

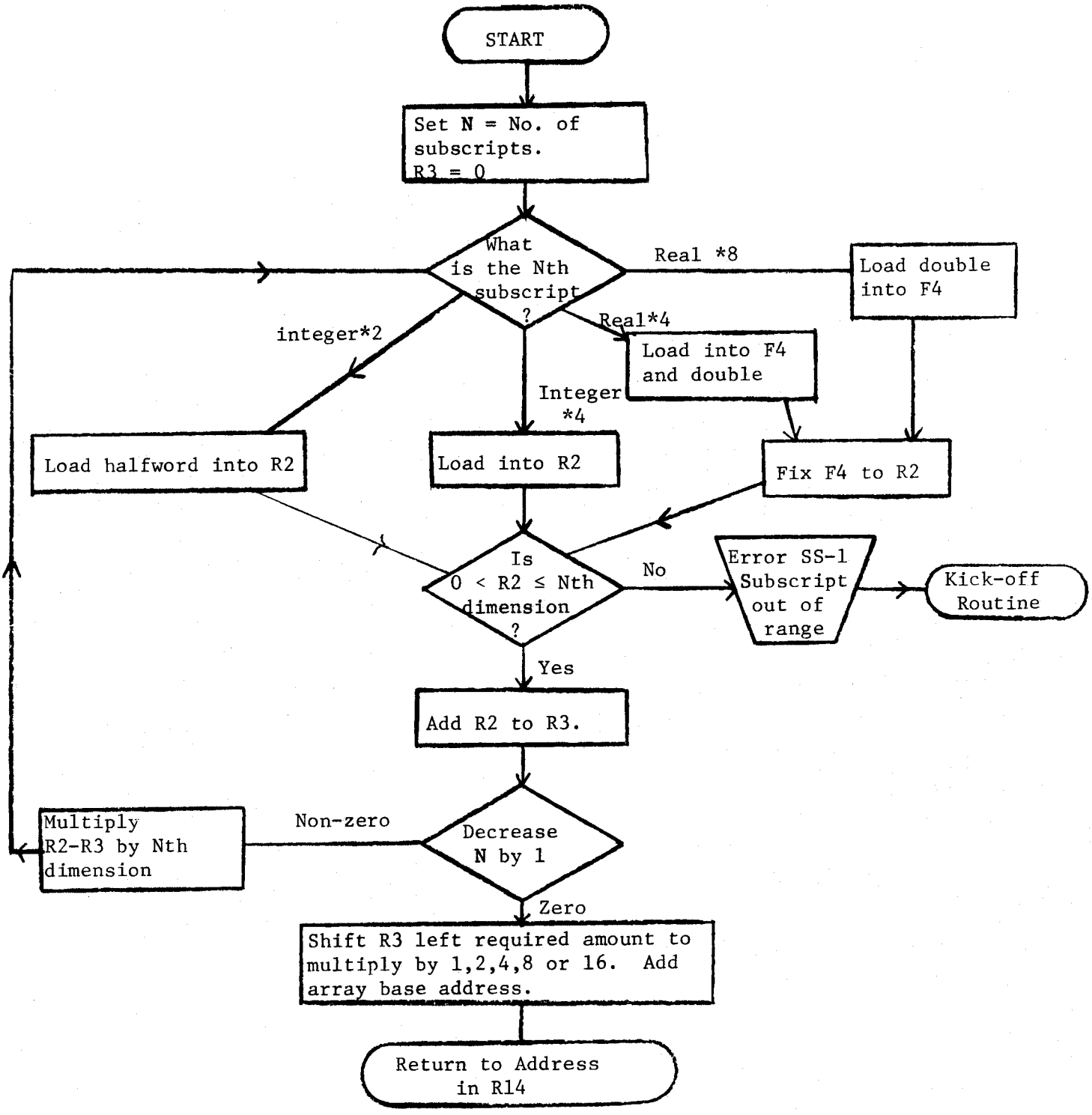
The subscript list contains the offset address of each subscript and a byte indicating its mode.

The dot routine list contains

1. 4N - 4 where N is the number of dimensions of the array,
2. the number of shifts required to multiply by the element width (1,2,4,8 or 16) for the array,
3. the array base address [A(X(o,o,...o) - START)],
4. the length of the array, and
5. the values of the dimensions.

For the internal workings of XAN and XA1 see the flow diagrams in figures 3.14.1 and 3.14.2. See also Appendix A, section A.2.

Subscripting Routine For Multiply Subscripted Variables (XAN)



Note: N is not a variable, but R4 = 4N - 4.

Figure 3.14.1.

Subscripting Routine For Singly Subscripted Variables (XA1)

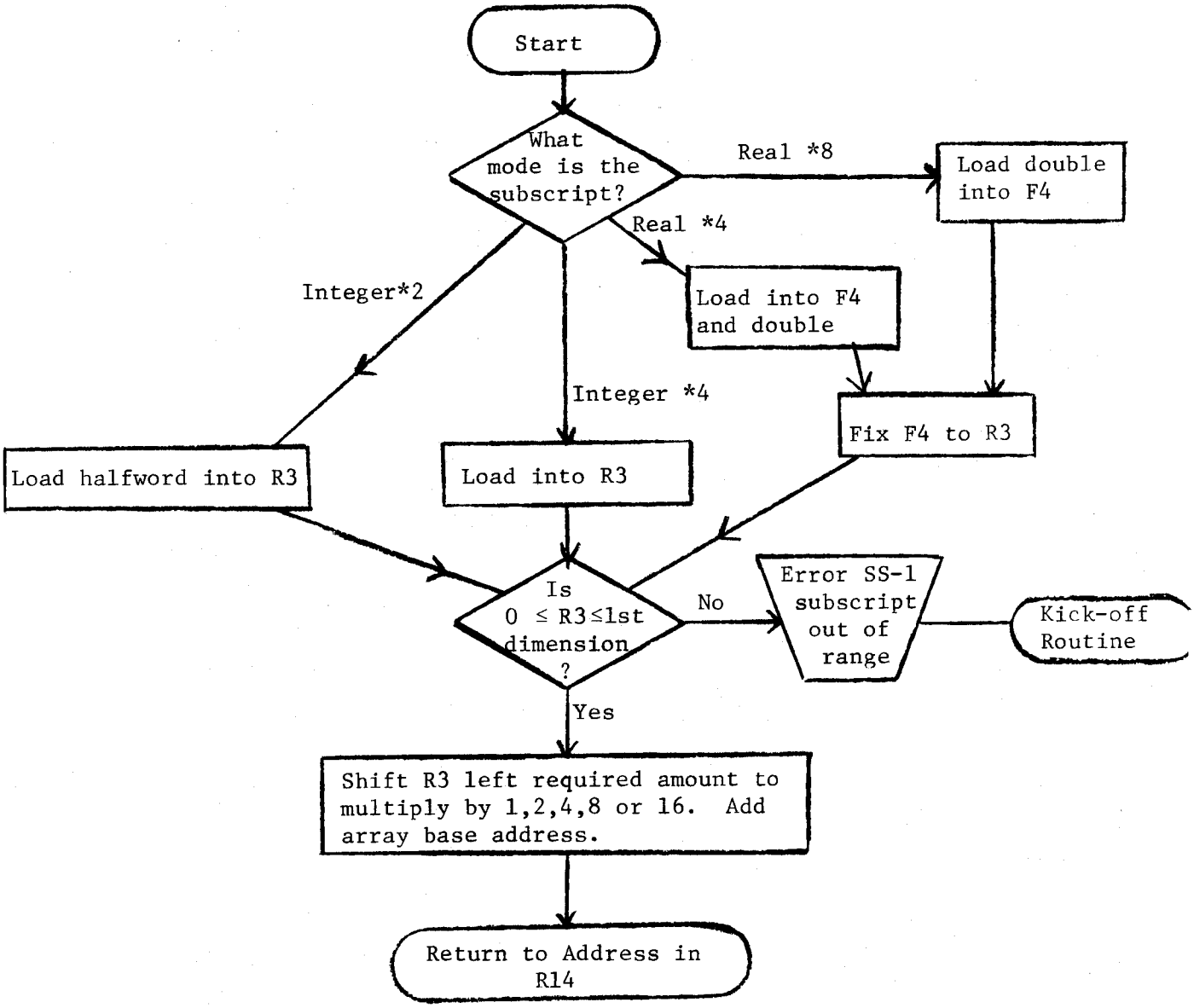


Figure 3.14.2.

Object Code and Dot Routine For A Subscripted Variable

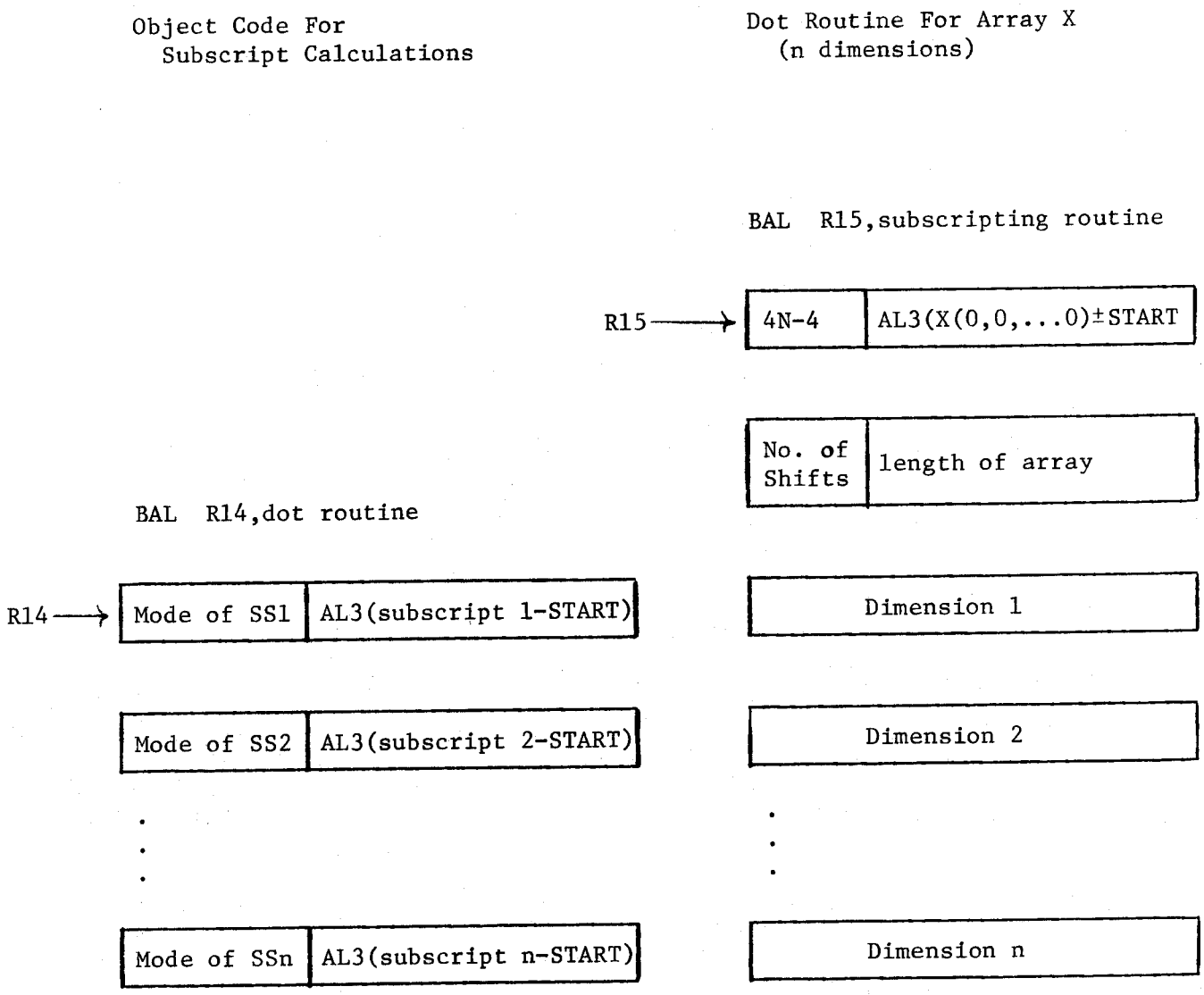


Figure 3.14.3.

3.15 Array Utility Routine - X1STELT

Since the object code constructed for any array reference always refers to the array's dot-routine, this service routine is provided for determining the address of the first element of the array. This is used for example by XARRAY and the variable-FORMAT decoder.

The inputs to X1STELT are:
- 4 times the number of dimensions of the array in R0
- the (address-START) of the dot-routine in R3

It is called via

```
BAL R1, X1STELT
```

and the outputs are:

- (address-START) of the first element of the array in R3
- total length in bytes of the array.

3.16 Exponential Routines

Routine to Raise An Integer to an Integer Power (XIEXPI)

XIEXPI raises the integer whose value is in R1 to the power whose value is in R4. The result is placed in R1. If R1 and R4 are both zero, error message EX-2 is issued and execution is terminated. If R4 is zero and R1 is non-zero the result is 1.
If R4 is negative, the result is calculated for $|R4|$ and then inverted.

Routine to Raise a Real *8 Number To An Integer Power (XR8EXPI)

XR8EXPI raises the real *8 number whose value is in F0 to the power whose value is in R4. The result is placed in F0. If F0 and R4 are both zero, error message EX-7 is issued and execution is terminated. If R4 is zero and F0 is non-zero, the result is 1.0. If R4 is negative, a MDR instruction in the routine is changed to a DDR instruction, so that the inverse result will be calculated. If R4 is negative and F0 is zero, the message EX-8 is issued and execution terminated.

Routine to Raise A Real *8 Number to a Real *8 Power (XR8EXPR8)

XR8EXPR8 raises the real *8 number whose value is in F0 to the real *8 power whose value is in F2.

If F0 is zero and F2 is less than or equal to zero, error message EX-6 is issued and execution is terminated. If F0 is zero and F2 is positive the result is 0.0. If F0 is negative and F2 is non-zero, error message EX-9 is issued and execution terminated. If F0 is non-zero and F2 equals zero, the result is 1.0.

In the case where F0 is positive and F2 is non-zero, the routine DLOG is called to calculate DLOG(F0) this is then multiplied by the value originally in F2 and the routine DEXP is called to find the exponential of this number which is returned to the calling programme in F0.

Routine to Raise a Complex *16 Number to an Integer Power (C16EXPI)

C16EXPI raises the complex *16 number in F0 and F2 to the power whose value is in R4. If R4 is negative, the result is calculated for |R4| and then inverted to give the proper result. The routine XCDIV116 is used for the inversion. Other complex multiplies and divides in the routine are executed in line.

3.17 Complex Multiply and Divide Routines

XCMULT8

XCMULT8 multiplies the two complex *8 numbers whose values are contained in (F0, F2) and (F4, F6) and places the result in (F0, F2).

XCMULT16

XCMULT16 multiplies the two complex *16 numbers whose values are contained in (F0, F2) and (F4, F6) and places the result in (F0, F2).

XCDIV18

XCDIV18 divides the complex *8 number in (F0, F2) by the complex *8 number in (F4, F6) and places the result in (F0, F2).

XCDIV116

XCDIV116 divides the complex *16 number in (F0, F2) by the complex *16 number in (F4, F6) and places the result in (F0, F2).

XCDIV28

XCDIV28 divides the complex *8 number in (F4, F6) by the complex *8 number in (F0, F2) and places the result in (F0, F2).

XCDIV216

XCDIV216 divides the complex *16 number in (F4, F6) by the complex *16 number in (F0, F2) and places the result in (F0, F2).

3.18 Fix and Float Routines

XFIX01 - fixes the real *8 number in F0, placing the result in R1.
XFIX61 - fixes the real *8 number in F6, placing the result in R1.
XFLOAT10 - floats the integer in R1, placing the real *8 result in F0.
XFLOAT30 - floats the integer in R3, placing the real *8 result in F0.
XFLOAT14 - floats the integer in R1, placing the real *8 result in F4.
XFLOAT34 - floats the integer in R3, placing the real *8 result in F4.

3.19 Undefined Variable Checking Routines (XROUT*n)

These routines check variables used in arithmetic to make sure they are defined. All variables in WATFOR are initially equal to X'8080...80'. If this bit pattern is found at the variable location to be used, the variable is assumed to be undefined.

XROUTSn (n = 1, 2, 4, 8, 16) - These routines assume a simple variable is to be checked. The instruction which uses the variable always follows the 'BAL R14,XROUTSn' instruction, so the base displacement address of the variable can be found from this instruction. The n bytes at the variable location are compared with X'8080...80'. If not equal, execution continues. If the variable was undefined, error UV-0 is issued and execution is terminated.

XROUTEn (n = 1, 2, 4, 8, 16) - These routines are for checking equivalenced, commoned or call-by-name simple variables. They assume that R3 contains the offset address of the variable. The n bytes at START(R3) are compared with X'8080...80'. If not equal, execution continues. If the variable was undefined, error UV-1 is issued and execution is terminated.

XROUTAn (n = 1, 2, 4, 8, 16) - These routines are for checking subscripted variables. They assume that R3 contains the offset address of the variable. The n bytes at START(R3) are compared with X'8080...80'. If not equal, execution continues. If the variable was undefined, error UV-2 is issued and execution is terminated.

All of the XROUTn routines destroy R15.

3.20 Execution Input/Output XIOINIT, XARRAY, XSIMPELT, XSUBSELT

These routines provide linkage between the object code generated for I/O statements and the conversion and I/O routines. The routine INOUT outputs object code for I/O statements. FORMCONV (section 5.4) and FIOCS (section 5.5) perform the conversion and I/O respectively. The routines in STARTA provide the necessary linkage. Consider the following FORTRAN I/O statement and the associated object code generated.

e.g.
```
      REAL A(20)
      PRINT25, A, B, C(2)

            BAL     R14,XIOINIT
†           DC      X'code1',AL3(address of unit number)
            DC      A(end of file addr)
            DC      A(error address)
†           DC      X'code2',AL3(address of format)
                    statement)
            BAL     R14,XARRAY
†           DC      A(address of A)
            BAL     R14,XSIMPELT
†           DC      A(address of B)

            BAL     14,.C
            DC      F'2'
            BAL     R14,XSUBSELT
†           DC      A(type code)
            BAL     R14,XENDLIST
```

† In some cases the address might occur at one or more levels of 'indirectness', (the code specifies this). However, for the following discussion, it will be assumed that the object code appears as above. The routines XIOINIT, XARRAY, etc. are described now.

XIOINIT

This routine located in STARTA is called at the beginning of each input or output statement. Its basic operation is to initialize FIOCS and FORMCONV for the impending input or output operation. XIOINIT in STARTA merely links to PIOINIT in the extended communications region STARTB.

Constants and switches

XBASADDR	The base register for the called routine (FORMCONV or XDATA)
XDSRN	Data set reference number and address of end-of-file and error returns
XTAB1	A set of 1 byte constants used to tell FIOCS what type of I/O (binary or BCD input or output).

XTAB2	A set of 6 address constants (in FORMCONV) for each type of I/O (BCD,binary and free input or output).
XTAB3	Similar table to XTAB2.
XXADDR	Address of particular I/O routine that XSIMPELT and XSUBSELT will use later.

Following are PIOINIT's functions:

1. Set up base register constant XBASADDR.
2. Move EOF and ERR addresses to XDSRN+4.
3. Obtain code for FIOCS from XTAB1.
4. Obtain unit number and store in XDSRN.
5. Determine the operation type from code 1.
   If BCD check if format is variable and if so go to FORMAT.
   Place address of format list in register 1.
6. Call FIOCS.
7. Set up XXADDR from XTAB3.
8. Get address of the FORMCONV initialize step from XTAB2.
9. Off to FORMCONV.

FORMCONV will return to the object code when it has completed its functions.

If the operation is a control operation (BACKSPACE, REWIND, or ENDFILE) control would have transferred at step 5 above to PCONTROL where the appropriate action is taken.

FORMCONV's base register is stored in XBASADDR and the end-of-file and error addresses are saved in XDSRN.

Following this set of operations the I/O and conversion routines have been initialized and are now ready to process the I/O list.

XSIMPELT and XSUBSELT

These routines are used if the list element is a simple variable or a subscripted variable. Register 3 is used to point to the required variable in the I/O list. The value previously stored in XXADDR determines where to go in FORMCONV. Again FORMCONV returns to the object code.

XARRAY

This routine is used if the list element is an array. After setting up the necessary loop control, finding the address of the first element and the length of the array (use X1STELT) XARRAY calls XSUBSELT the required number of times to output the array.

XENDLIST

This routine is used to terminate activity for an I/O or DATA list. For example on a PRINT statement a call to XENDLIST would cause a call to FIOCS to print the line. The call to XENDLIST is the last object code generated for an I/O list or the only code generated in case no list is present.

Error Handling (PERROR)

If an error occurs in FIOCS control transfers to PERROR (via MYIBCOM). Register 14 points to a constant containing an error code and the address of the unit number. If the error is an EOF or ERR type the addresses saved in XDSRN are used to possibly return to the object code or to print the appropriate error message.

3.21 Pre-execution DATA Statement Processor - XDATA, PDATA

The DATA statement compiler produces object code for source DATA and initializing type statements which basically, consists of two lists: one list is a sequence of executable instructions corresponding to the variables (an I/O list) and the other is a list of 8 byte constant descripters corresponding to the list of constants in the statement. The object code for all such statements is chained together. For a full description of this coding and its production see section 4.6.7. The discussion which follows will use the example statement:

```
DATA    A,I/1.,2/,C/Z12F3/
```

to describe the action of XDATA and PDATA.

The object code for this statement is:

```
            B       AROUND
            BAL     R11,XISNRTN
            DC      AL2(ISN)
            CNOP    0,4
            BAL     R14,XDATA
            DC      A(next data statement-START)
            DC      AL1(n),AL3(savearea-START)
SUBLIST1    DC      AL1(0),AL1(m ),AL2(CONLIST1-SUBLIST1)
IOLIST1     BAL     R14,XSIMPELT
            DC      AL1(t),AL3(A-START)
            BAL     R14,XSIMPELT
            DC      AL1(t),AL3(I-START)
            BAL     R14,XENDLIST
CONLIST1    DC      AL1(0),AL1(ℓ1-1),AL2(r1)
            DC      AL1(t),AL3(=1-START)
            DC      AL1(t),AL3(=2-START)
SUBLIST2    DC      AL1(0),AL1(m2),AL2(CONLIST2-SUBLIST2)
IOLIST2     BAL     R14,XSIMPELT
            DC      AL1(t),AL3(C-START)
            BAL     R14,XENDLIST
CONLIST2    DC      AL1(0),AL1(ℓ3-1),AL2(r3)
            DC      AL1(t),AL3(=XL16'12F3'-START)
            BALR    R11,0
AROUND      EQU     *
```

Where - n (=2) is the number of I/O-conlist pairs in the statement

- t is a type code for the constant or variable
- m_1, m_2 (=2,1) are the number of conlist descriptors
- r_1, r_2, r_3, (=1,1,1) are the replication factors for the constants
- ℓ_1, ℓ_2, ℓ_3 (=4,4,16) are the lengths in bytes of the constant (Hexadecimal constants are set up as 16 byte fields by the constant collector.)

The purpose of routines XDATA, PDATA is to match up the variable items with the constant items and effect the initialization. In this respect they perform the same functions as do XIOINIT and FORMCONV for formatted I/O statements; XDATA does initialization for each DATA statement and PDATA actually carries out the initializing of variables on an element by element basis as control is transferred to it via the I/O list item handling routines XSIMPELT, XSUBSELT, XARRAY.

XDATA does the following:

- sets up XBASADDR, XXADDR to provide linkage to and addressability for PDATA via XSIMPELT etc.
- saves the pointer to the next DATA statement in the chain (the last DATA statement is chained to the entry point of the mainline programme via XSTART11);
- saves the number of sublists n;
- loads the data area registers R5-R10 from the savearea of the programme segment this DATA statement happens to be in;
- sets a pointer to the first sublist header, SUBLIST1
- joins the coding which does sublist initializing.

The sublist initialization which is performed for each sublist is:

- set a switch which is used to insure that there are at least as many constants as locations to be initialized;
- saves the number of conlist descriptors m;
- returns to the I/O list coding via R14.

Eventually one of XSIMPELT, XSUBSELT, XARRAY sends control to PDATA with R3 containing the (address-START) of the variable to be initialized and R14 pointing to its type code t.

PDATA then does the following:

- tests the switch to see if the conlist has been exhausted: if so issue error DA-2 and proceed to next sublist;
- obtain the address of the constant from the saved conlist descripter;

- tests the saved conlist descripter type-code t for a hollerith or hexadecimal constant which require special treatment. (See below).
- adjusts the address of the constant to ignore its 2 high order bytes if the variable is half word integer;
- tests that the type of the variable and constant are the same (error DA-6 issued and next two steps bypassed if not);
- tests to see if the variable has already been initialized (e.g. DATA A,A/1.,2./) (warning DA-8 is issued and next step bypassed);
- moves the constant into the location specified for the variable (i.e. performs the initialization of the variable);
- decrements the replication factor r for this constant and returns to the I/O-list coding if it is not zero.
- decrements the conlist count m to see if there are any constants and sets the switch if not; if so it advances the conlist pointer to the next conlist descripter and saves both the pointer and the descripter before returning to the I/O-list coding via R14.

The special treatment of hollerith and hexadecimal constants is:

- right-pad with blanks if the variable is longer than the hollerith constant or truncate with a warning message if vice versa before initializing.
- right-pad with zeros if the variable is longer than the hex constants or truncate if vice versa.

When the end of the I/O-list is reached, XENDLIST sends control to that part of PDATA which does the following:

- tests the switch to see if the conlist has also been exhausted (if not, warning DA-7 is issued and the remaining constants ignored);
- decrements the sublist count n and repeats the sublist initialization if the count has not been reduced to zero; if zero, PDATA branches to the next DATA statement in the chain using the pointer saved above.

The flow charts for XDATA and PDATA are given in Figure 3.21.1.

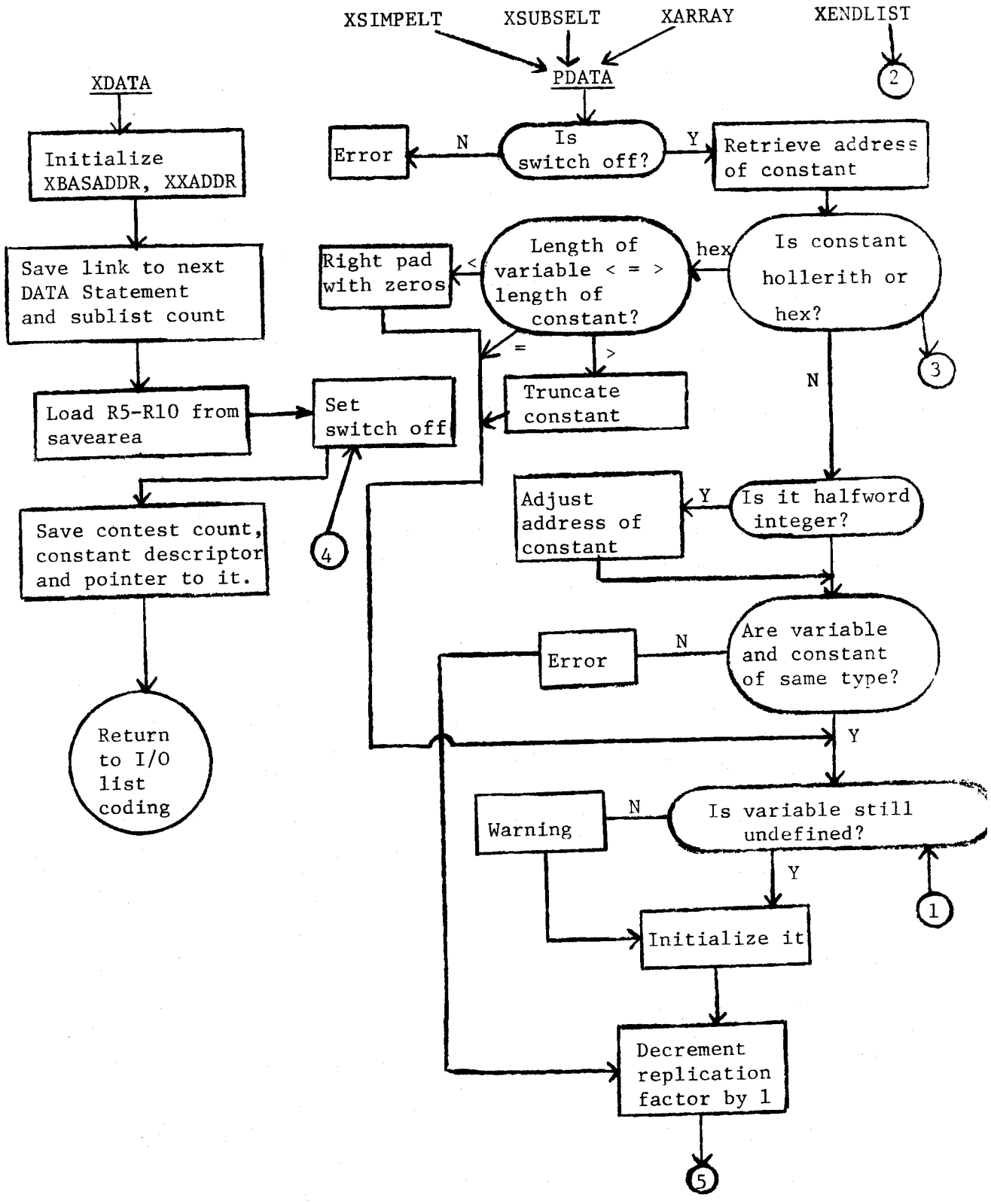

XSIMPELT
XSUBSELT
XARRAY
XENDLIST
PDATA
2
XDATA
Initialize XBASADDR, XXADDR
Error
N
Is switch off?
Y
Retrieve address of constant
Save link to next DATA Statement and sublist count
Right pad with zeros
<
Length of variable < = > length of constant?
hex
Is constant hollerith or hex?
3
=
>
Load R5-R10 from savearea
Set switch off
Truncate constant
N
Adjust address of constant
Y
Is it halfword integer?
Save contest count, constant descriptor and pointer to it.
4
Error
N
Are variable and constant of same type?
Return to I/O list coding
Y
Warning
N
Is variable still undefined?
Y
1
Initialize it
Decrement replication factor by 1
5

Figure 3.21.1.

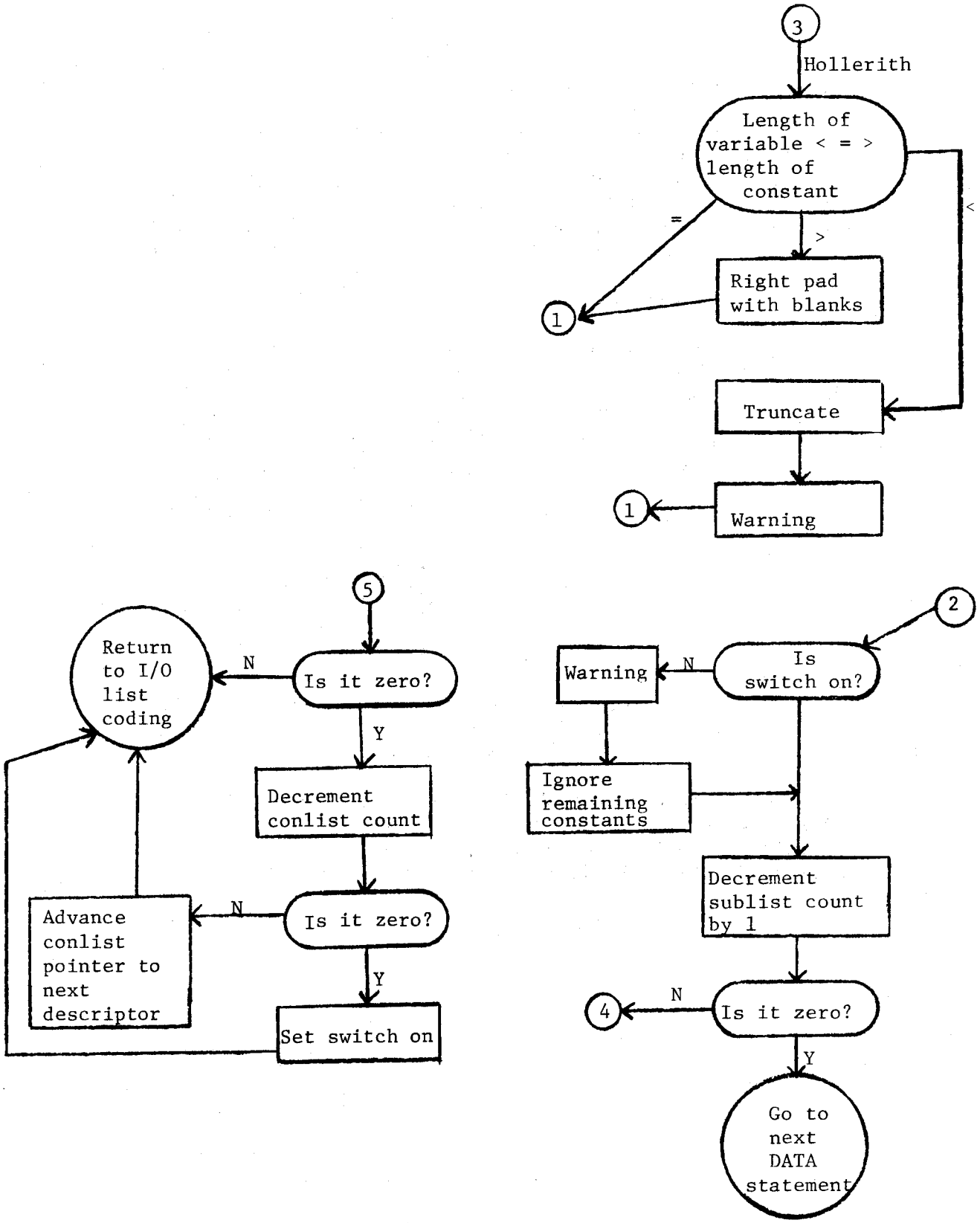


Figure 3.21.1. (Continued)

3.22 Compiler Output Routine XPRINT

This routine outputs source lines, error messages and accounting information on the output unit. (Usually a printer.) The routine uses FIOCS and assumes that FIOCS has been initialized to the proper "output state". At compile time the routine CREAD has accomplished this purpose since it expects to do output following input. If the routine XPRINT is to be called and the caller is not certain of the state of FIOCS a call to the routine CPRINT will rectify the situation.

XPRINT obtains the address of the current buffer supplied by FIOCS (from XBUFFER) and using the length supplied in register 2 and the address of the record in register 1 moves the line to be printed to the buffer. A call to FIOCS causes the line to be printed and on return register 2 points to the new buffer supplied by FIOCS. This is stored in XBUFFER.

Following is the calling sequence to FIOCS used by XPRINT:

```
LA      2,length of record
L       1,=V(FIOCS)
BALR    0,1
DC      X'02'
DC      X'00'
```

In general most routines will use the SPRIN macro to output a line.

```
SPRIN   AREA,n
```

where n is the number of bytes to be output from location AREA. Note SPRIN causes the current register to be saved and assumes that FIOCS is initialized for the output operation. (If the user is not sure a call to CPRINT (BAL R9,CPRINT) will initialize FIOCS.)

3.23 Execution-Time Interrupt Handling Routine (XRUPT)

During execution time, control is passed to XRUPT to process the interrupts indicated by the SPIE statement at XSTART11. These normally include all programme interrupts except fixed-point overflow and significance.

At XRUPT, R12 is loaded with the address of START in case the routine causing the interrupt has destroyed it. The interruption code in the PIE is inspected, and control passed to a routine to handle the particular interrupt.

For operation, privileged operation, execute, protection, addressing, data, decimal overflow and decimal divide interrupts, the error message KO-7 is issued. The address of STOP is put in the return address portion of the PIE and control is returned to OS which finishes its processing and then goes to STOP. These interrupts should not occur unless there is an error in the compiler.

The significance interrupt is ignored. (i.e. control is returned directly to OS which returns to the execution of the programme.)

The specification interrupt is assumed to be caused by improper boundary alignment of variables in common or equivalence if the instruction which caused the interrupt is located on a half-word boundary in core, and is four bytes long. If the instruction address is odd, or the length of the instruction is 2 or 6, a compiler error is assumed and the same procedure is followed as for the other invalid interrupts.

Otherwise, the instruction is assumed to be an RX type instruction in which the data is not located on the proper boundary. The $D_2(I_2,B_2)$ portion of the instruction is moved into an instruction 'LA R14,*-*' which is then executed. Following execution of this instruction, R14 contains the address of the data which is not located on the proper boundary. Eight characters from 0(R14) are moved to XDOUBLE which is located on a double word boundary. The programme's values of R15-R2 are restored from the PIE, and the OP code and R_1 portions of the instruction are placed in an instruction 'NOP 0,XDOUBLE'. This instruction is then executed, and the eight characters from XDOUBLE are moved back to the location from which they were obtained. The same effect has now been achieved as if the instruction causing the interrupt had been executed. The registers R15-R2 are stored in the PIE (in case the execution of the 'bad' instruction has altered their value) and a return to OS is executed. OS finishes its processing and then returns to continue execution of the programme.

For a fixed-point overflow interrupt, the value of XFXOFLOW is decreased by one and if non-zero, control is returned to OS which returns to execution. If XFXOFLOW becomes zero, the message KO-5 (too many fixed-point overflows) is issued and the return address in the PIE is set to XTRACEBK. A return to OS is then executed. OS returns to

XTRACEBK. XFXOFLOW can be set by calling the subroutine TRAPS (See section 5.1.

For a fixed-point divide interrupt, the value of XFXDVCNT is decreased by one, and if zero, the message KO-1 is issued and the return address in the PIE is set to XTRACEBK, etc. If XFXDVCNT does not become zero, the switch XDVCHKSW is set to X'01' and control is returned to OS which continues execution.

For a floating point divide interrupt, the value of XFLDVCNT is decreased by one, and if zero, the message KO-2 is issued and the return address in the PIE is set to XTRACEBK, etc. If XFLDVCNT does not become zero, the switch XDVCHKSW is set to X'01' and control is returned to OS which continues execution. XFXDVCNT and XFLDVCNT are initially equal to one and can be set by calling subroutine TRAPS. XDVCHKSW can be tested by calling the subroutine DVCHK (See 5.1.3.)

For an exponent underflow interrupt, the value of XEXUFLOW is decreased by one and if zero, message KO-4 is issued. The return address in the PIE is set to XTRACEBK, etc. If XEXUFLOW does not become zero, the switch XOVRFLSW is set equal to X'03', and control is returned to OS which continues execution.

For an exponent overflow interrupt, the value of XEXOFLOW is decreased by one and if zero, the message KO-3 is issued, the return address in the PIE is set to XTRACEBK, etc. If XEXOFLOW does not become zero, the switch XOVRFLSW is set equal to X'01' and control is returned to OS which continues execution. XEXUFLOW, XEXOFLOW are initially one and can be set by calling subroutine TRAPS. XOVRFLSW can be tested by calling the subroutine OVERFL (See 5.1.3.)

3.24 Run Time Operator Message Routines XSTOPN,XPAUSE

The coding for these routines may be eliminated by proper choice of the assembly parameter &STOPN in OPTIONS. (See page 27 of WATFOR Implementation Guide.)

The two entry points XSTOPN, XPAUSE merely establish addressability for and linkage to routines PSTOPN, PPAUSHOL which are now described.

1. PSTOPN

This routine is reached from the object code by a statement of the form STOPn by means of the compiler instructions

```
BAL     R14,XSTOPN
DC      AL4(n)
```

PSTOPN sets a switch to terminate execution following the issuing of the WTO macro, sets the operator message code to read IHC002I, edits 'n' into the message and branches to issue the WTO.

2. PPAUSHOL

The object code for a PAUSE statement is of two forms:

```
PAUSE n              BAL     R14,XPAUSE
                     DC      AL1(0),AL3(n)
PAUSE'literal'       BAL     R14,XPAUSE
                     DC      AL1(ℓ-1),AL3(literal-START)
```

where n is zero for simple PAUSE statement, ℓ is the length in bytes of the literal constant.

Thus by testing 0(R14), PPAUSHOL either converts to decimal the constant n or moves the literal constant into the operator message area. (The constant is truncated if longer than the message area.) It also sets a switch to execute a WAIT macro following the WTOR, sets up the operator message code to read IHC001A.

The WTO(R) macro is issued and a branch is taken to XSTOP (job termination) for a STOP statement or a WAIT is issued for the operator reply. When this is received the pause ECB is cleared and a return is taken to the object code via R14.

4.1 SCAN

4.1.1. Introduction

The SCAN routine of WATFOR performs two major functions. SCAN obtains the user's FORTRAN source statement and transforms it into the stack format (Section 2.7). The second function involves determining the statement type and giving control to the required statement processor.

In order to perform these tasks the following steps are required (See Figure 4.1.1.).

1. Initialize switches and constants
   Since WATFOR handles each statement as a logical quantity certain constants (e.g. bracket count) and certain switches have to be initialized.
2. Obtain a card image from the input unit.
3. Transform the statement into the stack format.
   The routines that perform this function could be described simply as a character manipulator. At each step the action taken is a function of the 'element' (variable, constant, etc.) being collected and the new character obtained.
4. Determine the statement type.
5. Process statement labels.
   Statement numbers in columns 1 - 5 are entered in the symbol table. At this time checking is done for errors involving statement numbers (e.g. multiply defined, illegal use, etc.)
6. SCAN performs some general functions such as generating ISN coding for executable statements, checking if the main-line entry coding has been generated.
7. Give control to the statement processor routine.
8. Check if end of do-loop coding is required.
   If the statement has a label we go to the routine DODO which performs the required check.

It should be noted that SCAN is called from two different locations in MAIN to handle card images either from the 'input' unit or from the 'library' unit. SCAN maintains control until the $ENTRY card or equivalent is obtained. At this time control returns to MAIN.

4.1.2. Scanning a Statement

1. Initialization

The only initialization for a particular job is to establish the keypunch mode specified by the user. A one byte switch CMODESWT is used.

This switch has the following settings:

X'00' - 26 keypunch
X'06' - 29 keypunch
X'0C' - S/R entry; leave mode as it was.

The table (SBCDEBC) is set up using this switch and SCAN can now issue warning messages if a user has punched from the 'wrong' character set.

2. Input/Output

SCAN uses the routines CREAD and XPRINT to read and print lines These routines are discussed in section 3.7 and section 3.22. respectively.

WATFOR attempts to allow the user some flexibility on choice and position of control cards. Our particular installation was 7040 IBSYS oriented and hence we decided that WATFOR should accept the same deck format that our users were accustomed to. Hence WATFOR allows control cards following the $JOB card or equivalent and preceding the first FORTRAN statement. Control cards are also allowed as separators between subprogramme modules and as an end of data or end of programme indicator. (Our installation uses coloured cards for this purpose and hence the operator's job is easier.)

e.g.
```
$JOB
$IBFTC
      mainline
$IBFTC
      subroutine
$ENTRY
      data
$IBSYS
```

In order to allow this feature and to check for 'empty' programmes ($JOB followed by $ENTRY) SCAN has an initial read loop to obtain the first statement of the programme. Now SCAN is ready to process the FORTRAN job and any future 'reads' required will occur in a second read loop (SMREAD).

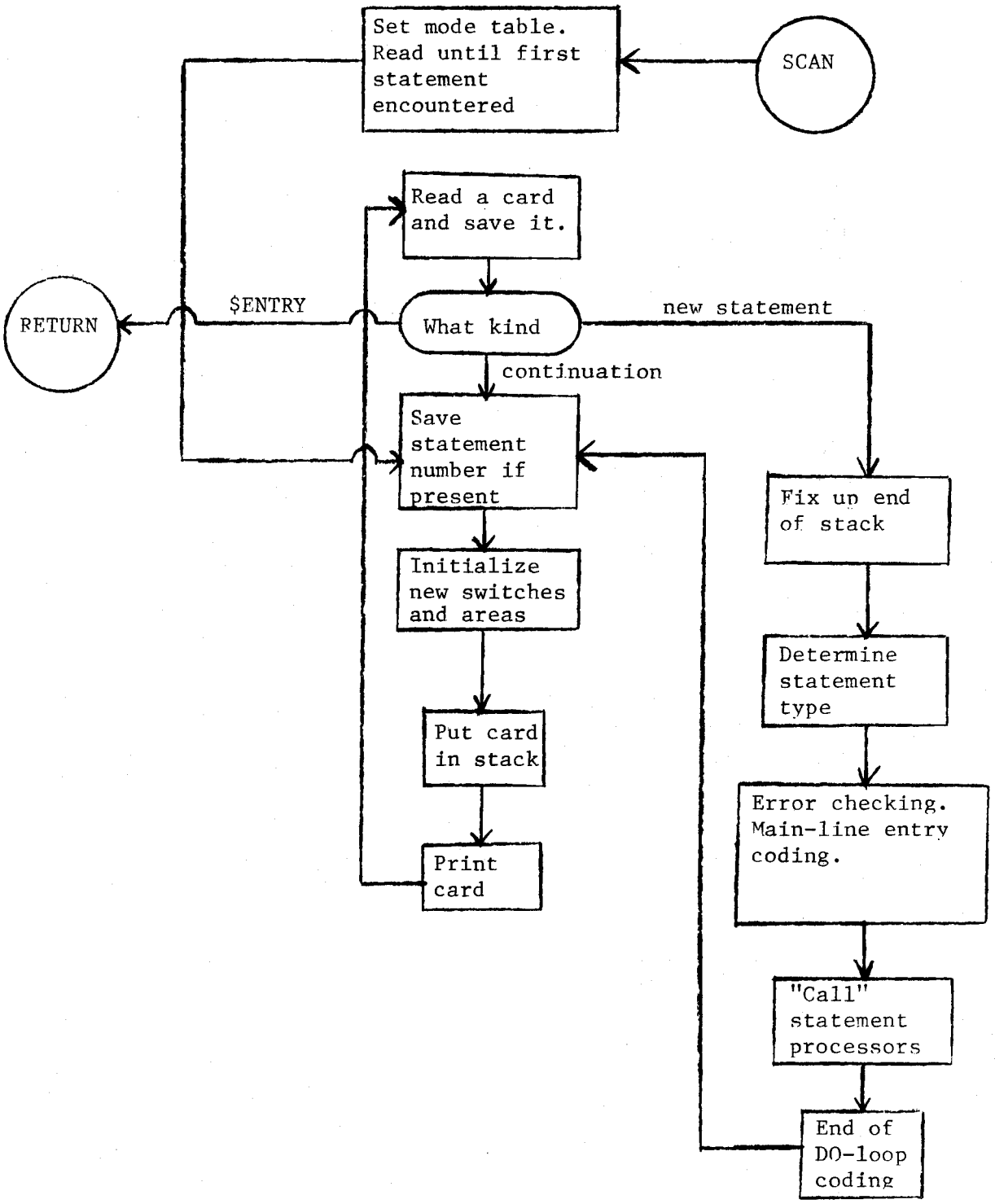

Set mode table. Read until first statement encountered
SCAN
Read a card and save it.
RETURN
$ENTRY
What kind
new statement
continuation
Save statement number if present
Initialize new switches and areas
Put card in stack
Print card
Fix up end of stack
Determine statement type
Error checking. Main-line entry coding.
"Call" statement processors
End of DO-loop coding

Figure 4.1.1.

SCAN processes each card image as it obtains it and a new read will destroy the previous card image. Hence, when we get a statement we save the statement number, if present, convert the card to the stack format, obtain a new card and if it is a continuation card, continue scanning. If it is not, we know we have a new statement and can proceed to the next step.

4.1.3. Transformation of Statement

At the beginning of each statement, a set of switches and constants is initialized by SCAN. These are now described:

LABEL	INITIAL SETTING	PURPOSE
SNLIST	zero	a double word used to collect digits of a constant.
SWITCHES		
SBOMBSW	zero	don't go to processor switch.
SLTSW	zero	first left bracket switch.
SLASHSW	zero	slash switch.
SHLCONS	zero	hollerith constant switch.
SIHPALNE	zero	already printed the line switch.
SWITCH1	zero	combined level 0 comma and equal switch.
SCC	&NOCCRDS	number of allowable continuation cards.
CDOEND	zero	used by end of do-loop
CDOBAD	zero	processor.
CSTNOLK	zero	pointer to current statement number entry in symbol table.
SBCNT	zero	parenthesis count.
SRT1	zero	pointer in stack to first and second
SRT2	zero	level zero right brackets.
SRTSW1	P'1'	counter of level 0 right brackets.
SADDR1	value in Reg 5	used for calculating the branch around address for Hollerith constants.
XISN	increased by 1	current ISN.
SOPTR	X'01'	start of stack deliminator.
SWITCH	X'03'	Operator and Operand type switch.
SERRVECT	zero	error vector
Reg 2	zero	for TRT inst.
Reg 15	addr of stack	Pointer to the stack.

SCAN is now ready to put the statement in the stack. The method used is a character collection process. Using the TRT (Translate and Test) instruction the address of the particular routine to process the character is obtained. The TRT uses a standard TRT character table (SJUMPC), set to ignore blanks and to process any other character. A second table (SJUMPR) contains the addresses of the various character processors. Certain characters, mainly operators, require some processing and switch setting to occur, before being placed in the stack. Following this processing, control transfers to the operator routine (SOPERTOR).

Character	Label	
*	SSTAR	If the previous character was '*' change the operator to '**' in stack.
/	SSLASH	Reverse the slash switch setting. We can check if we are collecting a list of constants, e.g. / 1.0, 2.0, 'ABC' /
,	SCOMMA	If we have had a level zero equal sign (not enclosed in brackets) we now have a level zero comma. (Used to determine DO statements)
Invalid	SERROR	Replace invalid character by &INVCHAR for printing.
.	SDOT	If previous character is a digit go to SCNUMB (we are collecting a constant). Check if the previous operator was a '.' and if so check if a relational operator, a logical operator, or a boolean constant is possible. If yes, make appropriate entries in the stack.

Note:

The characters +, =, (,), ' have two possible representations (26 or 29 keypunch) and are first checked to see if they are the same mode as specified by the user. If not, the appropriate error flag is set and processing continues assuming the character to be valid.

Character	Label	
+	SPLSBOK SPLSEOK SPLSBNO SPLSENO	Just check mode.
=	SEQUBOK SEQUEOK SEQUBNO SEQUENO	Check the mode and check if it is a level 0 equal sign. This is used to determine DO's and assignment statements.
'	SQUOTE	It's a Hollerith constant (described later).
(	SLTBRBOK SLTBREOK SLTBRBNO SLTRRENO	Check mode and go to SLEFTB.

Character	Label	
)	SRTBRBOK SRTBREOK SRTBRBNO SRTBRENO	Check mode and go to SRIGHTB.
	SLEFTB	Increase the bracket count. If it's the first left bracket check if first 6 characters in the stack are FORMAT. If yes go to SCAN's format processor (SFORM).
	SRIGHTB	Decrease bracket count and if negative set error flag. If zero set up a pointer to stack in SRT1 or SRT2 depending if it is the first or second time a level zero ')' has occurred.

Having processed the above characters, they and the rest of the possible characters, fit into one of three groups; letter, digit or operator. However, before inserting them in the stack, it is necessary to know what type of 'thing' we are presently or have just completed collecting.

A switch (SWITCK) is used for this and has the following settings.

1	OPERATOR
2	NAME (Anything with first character alphabetic)
3	NOTHING (beginning of statement)
4	BOOLEAN CONSTANT
5	HOLLERITH CONSTANT
6	NUMERIC CONSTANT

Depending on the current character and the setting of SWITCK one of two possible actions will occur. The first and simplest is just to insert the present character in the stack. For example, if our present character is an alphanumeric character and we are collecting a NAME,the character is concatenated to the name in the stack. The second possible action involves 'completing' the present stack entry and starting a new entry. The 'completing' of the stack entry is of course dependent on the operand type. It could involve one or more of the following steps.

1. Calculation of the link.
2. Inserting the appropriate code and length.
3. Conversion of a constant (see below).
4. Padding a variable name with blanks.

A set of routines (SFIXTAB) accomplish these steps for the various types of 'things'. (e.g. If we have been collecting a NAME and the next character is an operator, the link is determined and inserted, the appropriate code is inserted, and the name is padded with blanks to a full word boundary. The operator is then placed in the next entry of the stack.)

The three major routines (SOPERTOR, SLETTER, SNUMBER) which maintain control over the above process are now described.

SOPERTOR Upon encountering an operator a new entry in the stack is always required. Hence, using SWITCK the present entry is completed and the operator is inserted in the next entry. The operators are recoded before they are placed in the stack (Table 4.1.1.)

SNUMBER Constants are collected in groups of eight digits or less along with a digit count. These groups are inserted in the stack. Hence, if the previous 'thing' being collected was a constant we might continue collecting the constant or might be forced to do a conversion and then make an entry in the stack. If the previous 'thing' is a NAME the digit is just inserted in the stack. Any other type of entry other than operator will require a new stack entry to be generated.

SLETTER The letter routine follows the same basic pattern as described above; creating new entires if the previous 'thing' is number, boolean, hollerith, or null and just inserting the character if the 'thing' is letter. A test for the letter 'H' is made if the 'thing' is number, i.e. a possible hollerith constant.

Processing a FORMAT Statement (SFORM)

The list of specifications in a FORMAT statement are merely placed in BCD form in stack after FORMAT(has been recognised by the routine SLEFTB as described above.

Processing Hollerith Constants

The routine SHOLL is used to process hollerith constants. The constants are stored in-line in the object code. The stack contains a pointer to the symbol table and the symbol table points to the constant in the object code.

```
          e.g.          'ABCD', 3HXYZ, 'HHHHH'
               B        AROUND
               CNOP     0,4
               DC       CL4'ABCD'
               DC       CL4'XYZb'
               DC       CL8'HHHHHbbb'
AROUND         EQU      *
```

delimiters and =	0 0 0 0	0 0 0 0	Φ	0
	0 0 0 0	0 0 0 1	+	1
	0 0 0 0	0 0 1 0	[	2
	0 0 0 0	0 0 1 1	(	3
	0 0 0 0	0 1 0 0	)	4
	0 0 0 0	0 1 0 1	=	5
logical	0 0 0 1	0 0 0 0	OR	16
	0 0 0 1	0 0 0 1	AND	17
	0 0 0 1	0 0 1 0	NOT	18
relational	0 0 1 0	0 0 1 0	GT	34
	0 0 1 0	0 1 0 0	LT	36
	0 0 1 0	0 1 1 1	NE	39
	0 0 1 0	1 0 0 0	EQ	40
	0 0 1 0	1 0 1 0	GE	42
	0 0 1 0	1 1 0 0	LE	44
arithmetic operators	0 1 0 0	1 0 1 0	+	74
	0 1 0 0	1 0 1 1	-	75
	0 1 0 0	1 1 0 0	*	76
	0 1 0 0	1 1 0 1	/	77
	0 1 0 0	1 1 1 0	**	78
rt. bracket	1 0 0 0	0 1 1 0	)	134
	1 0 0 0	0 1 1 1	.	135
	1 0 0 0	1 1 1 1	&	

Transformed Operator Table

Table 4.1.1.

4.1.4. Determining the Statement Type (SFIND)

Determining the type involves one of four possible steps:

1. Is it a DO statement?
2. Is it an IF statement?
3. Is it an assignment statement?
4. Is it any other type of statement?

It should be noted that we already know if it is a FORMAT statement.

Several registers and switches are used or set up for future use. These are:

Reg. 2 - address of the statement processor routine

SFNDSWT - switch describing attributes of statement.

1	executable
2	possible for 'FUNCTION' to follow key-word (REAL FUNCTION)
4	statement valid in BLOCK DATA subprogramme
8	statement does not generate main-line entry coding (SUBROUTINE, FUNCTION etc.)
16	For 'IMPLICIT' processor
32	END statement
64	FORMAT statement

If the statement type is logical IF registers 3 and 4 are also set.

Reg. 4 - address of the statement processor routine following the IF.
Reg. 3 - number of characters in the key word portion of the statement (set zero for an assignment statement).

1. DO's (SFNDTYPE)

Control transfers to the routine SFNDDO to determine if we have a DO statement. This routine has a 'NO' and 'YES' return (0(14) and 4(14) respectively). If the first two characters are 'DO' and we have a level zero equal followed by a level zero comma it is assumed that we have a DO statement. The switch SFUNNY specifies that we had a level zero comma before a level zero equal and hence the statement is assumed not to be a 'DO'.

2. IF's (SFND1)

Control transfers to the routine SFNDIF. This routine also has a 'NO' and 'YES' return.

After determining that the first two letters are IF the pointer SRT1 is used to determine what follows the first level zero right bracket. If it is a digit, we have an arithmetic IF. If it is a letter we have a logical IF. The same 4 steps described above are now done to determine the type of statement after the logical IF. SRT2 is used to point to the second level zero right bracket in case another IF follows.

3. Assignment (SFND2)

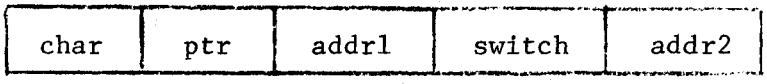
If we have a level zero equal and no level zero comma we now have an assignment statement.

4. Other (SFND10)

The routine SFIND is used to determine other statement types. The tables STYPET1 and STYPET2 used are now described.

STYPET1

This table consists of 1 entry of the following form for each statement.



where

char	4 bytes	- first four characters of the key-word.
ptr	1 byte	- a pointer to STYPET2 table (zero if key-word contains 4 or less characters).
addr1	3 bytes	- address of the processor routine
switch	1 byte	- attribute switch (same settings described above for SWITCK).
addr2	3 bytes	- this is present only if 'FUNCTION' can follow the key-word (e.g. INTEGER FUNCTION).

e.g. DC CL4'COMM',X'ptr',AL3(addr of COMMON proc),X'switch'

STYPET2

This table contains two entries for each key-word containing more than 4 characters. The first entry contains the length of the remaining characters minus one and the second the rest of the characters.

```
e.g.  DC X'01'
      DC CL2'ON'  last letters of COMMON
```

To determine the statement type the first four characters of the current statement are inserted as the last entry in the STYPET1 table. The processor address portion contains the address SERRST5 (i.e. an address in SCAN). Now a search is initiated and we are always guaranteed to be successful.

A test is now made 'to determine' if the key-word has more than r characters (if ptr is non-zero) and if not control can return with the address and switch. If there are more than 4 characters a check is made to see if 'FUNCTION' can follow. (Note: REAL has a dummy entry in the STYPET2 table and the ptr entry in STYPET1 is non zero so that it won't return after finding only 4 characters). If FUNCTION can't follow a comparison is made on the rest of the characters and the address is obtained or an error message is issued. If FUNCTION can follow the rest of the characters are compared and then a test is made to see if 'FUNCTION' follows the key-word in the stack. The appropriate address is obtained depending on a successful comparison. Note that if the statement is undecodeable the processor address obtained (SERRST5) will merely print out the appropriate error message.

4.1.5. Error Checking and Miscellaneous

Now that the statement type is known, it is now possible to do some error checking.

(a) ST-4 Statement after a Transfer Statement has no Statement Number

A switch 'CIFGOTSW' was set on if the last statement was a transfer statement. If the switch is on and the current statement is executable it turns the switch off. If there is no statement number on the current statement an error flag is set.

(b) Enter Statement Number in Symbol Table

The LOOKUP routine is used to do this (See section 3.3.2.) If the statement number is already in the symbol table but not defined or if it does not appear in the symbol table SCAN defines it (turn on appropriate bit in B1) and inserts the address. A pointer to the symbol table entry is stored in CSTNOLK. (This saves some LOOK-UP time for processors wishing to check and/or use it.) If it was there and was defined the ST-3 error is issued.

(c) Test for Illegal use of Statement Number

(i) ST-9 transfer to FORMAT statement
(ii) ST-7 transfer to non-executable statement
(iii) IO-2 referencing a non-format statement in an I/O statement.

These are accomplished by checking the various bits in the symbol table (See section 2.4).

(d) Set Switch for IMPLICIT Processor

Because there are certain rules about the placing of IMPLICIT statements SCAN sets COTHSTAT off for the SUBROUTINE, FUNCTION, 'type' FUNCTION and BLOCK DATA and on (X'FF') for all other statements. The switch SFNDSWT is used for this.

(e) Check If Statement is Valid in a BLOCK DATA Subprogramme

The switch CSRSWTCH is set X'92' if we are in a B/D S/R. Again the switch SFNDSWT is used and the error message BD-0 issued if statement is invalid.

(f) Generate MAIN-line Coding Entry

The CSRSWTCH is checked (X'40') and if so then we already have had main line coding generated. If not go to the routine 'LMAIN'. The statement number address in the symbol table has to be adjusted if we go to LMAIN.

(g) Generate ISN Coding

If the statement is non-executable no ISN coding is issued. If the statement is FORMAT a BALR 11,0 is issued in the object code and CBAR11 is updated (See section 2.8.3.). If the statement is executable a control transfers to CISN (See section 2.5.1.).

(h) Issue Error Messages

During the scanning process various error flags could have been set in the location 'SERRVECT'. If this word is zero we know that no error flags were set. If non-zero the appropriate error message is issued. Each bit is used to signify that a particular error occurred.

(i) Transfer to Statement Processors

The information saved after determining the statement type is now recovered. The processor address is placed in register 2. If the statement is a logical 'IF' the length of the keyword (zero if assignment) is placed in register 3 and the address of the processor required in register 4. Control now transfers to the statement processor which subsequently return to SCAN.

(j) End of DO-loop coding

If the statement has a statement number, control now transfers to 'DCSTN1'. The switch CDOEND is set X'02' if the statement is non-executable.

(k) All Done

We have now finished processing the statement and can now return and repeat the same procedure for the next statement.

4.2 LINKAGE STATEMENTS PROCESSOR (LINKR)

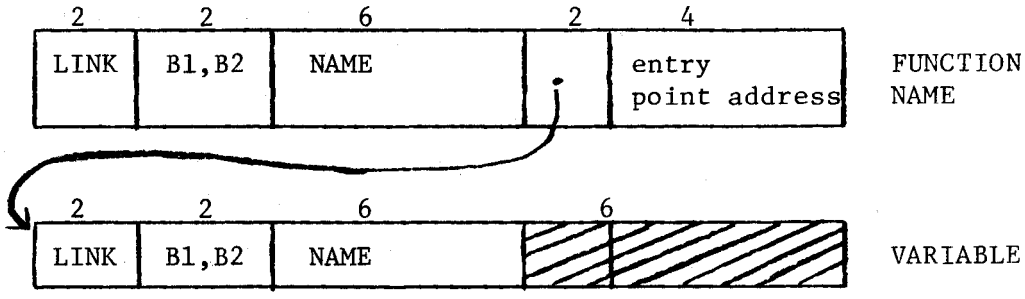
4.2.1. General

Compiler deck LINKR contains the sections of the compiler which deal with what may be grouped loosely under the title of linkage statements. These are statements which are concerned with the transfer of control to and from the function and subroutine call's. The actual call statements are processed by the arithmetic statement processor. But the entry and return statements are processed here. The loose definition has been expanded to include a main programme statement (simulated), the END statement, the BLOCK DATA statement, and simulated entry and return statements for arithmetic statement functions (ASF). Initialization for subprogramme segments is also done here in routine LENDPROG (somewhat misnamed).

The primary entry statements FUNCTION, SUBROUTINE etc. use a common exit routine and a common argument list SCAN routine to perform the tasks with a minimum of duplication of code.

4.2.2. Function Statement Processor (LFUNC)

The FUNCTION statement processor compiles the FUNCTION statement. First, it checks to see that we are at the beginning of a subprogramme segment and if not, simulate an END statement. Extract the function name and look it up in the symbol table and set aside space in the function name list for a duplicate symbol table entry, initializing it to a subprogramme entry. Branch to the common exit routine where we join with the processing of the various "TYPE" function statements (e.g. REAL FUNCTION). There we link the variable list entry to the function name list entry, save the entry point address, and initialize base register R11 at execution time (CBAR11).



We then output the object code for the entry sequence. The stack is checked for a left bracket and we go off to the argument list processor. Upon return, we output the last argument, and after initializing the "GO TO" switches, we return to SCAN.

4.2.3. Subroutine Statement Processor (LSUBR)

The subroutine statement processor does the same preprocessing as the FUNCTION statement processor until we look up the entry point name in the variable list. Here we enter it into the list as a subroutine entry point. We then save the entry point address in the symbol table entry and move the entry point coding into the object code area. If the stack contains a left bracket, we go off to the argument list processor. Then we move in the last argument of the model argument list and go off to the common exit routine to set the "GO TO" switches and return to SCAN.

4.2.4. Argument List Processor (LARGL)

This routine processes argument lists for main entry point statements. The syntax is checked in the normal manner as one scans the argument list. The variables are inserted into the variable list by calling the lookup routine. If an entry is "FOUND" when it is looked up, this indicates either duplicate entries in the ARG list (ERROR FN-3) or the entry point name is in the argument list (ERROR FN-8). If we come upon a STAR argument (i.e. multiple return) in a function subprogramme, then we print an error message (FN-5). For variable type arguments we simply enter them in the variable lists either as parameters or parameters by name (equivalent to simple variables, equivalenced). At the end of the argument list, we check for a closing right bracket and the end of statement symbol and return to the calling routine.

4.2.5. Type Function Statement Processors

1. COMPLEX FUNCTION (LCOMP)
2. DOUBLE PRECISION FUNCTION (LDOUB)
3. INTEGER FUNCTION (LINTE)
4. LOGICAL FUNCTION (LLOGI)
5. REAL FUNCTION (LREAL)

The above statement processors perform individual initialization and then converge to a common routine to perform statement analysis. With the exception of the DOUBLE PRECISION FUNCTION statement, each processor initializes CSRSWTCH, loads up R7 with the default mode and links to the common processing routine by R8 thereby transmitting an argument list whose values are the default and optional lengths for the respective statements. The statement is analysed to see if it has a length defined, and if it does, then the length is compared to see if it matches. If it doesn't, then an error message is given (FN-6).

Here the DOUBLE PRECISION FUNCTION statement rejoins the logic. Now we insert the name in the variable list and duplicate it for the function name list as was done for the FUNCTION statement. At this point, we join with the common processing for a FUNCTION statement.

Sample FUNCTION statement object code (unrelocated)

```
        FUNCTION    X(A,B,C)
        CNOP        0,4
X       STM         R14,R12,12(R13)
        BAL         R11,XENT
        DC          X'A0',AL3(0)
        DC          CL6'X',H'0'
        DC          X'A6',AL1(mode),AL2(pointer to A)
        DC          X'A6',AL1(mode),AL2(pointer to B)
        DC          X'A6',AL1(mode),AL2(pointer to C)
        DC          X'AC',AL1(mode),AL2(pointer to X)
```

Sample Subroutine statement object code

```
        SUBROUTINE  Y
        CNOP        0,4
Y       STM         R14,R12,12(R13)
        BAL         R11,XENT
        DC          X'A0',AL3(0)
        DC          CL6'Y',H'0'
        DC          X'AB',X'10',AL2(0)      LAST ARG.
```

Figure 4.2.1.

4.2.6. Entry Statement Processor (LENTR)

The entry statement processor receives control from SCAN, moves a branch instruction into the object code and then it performs the following checks before proceeding with the scan of the statement. It checks to see that we are not within a main subprogramme module (M/PROG). It checks to see that this is not the first statement of the subprogramme. It checks to see that the entry statement is not within a DO loop. The check that we are not within a BLOCK DATA subprogramme is performed by routine SCAN.

This done, we branch to different sections according to whether we are in a function or subroutine subprogramme.

1. ENTRY (SUBROUTINE)

The entry point name is extracted and looked up in the variable list. If found, an error message is printed (EY-0). The variable list entry is initialized to subroutine entry and the E.P. address stored. The E.P. name is moved into the prototype entry code and this code is moved into the object code area. If a left bracket follows the name in the stack, the subroutine (LEARG) which constructs the model argument list for entry statements is called. After the model argument list (if any) has been created, the last argument is added by this routine. The branch instruction which was moved into the object code prior to the entry sequence is patched up to branch to the current address of the object code. We then return to SCAN via the common exit routine for linkage statements.

2. ENTRY (FUNCTION)

The processing done here is much like the processing done for subroutine subprogramme entries with the following exceptions. When the entry point name is looked up in the symbol table it is permissable for it to be found in the variable list. With the provision that only its mode may have been previously declared (usage bit not on). This provision being met, space is set up for the function name list element which is then initialized and put in the FLIST. A pointer to the variable list entry is stored in this element along with the address of the entry point.

Of course, this being a function subprogramme entry, an argument list is mandatory rather than optional as in the case of subroutine subprogramme entries.

The last argument for function entries also includes a pointer to the variable list entry, so that the execution time return routine can load the appropriate value when the function is exited from at execution time.

3. ENTRY STATEMENT ARGUMENT LIST SCAN (LEARG)

This routine is much like the normal argument list scan routine with the exception that it allows names in the model argument list to be found in the symbol table when they are looked up. If they are found, checking is done that the previous declaration does not conflict with what is implied by the fact that the name appears in the entry statement. For example, mode may be declared, or even the type may be a dimensioned variable, but it is not allowed to have the variables in common or equivalenced as they are subroutine parameters.

Sample entry statement object code (unrelocated)

```
            ENTRY     X(A,B,C)      (subroutine)

            B         AROUND
            CNOP      0,4
X           STM       R14,R12,12(R13)
            BAL       R11,XEND
            DC        X'A0',AL3(0)
            DC        CL6'X',H'0'
            DC        X'A6',AL1(mode),AL2(pointer to A)
            DC        X'A6',AL1(mode),AL2(pointer to B)
            DC        X'A6',AL1(mode),AL2(pointer to C)
            DC        X'AB',X'10',AL2(0)      LAST ARG
AROUND      EQU       *
```

4.2.7. Return Statement Processor (LRETU)

This routine processes the return statement in FORTRAN. The following logical errors are detected upon entry from SCAN. Firstly, a return statement cannot be the first statement in a subprogramme. Then we determine whether it is a multiple return statement or not. If it is not, we output object code to branch to the execution time return routine (XRET) and then return to SCAN. If we do have a multiple return statement, then we check to see whether we are compiling a function subprogramme or a main subprogramme. If we are, then we terminate processing with appropriate error messages. (RE-2 and RE-4 respectively). Otherwise the statement is O.K., and we determine the mode of the multiple return number. It can be one of immediate value, or halfword or fullword integer, direct or indirect addressing. We then output corresponding object code for each case and return to SCAN.

4.2.8. End Statement Processor (LEND)

This routine merely scans the syntax of the END statement to see if it is correct. Then it calls Relocator Phases one and two to perform symbol table cleanup and object code relocation respectively. Then it calls for beginning of subprogramme initialization from LENDPROG and returns to SCAN.

4.2.9. End Statement Simulator (LENDS)

In the event that we are inside a subprogramme segment when we try to compile another start of programme statement (e.g. FUNCTION, SUBROUTINE) it is necessary to print an error message (EN-3), perform the processing normally done for an END statement and return to whatever statement processor called this routine. This processor does all the above. It should be noted that as there is a statement in the stack yet to be processed, LENDS (by choice) does not alter the contents of register R9.

4.2.10. Simulate Main Programme Entry (LMAIN)

This routine is called by SCAN if the first statement in a subprogramme segment is not a start of subprogramme statement (e.g. FUNCTION, SUBROUTINE, BLOCK DATA). It simulates the processing done for a special subroutine entry statement. That is, it creates an entry in the symbol table for the subroutine name 'M/PROG'. It then moves in object code for an entry point with no arguments for the main programme. This done, it returns to SCAN. It should be noted that R9 which points to the current statement in the stack, is not altered by this routine.

4.2.11. Block Data Statement Processor (LBLOC)

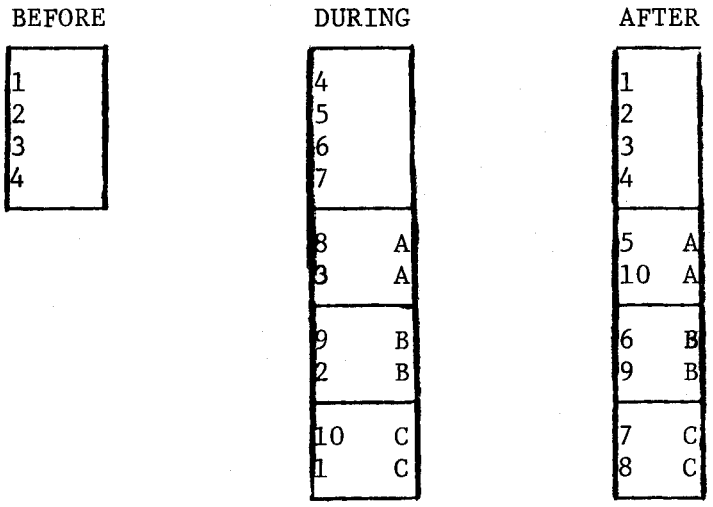
This statement does little more than check the syntax of the statement and initialize CSRSWTCH to a value which indicates that we are in a BLOCK DATA subprogramme.

4.2.12. ASF Entry Processor (LASFE)

This processor receives control from the arithmetic statement processor (ARITH) and proceeds to scan down the stack as if it were processing a FUNCTION statement. Before it does this, it moves a branch instruction into the object code which will be patched in later to branch around all code generated for this ASF. An entry sequence and model argument list are then generated and control is returned to ARITH. During the scan of the stack to create symbol table entries, duplicate entries are formed and added onto the front of the variable list. These will be the variable list entries in effect during the compilation of the rest of this statement.

At the end of the compilation, the sublist which was created by these local entries, will be removed from the front of the variable list and placed at the end of the variable list. Hence, during the rest of the subprogramme, the original variable list entries will be used. e.g. For X(A,B,C) = ...

The variable list looks like the following



where the order of the entries is indicated by the sequence number to the left.

Checking for duplicate entries is performed by storing the symbol table pointer in an unused portion of the ASF variable list entries, (the common and equivalence list pointers) to the variable list entry for the name of the ASF. In the example above, it would be the variable "X", when the lookup is performed for the model argument name. If the name is found, and the pointers are the same, then an FN-7 error is given. Another task performed is to calculate a cumulative link around the model argument list so that ARITH can eliminate this junk from the stack when it does its processing. The calculation is performed in R7 and the result shifted into R0 just before returning to ARITH.

4.2.13. ASF Return Processor (LASFER)

As described in the note just previous on the ASF entry processor, this routine cleans up the processing of the ASF statement.

This consists of

1. Moving a branch to the execution time return routine (XRETASF) into the object code.
2. Moving an end-of-ASF indicator into the object code

```
        DC      X'B6BO'
```

3. Shifting the local variable list to the end of the variable list.
4. Patching up the branch instruction to branch around all the generated code for this statement.

Control is then returned to ARITH.

Sample coding for ASF statement

X(A) = ...

```
            B       AROUND
            CNOP    0,4
X           STM     R14,R12,12(R13)
            BAL     R11,XENTASF
            DC      X'A1',AL1(0),AL2(pointer to X)
            DC      CL6'X',H'0'
            DC      X'A6',AL1(mode),AL2(pointer to A)
            DC      X'AB',AL1(mode),AL2(pointer to X)

            object code generated by ARITH for R.H.S.

            B       XRETASF
AROUND      DC      X'B6BO'     end-of-ASF indicator
```

4.2.14. Beginning of Subprogramme Initialization - (LENDPROG)

This routine initializes the compiler for the beginning of a subprogramme segment. Specifically this is: initialization for symbol table lookups, initialization of temporary lengths, initialization of the implicit mode table, and saving of the beginning of programme address and the bottom of the symbol table address (CSYMBASE). It should be noted that R9 is not altered by this routine. This was done because this routine can be called by the end statement simulator and an as yet unprocessed statement may be in the stack. This routine is also called at the beginning of job initialization from the compiler deck MAIN.

4.3 SPECS

4.3.1. Introduction

The routine SPECS contains processors for the following statements: DIMENSION, Type, COMMON, EQUIVALENCE, IMPLICIT and EXTERNAL. Table 4.3.1. shows the various sections.

In general this routine does not generate any object code but rather creates new, or changes old symbol table entries for variables.

```
e.g.    REAL          N,Z
        DIMENSION     N(20)
        EQUIVALENCE (N(1),Z)
```

On obtaining the first statement of the example SPECS would enter N and Z as new variables in the symbol table. Appropriate bits will be set on to indicate the mode, type and length etc. of the variables. The second statement merely adds new information about the variable N. Hence the appropriate bit would be entered in the symbol table to specify that N is subscripted and a "dimension" list established. (See description of VLIST section 2.4.1.) The third statement would require that an "equivalence" list be generated for N and Z and that again appropriate bits be set on to indicate that N and Z are equivalenced. While the above entries are being made or modified SPECS checks for various error conditions such as re-typing a variable.

SPECS will cause code to be generated if the type statement includes data initialization

```
e.g.    REAL  K/1./
```

CSECT name	Function
1. TDLIPCDP	Type and Dimension Processor
2. TEQUIVAL	Equivalence Processor
3. TCOMMON	Common Processor
4. TIMPLCIT	Implicit Processor
5. TEXTRNAL	External Processor
6. TSAVER	Save Area, Data Area, Service Routines

Table 4.3.1.

Entry points to TDLIRCDP

Entry Point	Purpose
TDIME	- Dimension Statements
TLOGL	- Logical Statements
TINTGR	- Integer Statements
TREAL	- Real Statements
TCMPLX	- Complex Statements
TDBLPREC	- Double Precision Statements

Table 4.3.2.

4.3.2. Tables & Switches used in SPECS

NAME	LENGTH	VALUE SET	USAGE
COTHSTAT	1	X'00'	Only a function, type function Subroutine, Block Data or Implicit Statement has occurred.
		X'FF'	Other Statements have occurred.
CIMPLIT	1	X'FF'	Implicit Statement has occurred.
		X'00'	Implicit Statement has not occurred. It is used to check that Implicit is first statement in a programme segment.
TAYBL (EQUATED TO XLENTAB IN STARTA)			Used by Implicit to check validity of optional lengths.
CTYPESW	1	X'02'	Call to INOUT is to process initialization.
		X'01'	Call to INOUT is for end-of-statement tidy up.
CIMPLT	42		Implicit table. Also used by LOOKUP.
CHECKINT	4		Checking integer for equivalence.
TML1	1		Set to Default bit configuration in VLIST by type and dimension processor.
TML2	1		Bit configuration to be put in B1 for a particular variable. Set by type and dimension processor.

NAME	LENGTH	VALUE	COMMENTS
TPVLINK	1	C'N'	No previously equivalenced variables have yet appeared in the current list of the equivalence statement being processed.
		C'Y'	At least one such variable has been found.
TSUDEQV	1	C'N'	Variable is not pseudo-equivalenced.
		C'S'	Pseudo-equivalenced, and not in common.
		C'T'	Pseudo-equivalenced variable appears in common statement.
TCRESS1	1	C'N'	No data initialization for current statement.
		C'Y'	At least one variable was initialized. Must go to INOUT to fix object code.
TRANGER	1	C'N'	No error yet in Implicit.
		C'Y'	Error Flag for Implicit.
TDATSW	1	C'D'	If slash occurs in type statement with no dimensioning it indicates initialization.
		C'N'	No Slash.
TBLNK	1	C'B'	Blank Common.
		C'L'	Labelled Common.
		C'E'	Invalid Common Block Name.
TSSVABL		C'Y'	Subscripting or Dimensioning follows.
		C'N'	No subscripting or dimensioning.
TERROR	1	C'N'	
		C'E'	Error Flag.
TFICLST	1	C'F'	Current Variable is first in Common Block.
		C'N'	Not first.
TDIMSW	1	C'D'	Dimension statement.
		C'T'	Type statement.
		C'C'	Common statement.
		C'E'	Equivalence statement.
TSPPARM	1	C'N'	Current variable is not a S/R parameter.
		C'Y'	Current variable is a S/R parameter.
TEQCOMER	1	C'E'	Error has occurred in compilation of subscript or dimension.
		C'N'	No error.
TSHIFTAB	8		Contains number of shifts, one for each data type and length, required to convert product of dimensions into an array length in bytes.
JUMPTAB	8		A macro which generates 8 1-byte address constants used as an index to process subscripts for dimensions.

NAME	LENGTH	VALUE	COMMENTS
TSS	32		First seven words contain the subscripts or dimensions associated with the variable. The last word contains the number of SS or DIMS.

4.3.3. Linking Operations in SPECS

Unfortunately the desired meaning of the word 'link' is not always clear when used in this documentation. Basically the word can be used in three different contexts depending on whether it is used as a noun or verb.

(a) Noun - in this case 'link' is a half word offset or displacement from some location, and is used to get a pointer to another entry.

e.g. ZVLINK, ZCOMLINK, ZEQVLINK, ZVDIM, ZVCOMM, and ZVEQV are all links in the symbol table. ZSTLINK is the forward link in the stack. Links in the symbol table are subtracted from the reference address to get the new entry. Stack links are added.

(b) Verb - to link means to create a link between two entries in the symbol table. If we create a link from entry A to entry B, whose addresses are in GPR R2 and R3, say, then the following code is used.

```
LR   R15,R2
SR   R15,R3
STH  R15,--
```

Note the use of a work register, R15. The reason is that the location in which the link is stored is usually dependent on the address in R2.

(c) Verb - to link from A to B can also mean to get a pointer to entry B using the pointer to entry A and a link relative to A.
e.g. if the address of A is in R2

```
LR   R7,R2
SH   R7,n(R2)   n = [0,2,4,...,14].
```

Again note that we may wish to preserve R2, hence the use of work register R7.

4.3.4. OBJECT CODE

The only object code generated by SPECS is done indirectly. INOUT produces the object code for initialization in type statements.

4.3.5. DETAILED DESCRIPTION OF PROCESSORS

Type & Dimension

At each entry point for TYPE statements R0 and R2 are loaded with the standard and optional lengths for the variable (other initialization is also performed). DOUBLE PRECISION and DIMENSION have identical syntax rules so they are treated in the same fashion. They have a separate call to the SETSTACK routine, after which they transfer control to label TESTNAME. All the others go to label TESTAR where TDIMSW is set to C'T' and the SETSTACK routine is invoked. Then the stack is examined to see if '*n' follows the statement identifier. If so, 'n' is checked against the contents of R0 and R2 for validity. If 'n' is the optional length, bit 7 of TML1 is set to 1.

All statements now transfer to TESTNAME. TML2 is set to TML1 and the STACK is checked for a symbol (i.e. the fourth byte in the current stack entry). Permissable values are of the form X'0n' where n is usually 1 or 2.

For Dimension Statements (TDIMSW = C'D') the type bits (bits 5, 6, 7) of TML2 are set by extracting the first character of the symbol in the stack and using this to pick up the corresponding element in table CIMLT, which is then OR'd with TML2. The next section, up to label TLOOK1, does some checking on the operator that follows the name. The operator in question is in the next stack entry (i.e. after the one containing the name). If the code for a left bracket (X'03') is detected, TSSVABL is set to C'Y' to indicate dimensioning, and control is passed to TLOOK1. If a terminator (X'01') is detected statement processing ends. If an asterisk (X'4C') is found, the operand in that stack entry is checked against the standard and optional lengths for that data type, which were saved previously. On detection of a slash TDATSW is set to C'D'. If either an '*' or a / is detected for a dimension statement, a diagnostic is issued. If none of these operators is found, a diagnostic is issued and processing resumes if the operand in that entry is a symbol (or a null operand). The current stack pointer is saved, and the one pointing to the previous entry is loaded. Control then passes to the symbol lookup routine, via the LOOKUP macro. There are two returns, for new and for old symbols.

For new symbols, the following processing is performed:

1. Fix up B1, by 'OR' ing TML2 into the low order 3 bits.
2. Set the type bit in B2 if this is a type statement.
3. Compile dimensions, if any, by going to TSUSDET. If there are dimensions, a dimension list is set up and the dimensions are set. The symbol table entry contains a link to this new list as follows:

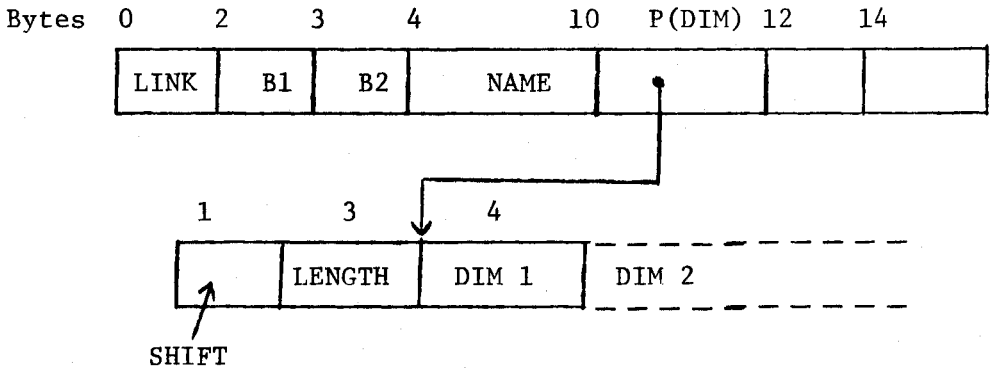


Figure 4.3.1.

P(DIM) is a half-word offset from the symbol entry. Note that it does not point to the start of the dimension list. This list will be described more fully under routine TGENDIME. Control then passes to label TESTINIT to check for initialization.

For old symbols, the following checking and processing is performed.

1. Check that the symbol is a variable name.
2. Check if dimension bit (bit 1 of B1) is on. If not, processing continues at label TNPREVDI. If it is on, check if variable actually is dimensioned. This is the case if the dimension field is non-zero. If the switch TSSVABL, has the value C'Y', an error is given for an attempt to re-dimension. In addition to these conditions if this is a type statement, the first word in the dimension list must be changed to reflect the new element length.
3. Compile dimensions if any, and generate a dimension list (routines TSUSDET, and TGENDIME). The second routine is not entered if this variable is pseudo-equivalenced.
4. Check that the variable is not

   (a) A DO parameter
   (b) An ASSIGN'd variable } Bit 3, 4, 5 in B2
   (c) Initialized

5. Check for the situation in which an equivalenced variable is being typed. If the element length of the new data type is different from its present length, an error condition exists. This is an implementation restriction, e.g.

```
e.g.    EQUIVALENCE (X(3), Y(4))
        COMPLEX X(5)            results in error
        INTEGER Y(10)           valid
```

   This condition can be avoided by having the type declarations precede the EQUIVALENCE statement.

6. If this is a type statement, the type bits in B2 are reset using TML2, and the type established bit in B2 is turned on.

7. If the pseudo equivalence condition (4.3.6.) exists and TSSVABL = C'Y' then routine TEQVFIX is entered. This routine is described in section 4.3.10.

8. TDATSW is checked for C'D', in which case, TCRESS1 is set to C'Y' to indicate that a call to IDATA was made. CTYPESW is set for the use of that routine, and control passes to it, to process the initialization. On return CTYPESW is checked for an error code in which case compilation of the current statement terminates with a branch to TIZEND.

Both new and old variables then are at label TNSUBSEQ where additional syntax checking is performed. Specifically the following is checked for:

1. A right bracket (dimensions).
   If one is found, a slash (initialization) can still occur. If a slash is found the same coding as in point No. 8 above is executed.

2. Check for a comma (more variables) or a terminator (end of statement). For the latter control is passed to TIZEND.

At location TIZEND end of statement "fixing up" is done i.e.

(a) If initialization occurred (TCRESS1 = C'Y') then another call to IDATA is made with a different switch setting for CTYPESW. This is done so that IDATA can patch up the object code it has generated.

(b) TCRESS1 is reset to C'N' and TML1 is moved into TML2 (in case this is a type statement).

For new symbols, the following processing is performed:

1. Fix up B1, by 'OR' ing TML2 into the low order 3 bits.
2. Set the type bit in B2 if this is a type statement.
3. Compile dimensions, if any, by going to TSUSDET. If there are dimensions, a dimension list is set up and the dimensions are set. The symbol table entry contains a link to this new list as follows:

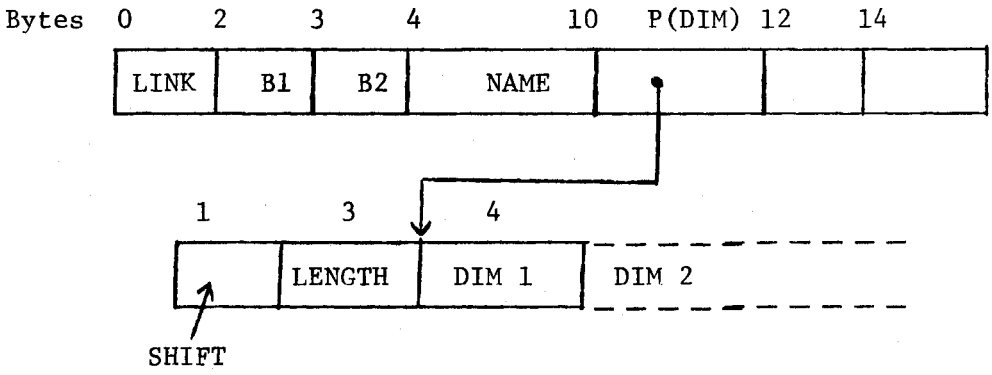


Figure 4.3.1.

P(DIM) is a half-word offset from the symbol entry. Note that it does not point to the start of the dimension list. This list will be described more fully under routine TGENDIME. Control then passes to label TESTINIT to check for initialization.

For old symbols, the following checking and processing is performed.

1. Check that the symbol is a variable name.
2. Check if dimension bit (bit 1 of B1) is on. If not, processing continues at label TNPREVDI. If it is on, check if variable actually is dimensioned. This is the case if the dimension field is non-zero. If the switch TSSVABL, has the value C'Y', an error is given for an attempt to re-dimension. In addition to these conditions if this is a type statement, the first word in the dimension list must be changed to reflect the new element length.
3. Compile dimensions if any, and generate a dimension list (routines TSUSDET, and TGENDIME). The second routine is not entered if this variable is pseudo-equivalenced.
4. Check that the variable is not

   (a) A DO parameter
   (b) An ASSIGN'd variable } Bit 3, 4, 5 in B2
   (c) Initialized

5. Check for the situation in which an equivalenced variable is being typed. If the element length of the new data type is different from its present length, an error condition exists. This is an implementation restriction, e.g.

```
e.g.    EQUIVALENCE (X(3), Y(4))
        COMPLEX X(5)            results in error
        INTEGER Y(10)           valid
```

   This condition can be avoided by having the type declarations precede the EQUIVALENCE statement.

6. If this is a type statement, the type bits in B2 are reset using TML2, and the type established bit in B2 is turned on.

7. If the pseudo equivalence condition (4.3.6.) exists and TSSVABL = C'Y' then routine TEQVFIX is entered. This routine is described in section 4.3.10.

8. TDATSW is checked for C'D', in which case, TCRESS1 is set to C'Y' to indicate that a call to IDATA was made. CTYPESW is set for the use of that routine, and control passes to it, to process the initialization. On return CTYPESW is checked for an error code in which case compilation of the current statement terminates with a branch to TIZEND.

Both new and old variables then are at label TNSUBSEQ where additional syntax checking is performed. Specifically the following is checked for:

1. A right bracket (dimensions).
   If one is found, a slash (initialization) can still occur. If a slash is found the same coding as in point No. 8 above is executed.

2. Check for a comma (more variables) or a terminator (end of statement). For the latter control is passed to TIZEND.

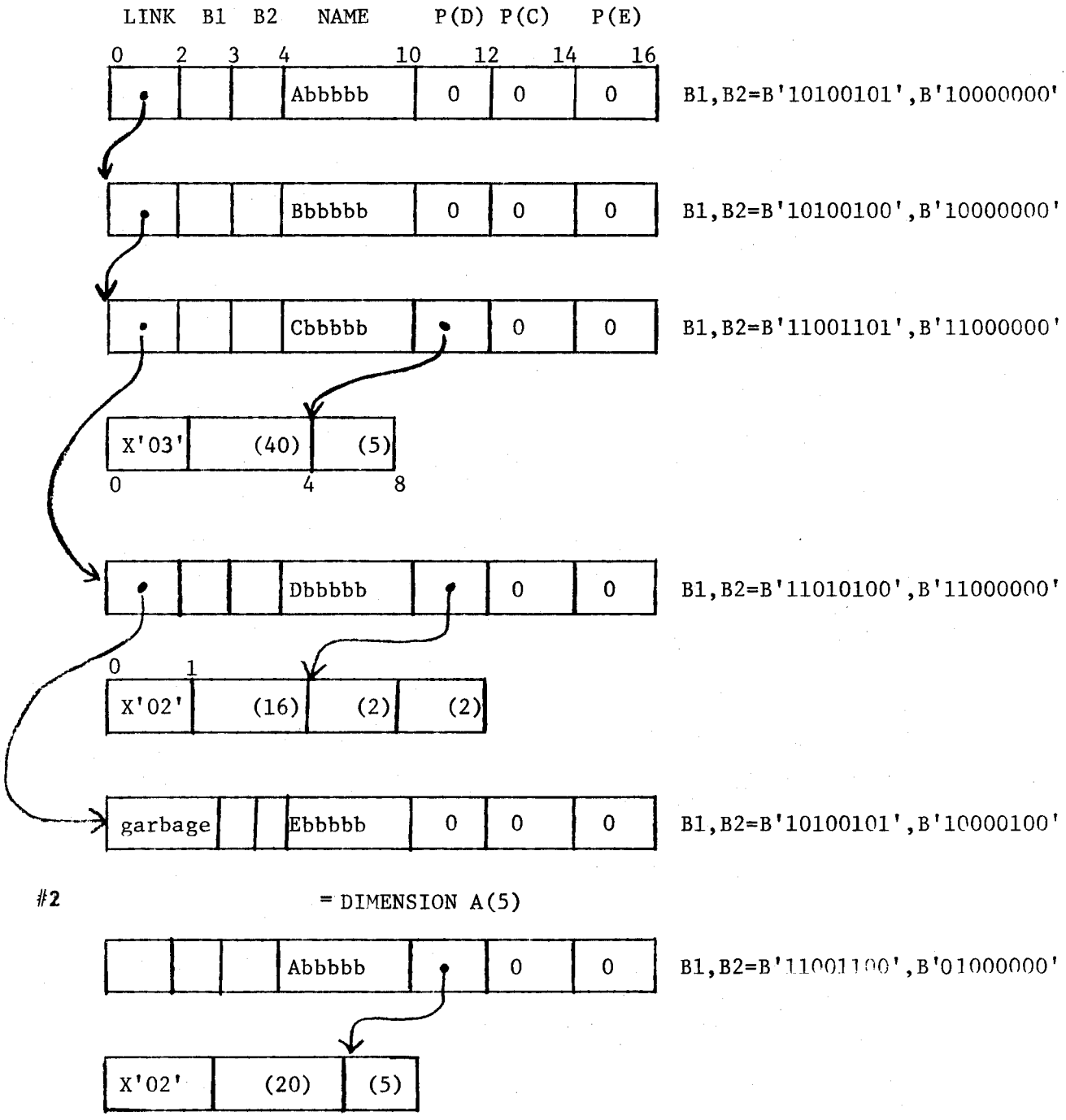
At location TIZEND end of statement "fixing up" is done i.e.

(a) If initialization occurred (TCRESS1 = C'Y') then another call to IDATA is made with a different switch setting for CTYPESW. This is done so that IDATA can patch up the object code it has generated.

(b) TCRESS1 is reset to C'N' and TML1 is moved into TML2 (in case this is a type statement).

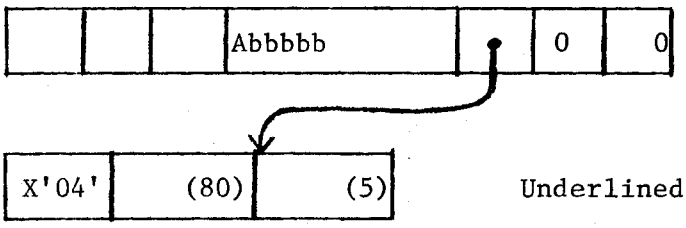
Examples of type and dimension statements.

#1 REAL*8 A, B*4, C(5), D*4(2,2)/4*0.1/, E*8/10D+0/



#2 = DIMENSION A(5)

COMPLEX *16A



Underlined areas have been changed.

4.3.6. EQUIVALENCE

At this point it would be wise to clarify some of the terminology used from here on in.

1. New list: could have either of two meanings:

   (a) a list of variables of the source statement, all of which are in equivalence e.g. EQUIVALENCE (A(3), B, C(4)), (X,Y,Z).

   (b) the linked list of symbol table entries resulting from the variables above (a).

Since these two meanings are almost synonymous no distinction is made between them.

2. Old list: If, while processing a new list, a previously equivalenced variable is encountered, the list to which that variable belongs is called the old list.

   Note: New list = new equivalence list
   Old list = old equivalence list

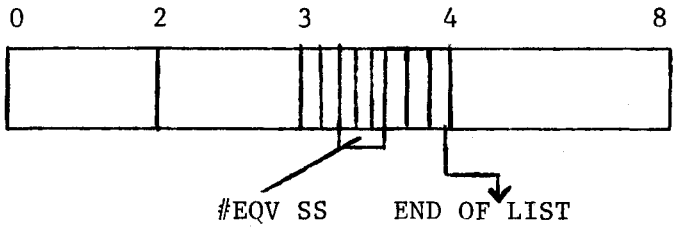
3. EQV is an abbreviation for equivalence.

4. VECT is the position of a subscripted element in the array. This holds only when #SS > 1 and the variable has been dimensioned. Its value is calculated by routine TVECTOR2 (see section 4.3.10.)

5. Dimensioned variable - is one that has been given dimensions in a DIMENSION, TYPE or COMMON statement.

6. Description of Equivalence Double Word

   - the EQV DWD (ZEQVENT) consists of two full words in the symbol table, aligned on a full word boundary. It contains information for use by the equivalence algorithm and dimensioning routines.



Byte 0,1 ZEQVLINK - a half word link to the next symbol entry in the equivalence list (filled in when the next variable is processed) or a link to the first symbol entry in the list if the current variable is the last element in the list.
Byte 2 ZEQVBYT1 - a one byte checking integer (see section 2.4)
Byte 3 ZEQVBYT2
Bit 0,1 - not used
Bit 2-4 - the number of equivalence subscripts. This field is filled in if the variable in question has not been dimensioned, but appears with subscripts in this equivalence statement. The variable is said to be pseudo-equivalenced (for lack of a better name). This is done in routine TDIME2 (see section 4.3.10). These bits have the same relative position in ZEQVBYT2 as the # dimensions bits do in ZVBYT1 i.e. B1 of symbol entry.
Bit 5 - unused
Bit 6 - this bit is set if an offset was assigned to the variable
Bit 7 - end of equivalence list indicator = 0 not end
= 1 end
Byte 4-7 ZEQVOFFS - a full word that contains an offset (in bytes) of the variable from an arbitrary reference point (0).

The FORTRAN statement

EQUIVALENCE (A(3),B(5),C(8),X,Y(1,2,3))

has the following core layout and will be used as an example. Calculations to determine the starting position of an array in the table are done relative to '0' and the amount of shifting to the right or left is called the offset.

	-7	-6	-5	-4	-3	-2	-1	0	+1
A -2						A(1)	A(2)	A(3)	A(4)
B -4				B(1)	B(2)	B(3)	B(4)	B(5)	B(6)
C -7	C(1)	C(2)	C(3)				C(7)	C(8)	C(9)
X 0								X(1)	
Y 0									

where A, B, C, X and Y are superimposed on each other. This graphical description is the basis for the offset algorithms.

Note that although X may or may not be an array it is nonetheless treated as an array of dimension one. Note also that since Y has not been dimensioned yet, (we assume it will be later on) and has more than 1 subscript we have no way of knowing where it fits into the above layout. Hence its offset is set to zero for the time being.

All non-dimensioned variables referenced with subscripts in an EQUIVALENCE statement, have their subscripts, saved (by routine TDIME2 (see section 4.3.10) in a list having a format identical to that of the dimension list, except that the subscripts occupy the locations where dimension would normally be. When dimensioning information is eventually provided, this list of subscripts is used for checking. In the case where the number of these subscripts is greater than 1, it is used to calculate the offset. If the number of dimensions (from dimensioning source) is equal to the number of equivalence subscripts, then the dimensions replace the subscripts in this list, otherwise (# dim > #SS) a new list is created. This processing is performed in routine TEQVFIX (see section 4.3.10.)

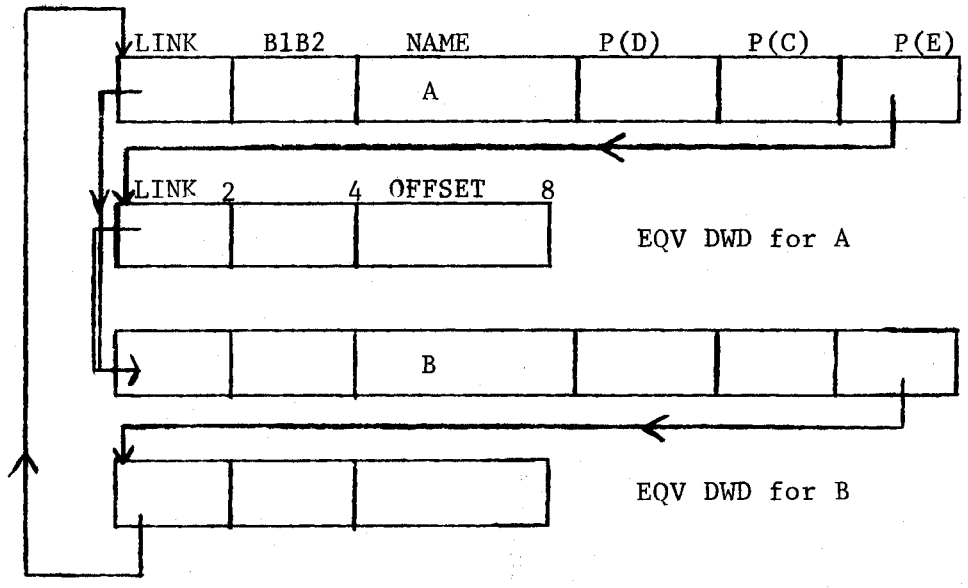
The offset is used by phase 3 of the relocator to assign core storage to these variables.

Algorithm for Calculating Offsets

a. For a new list whose elements were not previously equivalenced

1. Set up an equivalence double word (EQV DWD) for each new member of the list. The first half-word contains a link to the symbol table entry for the next member of the list. The last member of the list links back to the first member.

e.g. EQUIVALENCE (A,B) sets up the following symbol table.



2. In ZEQVBYT1, the third byte of the EOV DWD (referred to symbolically as ZEQVENT) set up an integer for checking. This value is contained in the full word CHECKINT in COMMR. Its value at any one time is the number of equivalence lists processed in the programme segment.

3. Calculate the offset and store it in 2nd word of EQV DWD and link the previous EQV DWD (if there was one) to the current symbol table entry.

   (i) OFFSET = TREFVALU - (SS-1)*ELTLEN for EQ...A(15), ...

   (ii) OFFSET = TREFVALU for EQUIV...(A, ...
   or EQUIV...(A(3,2,6),...
   if dimensions are known

   (iii) OFFSET = TREFVALU-(VECT-1)*ELTLEN for REAL A(4,4).
   EQUIVALENCE (A(2,3),...
   where TREFVALU has the value zero (0) in this case. ELTLEN is the number of bytes per element of this variable.

b. When a new list first links into an old list:

1. Set a switch (TPVLINK to C'Y') to indicate that the current list has linked into an old one.

2. Go through the old list resetting the checking integer (ZEQVBYT1) to agree with that of the current (new) list i.e. CHECKINT

3. REFERENCE VALUE (TREFVALU)

   = OFFSET of the Element of the Member of old list linked into or
   = OLD OFFSET (of the member of the old list) + (SS-1)*ELTLEN if NI = 11,TSSVABL=C'Y' or
   = OLD OFFSET + (SS-1)*ELTLEN if NI = 10,TSSVABL=C'Y' or
   = OLD OFFSET if TSSVABL = C'N' and NI = 10 or = 11

4. Reset offsets of all members already processed in the new list as follows:

   NEW OFFSET = OLD OFFSET + TREFVALU

5. Attach old list into the end of the new list and refer to the combined list as new.

6. Offset of all further new members in list is as described in (a) part 3 above.

c. When a new list links into an old list on subsequent occasion:

1. This situation is detected by testing TPVLINK which was set in (b.1.) to C'Y'.

2. Temporary Reference Value (TMPRFVAL)

= TREFVALU - OLD OFFSET - (i) 0
(ii) (SS-1)*ELTLEN
(iii) (VECT-1)*ELTLEN

(i) if TSSVABL = C'N'
(ii) if TSSVABL = C'Y' and #SS = 1
(iii) if TSSVABL = C'Y' and #SS > 1 (and has been dimensioned)

where **OLD** OFFSET is that of member of old list referred to

3. Go through old list

(i) Reset the checking integer in ZEQVBYT1 so that it agrees with that of the current list i.e. CHECKINT
(ii) Reset all offsets in the old list, as follows
NEW OFFSET = OLD OFFSET + TMPRFVAL

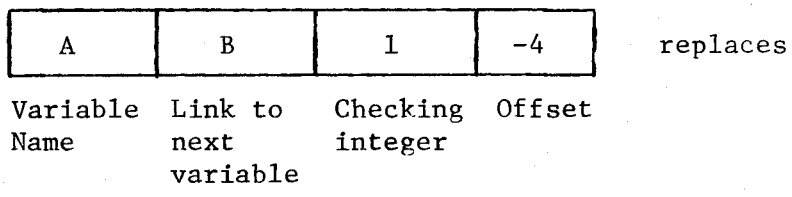
4. Attach the old list onto the end of the new (current) list by performing the proper linking. Call the resulting list new.

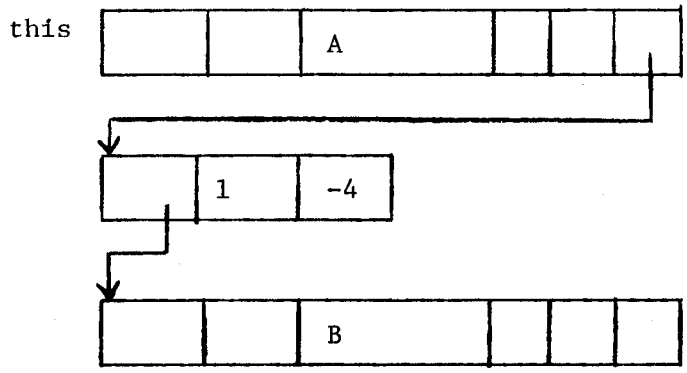
d. Purpose of the checking integer

1. This integer makes possible the detection of an attempt to equivalence a variable to itself. Such an occasion occurs if, when linking into an old list, a member of that list has a checking integer equal to that of the new list.

3. Examples

For the purpose of illustrating graphically a more simplified typical entry of the symbol table is used.





Also assume that the following statement has appeared.

IMPLICIT LOGICAL*1(A-Z,$)

This is for convenience in calculating offsets. Also, it is easier to see what is going on.

e.g. #1 EQUIVALENCE (A(3), B(5), C(8), X, Y(1, 2, 3))

results in the following list.

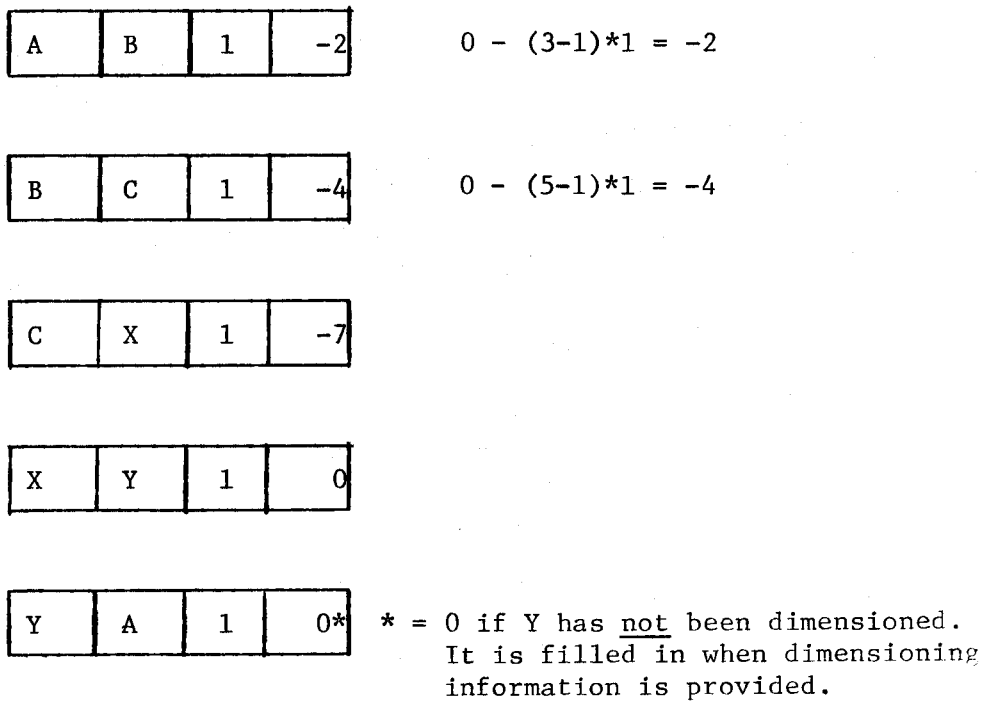


Figure 4.3.3.

EQUIVALENCE (D(19), E(11))

D	E	2	-18

E	D	2	-10

EQUIVALENCE (F(4), B(7), G, E(3), H(10))

F	A	3	-1

A	B	3	-2

TREFVALU = -4 + (7-1)*1 = 2
-4 is the offset for B in the old list.

B	C	3	-4

C	X	3	-7

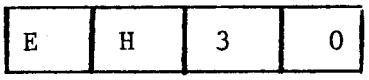
X	Y	3	0

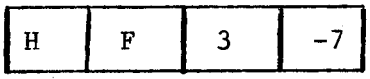
Y	G	3	0

G	D	3	2

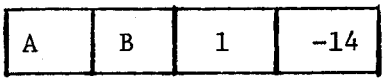
TMPRFVAL = 2 - (-10) - (3 - 1)*1 = 1

D	E	3	-8

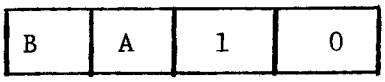





```
#2  DIMENSION A(2,3,4)
    EQUIVALENCE (A(1,2,3),B)
```




```
VECT = 1 + (1-1)1 + (2-1) 2+
       (3-1)2*3 = 15
Offset = -(VECT-1)*ELTLEN
```



4.3.7. COMMON PROCESSOR

Compilation of Common Statements begins with the setting of TDIMSW to C'C', followed by a call to the CSETSTAK routine (see section 3.10.). On return we check for a name in the operand field of the first stack entry in which case we are working with blank common and TBLNK is set to C'B' to indicate this. Otherwise we have a null operand and the operator field of the next entry should contain a slash (/). If this slash is followed by another slash in the succeeding entry, the operand between them is the common block name. If there is a name, i.e. not a null operand, TBLNK is set to C'L' for labelled common, but if the operand is null TBLNK is set to C'B'.

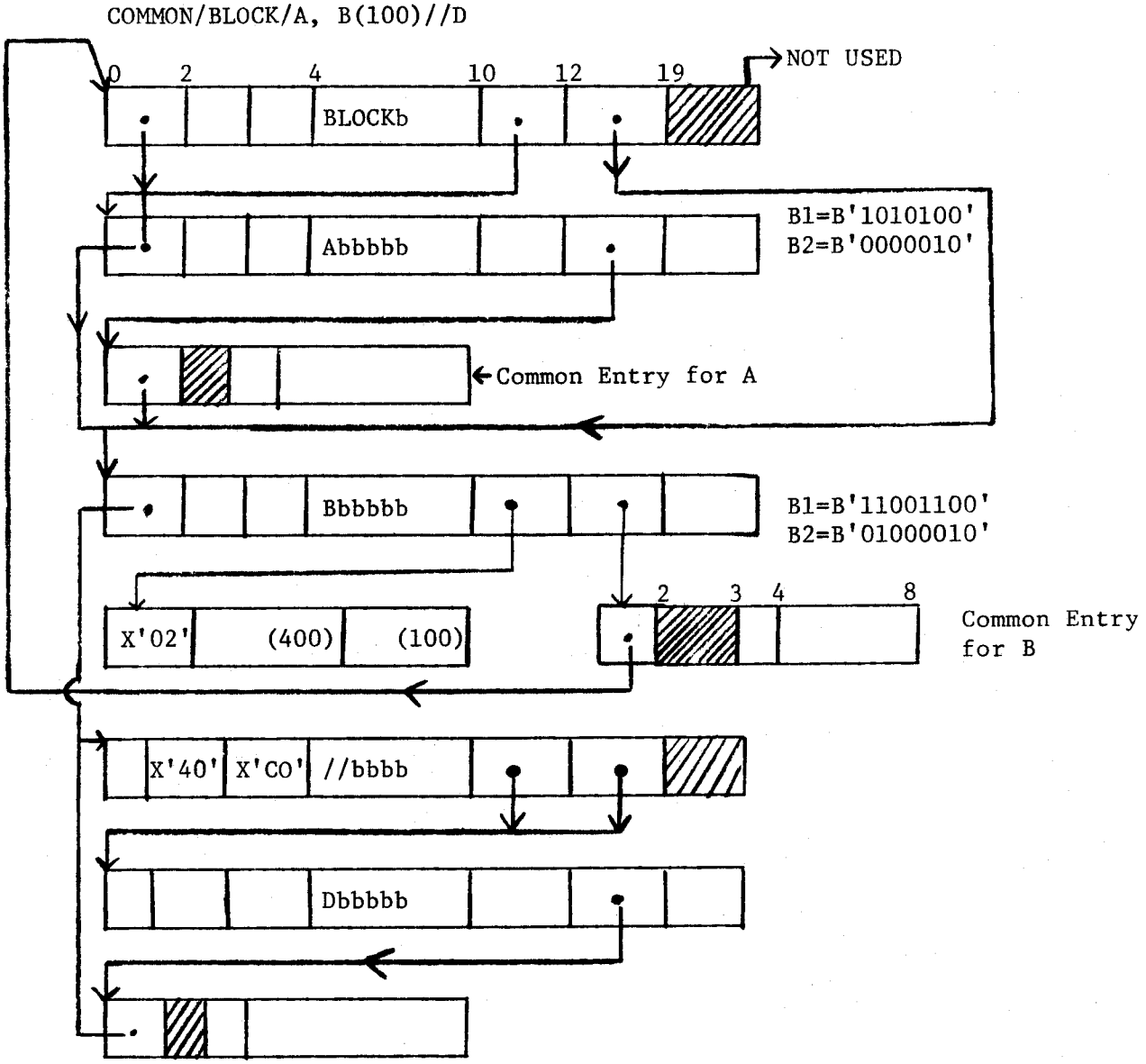
Next, a lookup on the common block name is performed. Depending on the type of common block name (blank or labelled) the CLBCOM or CLSYM lookup routine is used. If the symbol is new, B1 is set at X'40', the code for common block names. The type and usage established bits in B2 are also turned on. If the symbol is old, B1 is checked for the above code. If the test fails, an error is given and TBLNK is set to C'E'. Now the compilation of variables can begin.

The following are some of the conditions that are checked for in old symbols after the lookup of the symbol.

1. Does B1 have the code for a variable?
2. Is this variable already is common? If so - error.
3. Are there dimensions (TSSVABL = C'Y')? If so, link to TSUSDET and TGENDIME. This procedure is similar to that used for type and dimension statements.
4. If it's already dimensioned, is TSSVABL = C'Y'? If so - error.
5. Are the DO, ASSIGN and INITIALIZED bits in B2 on? If so - error.
6. Is this variable equivalenced? Check that none of the variables it is equivalenced with are in common.

Before continuing, a description of the linkage between common block entries, variable entries and common entries is necessary. A two word common entry is set up in the symbol table in much the same fashion as the equivalence double word, for a similar purpose.

The following symbol tables entries would result from this common statement.



Note that bytes 10,11 of the common block entry are a half word link to the entry of the first variable in the common block. Bytes 12 and 13 are a half word link to the entry for the last variable in the common block. Note also, that the linking structure of variables and their common entries are the same as equivalence, except that the link in the last common entry, points to the common block name rather than to the start of the list.

A description of the common entry is in order here. Byte 0,1 link to next variable in common block or to the common block if this is the last variable in the block. The latter is the case after compilation

of each variable has finished, no matter how many more variables there are to be processed in that block.

2	Not used
3	Bit 0-6 Not used
	7 = 1 End of list indicator
	= 0 normal i.e. not end of list
4-7	Not used by SPECS. RELOC sets up an offset here (see section 4.8)

The following is a brief description of how the links between the variable entry, common entry and common block entry are created.

1. If the current variable is the first in the common block, set TFICLST to C'F' and skip the next step.
2. If the current variable is not the first in the common block, link to the last (i.e. previous) member (element) of the common block using the link in bytes 12 and 13 of the common block entry. From this variable entry, link to its common entry using ZVCOMM, and turn off the end of list indicator in ZCOMBYT2 and keep the pointer.
3. Generate and zero a two word common entry and create a link to it in ZVCOMM of the current variable entry.
4. Create a link from the common block entry to the current variable in bytes 12, 13 of the common block entry.
5. If the current variable is the first in the common block also create the same link in bytes 10, 11. i.e. the latter is a link to the start of the linked variable list of commoned elements, and the former is a link to the tail of this list.
6. Turn the common bit (bit 6) in B2 on and also the end of list indicator in ZCOMBYT2.

4.3.8. IMPLICIT Processor

The Implicit processor has two phases, one, to update a table (CIMLT) using the information from the source statement, and two, to retype subprogramme arguments and not-explicitly-typed function names. The format of CIMLT is as follows:

- 42 bytes, one for each character in the binary collating sequence from A to Z inclusive, plus one more for the '$'
- the bit configuration of each byte is:

0	1	2	3	4	5	6	7	8
n	o	o	o	o	o	t	t	t

n = 1 if this letter has been typed by IMPLICIT
  = 0 otherwise
ttt = type and length bits as they would appear in B1 of variable list. The algorithm used is as follows:

- look for a valid data-type keyword and set TMODLEN accordingly i.e. B'10000ttt'
- check syntax for left bracket,
- check syntax for a single alphabetic character operand and get a pointer to the corresponding entry in CIMLT,
- check the operator in the next stack entry for '-', ',', ')'. For '-' check the operand in that entry for a single alphabetic character and get a pointer to its entry in CIMLT. Propogate TMODLEN between these two bounds. For a ',' or ')' simply put TMODLEN into entry pointed to by first pointer.
- the previous step is repeated if ',' appears after ranges e.g. A-E, or after single character specifications. After a right bracket a search for another data type keyword is initiated, or for the end of the statement.

Note that since the position of the letter '$' in the alphabetic sequence is not defined by IBM, specifications of the following type are illegal and generate error messages.

--$-F, M-$ --

Actually, the '$' is treated separately because its position in the table (after 'Z') does not agree with its binary representation.

Routine TPOSTIMP is entered after the end of statement has been encountered. CSRSWTCH is tested to see whether we are in a function or

subroutine. Block Data or M/PROG settings of this switch cause branching out of this routine. If we are in a subroutine, the first entry in the VLIST, the subroutine name, which cannot be typed, is skipped. Function names are retyped. Each succeeding entry, if any, is processed as follows:

1. Get the first character of the name and pick up the corresponding byte in CIMLT.
2. If bit 0 of the byte from CIMLT is on, then the low order three bits of ZVBYT1 (B1) are replaced by the low order three bits of the CIMLT entry.

4.3.9. EXTERNAL Processor

This processor is very trivial in nature, because of the simple syntax and the minimum of symbol table modification. As each name is picked from the stack it is checked if it was previously used for the following errors:

1. Is it an array?
2. Commoned or Equivalenced?
3. Previously externalled?
4. Usage other than in type statement?

If the name passes this test, bits in B1 and B2 are set to indicate that it is externalled, used as a function or subroutine, and it is assumed that it will be a subprogramme.

4.3.10. SERVICE ROUTINES

1. TVECTOR2 - This routine calculates the position in an array of an element with 2 to 7 subscripts, provided that the number of dimensions of the array match these. The value that is calculated is the following:

$$v = 1 + \sum_{i=1}^{n} [(s_i - 1) \prod_{j=0}^{i-1} D_j]$$

where n = #ss = # dims $\quad 2 \leq n \leq 7$

s_i = the subscripts

D_j = the dimensions, $D_o = 1$

This value, decremented by one is usually used as input to the ELTLON macro which produces a value in bytes rather than elements.

Inputs - R15 points to ZDIME, the dimension list
- R14 is the return address
- R4 #ss

Outputs - R0, R3 the value 'v' as calculated. This routine is used by the equivalence processor and by routine TEQVFIX.

2. TEQVFIX - This routine is called by the type, dimension and common processors when a non-dimensioned equivalenced variable is to be dimensioned. Four different cases must be distinguished before processing can begin.

(i) #ss (from ZEOVBYT2) = #DIMS(in R4) = 1
(ii) #ss (from ZEQVBYT2) = 1, #DIMS(R4) > 1
(iii) #ss (ZEQVBYT2) = #DIMS(R4) 1
(iv) #DIMS(R4) ≠ #ss (ZEQVBYT2) ≠ 1

The last case being an error condition.

In case (i) the offset has already been calculated, so the dimension has to be placed in the dimension list and the array length in bytes, placed at the head of the dimension list along with the shift byte. The # dims is also put into B1.

Case (ii) is almost the same as case one, except that a new dimension list must be created, because the old one is too small.

In Case (iii), the offset has not been calculated. In order to use routine TVECTOR2 to accomplish (part of) this, the subscripts which are in the dimension list, and the dimensions which are in TSS must be interchanged. Also the # dims is put into B1 and the array

length is calculated, as in cases (i) and (ii). Now a call is made to routine TVECTOR2. The offset is now created as described in the equivalence processor (see section 4.3.6.) and stored in TEQVOFFS.

Inputs: R14 - Return address
R9 - Pointer to symbol table entry
R4 - # dimensions

In addition the following registers are set up in the routine

R15 - Pointer to dimension list
R8 - Pointer to equivalence entry.

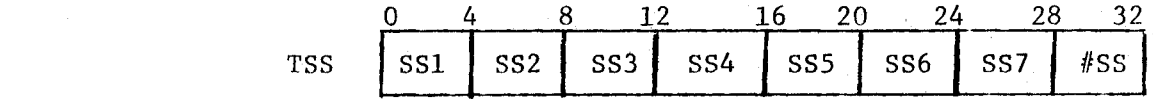
Routine TGENDIME is utilized by case (iii) above.

3. TGENDIME - This routine is called from Type, Dimension and Common processors as well as by TEQVFIX. Using the # of dims from TSUSDET as input, it creates a dimension list, puts the dimensions in it, calculates the array length and shift byte and puts the # dims in B1.

Inputs: R9 - Symbol entry pointer
R4 - # dims

Outputs: R15 - Pointer to dimension list

4. TSUSDET - This routine compiles subscripts or dimensions in Type, Dimension, Common and Equivalence statements. It produces a list of the binary equivalents and loads a register with their number. The list of full words where the SS or dims are placed is labelled TSS.



Checking is done here for variable dimensions, to see that they are simple integer *4 variables and are explicit or implicit parameters. In these cases the dimension is set to 1

Inputs: R14 - Return address
R1 - Stack pointer to first entry of ss or dim

Outputs: R4 - #ss or #DIMS
R1 - Stack pointer to last entry of ss or dim. i.e. to entry containing ')' as operator
TSS - The list of subscripts or dimensions

5. TDIME2 - This routine is used only by the equivalence processor. It is much the same as TGENDIME except that the #dims field in B1 is left at zeros. They are, however, put into ZEOVBYT2. The values put into the dimension list are subscripts. No array length is calculated.

4.4 ARITH

4.4.1. General Description (See Figure 4.4.1.)

ARITH is the routine which compiles all assignment statements. It also compiles CALL statements, parts of the logical IF, arithmetic IF statements, and elements in I/O and data lists (which are allowed to be expressions in WATFOR). It recognizes arithmetic and logical statement functions, and compiles the expression on the right-hand side of the equal sign.

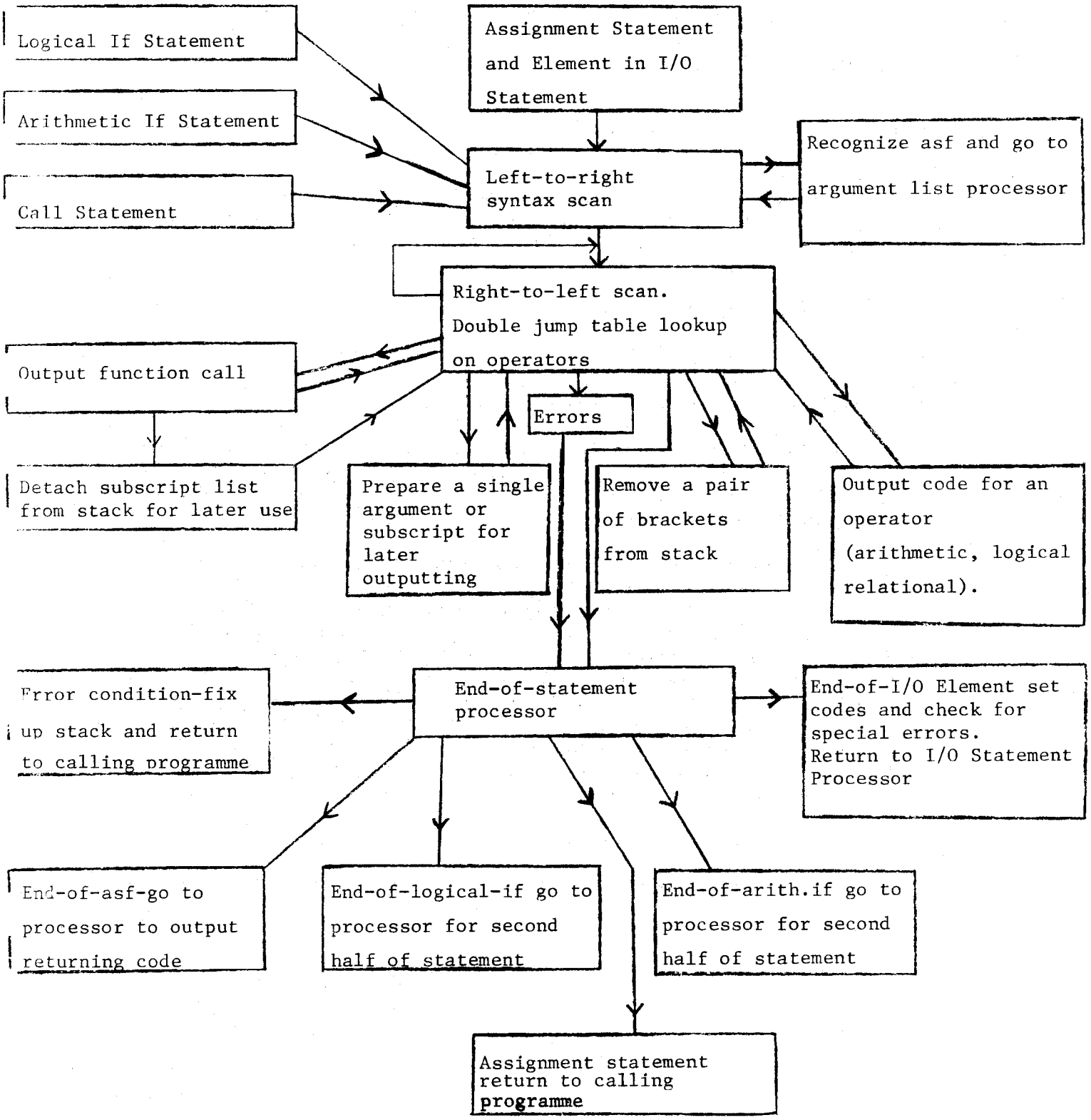
There are entry points in ARITH for logical IF, arithmetic IF, and CALL statements as well as the main entry point used for assignment statements and elements in I/O and data lists. Each entry point does some initialization and switch setting.

All entry points converge to a left-to-right syntax-checking scan which has as its input the stack formed by SCAN. The purpose and operation of this routine is described in section 4.4.3.

After completion of the syntax-checking scan, the code-generating routine is entered unless a serious error has been detected. The code-generating routine is a right-to-left scan of the stack formed by SCAN as modified by the syntax-checking routine. Its purpose and operation are described in section 4.4.4.

When the statement (or expression) has been compiled or when a serious error has been detected, control is passed to an end-of-statement routine. This routine determines which type of statement is being compiled and goes to one of several termination routines. These do special checking and/or cleanup depending on which type of statement is being compiled (See section 4.4.17.)

General Flow Diagram For ARITH



4.4.2. Object Code Generated

Since the design of the object code to be generated for arithmetic dictates the design of the compiler routines, the object code to be generated will be discussed before the routines in ARITH.

Execution-Time Routines

In order to reduce the size of the object code, and to do some execution-time error checking WATFOR uses out-of-line routines located in STARTA to perform many functions of arithmetic. Examples are: subscript calculation (and range checking), fix, float, exponentiation, complex multiply and divide. In some cases (e.g. complex multiply) execution-time is faster because of this practice, since the routines in STARTA are addressable and it is not necessary to do a formal call to get to them.

Register Usage (See Figure 4.4.2.)

At execution-time, R5-R11 are used as base registers for the programme and its data area, R12 and R13 are used to address STARTA and the programme's save area, respectively.

Therefore R14 and R15 are available for linking between the object code and the execution-time routines. (See section 3.15 etc.) R0-R4 and F0-F6 are available to the object code and execution-time routines for use as accumulators and work registers.

In order to be able to pass arguments to the execution-time routines in registers (rather than by use of argument lists), it was decided to use only one register as an accumulator for each mode of arithmetic. The accumulators are: logical-R1, integer-R1, real-F0, complex-(F0,F2) (See Figure 4.4.2). The object code generated for any operator will leave the result in the accumulator associated with the mode of the result. (e.g. The result of I+J would be in R1, the result of X+I would be in F0). R3 is the result register from the subscripting routines (See 4.4.14). It is also used to contain the offset address of equivalenced or commoned simple variables, since they are not addressable using R5-R10.

e.g. of arithmetic code:

Fortran Statement: Y = J * X + (I-1) where I is commoned.

Object Code Generated:

```
L     R3,p(I)          Pointer to I
L     R1,START(R3)
S     R1,p(1)
L     R3,p(J)
BAL   R14,XFLOAT30
ME    F0,p(X)
BAL   R14,XLOAT14
AER   F0,F4
STE   F0,p(Y)
```

For a description of the object code issued for subprogramme calls, see section 4.4.10.

Some Restrictions On The Use Of Registers:

1. F6 is used to convert single-precision numbers to double precision. It must always be left with zeros in the low-order half.
2. F4 is used by the subscript routine (as well as R2 and R4). Its contents may be destroyed during subscript calculations.

Figure 4.4.2 Use of Registers in Object Code Arithmetic

	R0	R1	R2	R3	R4	R14	R15	F0	F2	F4	F6	
Integer +, -, *		result										
/	result		work									
* *	work	result input	work	work	input	return						*
Float Routines		input		input		return		result		result		*
Fix Routines		result				return		input			input | 0	*
Real *4 +, -, *								result		work		
/								result	work			
Doubling											input | 0	
Real *8 +, -								result		work		
/								result	work			
**I					input	return		result input				*
**Real 8						return		input result	input			*
Complex *8 +, -								result	result			
*, /						return		input result	input result	input	input | 0	*
Complex *16, +, -						return		result	result			
*, /						return		input result	input result	input	input | 0	*
**I				input		return		input result	input result			*
Logical Operations	input	input result										
Subscripting Routine			work	result	work	return	dim list			work		**
Com. & Equiv. Vbls.				offset address								
Function Calls	Log. or int.result	arg.list		offset address of rtn.		return		Real or Complex Result			| 0	***

* See section ** See section *** See section 4.4.10

4.4.3. Syntax-Checking and Lookup Routine (See Figure 4.4.3.)

The functions of this routine are:

1. Check for as many errors as possible at this stage. (e.g. AS-2, CM-4, CN-6, EQ-A, EO-8, HO-4, SF-1, SF-2, ST-5, ST-8, SV-1, SX-0, SX-1, VA-A, VA-6, VA-8).
2. Set an error/continue switch to prevent further compilation of the statement if a serious error occurs. (Note that the complete statement is always scanned by this routine, even though the error switch may be on).
3. Recognize arithmetic and logical statement functions (ASF's) and transfer to an ASF entry list processor in LINKR. (See 4.2.12.) [The processor in LINKR outputs code for the entry sequence and returns to ARITH. After compiling the expression on the right-hand side of the equals sign, ARITH goes to an ASF return processor in LINKR which outputs the return code. See 4.2.13]
4. Look up names in the symbol table, and replace the names in the stack by B1 and B2 from the symbol table, and the symbol table pointer for the name.
5. Find argument lists and subscript lists. Replace the '(' by a function bracket '['. Recognize the first usage of a function and update the symbol table accordingly.
6. Turn on type and usage established bits in the symbol table for variables and subprogramme names.
7. Use the constant collector to enter constants into the symbol table, and link around them in the stack. Replace constants in the stack by a byte similar to B1 for variables and a symbol table pointer.
8. Convert &nnnnn constants to symbol table pointers and go to a routine in deck DODO to check for invalid transfers.
9. While scanning the stack, change the links so that each entry points to the one preceding it. Thus at the end of the routine, the pointer is at the end of the stack and the links point back to the beginning.
10. Change the code bytes in the stack to new codes:

CODE	OPERAND
A3	a name with no argument list following it
A4	a statement number constant (&nnnnn)
A5	a hollerith constant
A8	a name with an argument list or the name of an indirectly addressed variable (commoned, equivalenced, called by name)
A9	a constant

NOTE: Three further codes are developed later in the compilation:

A2	a quantity in temporary storage
AA	a quantity in R1
AB	a quantity in F0

These codes are used extensively in the code-generating phase of ARITH for table lookups on the types of operands. Some of them are also used in the object code as information to the relocator. (See section 4.4.10.)

On completion of the syntax-checking phase each entry in the stack has the format:

link back	operator	code	B1	B2	p(symtab)
2	1	1	1	1	2

(except for entries with null operands)

An example of the stack produced by this routine appears in Figure 4.4.6.

4.4.4. Code-Generating Phase

After completion of the syntax-checking routine, the code-generating routine is entered if no serious error has been detected.

This is a right-to-left scan of the stack produced by the syntax-checking routine. The order in which operations should be executed is determined by a table lookup on the operators in pairs. The entry in the table (AUPTABO table 4.4.4.) is found by

1. Calculating a function of the operator indicated by the stack pointer.
2. Moving the pointer up one entry in the stack.
3. Calculating a function of the new operator.

The entry thus found in the table indicates the routine which should be executed when these two operators occur in sequence in the stack (See Figure 4.4.4.).

The different routines referred to in the table are:

ARIT: Output code for an arithmetic, logical or relational operator or equals sign (AOUTARIT).
CALL: Output code for a function call or detach subscript list from stack (AOUTCALL).
PREP: Prepare an argument or subscript in a list for outputting later (APCALL).
REMOVE: Remove a pair of parentheses from the stack (AREMBR).

HOLD: Return to the table-lookup routine to examine the next pair of operators (AHOLD).
Error conditions, (e.g. LG-2, PC-0, SX-6, EQ-6, MD-2).
END: End of statement or expression (AEND).

Flow Diagram of Syntax-Checking and Lookup Phase of ARITH

Figure 4.4.3

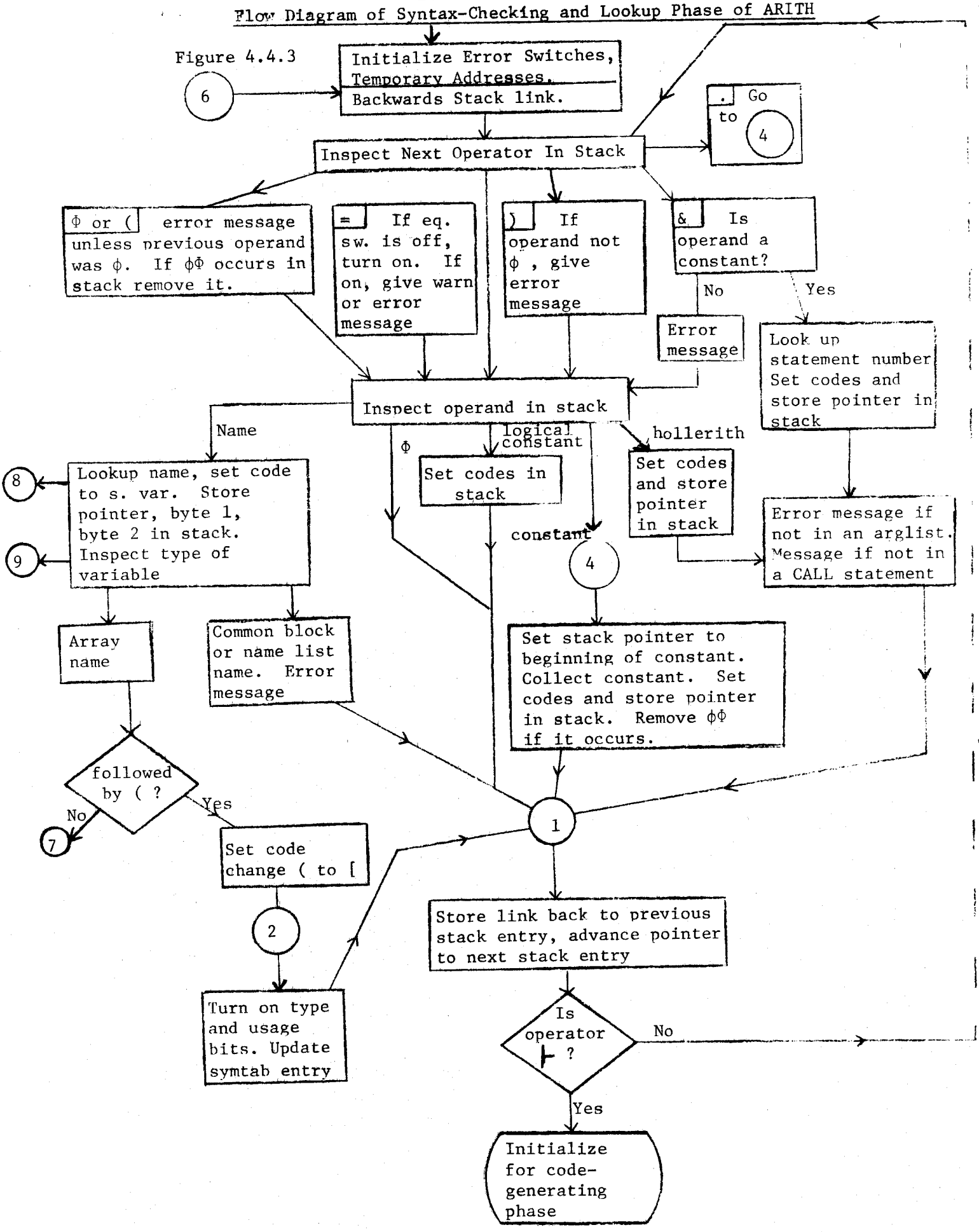
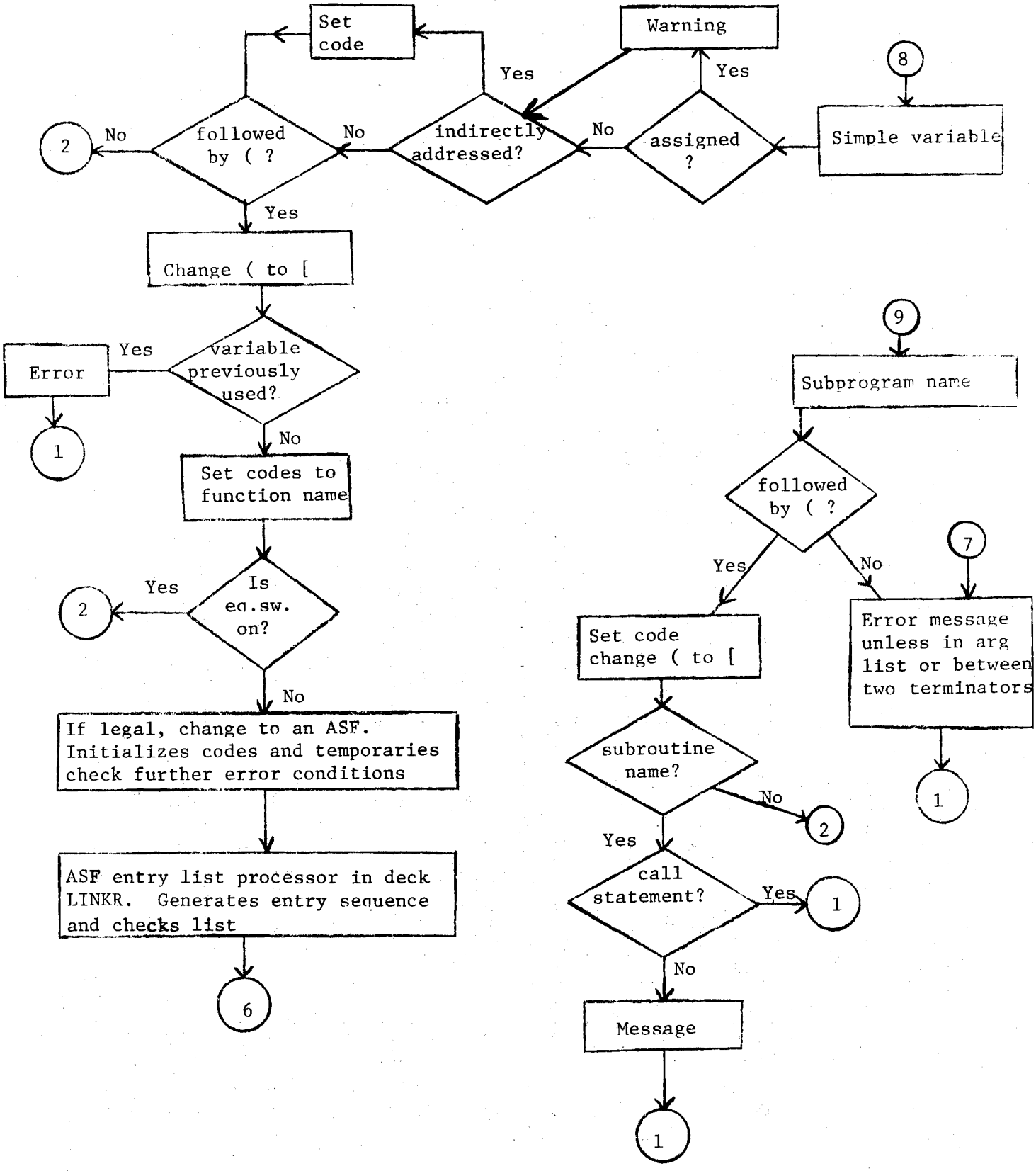


Figure 4.4.3 (cont'd)



Fi e 4.4.4 Table of Rout es for Operator Pairs

op1 \ op2	Φ	⊣⊢	[	(	,	=	.OR.	.AND.	.NOT.	.EO. etc.	+	-	*	/	**	)
Φ	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D	SX-D
⊣⊢	SX-D	END	CALL	REMOVE	SX-C	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	PC-0
[	SX-D	PC-0	CALL	REMOVE	PREP	EQ-6	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	PREP
(	SX-D	PC-0	CALL	REMOVE	SX-C	EQ-6	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	HOLD
,	SX-D	SX-C	CALL	REMOVE	PREP	EQ-6	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	PREP
=	SX-D	HOLD	CALL	REMOVE	SX-C	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	PC-0
.OR.	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	HOLD
.AND.	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	HOLD
.NOT.	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	HOLD	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	ARIT	HOLD
.EQ.etc	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	HOLD	LG-2	MD-2	ARIT	ARIT	ARIT	ARIT	ARIT	HOLD
+	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	HOLD	LG-2	HOLD	HOLD	HOLD	ARIT	ARIT	ARIT	HOLD
-	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	HOLD	LG-2	HOLD	HOLD	HOLD	ARIT	ARIT	ARIT	HOLD
*	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	HOLD	LG-2	HOLD	HOLD	HOLD	HOLD	HOLD	ARIT	HOLD
/	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	HOLD	LG-2	HOLD	HOLD	HOLD	HOLD	HOLD	ARIT	HOLD
**	SX-D	HOLD	CALL	REMOVE	HOLD	EQ-6	HOLD	HOLD	LG-2	HOLD	HOLD	HOLD	HOLD	HOLD	ARIT	HOLD
)	SX-D	HOLD	SX-D	SX-D	HOLD	HOLD	HOLD	HOLD	LG-2	HOLD	HOLD	HOLD	HOLD	HOLD	HOLD	HOLD

ARIT - go to AOUTARIT to output code for an arithmetic logical, relational operator or equals sign

CALL - go to AOUTCALL to output code for a function call, or to 'detach' subscript list

PREP - go to APCALL to prepare an argument or subscript in a list

REMOVE - go to AREMBR to remove a pair of brackets in the stack

END - go to AEND - the end-of-expression routine

HOLD - go to AHOLD - lookup the next pair of operators

SX-D etc. - error conditions

4.4.5. The Routine To Output Code For An Operator (See Figure 4.4.5.)

This routine sets up a table for the operands of the operator concerned (see below). It also finds the proper entry in the code table for the operator (see below) and calculates the mode of the result. Temporary storage is obtained for the register to contain the result, if this is necessary. Control is then passed to the piece of dummy object code indicated by the entry in the code table.

The dummy object code links to an outputting routine which uses the operand tables and dummy object code to produce instructions, which it outputs.

After the object code for the operation has been produced, the stack is updated to remove the entry containing the operator and operand 2. The entry containing operand 1 is updated to indicate that the new operand is a quantity in a register.

Control is then passed back to the operator-pair table lookup routine.

A. Contents of The Operand Tables

ASAND1 ASAND2	1	2	3		4	5	6
			AIN 1 AIN 2			APOIN1 APOIN22	APOIN12 APOIN22

1. The numeric portion of the code of the operand (X'00', X'A2' ... X'AB') taken from the stack entry in which the operand occurs.
2. The mode of the operand (X'00', ... X'07'), taken from byte B1 in the stack entry, plus X'80' if the operand is a quantity in a register.
3. The index register to be used in the instructions pertaining to the operand [R3 if the operand is subscripted or indirectly addressed, otherwise R0].
4. The offset address of the stack entry in which the operand occurs.
5. and 6. The primary and secondary pointers to be used to fill in the base displacement address portion of the instructions referring to the operand. The values of these will be as follows:

TYPE OF OPERAND	APOIN1 and APOIN22	APOIN12 and APOIN22
temporary	the pointer in the stack entry	APOIN1 + (the element length of the operand)
simple variable or constant	the pointer in the stack entry	APOIN1 + 2
indirectly addressed or subscripted variable	X'C000'	APOIN1 + (the element length of the operand)
quantity in a register	not used	not used

Routine To Output Code For An Operator

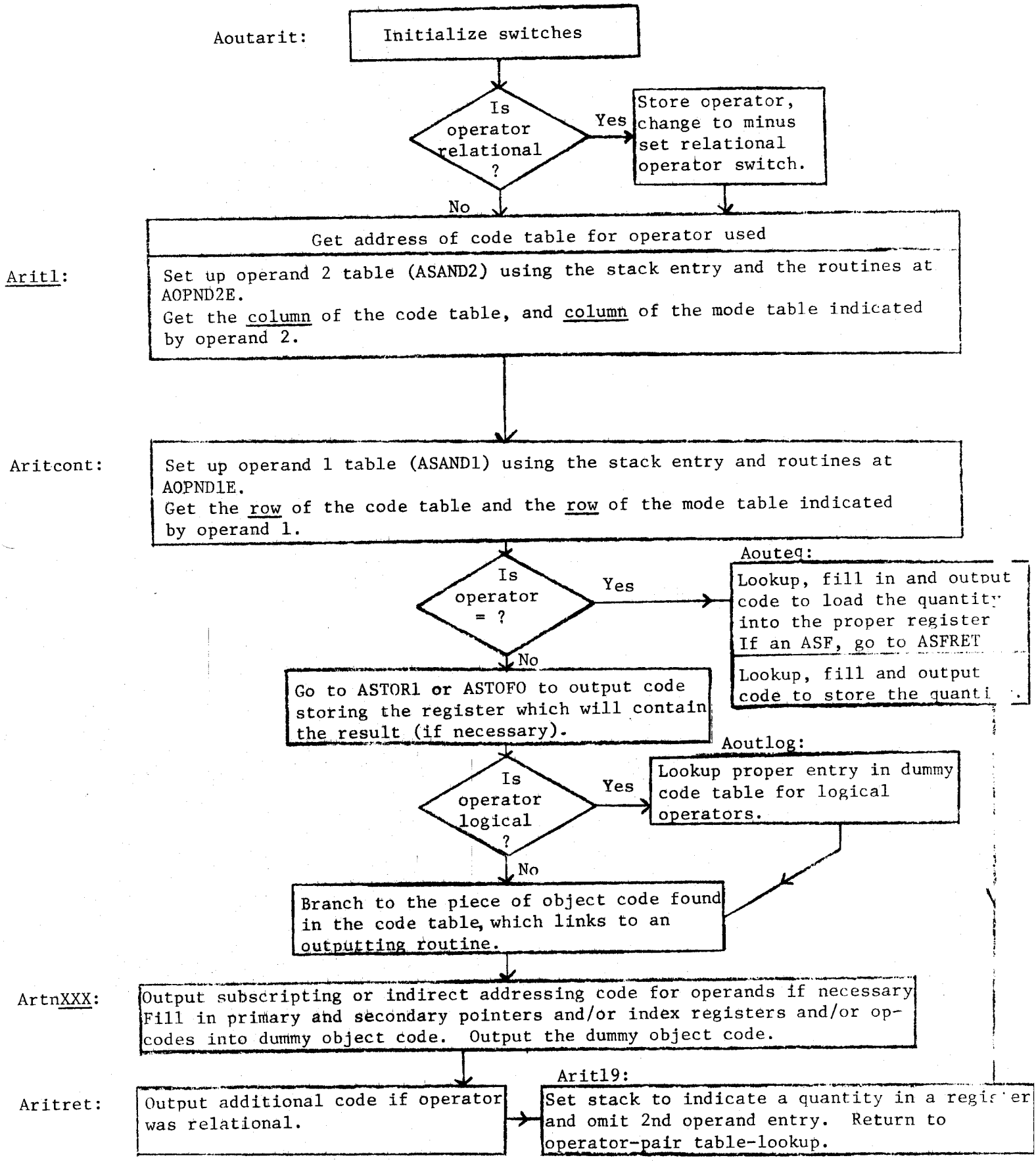


Figure 4.4.5

B. Contents of the Mode Table

The mode table (ARITAB6) is a table which gives the mode of the result as a function of the modes of the two operands of a binary operation. If the two modes are not allowed to occur together (e.g. integer with logical), the result mode is given as X'FF'.

C. Contents of the Code Tables For the Operators

There is a code table in ARITH for each of the operators +, *, /, ** [AADDTAB, AMPYTAB, ADIVTAB, AEXPTAB]. The tables have rows and columns corresponding to the following types of operands:

- null ( )
- quantity in R1
- integer *4 in storage
- integer *2 in storage
- real *4 in F0
- real *8 in F0
- real *4 in storage
- real *8 in storage
- complex *8 in F0, F2
- complex *16 in F0, F2
- complex *8 in storage
- complex *16 in storage

The entry for two given types of operands is an address indicating the piece of object code to be filled in and outputted when the operands occur with the given operator.

The operator '-' uses the same table as '+'. The op code is always filled into the object code for + and -, and extra instructions are outputted to change the sign of the result when necessary for -.

For commutative operators, the same dummy object code is used for AθB as for BθA. A routine (ASWAP) which interchanges the operand tables and looks up the address of the code for BθA is used for this purpose.

4.4.6. The Pieces of Dummy Object Code (ARICDnnn)

The pieces of dummy object code contain the 'skeleton' of the object code to be generated for an arithmetic operation. Each piece of code branches to a routine (ARTNnnn) using R1 to point to the 'skeleton'.

In some cases the same piece of dummy code is used for several operations. For example (real *4 in storage) + (real *4 in storage) uses the same dummy code as -, * and / with the same operands.

The code is:

```
LE      F0,*-*
AE      F0,*-*
LCER    F0,F0 (conditional)
```

The addresses *-* are filled in by the outputting routine, and the numeric part of the opcode AE is changed to suit the operator.

Some outputting routines are also used by several pieces of dummy code. For example the outputting routine used in the example above is also used for (integer *4) ± (integer *4) for which the skeleton code is:

```
L       R1,*-*
A       R1,*-*
LCR     R1,R1 (conditional)
```

This code is different from the code above, but has the same form. (i.e. it requires *-* addresses to be filled in at 2 bytes and 6 bytes from the beginning, and an op-code at 4 bytes from the beginning).

4.4.7. Outputting Routines

The outputting routines have as their inputs the operand tables (see section 4.4.5.A), and the dummy object code indicated by R1. They construct and output all the object code associated with an operation, inserting subscripting code and/or undefined-variable-checking code where required.

In general, the outputting routine goes to routine ANDSS1 which outputs the associated subscripting, indirect addressing and/or undefined-variable-checking code for operand 1. It then fills in the op codes, indexes and pointers required for the portion of the object code associated with operand 1, and outputs it. Routine ANDSS2 is used to output subscripting code, etc. for operand 2. The remainder of the dummy object code is then filled in and outputted. In the case of an operand in a register, ANDSS1(2) is not used, and therefore the object code for the whole operation can be output at once.

To generate the instructions for filling the object code, the macros AFILL and AFILC are used.

$$\text{AFILC } N_{op1},\ N_{in1},\ N_{p1},\ N_{op2},\ N_{in2},\ N_{p2}$$

generates instructions to:

1. Move the numeric part of the operator to $N_{op1}(R1)$ and $N_{op2}(R1)$.
2. Move the numeric part of the index for the operand concerned to $N_{IN1}(R1)$ and $N_{IN2}(R1)$.
3. Move the two characters of the primary pointer for the operand to $N_{p1}(R1)$
4. Move the two characters of the secondary pointer to $N_{p2}(R1)$. Any of the operands may be omitted.

The macro AFILL is the same, but has only 3 operands.

The macros AOUT1, etc. are used to generate the instructions which cause sections of object code to be output. For example AOUT1 N causes the N bytes of object code indicated by R1 to be output, and R1 to be increased by N. It then falls through to the next instruction.

AOUT2 N causes the N bytes or N+2 bytes of object code at R1 to be output depending upon whether the sign of the result should be changed. Control is then passed to ARITRET.

Example of an outputting routine:

The routine which outputs the object code for (real *4) ± (real *4) and (integer *4) ± (integer *4) given above is:

```
ARTN008   BAL     R14,ANDSS1     output ss code for opnd1
          AFILL   ,1,2           fill in index and pointer
          AOUT1   4              output 4 bytes and advance R1
          BAL     R14,ANDSS2     output ss code for opnd2
          AFILL   0,1,2          fill in opcode, index, pointer
          AOUT2   4              output 4 or 6 bytes and return
                                   to ARITRET
```

The reason that the object code is output in two sections is that the subscript or indirect addressing code leaves the offset address of the variable in R3, and so the addresses of two subscripted variables can not be available at the same time unless extra code is generated. Also, the code generated for undefined variable checking must immediately precede the instruction which uses the variable.

4.4.8. Routines to Output Subscript, Indirect Addressing and/or Undefined Variable Checking Code for Operands in ARITH

Routines ANDSS1 and ANDSS2 set R15 to indicate the operand table for operand 1 or 2, then converge to a single routine.

If the operand concerned is a constant or temporary, no object code need be generated. If it is a subscripted variable, control is passed to AOUTSS to generate the subscript list (See 4.4.14). If it is an indirectly addressed simple variable (commoned, equivalenced, called by name) an instruction 'L R3,P(V)' is generated, where P(V) is the symbol table pointer from the stack entry containing the variable.

Undefined Variable Checking:

If switches AUNDEFSW and CUNDEFSW are both on, an instruction to check for an undefined variable is generated. This instruction is:

```
BAL     R14, XROUTxn
```

where x is A for subscripted variable
E for indirectly addressed variables
S for simple variables
and n is 1, 2, 4, 8 or 16 depending on the mode of the variable. (See section 3.19. for a description of the XROUTxn'S).

The switch CUNDEFSW is set by MAIN depending on whether RUN = CHECK, NOCHECK or FREE.

The switch AUNDEFSW is on unless ANDSS1NC was called. ANDSS1NC turns off AUNDEFSW, then falls through to ANDSS1. This routine is used for quantities on the left-hand side of the equals sign, where it is necessary to generate subscript code, etc. but no undefined variable checking.

4.4.9. Routine To Prepare An Argument Or Subscript In A List

This routine is called from the operator pair table lookup when the operator sequences [, ; [ ) ; , , ; ,) occur.

A table lookup on the code of the operand in the stack is done, and the following actions are taken:

1. Temporary, statement number constant, hollerith constant (codes A2, A4, A5, A9) - no action.
2. (Code 00) - error ST-5
3. Simple name (code A3) - function or array - no action
   - simple variable, if the variable is a do-parameter or assigned variable, a bit in B1 in the stack is set on.
4. Quantity in a register (codes AA,AB) - control is passed to routine ASGETEMP which assigns temporary storage for the quantity, outputs the instructions to store it, and updates the stack entry to indicate a temporary (See 4.4.13.).
5. Subscripted variable or indirectly addressed variable (code A8) - the subscripting code or indirect addressing code is generated using AOUTSS if subscripted (See 4.4.14.).

If the operand is an indirectly addressed simple variable which is a do-parameter or assigned variable, the bit mentioned in (3) is set.

Instructions:

```
IC      R2,*-*
ST      R3,*-*
STC     R2,*-*
```

are then generated for both subscripted and indirectly addressed variables. These instructions will store the address of the variable in the argument or subscript list after it is calculated. The addresses *-* are presently unknown, but will be filled in when the list is output.

The offset address of the first *-* is stored in the stack, destroying B2 and the symbol table pointer. B1 in the stack is changed to look like a simple variable of the same mode.

4.4.10. Routine To Output A Function Call

The operator-pair table lookup branches to AOUTCALL for all valid operator-pairs with '[' as the second operator. If the operand before the '[' is an array name, control is passed to the routine ADETSS (See 4.4.12.). Otherwise, the operand must be a subprogramme name, and the code for a call is generated.

Code is generated to store R1 and F0 in temporary storage if in use. An instruction is generated if necessary to pad the object code to a full word boundary.

The instructions:

```
L       R3,P(F)
LA      R14,*-*
BAL     R1,START(R3)
```

are outputted. P(F) is the pointer to the function name and the return address *-* is not yet known. The argument list is now processed as follows:

1. If the code of the argument is A8, the base-displacement address of the argument in the object code is filled in to the three *-* addresses mentioned in 4.4.9.
2. If the code of the argument is A2, the temporary table (see 4.4.13.) is updated to indicate that the storage is now free to be used again.
3. For all arguments - 4 bytes are output, containing the code, B1 and the pointer from the stack. (The top bit of B1 is turned off) (See Figure 4.4.7.).

The code is used as an instruction to the relocator. B1 appears in the final argument list along with an address constructed from the pointer. (See 4.8.)

An extra 'last argument' is now generated containing the code X'B2'and the pointer to the function name. The base displacement address of the next instruction in the object code is now filled into the 'LA R14,*-*' instruction described above.

If the subprogramme called was a function, the code is changed to indicate a quantity in a register, and the register table (see 4.4.11.) is updated. If the function was integer or logical, the instruction 'LR R1,R0' is generated to move the result from the return register R0 to the accumulator R1.

Control is then returned to the operator-pair table lookup.

4.4.11. Contents of the Register Table

The register table consists of two full words ASGPR1 and ASFPR0. If bit 31 of ASGPR1 is 1, R1 is not currently in use. Otherwise ASGPR1 contains the offset address of a stack entry in which the operand is a quantity in R1. ASFPR0 operates similarly for F0.

4.4.12. Routine to 'Detach' Subscript List From Stack (ADETSS)

This routine counts the subscripts of an operand, and gives a message if the number is not correct. It also stores the link which points from the stack entry containing the array name, to the entry containing the first subscript into byte B2 of the entry with the array name (See Figure following). The link of this entry is then changed to point to the entry following the subscript list. Control is returned to the table lookup.

e.g. X = A(I,J)

Stack at beginning of ADETSS

0	⊣	A3	$B1_X$	$B2_X$	P(X)
8	=	A8	$B1_A$	$B2_A$	P(A)
8	[	A3	$B1_I$	$B2_I$	P(I)
8	,	A3	$B1_J$	$B2_J$	P(J)
4	)	00			
0	⊢	00			

Stack after ADETSS

0	⊣	A3	$B1_X$	$B2_X$	P(X)
28	=	A8	$B1_A$	8	P(A)
8	[	A3	$B1_I$	$B2_I$	P(I)
8	,	A3	$B1_J$	$B2_J$	P(J)
4	)	00			
0	⊢	00			

4.4.13. Routine To Assign Temporary Storage (ASGETEMP)

This routine assigns temporary storage for an operand in a register. Its inputs are a stack entry pointed to by R15, and the temporary table.

The temporary table consists of a double word ASNXTEMP. The first word of ASNXTMP contains a number of the form X'ooooDnnn' and the second word has X'oooоEmmm'. These two numbers represent the next available integer (or logical) temporary and the next available floating-point temporary.

The pointers 'Dnnn' and Emmm' are used in the address portion of instructions in the object code and are changed by the relocator into base displacement addresses (See 4.8.3.).

At the beginning of each programme segment, CANXTMP in COMMR is initialized to the numbers corresponding to the first temporary locations.

Upon entering ARITH to compile an expression, ASNXTMP is initialized to the numbers corresponding to the first temporary locations.

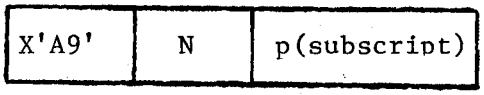
When temporary storage is required, the number in the appropriate half of ASNXTMP is aligned to the next boundary at which the value can be stored. Code is then generated to store the register(s) using the two bytes from the temporary table as address pointers in the instructions generated. The two bytes are also put in the pointer portion of the stack entry concerned and the code in the stack is changed to X'A2' to indicate a quantity in temporary storage. The register table is updated to indicate that the register stored is no longer in use.

The number of bytes used by the quantity stored (1, 2, 4, 8, 16) is added to the number in the temporary table to give the next available temporary location.

If the number in the temporary table is greater than the number in the corresponding position of CANXTMP, CANXTMP is updated to contain the new value. Thus, at the end of compilation of a programme segment, CANXTMP contains the pointers corresponding to the amount of temporary storage needed. The relocator then uses CANXTMP to assign the temporary storage area.

4.4.14. Routine To Output Subscript Coding (AOUTSS)

This routine begins by padding the object code to a full word if necessary. An instruction BAL R14,P(array name)' is output. A word for each subscript follows, containing:



where N is a code telling if the subscript is integer *4, integer *2, real *4, real *8. Logical subscripts are not allowed. Complex subscripts are also not legal in Fortran IV but are allowed in WATFOR. A warning is given for complex subscripts, and the real portion is used. Error messages are given for subscripts which are hollerith constants, statement number constants or subprogramme names. A warning is given for subscripts occurring on the right hand side of an ASF, and for non-constant subscripts occurring in a data list. For subscripts which are subscripted or indirectly addressed variables, the *-* addresses in the IC, ST, STC instructions generated earlier (See section 4.4.9.) are filled in with the base-displacement address of the subscript in the list. For subscripts which are quantities in temporary storage the temporary table is updated to make the storage available to be used again.

A description of what happens at execution time for subscripting is given in section 3.14.

4.4.15. The End of Expression Routine (AEND)

The end of expression routine is called from the operator-pair table lookup when the operator sequence ⊣ ⊢ occurs.

If the statement being compiled is a logical IF, arithmetic IF, DATA, READ or WRITE statement, control is passed to AIFLRET, AIFARET, or ARTTIO. If ACALLSW is on (see 4.4.17.A), control is passed to the routine AOUTCALL to output a CALL to a subroutine with no arguments. Otherwise a return to SCAN is executed.

4.4.16. Error End-of-Statement Routine

If a serious error occurs, the compilation of the statement is terminated, and an error end-of-expression routine is executed. This routine fixes up the stack so that the stack pointer is positioned at an entry containing a ⊣ and linking to another entry containing a ⊢. The code of the first entry is 00.

Control is then passed to the appropriate other routine, depending on the statement type.

4.4.17. Entry Points and Termination Routines For Various Types of Statements

There is one byte switch in COMMR called CADSSW. It contains:

EQ	CALL	DATA	ASF	LIF	AIF	IN	OUT

where:

EQ = 0 if an equals sign is expected
CALL = 1 if a call statement is being compiled
DATA = 1 if a data statement is being compiled
ASF = 1 if an ASF statement is being compiled
LIF = 1 if a logical if statement is being compiled
AIF = 1 if an arithmetic if statement is being compiled
IN = 1 if a read or data statement is being compiled
OUT = 1 if a write or print statement is being compiled

These switches are set by the programme calling ARITH or in the entry point used, and are tested by the end-of-expression routine to decide which termination routine to use.

A. CALL Statement

Entry Point

The entry point ACALL is used to compile a CALL statement. A special switch ACALLSW is set on if there are no arguments. This switch is tested by the end-of-expression routine.

The EQ and CALL switches in CADSSW are set on, and the stack pointer is advanced to remove the word CALL.

CSRT1 is used to find the first level-0 right bracket in the stack. The stack is checked to make sure it was the last character in the statement.

The name used in the CALL is looked up in the symbol table and changed to a subroutine if it has not been used previously in another context. Control is then passed to ARITH1 to compile the statement.

Termination Routine:

ACALLSW is tested by the end-of-expression routine, but there is no termination routine for CALL statements with an argument list.

B. Input, Output and Data Statements

Entry Point

The routine INOUT delimits an expression to be compiled by ARITH with ⊢'s in the stack. The stack pointer is set at the first ⊢. The DATA, IN and/or OUT portions of CADSSW are set, and a call is made to the regular entry point (ARITH) of ARITH. (i.e. ARITH has no special entry point for I/O expressions)

Termination Routine:

The end-of-expression tests the IN and OUT portions of CADSSW and if either is on, goes to the routine ARTTIO.

Routine ARTTIO checks for error conditions DA-3, DO-9, AS-5, IO-8, and IO-C.

For elements which are subscripted, AOUTSS is used to generate subscripting code. For indirectly addressed variables, the instruction 'L R3, P(V)' is output. The presence of an array name with no subscripts is detected, and a special value X'AC' is put in the code to indicate this condition.

C. ASF's

Entry Point

There is no entry point in ARITH for ASF's, since they are identical in syntax to assignment statements. See section 4.4.3 for detection of ASF's and handling of the entry list.

Termination Routine:

The routine which outputs code for an equal sign detects the presence of an ASF after it has output the object code which places the result in the proper accumulator. Control is then passed to routine ASFRET.

ASFRET tests for the error SF-3, and stores in the symbol table the maximum temporary values needed for the ASF. If the ASF is logical or integer the instruction 'LR R0,R1' is output to move the result from the accumulator to the function result register.

Control is passed to LASFR in deck LINKR to output return code, and is then returned to SCAN.

D. Logical If Statements

Entry Point (IFLOG)

This entry point has its own save area, since it calls a routine (which may be another entry point of ARITH) to compile the second half of the statement.

The routine adjusts the stack to remove 'IF' and changes the opening (to a ¬. It then sets R13 at ARITH's save area, turns on the EQ and LIF portions of CADSSW and goes to ARITH1.

NOTE: SCAN changes the first level zero ')' to a ¬ before calling IFLOG.

Termination Routine (AIFLRET):

AIFLRET gives error message IF-3 for result expressions which are not logical, or for hollerith or statement number constants. For expressions which are subscripted or indirectly addressed variables, the appropriate code is generated. Instructions are output to put the logical result in R1 if necessary.

```
The instructions:   N     R1,XTRUESP
                    BZ    *-*
```

are output. These instructions branch around the second half of the statement if the result is false. The EQ and LIF portions of CADSSW

are turned off, the address of the second level-0 right bracket is put in CSRT1 (See 4.4.17.A) and the processor is called to compile the second half of the statement. After returning from this processor, the *-* address above is filled in and control is returned to SCAN.

E. Arithmetic If Statements

Entry Point (GIFIF)

This routine adjusts the stack to remove 'IF' and changes the opening '(' to a ⊣. It then sets on the EQ and AIF bits in CADSSW and goes to ARITH1.

NOTE: SCAN changes the first level zero ')' to a ⊣ before calling GIFIF.

Termination Routine (AIFARET):

AIFARET gives error message IF-4 if the expression compiled was logical or complex or was a hollerith or statement number constant. If the expression was a subscripted or indirectly addressed variable, the associated code is output. Code is generated to put the result in the proper accumulator (if necessary) and to set the condition code according to its value. Control is passed to GIF in deck DODO to compile the branching portion of the statement, then returned to SCAN.

Figure 4.4.6. Example of Compilation of a Complete Statement

(1) Input Statement: **B** = A(X + I) + Y*SIN(Z) + 3.0
(A is an array)

(2) Stack Generated By Scan

→	⊣	01	B b b b
	=	01	A b b b
	(	01	X b b b
	+	01	I b b b
	)	ϕ	
	+	01	Y b b b
	*	01	S I N b
	(	01	Z b b b
	)	ϕ	
	+	81	(3)
	.	81	(0)
	⊢	ϕ	

(3) Stack after Syntax Scan

	⊣	A3	$B1_B$ $B2_B$P(B)
	=	A8	$B1_A$ $B2_A$P(A)
	[	A3	$B1_X$ $B2_X$P(X)
	+	A3	$B1_I$ $B2_I$P(I)
	)	ϕ	
	+	A3	$B1_Y$ $B2_Y$P(Y)
	*	A8	$B1_{SIN}$ $B2_{SIN}$P(SIN)
	[	N	$B1_Z$ $B2_Z$P(Z)
	)	ϕ	
	+	A9	$B1_{3.0}$ P(3.0)
→	⊢	ϕ	

(4) Routines Used in Code-Generating Scan

```
                    1   2          3
B = A(X+I) + Y*SIN(Z) + 3.0
```

	Operator Pair		Routine Indicated by Double Jump Table
1	+	⊢	return to table look up (HOLD)
2	)	$+^3$	HOLD
3	[	)	prepare an argument or subscript for later output
4	*	[	output a function call or "detach" subscript list (in this case - output code to calculate sin(Z).)
5	*	$+^2$	HOLD
6	$+^2$	*	output code for * (Y*SIN(Z))
7	$+^2$	$+^1$	hold
8	)	$+^2$	hold
9	+	)	hold
10	[	$+^1$	output code for + (X+I)
11	(	)	prepare an argument or subscript for later output (in this case output code to store X+I at a temporary)
12	=	[	output function, or "detach" subscript list. (In this case "detach" subscripts for A from stack).
13	=	$+^2$	output code for + (A(X+I) + Y*SIN(Z))
14	=	$+^3$	output code for + (above +3.0)
15	=	⊢	hold
16	⊣	=	output code for =
17	⊣	⊢	end-of-expression

Figure 4.4.7.

Object Code Generated For Statement in Figure 4.4.6.

```
                CNOP    0,4
                L       R3,P(SIN)
                LA      R14,*+12
                BAL     R1,START(R3)
4               [A3     B1(Z),P(Z)]          Pointer to Z
6               ME      F0,P(Y)
                STE     F0,P(TEMP1)
                L       R3,P(I)
                BAL     R14,XFLOAT30
10              AE      F0,P(X)
11              STE     F0,P(TEMP2)
                BAL     R14,P(A)
                [A9,code, P(TEMP2)]          Pointer to Z
                LE      F0,START(R3)
13              AE      F0,P(TEMP1)
14              AE      F0,P(3.0)
16              STE     F0,P(B)
```

4.5 DODO

4.5.1. Introduction

The routine DODO processes most of the FORTRAN 'control' statements. This includes the simple, assigned and computed GO TO statements and the ASSIGN statement. DODO also processes all DO loops including the usual DO as well as implied DO's in Input/Output statements or in DATA statements. While processing DO statements DODO also checks for illegal transfers into the range of a DO-loop. Finally, DODO processes the statement number portion of arithmetic IF statements (ARITH compiles the arithmetic expression portion).

4.5.2. COMPILE GOTO

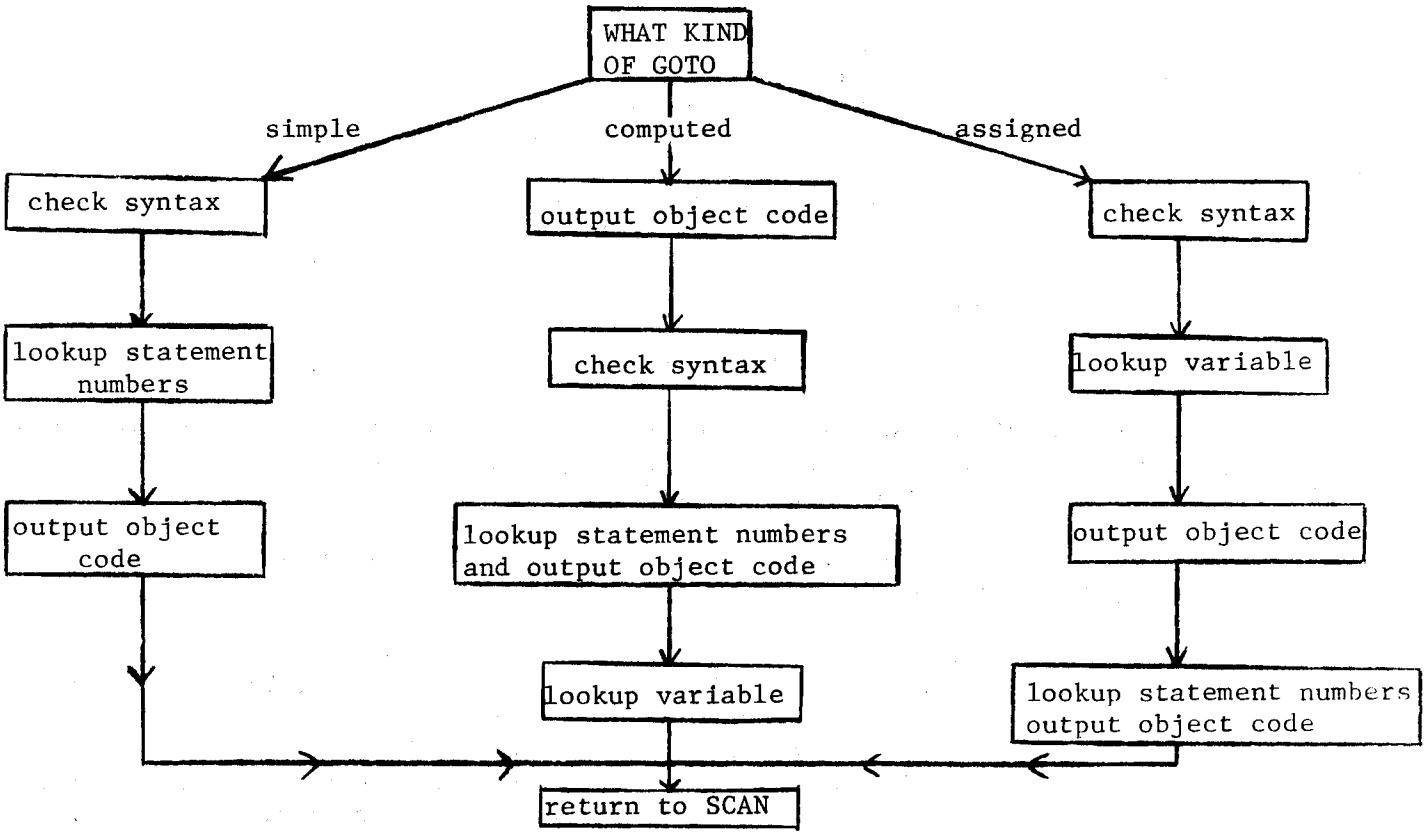


Figure 4.5.1.

To determine the kind of GOTO:

if a number follows "GOTO" is is simple GOTO,
if a variable follows "GOTO" it is assigned GOTO,
if a '(' follows "GOTO" it is a computed GOTO.

1. SIMPLE GOTO

- Lookup statement number and check for illegal branches into DO LOOPS by DCSTN2.
- Output the following object code:

```
        L     R3,ptr to statement numbers in symbol table
        B     START(R3)
```

2. COMPUTED GOTO

Output the following object code:

```
        B     XBOOT
        NOP   START(R3)
        LTR   R3,R3
        BNP   error GO-2
        LA    R4, maximum value of index
        CR    R3,R4
        BH    next FORTRAN statement
        SLA   R3,2 multiply R3 by 4
        EX    0, LABEL(R3)
LABEL   B     START(R3)
        L     R3,ptr to first statement number
        L     R3,ptr to second statement number
        etc.
```

The B XBOOT is necessary in case there is an error in the GOTO statement. We do not want to execute the code (RUN=FREE) which may not have all the necessary information filled in.

If no errors are encountered the following changes are made in the above code:

(i) - the B XBOOT is replaced by a L R3,ptr to index.
(ii) - the NOP START(R3) becomes a L or LH R3,START(R3) if the index is commoned or equivalenced.
(iii) - the value of the index is now in R3 so that we can test if it is positive. If it is negative the error GO-2 is given (index of compiled GOTO negative or undefined) - note that the special number to mark undefined variables is negative.
(iv) - now put maximum value of index into R4.
(v) - compare R3 and R4
(vi) - if R3 is larger than R4, the index is greater than the number of statement numbers present - therefore, we go to next FORTRAN statement.

(vii) - multiply R3 by 4 and execute the proper L instruction (see below), then branch to START(R3).

During compilation, the statement number list is scanned in a loop which looks up each statement number and checks for illegal transfer into DO loops (DCSTN2). Then "L R3,ptr to statement number" are outputted (forming a list of them right after the first block of object code). These loads are executed by the EX as noted above. When the index is looked-up, its address is patched into the object code.

4.5.3. Assigned GOTO

The variable is looked-up and marked as an assigned variable. An assigned variable has as its value the address of the statement number, thus it cannot be a half word integer.

Output the following object code:

```
          L       R3,ptr to assigned variable
          NOP     START(R3)
          LA      R15,LABEL1
LABEL2    EX      0,0(R15)
          CR      R2,R3
          BE      START(R3)
          LA      R15,4(R15)
          B       LABEL2
LABEL1    EQU     *
          L       R2,ptr to 1st statement number
          L       R2,ptr to 2nd statement number
          .
          .
          .
          L       R2,ptr to last statement number
          B       IRGO3
```

The following modification and insertions are made to the object code:

(i) - The NOP START(R3) is changed to L R3,START(R3) if the assigned variable was commoned or equivalenced.
(ii) - R3 has address of statement number.
(iii) - We execute the loads in turn looking for an equal compare of addresses (**i.e.** statement numbers).

During compilation each statement number is looked up and checked for illegal transfers. The object code consisting of

"L R2,ptr to address of statement number" is generated. Thus we have a series of L R2's coming after previous block of coding. Finally a B IRGO3 is outputted. If no equal compare is found this instruction is executed giving a GO-3 error. (IRGO3 is in STARTA)

4.5.4. ARITHMETIC IF STATEMENT

ARITH branches here after setting up the code to generate the condition code. This routine lookups and checks the statement numbers and outputs the following object code.

```
L     R2,ptr to 1st statement number
BL    START(R2)
L     R2,ptr 2nd statement number
BE    START(R2)
L     R2,ptr to 3rd statement number
B     START(R2)
```

4.5.5. ASSIGN STATEMENT

The statement number and variable are looked-up in the symbol table. The variable must be 'assigned' variable. The following object code is outputted.

```
L     R3,address of statement number
ST    R3,variable
```

These instructions cause the address of the statement number to be stored in the assigned variable.

4.5.6. DO STATEMENT

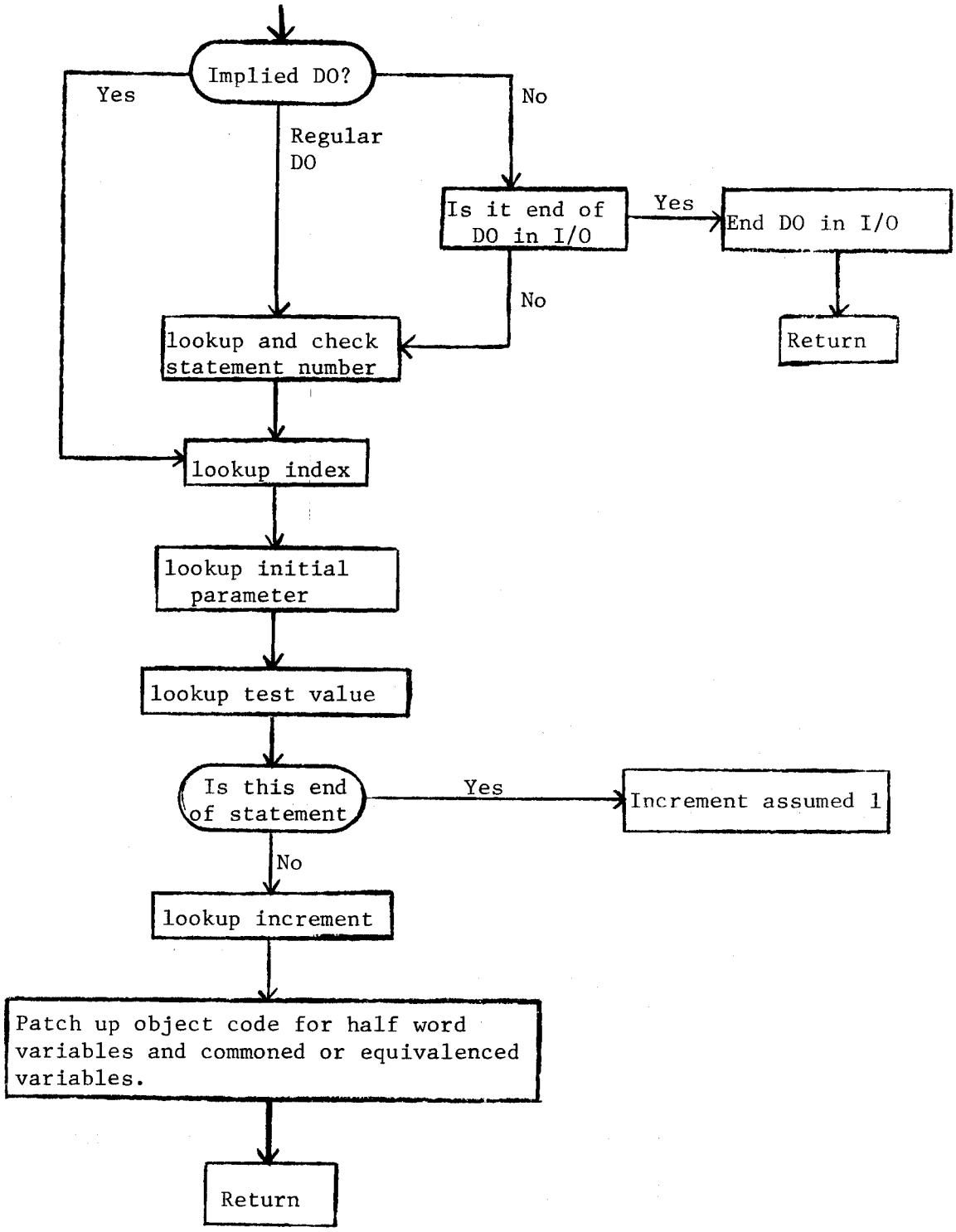


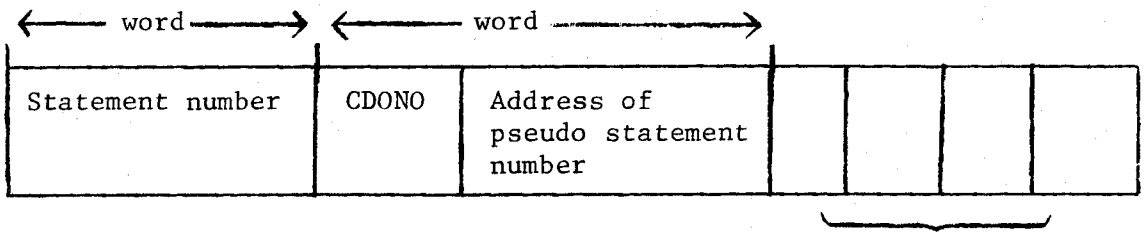
Figure 4.5.2.

There are two aspects of a DO statement.

1. Compiling the statement itself
2. Setting up the DO Loop

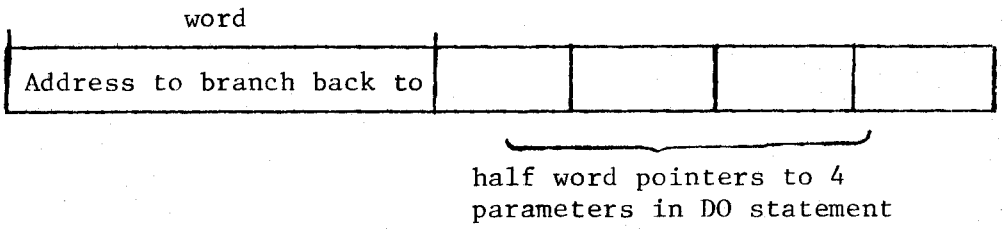
A. DOSTACK

Before describing the DO statement the DO-STACK located in COMMR, will be described. The stack is an area of fixed size. Each entry is of the form



The halfword pointers are used for turning off the bit which marks certain variables as DO parameters. This occurs when ending a DO loop.

For implied DO loops the stack is another area of core and its entries are of the form



The purpose of the DOSTACK is to have a record of all DO loops, which still have to be ended. It is useful when checking for illegal branching into DO loops.

The end of the DOSTACK is denoted by the statement number X'7FFFFFFF'. Zero cannot be used, as 0 is a legal statement number.

B. COMPILING THE DO STATEMENT

The statement number is looked up and checked by DCSTN2 for validity.

Each of the parameters is looked up in turn and the information needed to generate object code (whether full or half word variable or constant, commoned, equivalenced or not) is stored in DOPARM and DOPARMH. Pointers are stored in DOSTACK. If the parameter was a constant it is verified not to be zero.

Now the object code is generated, A few examples give an idea of what must be generated in various cases. The pseudo statement number is the branch back point from the end of the loop.

(i) Object code when all parameters are simple, full-word integer variables:

```
L     R2, increment
LTR   R2,R2
BNP   IRDO7 error return
L     R2, test value
LTR   R2,R2
BNP   IRDO7
L     R2, initial
LTR   R2,R2
BM    IRDO7
ST    R2, index
B     around incrementing code
pseudo statement number here
L     R2, index
A     R2, increment
C     R2, test value
ST    R2, index
BHR   R14 statement after end of DO LOOP
```

(ii) Object code when parameters all constant:

```
L     R2, initial
ST    R2, index
B     around incrementing
pseudo statement number here
L     R2, index
A     R2, increment
C     R2, test value
ST    R2, index
BHR   R14
```

(iii) If all parameters are commoned or equivalenced, the object code is:

```
L     R2,ptr to increment
L     R2,START(R2)
LTR   R2,R2
BNP   IRDO7
L     R2,ptr to test value
L     R2,START(R2)
LTR   R2,R2
BNP   IRDO7
L     R2,ptr to initial
L     R2,START(R2)
LTR   R2,R2
BM    IRDO7
L     R3,ptr to index
ST    R2,START(R3)
B     around incrementing
pseudo statement number
L     R2,ptr to index
L     R2,START(R2)
L     R3,ptr to increment
L     R3,START(R3)
AR    R2,R3
L     R3,ptr to test value
ST    R2,START(R3)
BHR   R14
```

In each case the code generated above the pseudo statement numbers tests if the parameters are positive and initialize the index to its initial value. If not positive control transfers to IRDO7 where the error message DO-7 is issued.

The pseudo statement numbers are created so that it is possible to branch back to the beginning of a DO loop. They are described in more detail in the sections about DO loop construction.

C. DO-loop Construction in Programmes

CDONO is a variable which is increased by 1 with each DO statement. Thus it is a number which is uniquely identified with each DO loop. The pseudo statement number is 100,000 plus the CDONO (larger than one can use in a FORTRAN programme).

Then end of DO loop coding branches to the pseudo number statement code by means of a BAL. It is now possible to exit from the DO loop if necessary.

For implied DO loops, the coding is simpler since it is assumed that they will not have more than 4096 bytes of object code (range of addressability). The address of the point which we branch back to is stored in the I/O DO-stack. At the end of the loop, the displacement is calculated.

COMPILE Implied DO's - I/O LOOPS

As mentioned above implied DO loops in I/O statements are compiled by DODO. The end of I/O DO loop is there.

For DATA statements, all the parameters must be constants since DATA statements are executed before anything else. This is checked in DODO.

D. ERROR CHECKING

The important items in the error checking routine are the following:

1. Each statement number in the symbol table is assigned a level. This level is the current CDONO when the statement number is encountered in columns 7-72. It is changed to the level of the statement when encountered in columns 1-5.
2. CDONO is a variable which increases by 1 with each DO statement encountered. It does not indicate the level of nesting but is a unique number corresponding to each DO loop.
3. DOSTACK. Its purpose is to keep a record of all DO loops which are 'open' at any time - the number of entries in the DO-stack is the depth of nested DO loops. (In the examples it is denoted as 'statement number' - CDONO)

(i) Illegal transfers into DO loops.

There are two cases to consider when checking for illegal transfers into DO loops.

Case 1: A statement number has not appeared in columns 1-5 e.g. GOTO5. The course of the action is:

1. Look up the statement number and put CDONO in the symbol table entry. If the statement number is found in symbol table, do not alter its level.
2. Check for illegal transfer using routine DCSTN2.
3. Transfer to the statement number in columns 1-5 is legal if the level in the symbol table is greater or equal to CDONO on the top of the stack. This is determined in DCSTN1 and a DO-1 error is issued for an invalid transfer.
4. For a DO statement, do not alter the level (See e.g. 6).

Case 2: The statement number has appeared in columns 1-5. When the statement number appears in columns 7-72, the transfer is legal if the level in the statement number can be found in the DO-stack. If it cannot be found that DO-loop must have ended so we must be branching into the DO-loop. For DO statements give a DO-2 error - the object of the DO-loop is before the DO statement.

Some examples follow:

1.

				Level
1		GO TO 200		0
2		DO 100	I = 1, N	0
3	200	CONTINUE		1
4	100	CONTINUE		1

DO-stack in line 1 is 0 - 0
DO-stack in line 2 is 100 - 1
0 - 0

In line 3 the symbol table entry for 200 had level 0. Since the level on top of stack (1) is greater than 0, this is an illegal transfer.

2.

				Level
1		DO 100	I = 1, N	0
2	200	CONTINUE		1
3	100	CONTINUE		1
4		GO TO 200		1

In line 1 the DO-stack is 100 - 1
0 - 0
In line 2, the level 1 is put in the symbol table entry for 200
In line 3, the DO loop is ended leaving the DO-stack as 0 - 0
In line 4 since 200 has 1 at its level and there is no level 1 in the DO-stack we have an error (DO-1)

3.

```
                                                  Level
1            DO    100     I = 1, N               0
2            GO TO 300                            1
3     100    CONTINUE                             1
4            DO    200     I = 1, N               2
5     300    CONTINUE                             2
6     200    CONTINUE                             2
```

In line 1 the DO-stack contains 100 - 1
0 - 0
In line 2 300 is assigned level 1 in the symbol table.
In line 3 the DO-stack is 0 - 0, a DO loop has ended.
In line 4 the DO-stack contains 200 - 2 (CDONO increases by 1 with each DO statement)
In line 5 the level for 300 is 1 in the symbol table but the level at the top of the DO-stack is 2, hence, an error.

4.

```
                                                  Level
1            DO    100     I = 1, N               0
2     300    CONTINUE                             1
3     100    CONTINUE                             1
4            DO    200     I = 1, N               2
5            GO TO 300                            2
6     200    CONTINUE                             2
```

In line 2, 300 is assigned a level 1 in the symbol table. After line 4 the DO-stack contains 200 - 2 and hence in line 5 since no level 1 occurred in the DO-stack we have encountered an error.

5. The DO-stack makes it possible to check for badly nested DO loops

```
                                                  Level
1            DO    100     I = 1, N               1
2            DO    200     J = 1, N               2
3     100    CONTINUE                             2
4     200    CONTINUE                             2
```

DO-stack in line 1 100 - 1
0 - 0
DO-stack in line 2 200 - 2
100 - 1
0 - 0
In line 3 we attempt to end DO loop ending on statement 100. Since 100 is not on top of stack, the loops are incorrectly nested.

6. An example showing why DO statements are special as far as the illegal branching is concerned.

```
                                                    Level
1            DO    100     I = 1, N                  1
2            GO TO 100                               1
3            DO    100     J = 1, N                  2
4     100    CONTINUE                                2
```

In line 1, DO-stack is 100 - 1. 100 gets dummy level X'FF'
0 - 0
In line 2, 100 gets level 1 in the symbol table
In line 3, stack is 100 - 2 and 100 still has level 1
100 - 1
0 - 0
in the symbol table
In line 4, the level in symbol table is 1, and level on top of DO-stack is 2, thus an error is given.

4.5.7. End of DO loop coding

The code for the end of a DO loop is

```
L     3,ptr to pseudo statement number
BAL   14,START(R3)
```

This code branches back to the section of DO loop coding which increments the index and tests for end of DO loop.

For implied DO loops, the end of DO loop coding is

```
BAL   14,address where pseudo statement number
      is put out in ordinary DO loops.
```

4.5.8. COMMON ROUTINES

DCSTN1	This routine checks for illegal branches into DO loops, improperly nested DO loops, and end of DO loops. When SCAN finds a statement number in columns 1-5 it goes on to process the statement. Then, it looks up the statement number and calls DCSTN1.
DOENDCD	The end of DO loop coding is "outputted" here. The DO parameter bit in the 4 DO parameters is turned off in the symbol table.
DCSTN2	All statement numbers which can be transferred to (i.e. the statement number list in a GOTO) are looked up and checked for legality here.
DOUNP	This routine cancels DO parameters by turning off their DO parameter bit, for DO statements and implied DO's.

4.5.9. SWITCHES

CIFGOTSW:	When set to X'01' the next statement must have a statement number. Used in arithmetic IF and GOTO statements.
CDOBAD:	X'01' means this statement can't be end of DO loop.
CDONO:	Has the number of DO statements processed so far. Used to give each DO loop a characteristic number.
DOSWCH:	X'01' means that DCSTN was called from DODO. A special return is required.
DATASTW:	X'01' means at least one of the parameters in implied DO loop was a variable. This is illegal for DATA statements.
CTYPESW:	Not X'00' means this is a DATA statement.
DOSTA:	X'01' means this is a DO statement used in DCSTN1.
DOPARM:	Stores information on DO parameters. 2 bits are used for each parameter. The 1st bit indicates whether constant or variable (0 or 1). The 2nd bit indicates whether commoned or equivalenced or neither (1 or 0).
DOPARMH:	Stores information about DO parameters. 0 means not half word, 1 means half word variable.
GASSNSW:	X'01' means an ASSIGN statement is being processed. Used in DCSTN2 so that the statement number will not be treated as if it were GOTO n, i.e. will not be checked for illegal transfer.
DHIGHEST:	The CDONO on top of stack when entering DCSTN1 is stored here.

4.6 INOUT

4.6.1. General Organization

The compiler module INOUT contains the coding to process all I/O statements allowed by version 0 of WATFOR and in addition compiles DATA, STOP, PAUSE and CONTINUE statements. The statements processed and their entry points in INOUT are given in the following table:

1	CONTINUE	GCONT
2	STOP	ISTOP
3	PAUSE	IPAUS
4	BACKSPACE	IBACK
	ENDFILE	IENDF
	REWIND	IREWI
5	PUNCH	IPUNC
	PRINT	IPRIN
	WRITE	IWRIT
	READ	IREAD
6	DATA	IDATA

Each entry point is entered via the CENT macro which establishes addressability for that point as well as for the save area.

The module is divided roughly into seven sections because of the similarity in syntax of some of the statements. The sections which should be considered together and which share coding are indicated in the left column of the table above. The seventh section is a set of utility routines used by these various processors and all are contained in the one save area used by the module. Some examples of these utility routines are a processor for units in I/O statements, an I/O list compiler, error message routines etc. These will be described individually later on since their use is common to several of the statement processors. For example the unit processor is called by BACKSPACE and READ, the I/O list compiler is called by READ and DATA.

The module INOUT calls entry points in other compiler modules to perform certain functions. Specifically, these are:

- ARITH to compile expressions in I/O lists
- DODO to compile implied DO's in I/O lists
- DCSTN2 (entry point in deck DODO) to perform error checking on statement numbers
- FRIOSCAN to process hexadecimal constants in DATA statements

Entry point IDATA of INOUT is called by the Type Statement processor (deck SPECS) to handle initializing information in type statements.

The principal switches used by INOUT are:

- CADDSW for communication with ARITH
- CDOIO to indicate to DODO that start or end-of-DO coding is requested
- CIFGOTSW to indicate a transfer statement is being compiled (GOTO, STOP, etc.) and next statement should be numbered
- CDOBAD to indicate that the statement being compiled may not be the object of a DO loop
- CTYPESW for communication between the DATA and TYPE processors - DATA and DODO.

The register conventions used in the module are roughly as follows:

R1,R9	-	used as stack pointers
R11	-	used for programme addressability and for calling some utility routines
R13	-	covers save area and utility routines
R14,R15	-	used for calling some utility routines

The rest of the description of INOUT is split into sections which describe the logical organization just outlined and each section is headed by the statement types considered.

4.6.2. CONTINUE

No object code is produced for this statement. A simple check is performed to verify that the keyword CONTINUE is not followed by any other characters.

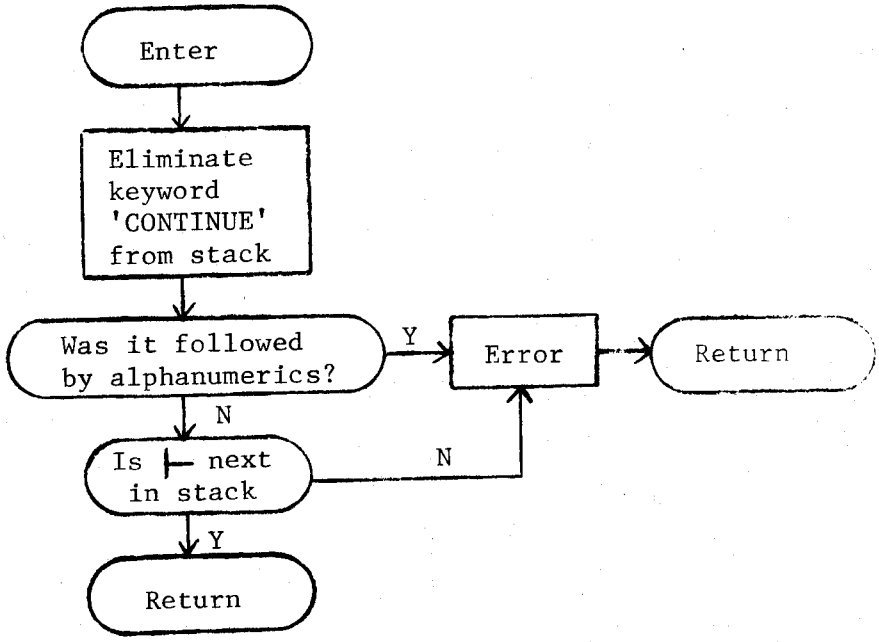


Figure 4.6.1.

4.6.3. STOP

Since WATFOR was to be used principally for fast processing we decided that the operator messages which result from the execution of STOP statements might impair the performance of the compiler. Thus it was decided that statements of the form STOPn would be screened at compile time with a warning message to the programmer that the constant n was being ignored. For the same reason the simple STOP statement would produce no operator message whatsoever.

However since some installations might wish to allow these messages we included an assembly parameter &STOPN which could modify the compiler to allow these statements just as the other compilers would. Thus the description of the STOP statement processor provides the logic for both cases.

```
1.      &STOPN      SETC      'NO'
```

Switches CDOBAD, CIFGOTSW are set to indicate that this statement can't be the object of a DO or a logical IF. The keyword STOP is eliminated from the stack. A test is made for concatenated characters. If concatenated characters are present, the stack is tested to verify they are numeric. If so, a warning message PS-0 is given; if non-numeric, an error is given and the routine returns. The stack is then checked for ⊢. The object code output is B XSTOP.

```
2.   &STOPN     SETC     'YES'
```

The switches are set as above. The same checks are made. In the event of a numeric character string the following object code is output:

```
BAL     R14,XSTOPN
DC      AL4(n)
```

For a statement of the form STOP the code B XSTOP is produced.

4.6.4 PAUSE

The same remarks concerning operator messages apply here with the added comment that the PAUSE statement in the hands of mischievous students could disrupt the operation of the computer installation.[1]

The CDOBAD switch is set to indicate this statement can't be the object of a DO. After the keyword PAUSE is eliminated from the stack a test is made for a following numeric string, hollerith constants, null string or error conditions.

```
If &STOPN  SETC  'NO', a warning PS-1 message is given and no code produced.
If &STOPN  SETC  'YES' the following object code is output
           BAL   R14,XPAUSHOL      for PAUSEn
           DC    X'00',AL3(n)      (n = 0 for PAUSE)
or         BAL   R14,XPAUSHOL      for PAUSE'hol'
           DC    AL1(r),AL1(ℓ - 1),AL2(hol*)
```

where r is a relocator code, ℓ is the length of the hollerith constant (possibly right padded with blanks by SCAN to a multiple of 4) and hol* is a pointer to the symbol table entry for this hollerith.

---

1 For example, the following programme showed up several times on the day the use of '+' as carriage control was described to one undergraduate class:

```
      DO 1 I=1,1000
    1 PRINT 2
    2 FORMAT ('+ THIS PROGRAMME CHEWS UP THE PRINTER RIBBON')
      STOP
      END
```

The latter code after relocation is

```
BAL     R14,XPAUSHOL
DC      AL1(ℓ - 1),AL3(hol - START)
```

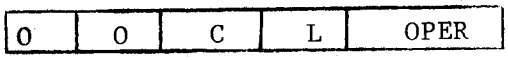
where hol is the location of the hollerith constant stored by SCAN.

4.6.5. BACKSPACE, ENDFILE, REWIND

The relocated object code for these statements is

```
BAL     R14,XIOINIT
DC      AL1(OP),AL3(unit - START)
```

where OP is a bitfield which describes the operation to be carried out by run-time routine XIOINIT. This field is as follows:



where
- C = 1 if unit is COMMONed or EQUIVALENCEd
  = 0 otherwise
- L = 1 if unit is 2 byte integer
  = 0 if unit is 4 byte integer
- OPER is a 4 bit subfield which specifies operation by the codes:

  0 formatted input
  1 formatted output
  2 unformatted (binary) input
  3 unformatted (binary) output
  4 free input
  5 free output
  8 Backspace
  9 Rewind
  10 Endfile

Each entry point does initialization of OP in the skeleton object code and to eliminate the keyword. All three converge to a routine which does the following:

- eliminates the keyword from the stack
- calls the unit processing utility routine
- outputs the unrelocated code

```
BAL     R14,XIOINIT
DC      AL1(r),AL1(OP),AL2(unit*)
```

- checks the stack for ⊢ following the unit.

4.6.6. PUNCH, PRINT, WRITE, READ

These statements are considered together since there is basically one routine which compiles all four with some initialization being done at each entry point. The main compilation routine, in itself, consists for the most part of calls to utility routines, and so is fairly straightforward. One complication is the fact that the READ entry point must distinguish the three variations allowed in WATFOR

```
e.g.   READ1,X
       READ(5,1)X
       READ,X
```

Similarly PRINT and PUNCH each must distinguish two cases.

The following description is a paraphrasing of the coding for these routines in the order it appears in the module.

1. PUNCH: Initialize a switch for punch unit number (&PUNCH) and join coding for PRINT.
2. PRINT: Initialize a switch for print unit number (&PRINT). If a non-blank character follows the keyword (PRINT or PUNCH) in the stack the operation is formatted.

Otherwise initializing is done for free output and a branch to the free I/O operation test routine is taken. (See below under READ). In case of formatted output, the stack fixup routine is called with a return to the main compile routine. (The stack fixup routine changes the stack configuration so that a statement of the form PRINT1 or READ2 is transformed to look as if statement WRITE(6,1) or READ(5,2) were actually being compiled.)

3. WRITE: The keyword is eliminated from the stack and initialization is done for output before branching to the main compile routine.
4. READ: Initialization for input is performed. If there is a concatenated format, e.g. READ1, the stack fixup routine is called with a return to the main compile routine. Otherwise the free operation test is performed: if the delimiter following the keyword is a ',' we have a free operation; if not we have an error in the case of PRINT e.g. PRINT+ or we assume a READ operation of the form READ(and go to the main compile routine. For free operations, the unit involved is processed by a utility routine with the return set to the code output point in the main routine. This way all checking for format, END/ERR returns is bypassed since we know none are present.

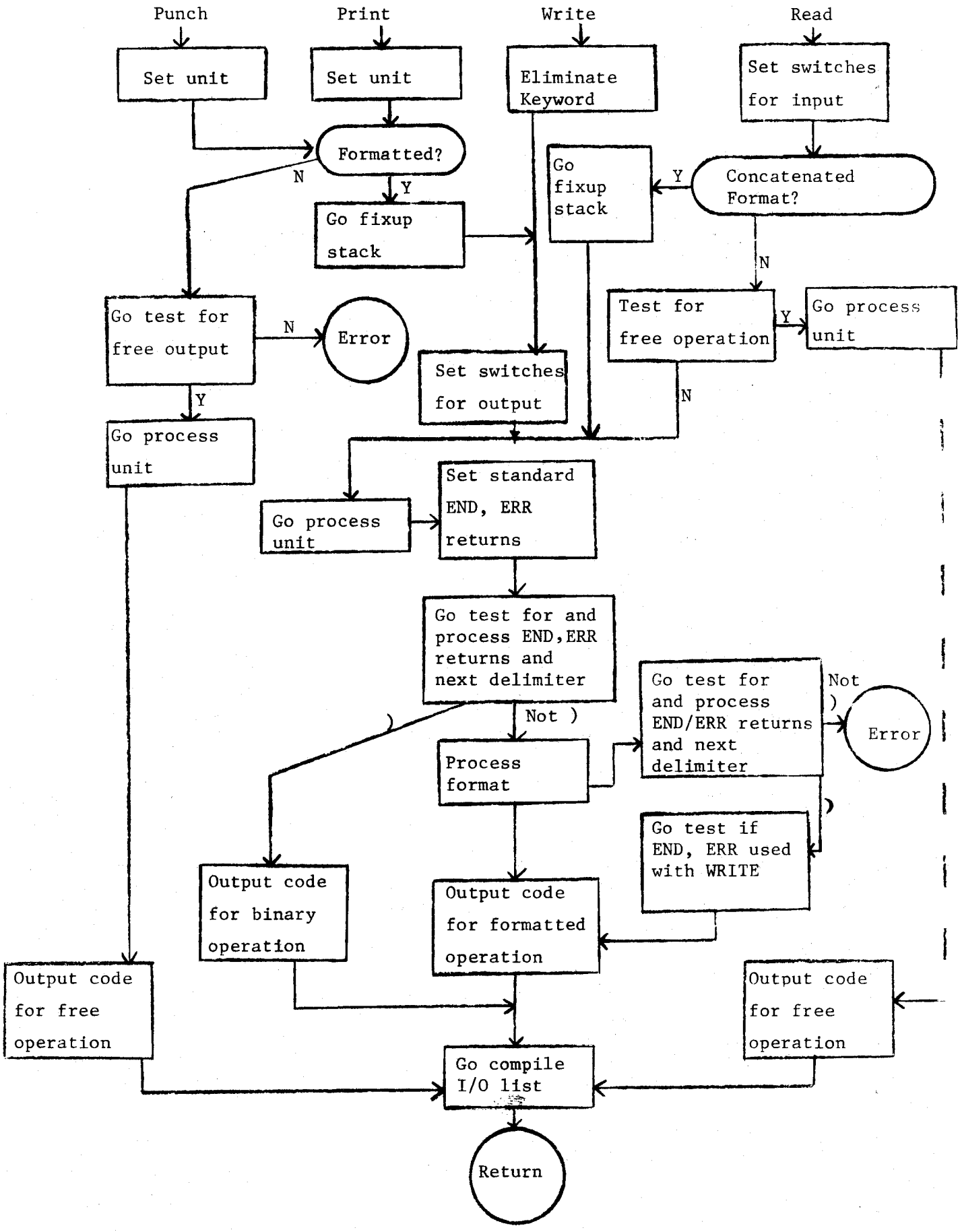

Punch
Print
Write
Read
Set unit
Set unit
Eliminate Keyword
Set switches for input
Formatted?
N
Y
Go fixup stack
Y
Concatenated Format?
Go fixup stack
N
Go test for free output
N
Error
Test for free operation
Y
Go process unit
Set switches for output
Y
N
Go process unit
Go process unit
Set standard END, ERR returns
Go test for and process END,ERR returns and next delimiter
Not)
Go test for and process END/ERR returns and next delimiter
Not)
Error
Process format
Go test if END, ERR used with WRITE
Output code for binary operation
Output code for formatted operation
Output code for free operation
Output code for free operation
Go compile I/O list
Return

Figure 4.6.2

5. Main compile routine: At the entry to this routine the stack register points to the first delimiter after the keyword. If this is '(' we set a flag to assume the operation is binary. The unit processing routine is called. This checks for and processes a valid variable or constant unit. Next the skeleton object code is set for the default END and ERR returns (XIOEND,XIOERR) and the utility routine IOENDERR is called to check for the presence in the stack of either or both. (This routine advances the stack pointer beyond the END/ERR parameters if they are present and returns to the first instruction following the call if the delimiter pointed to is ')' or the second if no END/ERR parameters are found and the delimiter is a ','. Thus binary operations of either type e.g. READ(1) or READ(1,END=2) are easily identified.)

In case of a binary operation we branch to output object code. Otherwise a test is made for the presence of a format specification. This is processed by one of two routines which handle the cases of variable format or statement number. The former looks up the format symbol and checks it as being an array. If so the number of dimensions is placed in the skeleton object code for run-time use. The latter routine tests or sets the format bit in the symbol table in case the statement number is old or new respectively.

The format address is entered into the skeleton code and the END/ERR utility routine is called. Following this, if the next delimiter is not ')' an error return is taken. Otherwise we proceed to output object code.

Before producing object code a test is performed to see if END/ERR parameters were present in a WRITE statement. Object code depending on the operation is output prior to calling the I/O list compiler. For formatted I/O the code is:

```
CNOP    0,4
BAL     R14,XIOINIT
DC      AL1(r),X'OP',AL2(unit*)
DC      AL1(r),AL3(END return)
DC      AL1(r),AL3(ERR return)
DC      AL1(r),AL1(Dims),AL2(format*)
```

Object code for binary operations is the same except the last 4 bytes are not output.

For free I/O the object code is

```
CNOP    0,4
BAL     R14,XIOINIT
DC      AL1(r),X'OP',AL2(unit*)
DC      AL4(XIOEND-START)
DC      AL4(XIOERR-START)
```

'OP' is as described above under REWIND etc. (Section 4.6.5.)

'Dims' is - 0 for a format statement
- 4 times the number of subscripts for a variable format array

Here 'r' stands for various relocator codes which tell Relocator Phase II how to process this object code.

XIOINIT is the runtime object code - FIOCS/FORMCONV interface routine. (See section 3.20.).

Following the production of this code the I/O list compiler is called to compile any I/O list elements. This is described below.

Several examples follow showing the relocated object code for the specification part of an I/O statement.

e.g. 1. `READ(5,1)`

```
CNOP    0,4
BAL     R14,XIOINIT
DC      AL1(0),AL3(=5 - START)
DC      AL4(XIOEND - START)
DC      AL4(XIOERR - START)
DC      AL1(0),AL3(#1 - START)
```

e.g. 2. `READ(I,END=2)`

```
CNOP    0,4
BAL     R14,XIOINIT
DC      X'02' AL3(I - START)
DC      AL4(#2 - START)
DC      AL4(XIOERR - START)
```

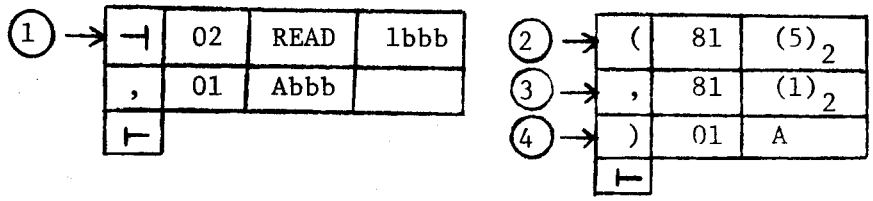
e.g. 3.

```
DIMENSION A(20)
COMMON I
READ(I,A,ERR=3)
```

```
CNOP    0,4
BAL     R14,XIOINIT
DC      X'20',AL3(=A(I - START))
DC      AL4(XIOEND - START)
DC      AL4(#3 - START)
DC      AL1(4),AL3(.A)
```

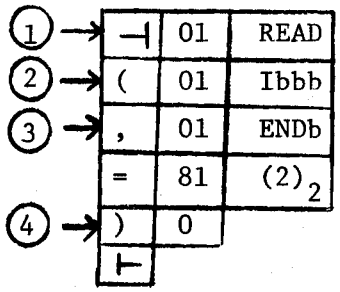
Stages in scanning the stack are given for several cases. (Stack links not shown)

1. READ1,A



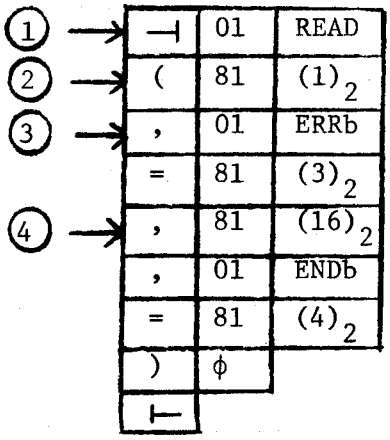
Position of stack pointer (1) - on entry to READ processor.
(2) - after calling stack fixup routine and for call to unit processor.
(3) - at processing of format.
(4) - before and after calling END/ERR processor and before calling list compiler.

2. READ(I,END=2)



Position of stack pointer (1) - on entry to READ.
(2) - after test for free input and at call to unit processor.
(3) - at call to END/ERR processor.
(4) - at return from END/ERR processor and at call to list compiler.

3. READ(1,ERR=3,16,END=4)



(1), (2), (3), same as previous example.
(4) position of pointer at error exit in END/ERR processor. (Error since ERR=3 not followed at (4) by ')'.)

4.6.7. DATA

This section processes DATA statements and initialization in Type Statement e.g. INTEGER A(10)/10*2/,B,C/3/. The processing is almost the same for both statements except that there is a jumping back and forth between the DATA compiler and the type statement compiler. The result of this action is that in essence the type statement shown above is treated as if it were really the statements

```
INTEGER A(10), B,C,
DATA    A/10*2/,C/3/
```

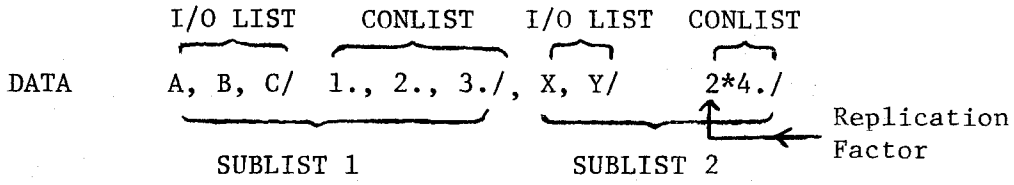
For an ordinary DATA statement, control remains in this routine until the end of the statement is encountered.

Data initialization statements present somewhat of a problem for a one-pass compiler since at the time the DATA statement is processed, no storage locations have been assigned for the variables involved. Thus the information for initializing must be kept around until address assignment is completed. In the case of WATFOR, the initializing can take place only following the processing of Relocator Phase III since array and common/equivalenced storage is assigned then. The way this is accomplished in WATFOR is to compile object code for DATA statements which is 'executed' following phase III of the Relocator but preceding the actual entry of the mainline programme.

Since initialization for all DATA statements, whether in main or subprogrammes must be done before execution is initiated, the object code for all such statements is chained together with the final DATA statement being linked to the main entry point. Thus there are really three distinguishable times in WATFOR's processing of a job viz. compile time, DATA-statement pre-execution time and execution time.

The object code that is produced for a DATA statement is very similar to that for a READ statement since the syntax of the variable lists in both statements is virtually the same. The difference is that when the DATA object code is executed, values are moved into memory locations, not from buffers, but from other memory locations where constants were stored at compile time. Thus much of the machinery that exists for other purposes in the compiler is useable in DATA processing, e.g. XSIMPELT, XENDLIST, the I/O list compiler, etc.

It is probably instructive at this time to establish some of the terms which will be used in the following discussion by considering the DATA statement:



This statement is composed of two DATA sublists, each of which is further considered as two parts, the I/O list and the constant list (conlist). The object code produced for this statement reflects its structure and looks as follows (after relocation).

```
                B         AROUND
                BAL       R14,XISNRTN
                DC        AL2(ISN)
                CNOP      0,4
                BAL       R14,XDATA
                DC        A(next DATA statement in chain-START)
                DC        AL1(s),AL3(savearea-START)
               {coding for
               {sublist 1
               {coding for
               {sublist 2
AROUND          EQU       *
```

The instruction B AROUND insures that, after normal programme execution has been initiated, the DATA statement object code is never performed again.

Here 's' is the sublist count (s = 2 for the example) and 'savearea' is the address of the savearea for the subprogramme this DATA statement appeared in. This is used by the initializer XDATA to set up registers R5 - R10 for addressing of the variables involved in the initialization.
(XDATA is the 'pre-execution' DATA statement interpreter).

The coding for sublists is composed of three parts as follows:

```
DC AL1(0),AL1(n),AL2(p)
{I/O list
{coding
{conlist
{coding
```

The I/O list coding is the same as would be constructed for the I/O list part of READ, A, B, C.

The conlist coding consists of a sequence of 8 byte constant descripters, one for each of the n constants in the conlist. The value p is a pointer to the first descripter in the conlist and is calculated as the displacement in bytes of the descripter from the location of the DC.

Each conlist descripter is constructed as follows:

```
DC   AL1(0),AL1(ℓ - 1),AL2(replication factor)
DC   AL1(t),AL3(constant-START)
```

Here ℓ is the length in bytes of the constant, t is the type of the constant (t = 8 for literal, 9 for hexadecimal, same as variable types for logical, integer, real and complex i.e. low 3 bits of B1).

The byte switch CTYPESW is used for communication between the DATA and type statement processors and can have the following settings and meanings:

-X'00'	-	DATA statement to be compiled (entry from SCAN)
or	-	an error was encountered in the conlist compilation. (This is used by the type processor as a signal to abandon this statement e.g. REAL A/B/,C,D)
-X'01'	-	last entry from type processor to close out the object code. Set by type processor.
-X'02'	-	first call from type processor (set by type processor
-X'03'	-	subsequent calls from type processor (set to this by DATA processor)

A rough flowchart of the processor is given in Figure 4.6.3. and a description of it follows.

On entry CADSSW is set for ARITH to indicate a DATA statement. A test on CTYPESW is performed to see what sort of an entry this is:

(i) if last call from type we branch to close out the object code.

(ii) if from type processor we fixup the stack to make it look like an ordinary DATA statement e.g. for INTEGER *2A*4/3/,--- the stack pointer (R1) received from type points to the opening '/' of the conlist and R9 points to the symtab entry for A.

Thus

R1 →

ϕ	01	Abbb
*	81	$(4)_2$
/	81	$(3)_2$
/	ϕ	
,	- - -	

is transformed to

R1 →

⊣	02	Abbb	bbbb
/	81	$(3)_2$	
/	ϕ		
,	- - -		

which is the stack configuration for the statement

DATA A/3/,...

If this is the first entry from type we set CTYPESW to X'03' and join the coding for the entry from SCAN, (iii) below, which first puts out the statement initialization coding; if a subsequent entry from type we bypass this processing.

(iii) if entry from SCAN we
- initialize the sublist count
- output the (unfilled) branch around, the ISN coding, and the CNOP 0,4
- link this data statement in to the last one compiled using CENTRYPD (initialized by the new job initializer)
- output the statement initialization coding.

Next, the I/O list compiler is called to process the list of variables; this returns when the opening '/' of the conlist is encountered and the conlist is compiled.

This compilation proceeds as follows: Output the (unfilled) 4 byte conlist count and pointer. Then: test the stack delimiter for '-', '.', '(' or '+' which signal a possible numeric constant e.g. ...A/+1./...

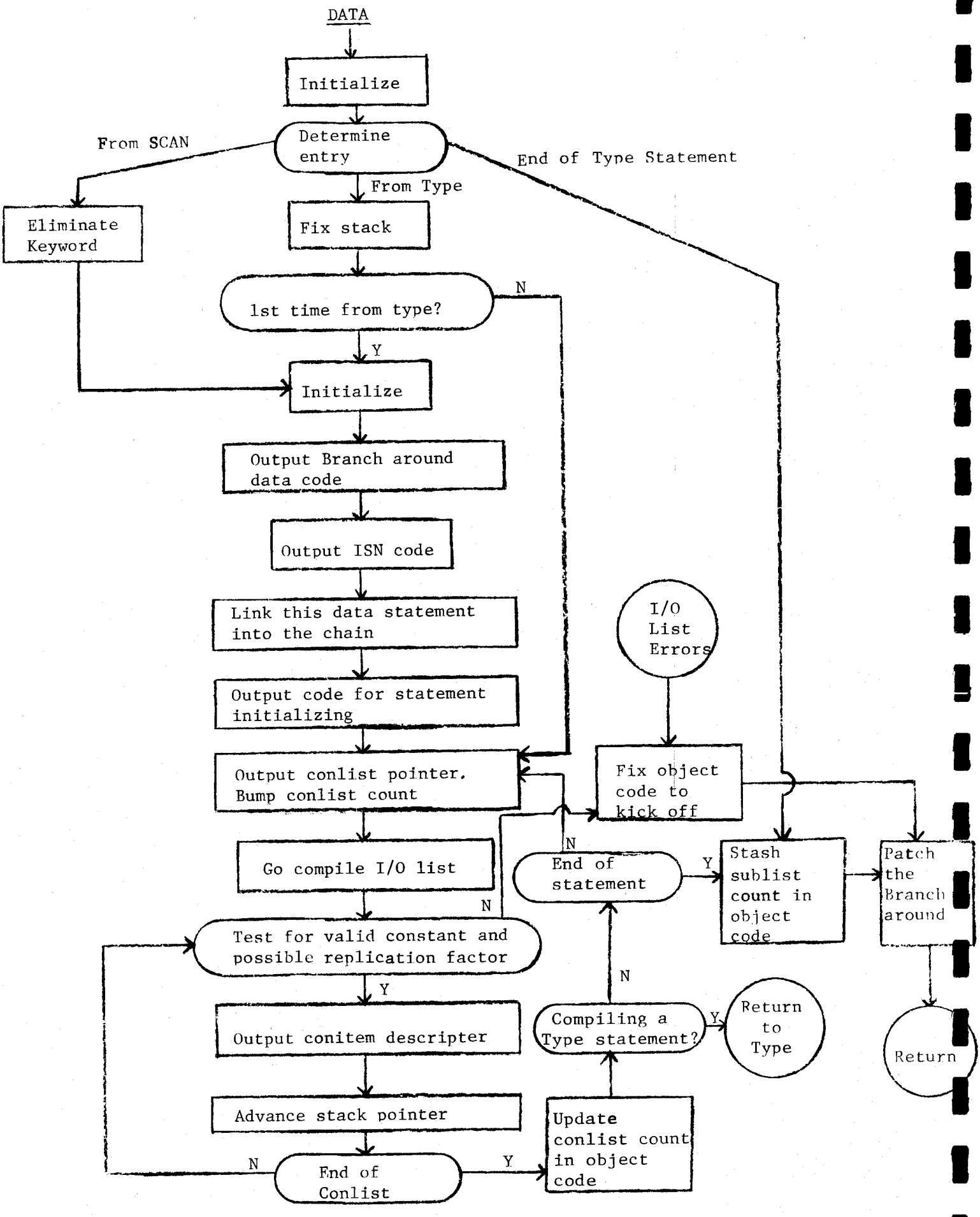
- test for logical constant of form .TRUE. or .FALSE. (stack code is X'41')
- test for undelimited numeric constant ...A/3./... (stack code is X'81')
- test for hollerith constant (stack code is X'21')
- test for logical constant of form T or F (stack operand is Tbbb or Fbbb)
- test for hexadecimal constant. (1st letter of stack operand is Z)

If these tests fail an error exit is taken.

Numeric constants are processed by the calling of a special entry point CONTEST in the constant collector with an error return in R0 in case no valid constant is found e.g. ... A/+B/ ...

Hexadecimal constants are converted to internal form by calling the runtime input field scanner FRIOSCAN with a pointer to the start of the first hex digit in R1, the length -1 of the field to be converted in R3 and a code in R7 to specify hex conversion. Return is to the first statement following the call in case of an error in the hex constant or to the second statement for no error. In the former case the conlist error exit is taken; in the latter, the converted constant, right justified in a 16 byte field is looked up in the constant list by routine CONLOOK.

Figure 4.6.3.


DATA
Initialize
Determine entry
From SCAN
End of Type Statement
From Type
Eliminate Keyword
Fix stack
1st time from type?
N
Y
Initialize
Output Branch around data code
Output ISN code
Link this data statement into the chain
Output code for statement initializing
Output conlist pointer. Bump conlist count
I/O List Errors
Fix object code to kick off
Go compile I/O list
N
End of statement
Y
Stash sublist count in object code
Patch the Branch around
N
Test for valid constant and possible replication factor
Y
Output conitem descripter
Compiling a Type statement?
Y
Return to Type
N
Return
Advance stack pointer
Update conlist count in object code
N
End of Conlist
Y

For each type of constant the con-item descripter is prepared and output to the object code. If the closing '/' is not encountered we loop to process another constant, bumping the conlist count for each one.

When the closing '/' is encountered, we fill in the conlist count and pointer and test if we are really compiling for a type statement; if so return to the type processor if not test for another DATA sublist and if present repeat the above process.

If no more sublists are present or this is the last entry from the type processor we finish off by filling in the displacement part of the branch around and the sublist count which were output previously and then return by the INOUT exit routine which resets CTYPESW to X'00'.

The conlist error exit performs the same processing as described in the previous paragraph, but in addition modifies the initialization coding instruction BAL R14,XDATA to BAL R14,XBOOT to kick off when this faulty DATA statement is encountered at pre-execution time.

An example of the complete relocated coding generated for a data statement follows:

```
DATA A,B,I,J/1.,2.,2*3/,X/'ABCb'/,L/T/

        B       AROUND
        BAL     R11,XISNRTN
        DC      AL2(ISN)
        CNOP    0,4
        BAL     R14,XDATA
        DC      A(next DATA statement in chain - START)
        DC      AL1(3),AL3(savearea)
D       DC      AL1(0),AL1(3),AL2(M-D)
        I/O-list coding for A,B,I,J
M       DC      AL1(0),AL1(3),AL2(1)
        DC      AL1(4),AL3(=1.-START)
        DC      AL1(0),AL1(3),AL2(1)      }
        DC      AL1(4),AL3(=2.-START)     }  CONLIST 1
        DC      AL1(0),AL1(3),AL2(2)      }
        DC      AL1(2),AL3(=3 -START)
E       DC      AL1(0),AL1(1),AL2(N-E)
                I/O list coding for X
N       DC      AL1(0),AL1(1),AL2(1)      }CONLIST 2
        DC      AL1(8),AL3(='ABCb'-START) }
F       DC      AL1(0),AL1(1),AL2(P-F)
                I/O list coding for L
P       DC      AL1(0),AL1(1),AL2(1)      }  CONLIST 3
        DC      AL1(0),AL3(XTRUE-START)   }
        BALR    R11,0
AROUND  EQU     *
```

4.6.8. INOUT Utility Routines

The various utility routines mentioned in the previous sections all follow INOUT's savearea and hence are addressable by R13 from any part of the module. As well as utility routines, the skeleton object code and some constants and work areas used by INOUT are in this section.

There are six utility routines and their entry points and functions are tabulated below in order of appearance in INOUT.

I/O-list compiler	....	IOLISTNT
		ILISTCMP
Unit Processor	....	ICHKUNIT
END/ERR processor	....	IOENDERR
Stack fixup routine	....	IFIXSTK
Error message routines	....	too numerous to list
Exit routine	....	too numerous to list

With the exception of the I/O list compiler, these routines are relatively short and straightforward and hence an attempt will be made to keep their descriptions brief.

1. I/O-list Compiler

The I/O-list compiler is a utility routine called by the READ, WRITE, PRINT, PUNCH and DATA processors as mentioned above. Its purpose is to construct object code which provides linkage to the execution time conversion and/or data movement routines (e.g. FORMCONV, INBINI, PDATA). The object code must also provide to these routines location and type information about the items being handled, be they simple variable names, subscripted array elements or array names. Since the arithmetic expression compiler ARITH has the ability to handle these sorts of entities, it was decided that ARITH would actually produce the object code to generate item addressing while the list compiler would merely produce linkage code. Thus, it was easy to allow expressions in I/O statements since the I/O-list compiler is really more of an 'isolater' in the sense that its processing is roughly as follows:

- isolate in the stack an expression (which might be simply a variable name) and call ARITH to process it
- produce linkage coding for the item
- repeat for each item in the I/O list.

It is also an isolater, in the sense that the processing just mentioned may have to be interrupted occasionally to handle implied DO-loops and the I/O-list compiler isolates the implied loop information in the stack for processing by the DO-compiler DODO.

DODO and the list compiler communicate by the byte switch CDOIO which is set to indicate that start- or end-of-DO coding is required.

Communication is carried on between the list compiler and ARITH by means of the byte switch CADSSW and the stack. For example, READ sets CADSSW to X'82' and ARITH uses this to give an error message should the programme try to 'read' an expression e.g. READ, X+3.*Y

In short ARITH uses CADSSW to test for errors which are unique to input, output or DATA lists.

The stack is used to communicate type and mode information about the item ARITH has just processed when control is restored to the list compiler. There are basically four things ARITH reports back about the item compiled:

(i) coding was produced to put the item's run-time address in R3 e.g. for X(I).
(ii) the item was a simple expression; its location is in the stack.
(iii) the item was an array name; the location of its dot-routine is in the stack.
(iv) an error was detected; don't produce linkage coding.

Information about the item's type and length is also obtainable from the stack.

In cases (i), (ii), (iii) above the I/O-list compiler finishes processing the item by producing coding which has the ultimate effect of providing, in register R3, the memory location of the item to be processed by the appropriate run-time routine and in R14 a pointer to its type and length information.

An example may clarify this. (Assume Y, Z are arrays.) For WRITE(6,1)X,Y,(Z(I),I=L,M) the relocated object code produced consists of the following eight parts:

```
(1)  CNOP    0,4
     BAL     R14,XIOINIT
     DC's    etc. (see above, section 3.20.)

(2)  CNOP    0,4
     BAL     R14,XSIMPELT
     DC      AL1(t),AL3(X-START)
```

```
(3)  {  CNOP     0,4
        BAL      R14,XARRAY
        DC       AL1(t),AL3(.Y-START)

(4)     Start of DO coding

(5)     Coding to leave A(Z(I)-START)) in R3

(6)  {  CNOP     0,4
        BAL      R14,XSUBSELT
        DC       AL1(t),AL3(0)

(7)     End of DO coding

(8)     BAL      R14,XENDLIST
```

The coding (1) is the statement initialization coding (described above) which establishes linkage to the proper runtime routines depending on the operation to be performed.

XSIMPELT puts the location information A(X-START) in R3 and goes to some runtime routine established by coding (1).

XSUBSELT merely goes to the runtime routine since P3 already contains the location of Z(I).

XARRAY repetitively generates in R3 the address of successive elements of Y and goes to the runtime routine for each one. Control returns to the object code above when the array is exhausted.

XENDLIST closes off activity for the statement (e.g. writes out a buffer).

In each case above 't' is coded information about the item type and length and in (3) also provides the number of dimensions of array Y.

(See section 4.5 for description of coding (4) and (7).

The processing performed by the list compiler is somewhat different for I/O and DATA lists since the syntax of these is slightly different e.g. an unbracketted '/' is end-of list delimiter for DATA whereas it may be part of an expression for PRINT

```
e.g.    DATA      X/2/
        PRINT ,   X/2/B
```

Also, for items in DATA lists, the symtab initialization bit must be turned on but ignored for I/O statement. In order for the list compiler to accomplish these operations the call to it is followed by a set of five instructions which are executed by the list compiler. Thus, the call from the I/O statements looks like

```
BAL     R11,IOLISTNT
B       12(R8)
NOP     0
NOP     0
B       IERXIT
B       INOINIT
```

while the call from DATA is essentially

```
BAL     R11,ILISTCMP
B       IBADEND
BZ      12(R8)
CLI     2(R1),X'4D'
B       ITUFF
NOP     0
```

The first instruction following the BAL is executed (via R11) when a '⊢' is encountered in the list (e.g. PRINT,X is valid but DATAX is invalid).

The second and third instructions are used to test for the significance of '/' as mentioned above.

The fourth is used in case syntax errors occur in the list. For DATA statements, the object code must be closed off. (See under DATA above.) No special processing is required for I/O statements.

The fifth is used to test if the initialization bit should be turned on or ignored.

The actual instructions shown above have been designed to accomplish these ends.

The entry point IOLISTNT is provided in the list compiler to test for I/O statements with no list e.g. READ(5,1).

A description of the routine now follows (See chart 4.6.4.)

Register 1 is assumed to point to the entry in the stack where the I/O list starts. The DO-loop counter is initialized to zero.

List item processing commences by checking if the stack delimiter pointed to is '('. If it is, we assume this signifies an implied DO, the current stack position is saved and a search is made in the stack for an '=' at the same bracket level. The ',' which precedes it is overlaid with ⊙ (pseudo-comma or implied-DO mark), the DO-loop counter is incremented by 1, a warning message is issued if this implied DO is in a DATA statement and DODO is called with CDOIO set to X'01' to indicate start-of-DO coding is to be compiled for the loop specification (e.g. I=M,N) starting in the stack entry pointed to by R1. (The entry which now contains ⊙).

DODO returns with R1 pointing to the ')' which terminates the implied DO. The link part of the stack entry containing ⊙ is modified to point to the entry following the ')', thus effectively eliminating the 'I=M,N)' part of the stack. The stack pointer is restored to the opening '(' (saved above) and this item processing is repeated.

If the stack delimiter is not a '(' it is overlaid with a '⊣'. A search is then made in the stack for the delimiter, say θ, which marks the end of this expression (i.e. ',' or '⊢' for I/O, ',' or '/' for DATA).

This is saved and overlaid with a '⊢'. The expression is now isolated between '⊣' and '⊢' and ARITH is called with R9 pointing to the stack entry containing '⊣'.

Upon return, the stack contents at R9 are examined to determine the results of ARITH's processing. If ARITH has indicated an error we skip to check for more items. Otherwise, the stack contains enough pointers etc. to perform the following:

- turn on the initialization bit of a symbol table entry if this is a DATA statement. Also test if COMMON is being initialized and this is not a BLOCK DATA subprogramme.
- prepare and output one of the following sets of object code: (Stack entry returned by ARITH shown to right).

```
BAL     R14,XSIMPELT
DC      AL1(r),AL1(t),AL2(item*)
```

R9→ | links | ⊣ | A4 | B_1 | B_2 | item* |

```
BAL     R14,XSUBSELT
DC      AL1(r),AL1(t),AL2(0)
```

R9→ | link | ⊣ | A8 | B_1 | |

```
BAL     R14,XARRAY
DC      AL1(r),AL1(8d+t),AL2(array*)
```

R9→ | link | ⊣ | AC | B_1 | B_2 | array* |

Here 'r' represents relocator codes, 't' is type and length code, 'd' is number of dimensions. ARITH also adjusts the 'link' to point to the '⊢' at the end of the expression. Thus the delimiter θ is easily restored and tested. If this is a ',' we repeat the processing for another list item. If it is '⊙', end-of-DO coding for an earlier processed implied DO must be output. The DO-loop counter is decremented by 1 and DODO is called with CDOIO set to X'FF' to unstack the DO-loop. We repeat delimiter testing on the next delimiter.

If the tested delimiter is '⊢' (or '/' in a DATA statement) the list compiler outputs the list closing object code

```
BAL     R14,XENDLIST
```

and returns. Otherwise an error exit is taken.

The example which follows shows the various stages in the compilation of the I/O-list for PRINT 1,X,((B(I), I=1,5), J=1,3). Figure 4.6.5. shows the stack as it appears at the various stages and the object code for the statement: (The stack code field is not shown and links not explicitly shown are assumed to point to the next stack entry.)

Description of example:

(i) List compiler called by PRINT with input pointer (1).
(ii) Expression X isolated and ARITH called with pointer (2).
(iii) ARITH returns (3); list compiler produces code 2 restores ','. The '(' at (4) causes scan for '=' and calls DODO with pointer (5).
(iv) DODO produces code 3 and returns pointer (6); list compiler sets new link.
(v) The '(' at (7) causes scan for '=', list compiler calls DODO with pointer (8).
(vi) DODO produces code 4 and returns pointer (9), list compiler sets new link.
(vii) Expression B(I) is isolated and ARITH called with pointer (10).
(viii) ARITH produces code 5, sets new link and returns pointer (11).
(ix) List compiler produces code 6, restores , at (12). The , signals call to DODO for coding 7.
(x) Upon return from DODO, list compiler advances stack pointer to (13) and calls DODO for coding 8.
(xi) DODO returns and list compiler advances stack pointer to (14), produces code 9 and returns to PRINT.

2. Unit Processor

This routine is called by a BAL R11,ICHKUNIT to check for presence in the stack of valid constant or variable units in all I/O statements and to set up the unit address in the skeleton object code. The first 8 bytes of object code is the same for all I/O statements, viz,

```
BAL     R14,XIOINIT
DC      AL1(r),X'OP',AL2(unit*)   (unrelocated)
```

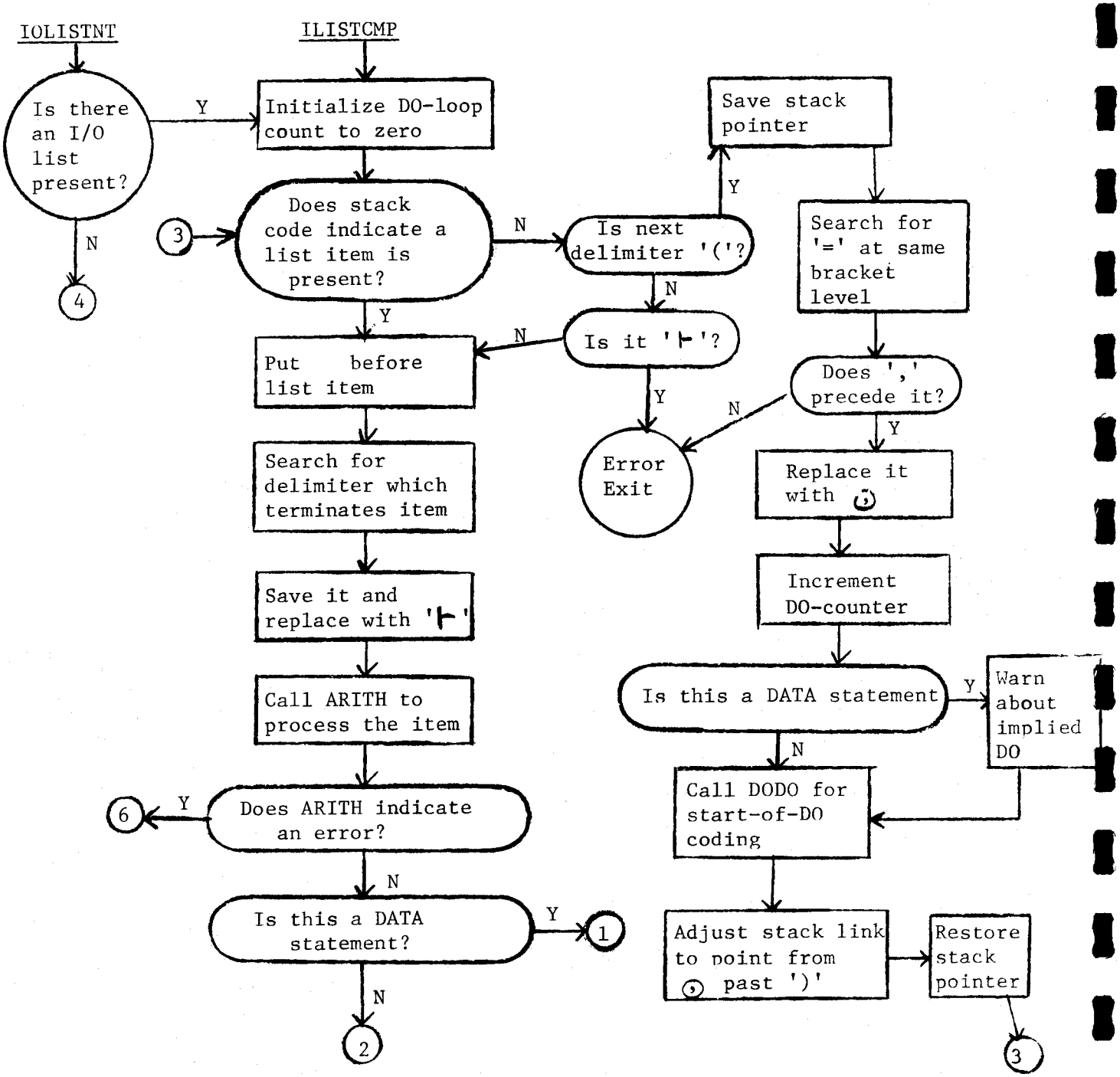

IOLISTNT
ILISTCMP
Is there an I/O list present?
Y
N
Initialize DO-loop count to zero
Does stack code indicate a list item is present?
3
N
Y
Is next delimiter '('?
Y
N
Save stack pointer
Search for '=' at same bracket level
Does ',' precede it?
N
Y
Is it '⊢'?
N
Y
Put before list item
Error Exit
Replace it with ,
Search for delimiter which terminates item
Increment DO-counter
Save it and replace with '⊢'
Is this a DATA statement
Y
Warn about implied DO
N
Call ARITH to process the item
Call DODO for start-of-DO coding
6
Y
Does ARITH indicate an error?
N
Is this a DATA statement?
Y
1
N
Adjust stack link to point from , past ')'
Restore stack pointer
2
3

Figure 4.6.4.

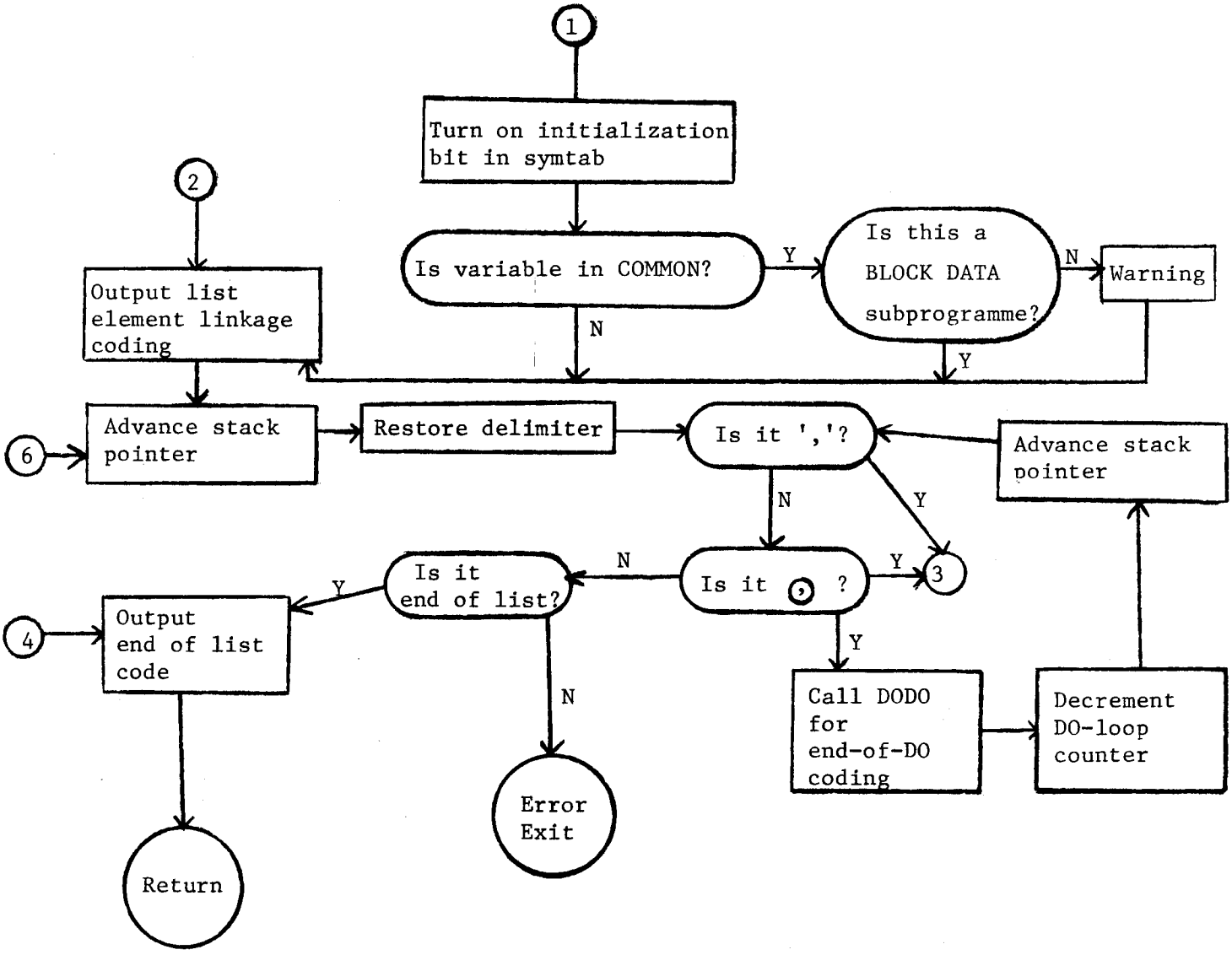


Figure 4.6.4. (Continued)

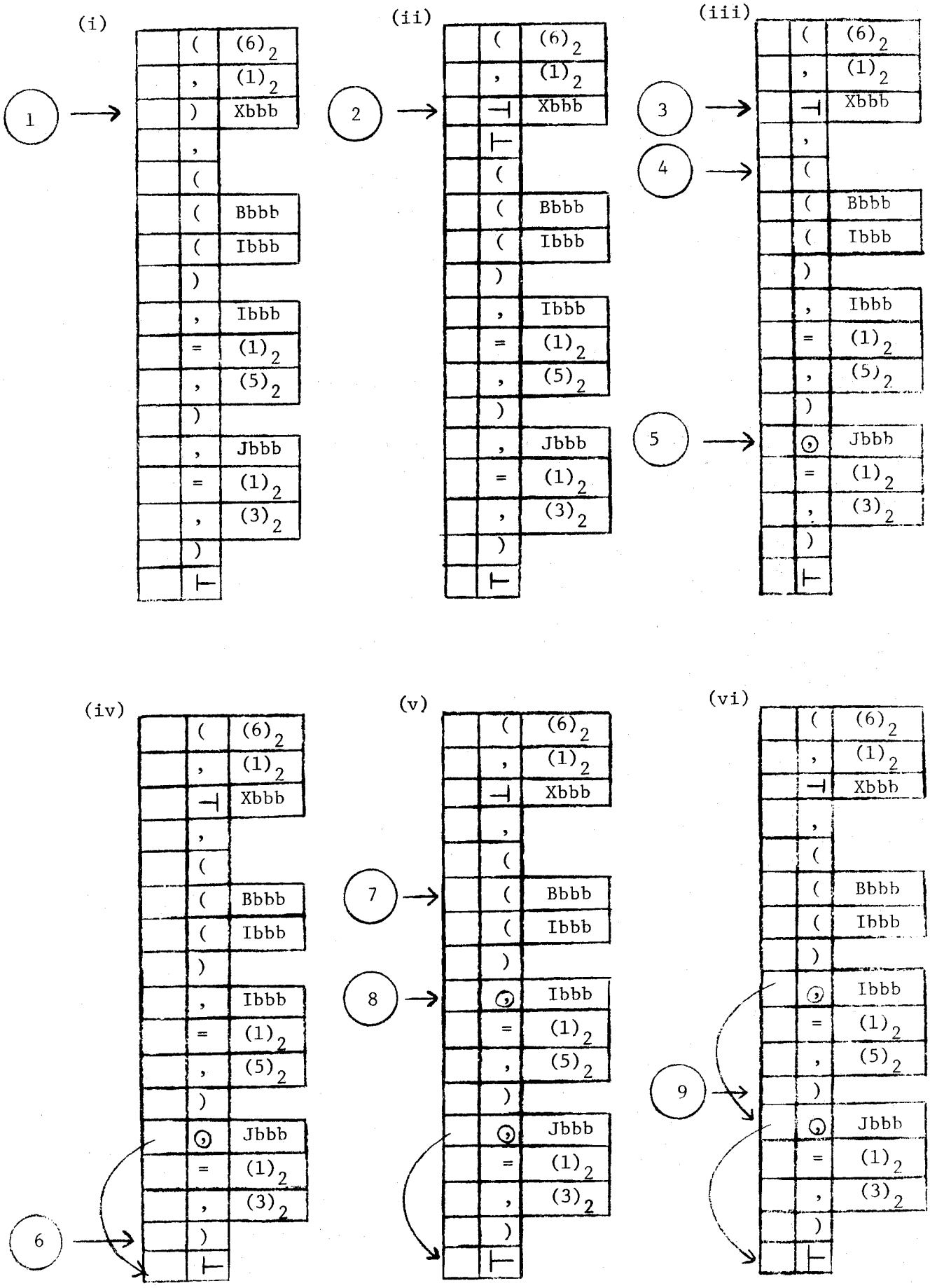
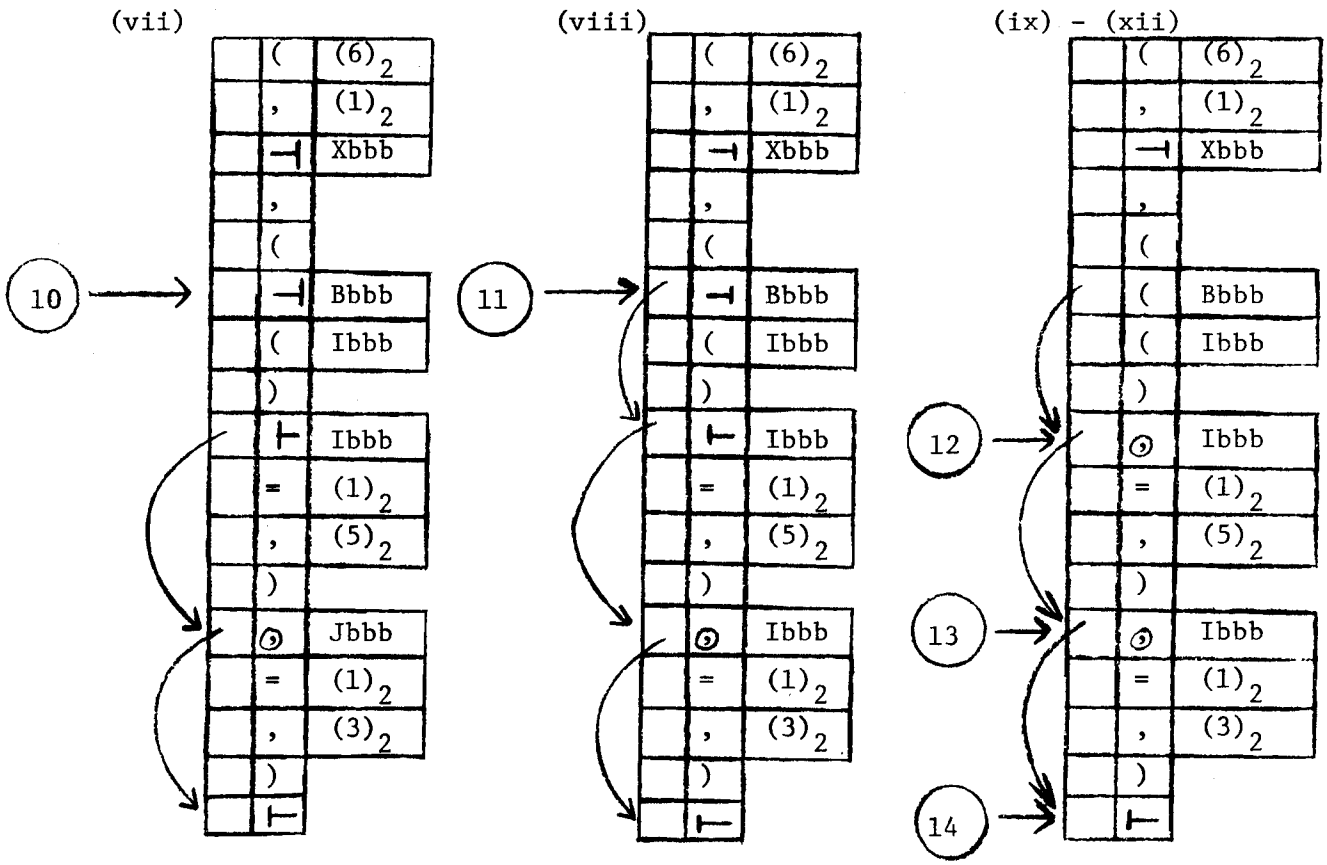

(i)
(ii)
(iii)
(iv)
(v)
(vi)
1
2
3
4
5
6
7
8
9
((6)2
, (1)2
) Xbbb
, (
(Bbbb
(Ibbb
)
, Ibbb
= (1)2
, (5)2
)
, Jbbb
= (1)2
, (3)2
)
⊢
⊣ Xbbb
⊙ Ibbb
⊙ Jbbb

Figure 4.6.5.

Figure 4.6.5. (Continued)



Object Code

1. BAL R14,XIOINIT
   DC
   DC
   DC
   DC

2. BAL R14,XSIMPELT
   DC AL1(r),AL1(t),AL2(X*)

3. Start-of-DO coding for J

4. Start-of-DO coding for I

5. Coding to put A(B(I)-START) in R3

6. BAL R14,XSUBSELT
   DC AL1(r),AL1(t),AL2(0)

7. End-of-DO code for I

8. End-of-DO code for J

9. BAL R14,XENDLIST

Upon entry, R1 points to the stack where a unit operand should be. The stack code is checked for being variable or constant operand and one of two routines is used. A variable unit is looked up in the symbol table and checked as a non-ASSIGNed, non-DIMENSIONed integer variable. Bits are set in OP for COMMONed/EQUIVALENCEd and half-word variables and unit* is set in the skeleton code before returning via R11. A constant is looked up by a call to the integer constant collector COLINTGR and is checked for exceeding &NOUTILS before unit* is set and return taken via R11.

3. END/ERR Processor

This routine checks for the presence of END= or ERR= parameters in the stack, sets up the skeleton object code and advances the stack pointer R1 if so.

The routine is called by a BAL R11,IOENDERR and the routine returns to 0(R11) if input delimiter is ')' or if valid END= or ERR= parameters are followed by ')'; the return is to 4(R11) if the input delimiter was ',' and no END/ERR parameters are present or are followed by ','.

The processing is most simply described by Figure 4.6.6.

The routine DCSTN2 looks up and checks statement numbers as executable and for illegal branching into DO-loops (See section 4.5)

4. Stack Fix-Up

The purpose of this routine is to modify the stack configuration for a statement of the form READ1 to look as if statement READ(5,1) were being compiled. Similarly for PRINT, PUNCH. An example of this process was given in section 4.6 above.

5. Error Message Routine

There are 8 different ERROR macros to call the error message editor. They are included here as callable routines to save storage by reducing the number of uses of the ERROR macro throughout INOUT.

6. Exit Routine

This routine has 3 entry points and merely serves to reset the switches CDOIO, CTYPESW before returning and to call the DO-compiler to unstack any implied DO-loops uncompleted because of syntax errors in I/O lists

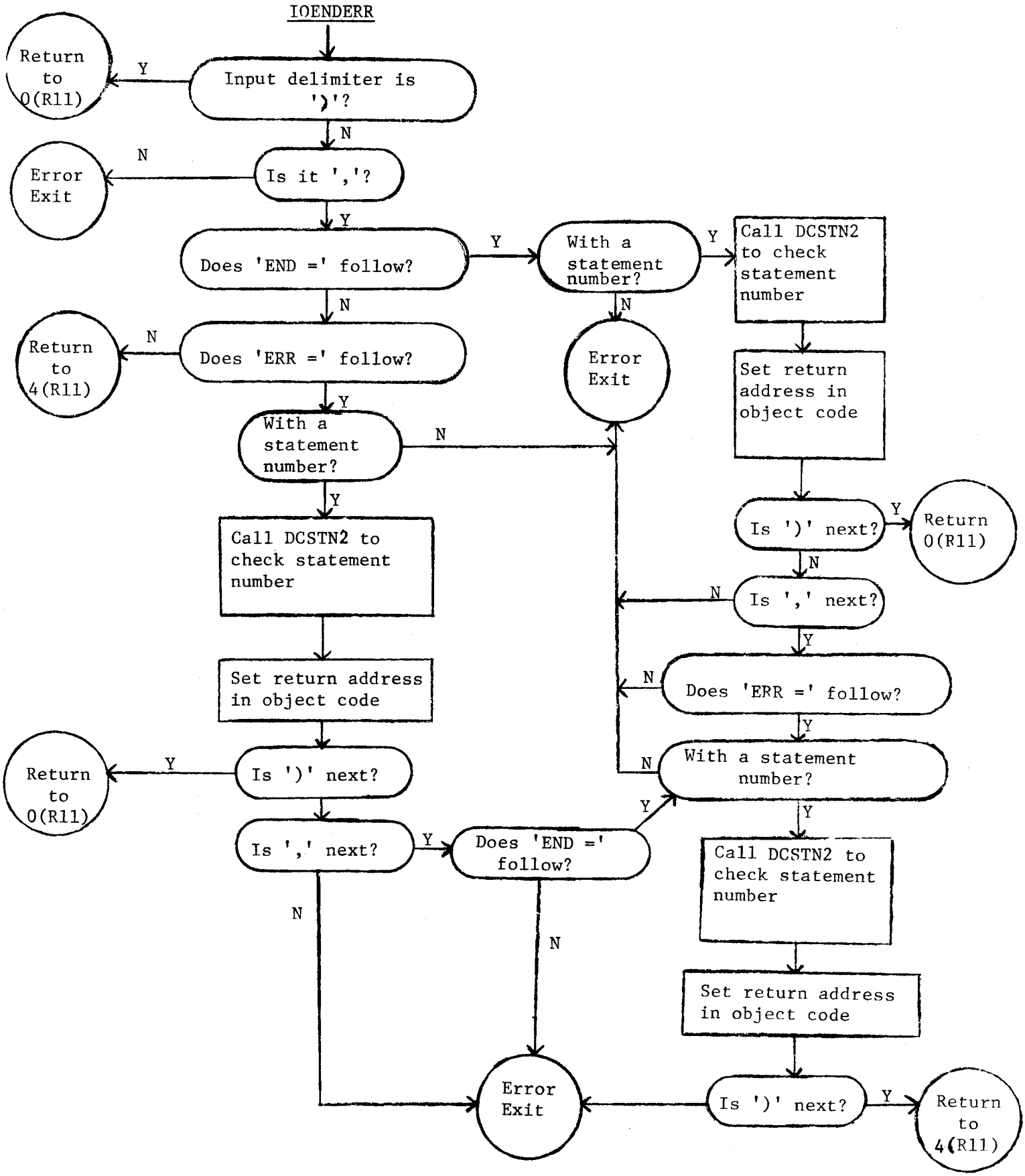

IOENDERR
Input delimiter is ')'?
Y
Return to 0(R11)
N
Is it ','?
N
Error Exit
Y
Does 'END =' follow?
Y
With a statement number?
Y
Call DCSTN2 to check statement number
N
Error Exit
Set return address in object code
N
Does 'ERR =' follow?
N
Return to 4(R11)
Y
With a statement number?
N
Y
Call DCSTN2 to check statement number
Set return address in object code
Is ')' next?
Y
Return 0(R11)
N
Is ',' next?
N
Y
Does 'ERR =' follow?
N
Y
With a statement number?
N
Y
Is ')' next?
Y
Return to 0(R11)
Is ',' next?
Y
Does 'END =' follow?
Y
Call DCSTN2 to check statement number
N
N
Set return address in object code
Error Exit
Is ')' next?
Y
Return to 4(R11)

Figure 4.6.6.

e.g. READ(5,1)((X(I,J) + I = 1,5), J=1,5)

Start-of-DO coding will have been output before the invalid '+' which terminates compilation is discovered. The DO-loop must be unstacked so that spurious DO-4 (redefinition of DO-par within range) errors will not be issued in subsequent statements.

4.7 FORMAT

4.7.1. Introduction

The routine FORMAT processes both compile and execution time FORMAT lists. This routine is the only statement processor that uses the actual statement as input rather than having SCAN transform the statement. The input list of specifications is transformed into a list of codes which will be interpreted by the routine FORMCONV at execution time.

4.7.2. Compile-Time Entry (FORMAT)

At compile time the input list is found in the STACK by SCAN and the output list is placed in-line in the OBJECT code area.

If the OBJECT area is not full, enough space is left to output a branch around the list of transformed specifications. SCAN has previously set up a pointer to the statement number (CSTNOLK) in the symbol table and this is now updated to point to the list of specifications. Control now passes to the routine (FORSCAN) to transform the list. Two returns are possible from this routine. The first indicates that an error was encountered in the FORMAT statement. An error code is inserted as the first entry of the transformed list. The branch around instruction is inserted and control returns to SCAN. The second return indicates that no errors were encountered. The branch around instruction is inserted and control returns to SCAN.

4.7.3. Execution-Time Entry (FORMATEX)

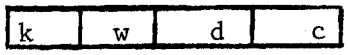
The input list is, of course, stored in an array by the user. The stack is used as the location to store the output list. The address of the first element of the array is obtained using the routine X1STELT(section 3.15) which also indicates the length of the array. After checking that the first non-blank character is a left bracket, control transfers to FORSCAN.

On an error free return the address of the new list is placed in register 4 and control returns. If an error occurred in the list (the error message has already been issued) control transfers to the "trace back" (section 3.13) routine.

4.7.4. Specification Converter (FORSCAN)

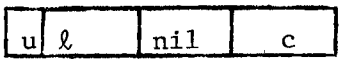
For each specification in the input list a one word coded entry is generated. The codes have the following forms:

(i)



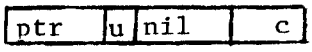
k group count 3F16.2
(set 1 for case F16.2)
w field width 3F16.2
d number of digits after decimal 3F16.2
c code for specification 3F16.2

(ii)

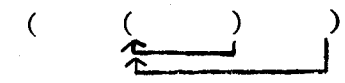


ℓ field count 32X
c code 32X
u unused

(iii)



If c indicates a right bracket a pointer (ptr) is set up to the corresponding left bracket according to the rules given in the FORTRAN manual.

e.g. 

In the case of Hollerith constants the appropriate word is output followed by the Hollerith constant.

The codes are now listed

X'04'	Start of Format list
X'08'	First level '('
X'0C'	Invalid Format
X'10'	P scaling
X'14'	F Format
X'18'	E Format
X'1C'	D Format
X'20'	I Format
X'24'	T Format
X'28'	A Format
X'2C'	L Format
X'30'	X Format
X'34'	H Format or ' '
X'38'	First level ')'
X'3C'	/ format
X'40'	G Format
X'44'	End of list
X'48'	Z format
X'4C'	Second level ')'
X'50'	Second level '('

On entry to FORSCAN it is assumed that Register 4 contains the address of the input list and Register 6 the address of the output area. These have been set up by FORMAT or FORMATEX. The initial entry is placed in the output list and now we can proceed to scan the list of specifications.

A TRT instruction is used in conjunction with a table (FTPAN). This table has an entry of zero for 'blank', one for invalid characters, (both bad punches, e.g. (≠) and characters not allowed as specifications in a format list (e.g. B)) and entries 2 - 27 for the various valid characters. A set of routines is used to process each type of character. Before describing these, the routines FGETSW and FCOLL, used for finding and collecting constants, and the switch FSW will be described.

FCOLL (BAL R15, FCOLL)

This routine is given a pointer to a digit in register 1 and proceeds to scan the list until a non-digit is encountered. The collected number is converted, stored in register zero, and also, in the output list as a possible field count. Control returns after checking that the constant is not greater than 255. If it is greater an error message is issued.

FGETWD (BAL R14, FGETWD)

This routine determines if a constant is of the form w or w.d. (I22 or F16.8) and returns to 0(R14) or 4(R14) respectively. It uses the routine FCOLL to collect the required constant.

FSW

This is a one byte switch with settings used to determine various error conditions

X'01'	We have already collected a constant e.g. 5 PE16.8
X'02'	We have a comma
X'04'	We have completed collecting a specification and are now ready to start on a new specification

This switch is tested and set by the various character routines.

Processing of Characters

The following routines process the various types of specifications. Each one inserts the appropriate code described above in the entry in the object code.

FNUMA (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

After calling FCOLL the constant is stored in the list as a field count.

FALFA (A, D, E, F, G, I, L, T, Z)

After possibly having changed a field count, stored by FNUMA to a group count, the code is inserted in the output area. FGETWD is then called and upon return the constant and specification type are checked to see if a valid specification has been obtained.

FMINUS (-)

Since a minus must be followed by a constant FCOLL is called. It is then determined if the next character is P. (If not an error message is issued). Control now transfers to FP1.

FCOMMA (,)

Check if last character was a comma and if so issue an error.

FPALIGN (P)

Did a number precede and if not issue an error.

FERROR (invalid character)

Issue an error message.

FLT1 (left bracket)

The bracket count (FBCNT) was initialized upon entry to FORSCAN to four. FLT1 increases this count by four and depending on the count being 8 or 12 saves the current address of the output list in FSAVE1 or FSAVE2 respectively.

FRT1 (right bracket)

The bracket count is decreased by four. If it is zero the end of the list has been encountered and control transfers to FDONE. If non-zero the appropriate pointer is inserted from FSAVE1 or FSAVE2.

FHOLL1A (H type holleriths)

FQUOTEA (Quote type Holleriths)

The Hollerith constant is placed in the output area. Register zero contains the length of the constant. The length is inserted in the list at FQDONE and control returns to process the next specification.

Errors (FERROUT)

```
BALR 14,9
DC   (error code)
```

Register 9 contains the address of the error processor. The error message is issued in the standard manner and upon return a check is made to determine if it is a compile or execution time error. If execution, FORSCAN returns (error return 0(R14) and if compile time the object code register is re-aligned to a full word boundary and control then returns to FORMAT via the error return.

4.8 Relocator

4.8.1. Introduction

The relocator can be thought of as a loader or "tidy-up" phase for a programme segment and for the entire programme. It is invoked each time an END card is encountered by SCAN and also for a final time when the $ENTRY card (or equivalent) is read.

The relocator deck of WATFOR is partitioned into three phases which perform the following functions. Phase one scans the various symbol tables generated by the statement processors and assigns storage for the entries in the symbol table. Phase two passes over the object code generated by the statement processors and changes symbol table pointers into the addresses of the storage assigned by phase one. Phases one and two are invoked by the compiler at the end of every programme segment. Phase three processes the relocator phase three variable list generated by phase one. This is a list of the uncompleted tasks of phase one. It consists of the assignment of storage for common, dimensioned and equivalenced variables and the linking of external address constants with their entry points. In this last respect, its operation corresponds to that of the LINKAGE EDITOR.

4.8.2. Phase One

Phase one of the relocator assigns storage positions in the data area of the subprogramme. The data area has two sections; first, storage for a save area and temporaries which are addressed by register R13 at execution time. The rest of the storage area (addressed by registers R5 through R10) is devoted to:

1. Simple variables not appearing in common or equivalence statements.
2. Address constants pointing to simple variables appearing in common and equivalence statements.
3. Dimensioning routines for dimensioned variables.
4. Address constants of subroutines/functions used.
5. Save areas and temporary areas for arithmetic statement functions.
6. Integer and floating-point constants.
7. Address of the Hollerith constants used in the programme.
8. Addresses of statements branched to in the programme (including pseudo branch points generated by DO statements).

The Flow Of Logic Of Phase One Is As Follows:

1. Phase one assigns storage for the main save area and the main temporary areas, and notes these addresses for later use by phase two.

2. If we are compiling a function subprogramme, the function list is concatenated onto the front of the variable list. Remember that the function list is a list of the entry point names of the function. Corresponding function variable list entries already exist in the variable list. Hence there are now two entries in the variable list for each function entry. Note also, the first entry in the variable list is that of the main function entry point. This is required in the processing of the relocator phase three variable list and will be explained in more detail at that time.
3. The variable list is scanned and the following processing is done on various elements.

   (a) Dimensioned Variables.
   A dimensioning routine is generated in the data area from information supplied by the dimension list of the entry. The base/displacement (B/D) address of this routine is stored over the variable name and the element placed in the relocator phase three variable list for phase three storage allocation and patch up. If the array is equivalenced, we transfer to the equivalence list processor.

   (b) Simple Variables, Not Commoned or Equivalenced.
   Storage in the data area is assigned according to the type and mode of the variable and the B/D address is stored in the primary address field of the entry. The undefined quantity is placed in the storage allocated.

   (c) Simple Variables, Commoned or Equivalenced.
   A word is set aside in the data area for the address of the storage to be assigned by phase three. If the variable is equivalenced we transfer to the equivalence list processor.

   (d) Equivalence Lists (See Figure 4.8.1.)
   If an equivalenced variable is encountered in the VLIST, control is passed to RVGENEQL <u>after</u> the normal processing for a simple-equivalenced variable or array name has been completed.

   The equivalence list of which the variable is a member is scanned if the equivalence offset of the variable is negative. The maximum negative offset is found, and each variable is tested to see if any are in common. If any of the variables are in common, no further processing is done on the list.

   After the list has been scanned to find the maximum negative offset it is scanned again, adjusting the offsets so that all are positive or zero. The offsets are checked and a warning message given if equivalencing causes any

Flow Diagram for Equivalence List Processor

In Relocator Phase One

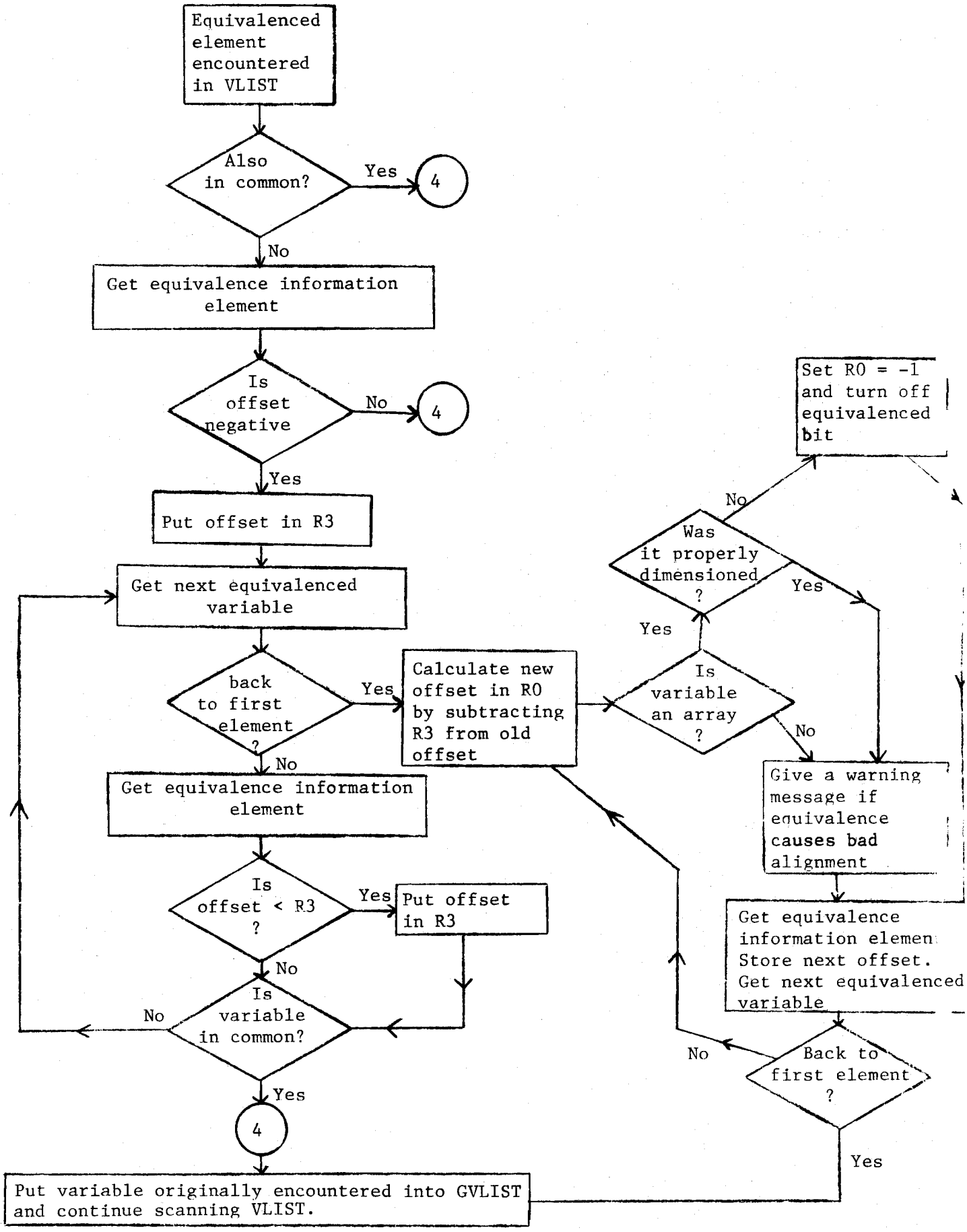


Figure 4.8.1

variable not to be aligned on the proper boundary. Arrays are checked to make sure they have been properly dimensioned, and if not, the offset is set negative as a signal to the equivalence processor in relocator phase three to ignore the variable. The equivalence bit of the variable is also turned off.

After the processing of the equivalence list has been completed, the variable which was originally encountered in the VLIST is added to the GVLIST and the processing of the VLIST continues.

(e) Main Programme Entry.

The programme name is looked up in the library list and entered as a valid entry point name. The entry point address is placed in the library list for phase three processing. Checking is done if the name already appears in the library list (caused by a previous external reference) to see that the type and mode agree. If they do not, this entry takes precedence and an error message is given. The address of the data area (effectively) is stored in this element and the element shifted to the relocator phase three variable list to serve as a marker for the beginning of the sublist for this programme segment.

(f) Multiple Entry Point.

The same processing is performed for this entry as for a main entry point except that the entry is not shifted into the relocator phase three variable list.

(g) External Entry Point.

A word is assigned in the data area for the address of the routine (to be inserted by phase three). Then a library list lookup is performed and a link (to the library entry) is inserted into the variable list entry. The entry is then shifted into the relocator phase three variable list.

(h) ASF Entry Points.

The entry point address is moved into the data area and its relocated address stored in the variable list entry. A save area is created in the data area, and its address stored also in the entry. Storage for temporaries for this entry is assigned and the offset of the floating point temporaries is stored in the variable list entry. All this is done as a preliminary to the phase two relocation of the object code. Note that a full save area need not be generated, as the base registers R5 to R10 for the data area do not need to be reloaded at execution time, when the ASF is called.

(i) Common Block Names. (See Figure 4.8.2.)

If a common block name is encountered in the VLIST, control is passed to RVGENCBL.

RVGENCBL scans the list of variables in common and assigns common offsets for them. The offset of the first variable is zero. The offset of any variable in the list is equal to the offset of the variable preceding it plus the length of the variable preceding it. A warning message is given for initializing a variable in blank common. A warning is also given if the offset of a variable indicates that it will not be aligned on the proper boundary.

If a variable in the common list is equivalenced the equivalence list of which it is a member is added into the common list being processed; see figure 4.8.3. The equivalence offsets of the variables are adjusted to make them common offsets. Error messages are given for:

1. Variables which were subscripted in an equivalence statement but not dimensioned.
2. Variables in the equivalence list which are also in common.
3. Variables which have negative common offsets. (i.e. extending common downwards with equivalence.)

Warning messages are given for initializing blank common and for bad alignment.

RVGEN6MX is set equal to the common offset of the first byte of storage not used by any element in the equivalence list. (This will be the length of the common block if the equivalence list extends past the end of the common list.) When all elements of the equivalence list have been processed, the processing of the common list continues.

When all elements of the common list have been processed, the length of the common block is calculated. The common block name is looked up in the LLIST. The common block length is stored in the LLIST entry if the name had not occurred before or if the length is longer than the previous value. The offset address of the LLIST entry is stored in the VLIST entry.

An error message is given if the common block name was previously used in a different context. A warning message is given if the length is not the same as the previous value.

The common block name is put in the GVLIST and the processing of the VLIST continues.

Common Block Processor in Relocator Phase One

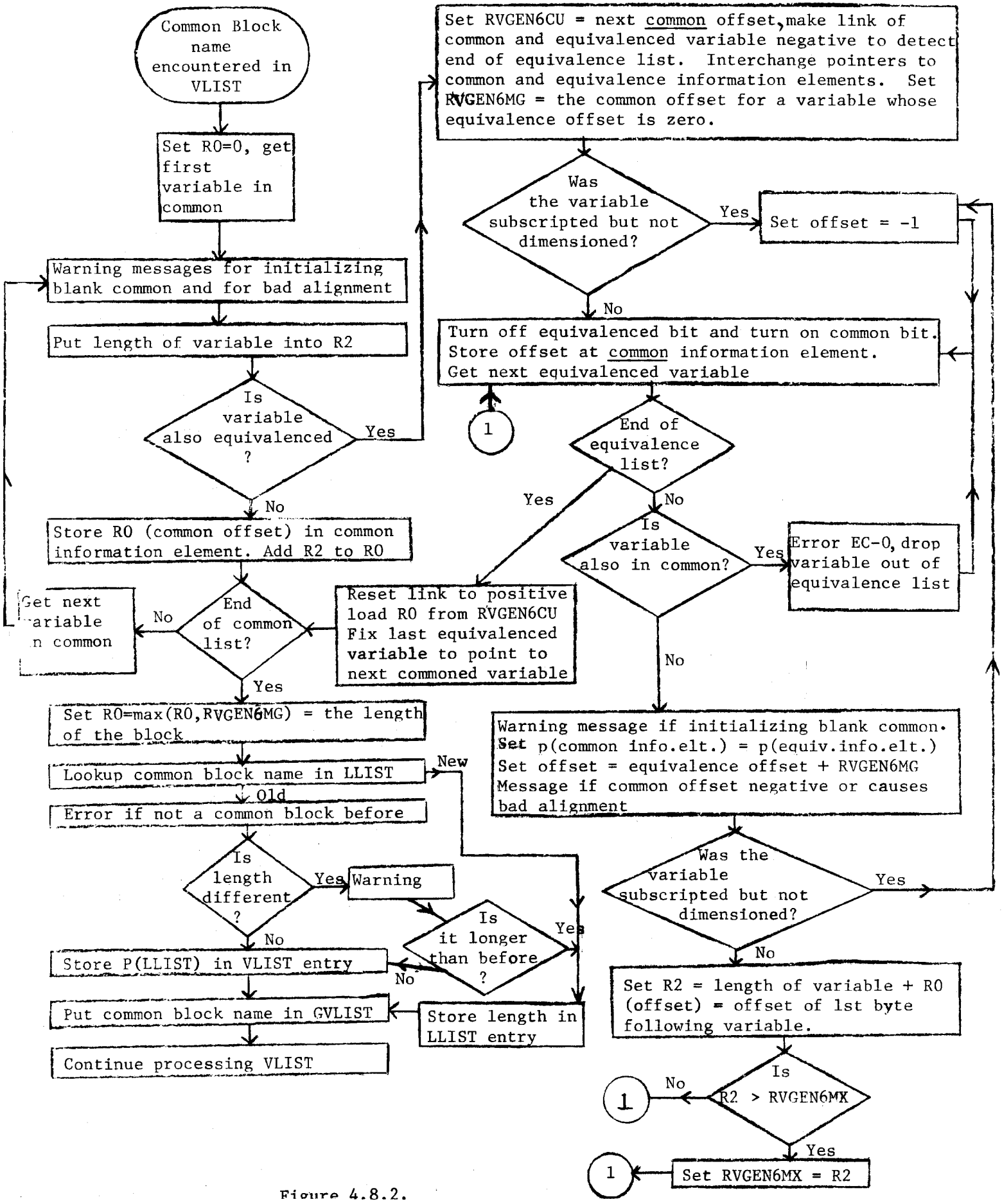


Figure 4.8.2.

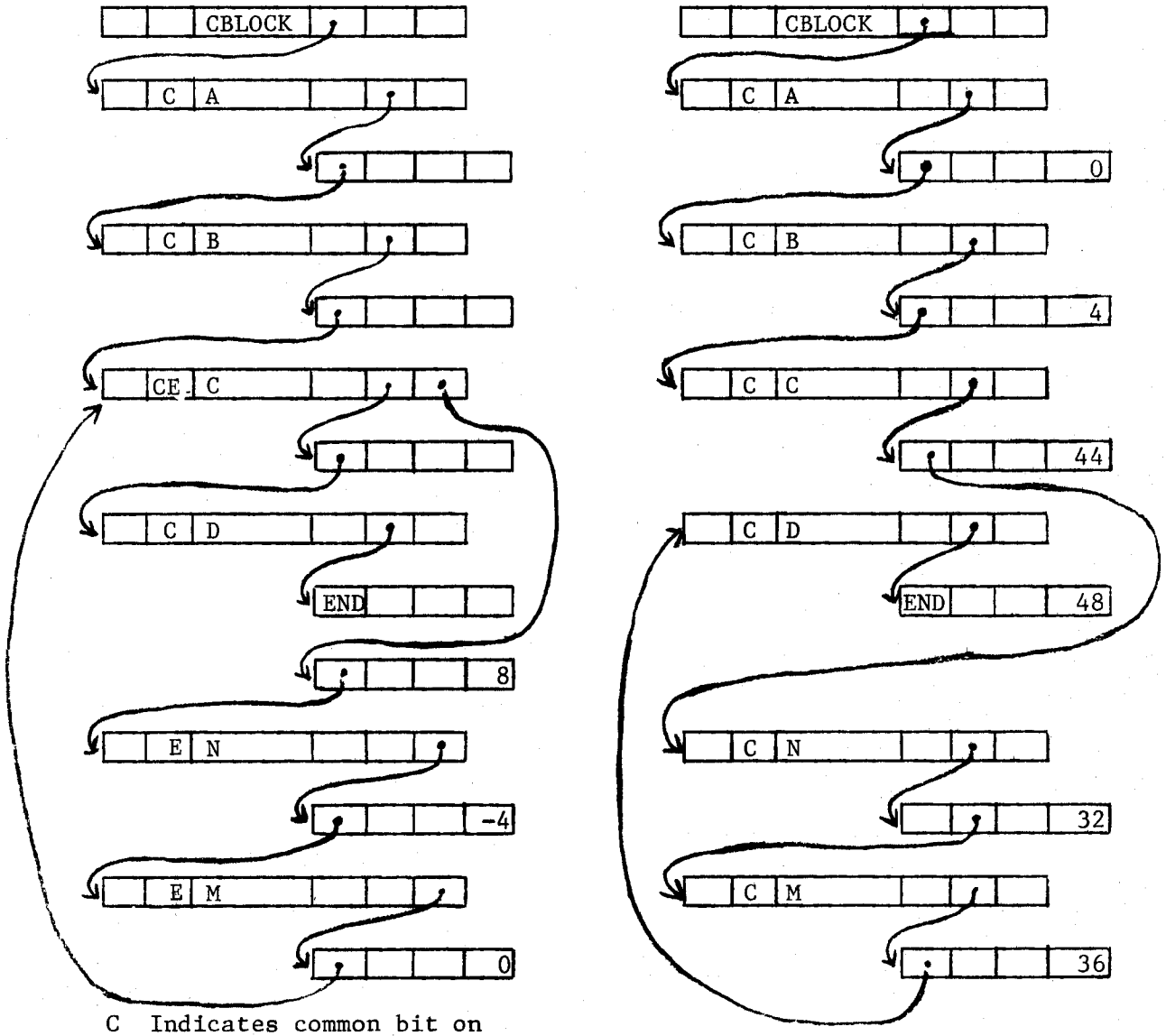
Example of a Combined Common-Equivalence List

Fortran Statements:

```
DIMENSION N(10), M(5)
COMMON/CBLOCK/A,B(10)C,D
EQUIVALENCE (M(3), C, N(4))
```

VLIST Before Entering RVGENCBL

VLIST After RVGENCBL



C Indicates common bit on
E Indicates equivalence bit on

Length of common block = 72 (bytes) because N extends past D.

Figure 4.8.3.

(j) Subroutine Parameters.

These entries are treated in the same manner as other variables except that the entry is not shifted into the relocator phase three variable list.

4. The constant list is scanned and the constants moved down into the data area. For constants of length eight and 16 bytes, two relocated addresses are stored into the constant list entry, one for the first and second half of the constants (i.e. the real and complex parts). Constants of length eight are aligned on a double word boundary but are also treated as complex, single word entries since they may be both. e.g. The constants 1.D0 and (1.,0.) require only one constant list entry as they both have the same length and the same bit pattern.
5. The statement number list is scanned, and those executable statement numbers which are referenced, have an address constant generated in the data area. The relocated address is stored in the list entry.
6. The Hollerith constant list is scanned (Hollerith constants are treated as single dimensioned real *4 arrays) and an appropriate dimensioning routine is generated (except for Hollerith constants used in DATA or PAUSE statements and marked by INOUT).

At this point, all storage for the data area has been allocated and a check is made that this area has not overflowed the maximum permitted (6 pages = 24K bytes) before control falls through to phase two.

4.8.3. Phase Two

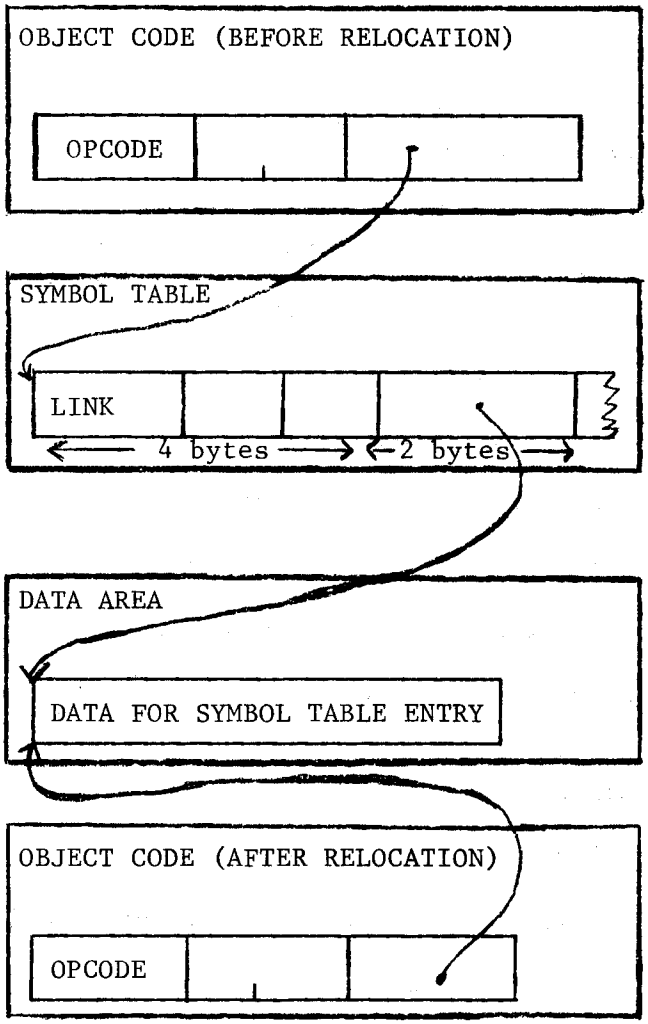
As stated before, the purpose of phase two of the relocator is to pass over the code generated by the various statement processors and to perform such actions as are necessary to change this code into an executable programme. This consists largely of transforming pointers from symbol table pointers to base/displacement addresses in the data area. This transformation was determined by phase one.

The decision process used in passing over an instruction, is two stage. That is, firstly the opcode is interrogated and control is passed to a corresponding routine. These routines fall roughly into four classes.

1. No relocation necessary, skip over instruction.
2. Relocation may be necessary.
3. Additional processing necessary.
4. Pseudo-operations requiring special relocation.

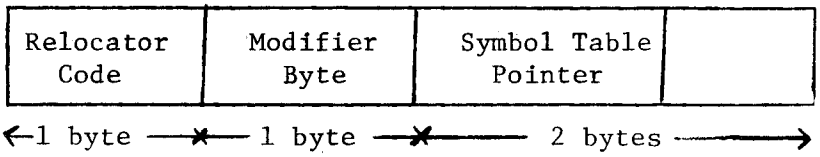
Rather than list all the instructions, in this manual, only a few examples from each class will be given. Class one contains register to register operations, and the load and store multiple instructions. Class two contains most register-indexed instructions. Class three contains instructions such as "BAL" and "BALR" which may change the value of the base address register for the programme (R11). Class four has opcodes which were assigned from the section of unused opcodes (they were assigned as needed and in an arbitrary manner). Their hex values start at X'A0' and increase sequentially.

The second stage of the relocation comes into play at the point when control is passed to the specialized routine. Class two opcodes interrogate the base register of the second operand of the instruction and if it lies in the range 1 to 10 then symbol table relocation is performed. This consists of looking at the symbol table entry pointed to by the second operand and replacing it with the corresponding address pointer. Thus relocation of the bulk of the code is a rather simple matter.



Second stage relocation processing for class three opcodes involves merely noting the new value to which register R11 is set in register R8 and possibly doing the class two relocation as well.

A list of these special opcodes follows along with a brief explanation of the action performed. The majority of these have a constant format (referred to as "normal form"). The first byte is a relocator code. The second byte is a modifier byte used in constructing the high order byte of the address constant which will replace this word. The third part (a half word) is a symbol table pointer.



Relocator Codes

X'A0' - Replace this word with the address of the save area for this subprogramme. Skip over entry point name.

X'A1' - Replace this word with the address of the save area for this arithmetic statement function. The location of the temporary storage area is noted for use in relocating temporaries. Skip over entry point name.

X'A2' - Temporary address (Normal form).

X'A3' - Variable list address (Normal form).

X'A4' - Statement number address (Normal form).

X'A5' - Hollerith constant address (Normal form).

X'A6' - Variable list address (from model argument list) (Normal form).

X'A7' - Statement address (from model argument list) (Normal form).

X'A8' - Subscripted variable (address to be filled in at execution time). (Normal form).

X'A9' - Constant or subscript address (Normal form).

X'AA' - Terminate phase two of the relocator (At end of object code).

X'AB' - Last argument for subroutine model argument list (Normal form).

X'AC' - Last argument for function model argument list (Normal form).

X'AD' - Not used. Reserved for

X'AE' possible expansion.

X'AF' - Omit relocator processing for following code. Change relocator code X'AF' into opcode X'47' (BC).

X'B0' - Internal statement number. Change relocator code X'B0' into opcode X'45' (BAL). Update register R8 to reflect result of BAL instruction on R11 at execution time. Skip over 6 bytes of ISN code.

X'B1' - I/O list entry address (Normal form).

X'B2' - Last argument in calling programme argument list. (Normal form).

X'B3' - Statement number address in I/O list (Normal form).

X'B4' - Address of save area for initial coding in a DATA statement. It is similar in some respects to the relocator code X'AO'. The address is used during the execution of the DATA statements to load registers R5 thru R10 with the address of the data area.

X'B5' - Address of hollerith constant as required by DATA statement at execution time (Normal form).

X'B6' - Indicates end of ASF routine. Restore the temporary pointers to their original condition. Change relocator code X'B6' into opcode X'05' (BALR) and update register R8 to reflect the result of the BALR instruction at execution time.

In addition to the above series of relocator codes, the code X'00' was set aside to indicate that no relocation be done on the word following. In essence, this allows the statement processors to store integer constants of a magnitude $0 \leq K < 2^{24}$ directly in the object code. In addition, this code is used by some execution time routines to indicate kickoff due to an error at compile time. (KO - 0 error message.)

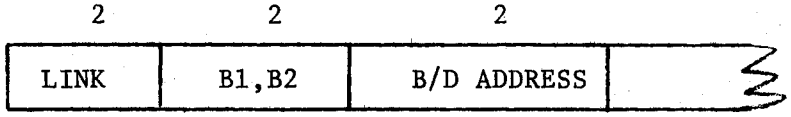
4.8.4. Relocator Phase Three

As stated earlier, the function of Relocator Phase Three is to finish up all the unresolved tasks of Relocator Phase One. This includes the linking up of subroutines and assignment of storage for equivalenced and common variables and for arrays. The processing for members of the global variable list, from which it gets its tasks, is as follows.

1. Subprogramme Header

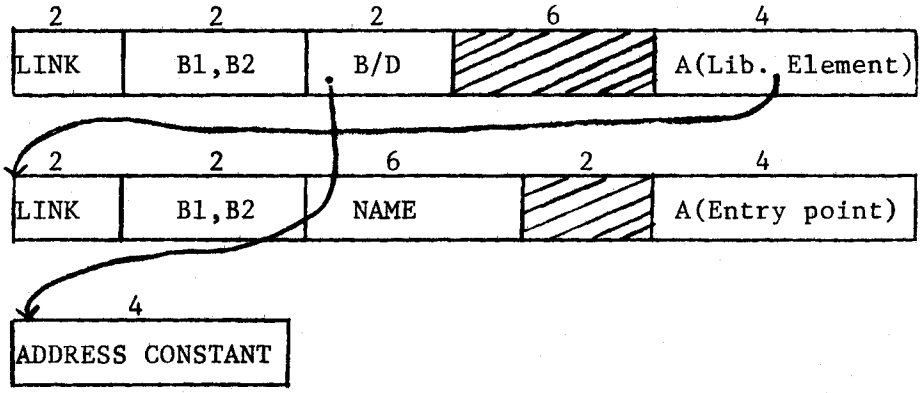
This element, which was constructed from the main entry point element in the variable list, contains the address of the beginning of the data area for the list elements which follow it. This quantity is used in reconstructing the address of the address constants which point to variables whose storage is to be assigned during this phase. Remember that this address constant is pointed to by a halfword base/displacement address located in the third halfword of the list element.

i.e.



2. External References

An external element contains a pointer which points to the corresponding library list element. From the library list element we get the address of the entry point for the subroutine/ function. This is placed in the address constant for this element in the data area of the subprogramme.



3. Simple Arrays

Storage is assigned according to information contained in the dimension routine for this variable. This dimension routine is pointed to in the same manner as all the information in the data area of the subprogramme, by the third halfword of the list element.

4. Equivalence List Processor In Relocator Phase Three (RVGEQL) (See Figure 4.8.4.)

When an equivalenced variable is encountered in the GVLIST, control is passed to RGVEQL to assign storage for the variables in the equivalence list.

R5 is aligned to a double-word boundary, so that the variable(s) with an equivalence offset of zero will fall on a double word boundary.

The variables in the equivalence list are processed as follows:

(a) The equivalence bit is turned off, and the commoned bit turned on. This is done so that storage will not be assigned again if the variable is encountered in the GVLIST.
(b) The address of the word in the data area which should contain the offset address of the variable is calculated using the primary pointer in the GVLIST entry. (See 2.4)
(c) If the equivalence offset is negative, no further processing is done for the variable. [This condition exists only when the variable was equivalenced using subscripts but was never dimensioned.]
(d) The address to be stored in the data area is calculated by adding together the contents of R5, the equivalence offset and (for arrays only) the array base offset. This quantity is stored at the address calculated in (b).
(e) The offset address of the first byte after the variable is calculated by adding together the contents of R5, the equivalence offset and the number of bytes occupied by the variable. If this number is larger than RGVEQLMX, it is stored at RGVEQLMX. After processing all variables in the list, RGVEQLMX will contain the offset address of the next variable storage position.

After all the variables in the equivalence list have been processed, R5 is set equal to RGVEQLMX and the processing of the GVLIST continues.

Equivalence List Processor in Relocator Phase Three (RGVEQL)

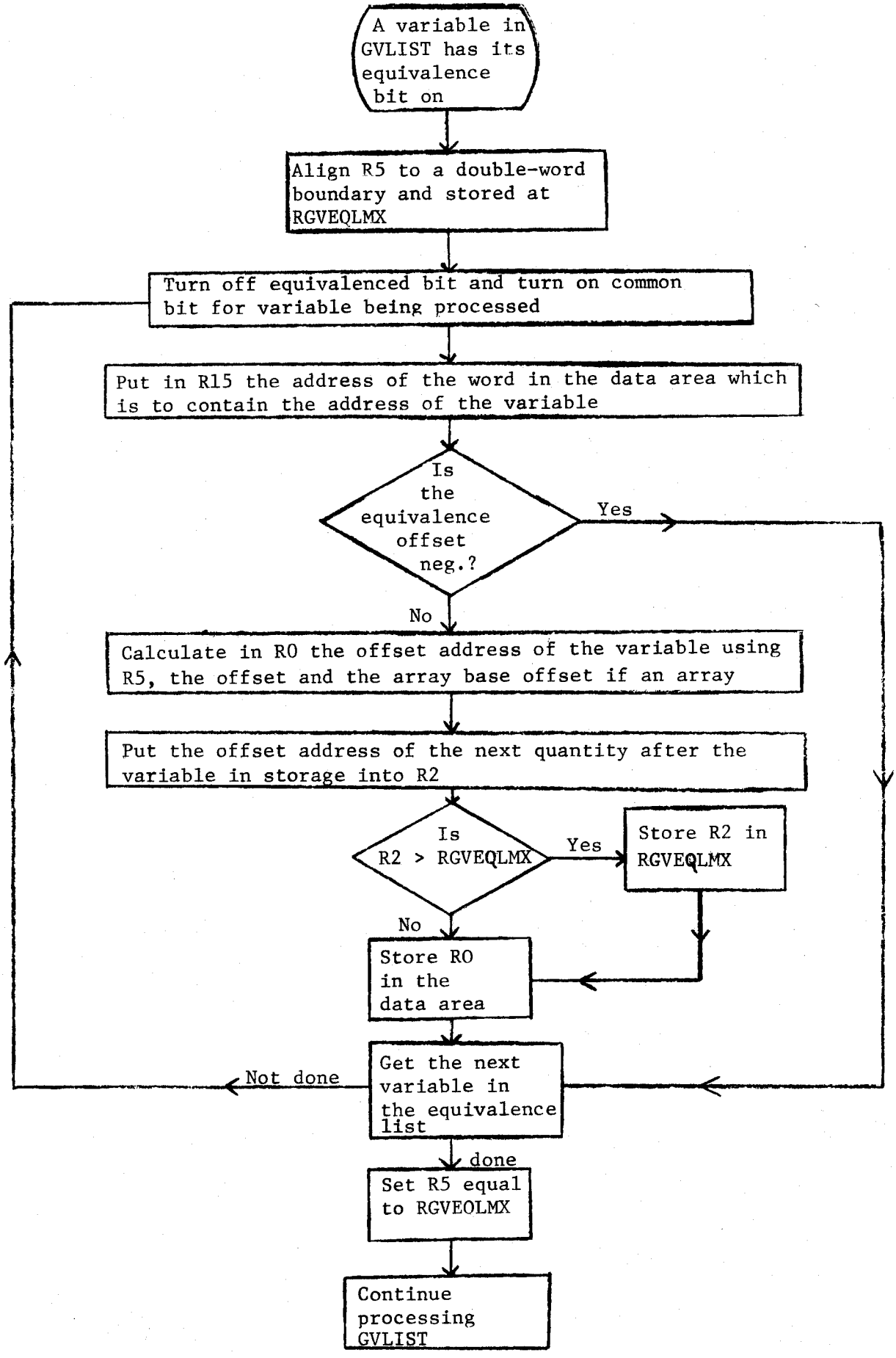


Figure 4.8.4.

5. Common Block Processor in Relocator Phase Three (RGVSC50) (See Figure 4.8.6.)

When a common block name is encountered in the GVLIST, control is passed to RGVSC50.

The entry in the LLIST for the common block name is obtained and checked to see if the address has been assigned. If the address has been assigned, R7 is set equal to this address. If the address has not been assigned, storage is assigned by storing R5 in the address portion of the LLIST entry and adding the length of the common block to R5. R7 is also set equal to the offset address of the common block.

The list of variables in the common block is scanned and the offset address of each variable is calculated by adding R7 to the common offset and array base address for arrays. This address is stored in the proper word in the data area.

After assigning addresses for the variables in the common list, the processing of the GVLIST is continued.

The example below and figure 4.8.6. show the handling of a common block name which occurs in several programme segments.

FORTRAN STATEMENTS:

```
                    .
                    .
                    .
                    COMMON/CBLOCK/A,B
                    .
                    .
                    END
                    SUBROUTINE SUB1
                    .
                    .
                    COMMON/CBLOCK/X,Y,Z
                    .
                    .
                    .
                    END

                    SUBROUTINE SUB2
                    .
                    .
                    COMMON/CBLOCK/M,N
                    .
                    .
                    .
                    END
          $ENTRY
```

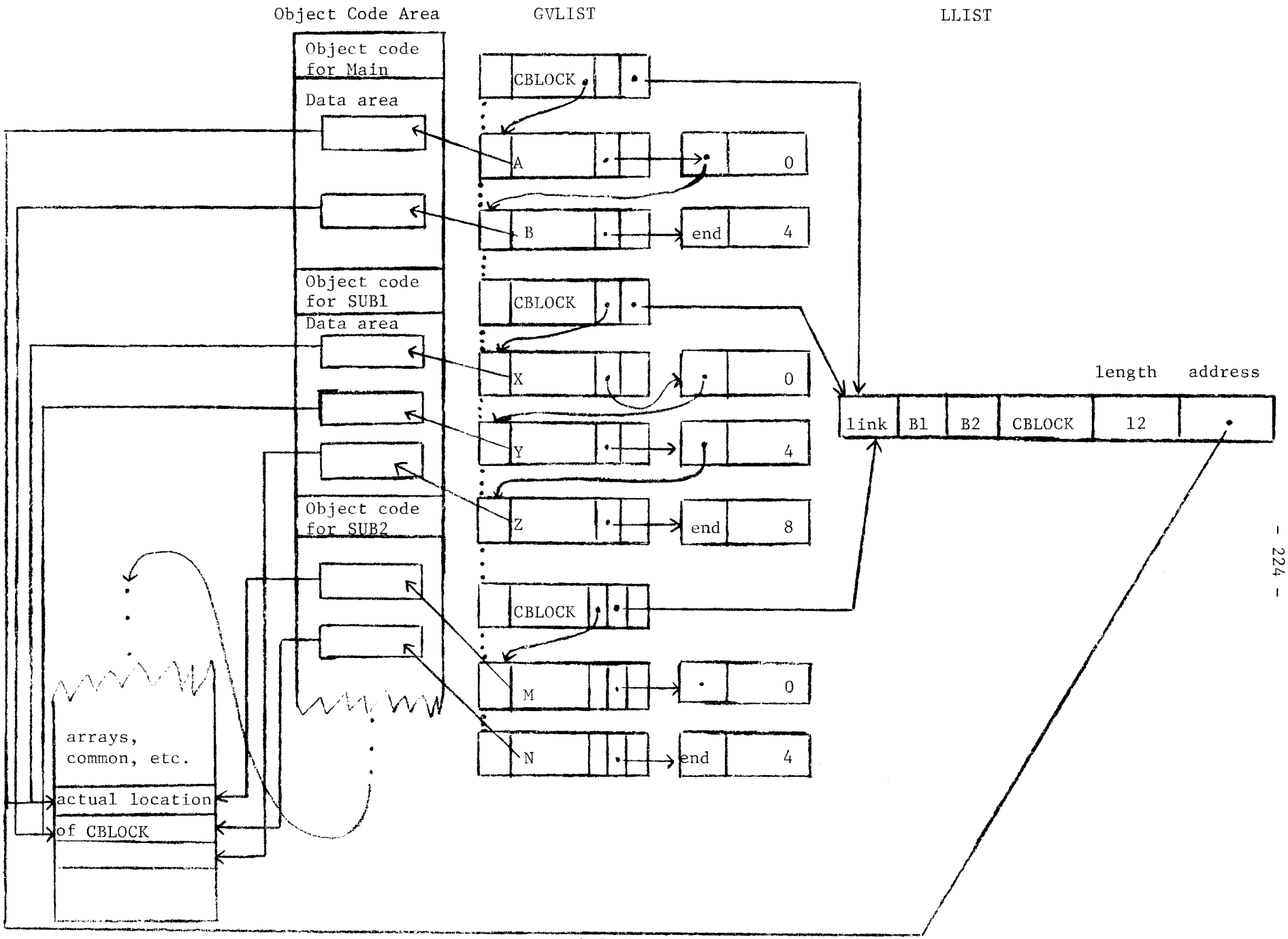

Object Code Area
GVLIST
LLIST
Object code for Main
Data area
Object code for SUB1
Data area
Object code for SUB2
CBLOCK
A
0
B
end
4
CBLOCK
X
0
Y
4
Z
end
8
CBLOCK
M
0
N
end
4
length
address
link
B1
B2
CBLOCK
12
arrays, common, etc.
actual location of CBLOCK

Figure 4.8.5.

Common Block Processor in Relocator Phase Three

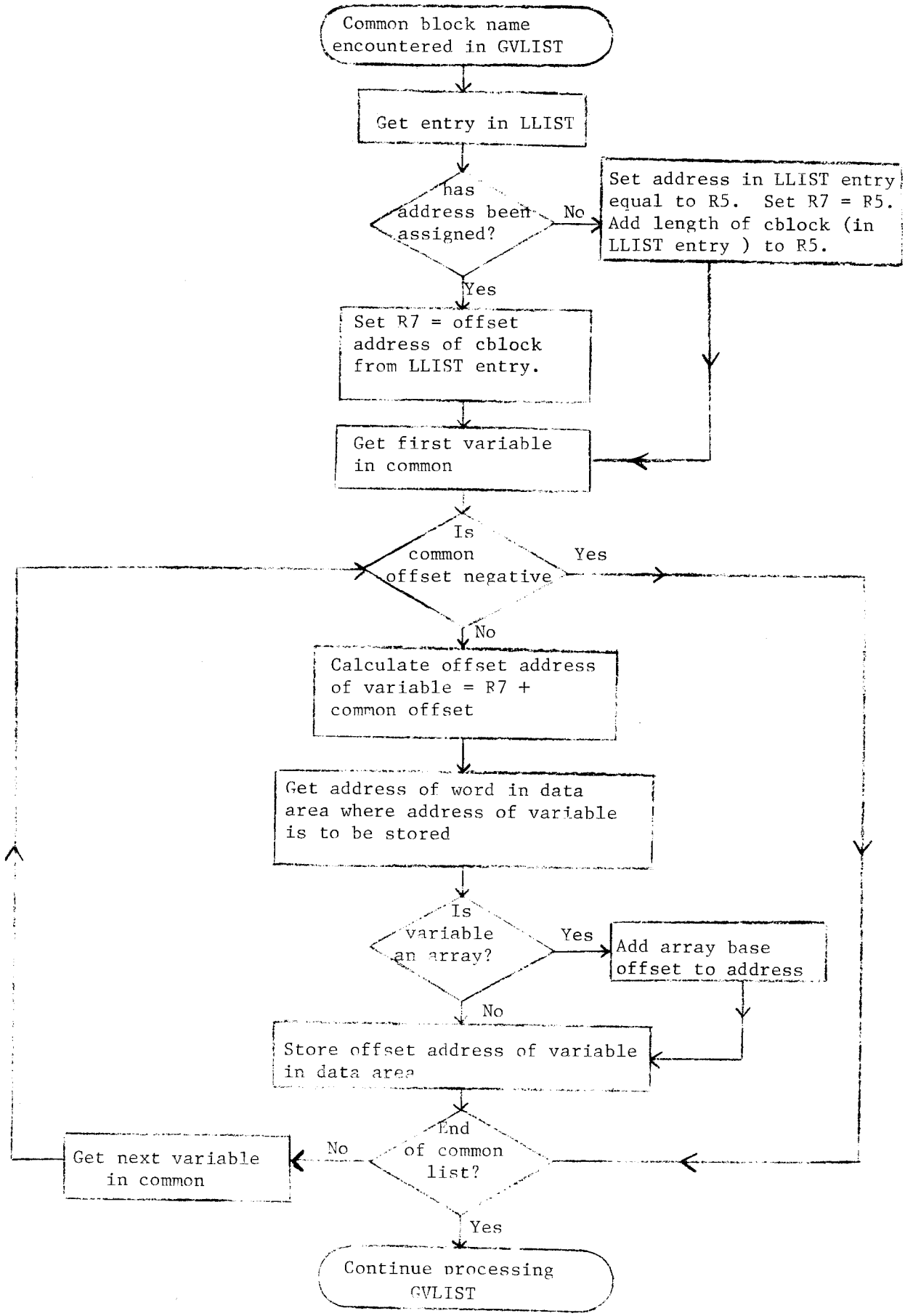


Figure 4.8.6.

Since storage is assigned but not initialized by this stage of the compiler, the addresses of the beginning and the end of this area are stored in locations XBEGDATA and XENDDATA for initialization to the undefined value just prior to execution.

As well, the main entry point of the programme is found and moved into location XENTRYP.

5.1 FUNCTION

The deck FUNCTION contains the in-core function library. Included in the library are the standard FORTRAN mathematical functions (e.g. SIN, EXP) and the functions normally compiled as in-line functions (e.g. ABS, REAL).

5.1.1. THE MATHEMATICAL FUNCTIONS:

The mathematical functions used were obtained from the Release 6 FORTRAN H compiler, and were modified to conform to WATFOR conventions. The entry and returning sequences were modified to use WATFOR's register conventions, and the entry sequence was changed to check the number, mode and definition of the arguments and the mode of the function in the calling programme.

In order to save core space, only the double-precision functions were used and a special entry sequence was written for single precision routines. This sequence is used to call the corresponding double precision routine with the result truncated for the single precision value.

A tree structure was drawn, indicating which routines are called by other routines (See Figure 5.1.1.). Those routines which do not call any further routines are called level-0 routines. Routines which call only level-0 routines are called level-1 routines, etc. The highest level is two.

Two 19-word save areas, XSAVEFN1 and XSAVEFN2, are set up in FUNCTION. Level-2 functions store their registers in the calling programme's save area and use XSAVEFN2 as their save area. Level-1 functions store their registers in the calling programme's save area and use XSAVEFN1 as their save area. Level-0 functions do not restore the registers they use except for F6. [F6 must always contain zeros in the low-order half. See section 4.4.]. The other registers used by the level-0 routines are F0, F2, F4 and R0-R4 (depending upon the routine).

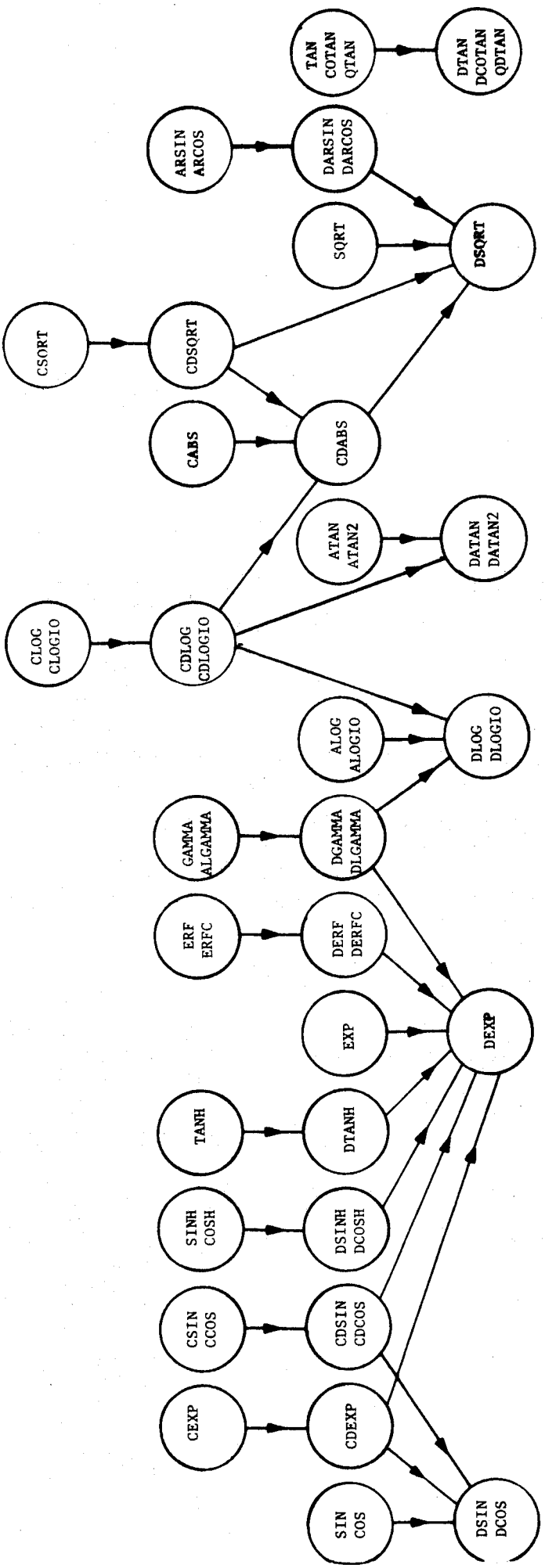


Figure 5.1.1.

Function Dependency Table

5.1.2. Macros Used By Function:

(a) NAME FENT (ARG1,MODE1),(ARG2,MODE2), ..., LEVEL

FENT is used to generate a WATFOR-like entry sequence for level-1 and level-2 functions. The entry sequence is:

```
NAME    STM     14,12,12(13)              store registers
        BAL     R11,XENTSPEC              go to argument passing routine
        USING   *,R11
        DC      A(XSAVEFN1(or 2)-XTART)   save area
        DC      CL6'NAME'
        DC      H'0'
        ARG     ARG1,MODE1                see ARG
        ARG     ARG2,MODE2                see ARG
         .
         .
         .
        LARG    NAME,MODE1                see LARG
        CLC     XUNDEF(LMODE1),ARG1       is argument 1 undefined?
        EX      0,XNOPDEFN
         .
         .
         .
        CLC     XUNDEF(LMODEn),ARGn
        EX      0,XNOPDEFN
```

Control is passed to XENTSPEC to pass the arguments to the function from the calling programme. XENTSPEC is an entry point in the routine XENT and does the same argument-checking as XENT except that it does not check for recursive calls. R11 is set up as the base register for the function, and the arguments are passed (by value) to the locations ARG1,ARG2,..., etc. After returning from XENTSPEC the arguments passed are compared with XUNDEF to make sure that they are defined. FENT assumes that the function has the same mode as the first argument.

(b) ARG ADDR,MODE

ARG is used to generate an argument in the entry sequence in FENT. The word generated indicates that the argument is a simple variable of mode MODE, called by value, and to be stored at ADDR.

(c) LARG ADDR,MODE

LARG is used to generate the last argument in the entry sequence in FENT. The word generated indicates that the function has the mode MODE, and the entry point is at ADDR.

(d) FCALL (NAME,MODE),(ARG1,MODE1),(ARG2,MODE2), ...

FCALL generates a call from a level-1 or -2 function to another function. The code generated is essentially the same as the code generated for a call in a WATFOR programme.

The calling sequence is:

```
                LA      R14,RETURN
                L       R3,ADDRFUNC
                BAL     R1,START(R3)
                ARG     ARG1,MODE1
                ARG     ARG2,MODE2
                .
                .
                .
ADDRFUNC        LARG    NAME,MODE
RETURN          ...
```

(e) FRET

FRET is used to generate a return from a level-1 or -2 function. It generates a 'B XFRET' instruction. XFRET restores the calling programme's registers and returns to the calling programme.

(f) CALLDUB DNAME,LEVEL,TWO

CALLDUB generates the special sequence for a REAL*4 routine to call the corresponding REAL*8 routine.

The sequence is:

```
NAME    BAL     R15,XCALLDUB
        DC      CL6'NAME',H'0'
        DC      AL1(NARGS-1),AL3(DNAME-XTART)
```

where NARGS is the number of arguments and is assumed to be one unless the argument TWO is present.

The routine XCALLDUB is in FUNCTION. The arguments are checked to make sure they are REAL*4 and are defined. The values of the arguments are then placed in double-word locations which have been initialized to zero.

R1 is set to point to an argument list which indicates the new locations of the arguments and has REAL*8 as their mode and the mode of the function being called. Control is passed to the routine DNAME as though it had been called directly from a WATFOR programme.

(g) NAME CALLDC DNAME,LEVEL

CALLDC generates the special sequence for a COMPLEX*8 routine to call the corresponding COMPLEX*16 routine. The sequence generated is:

```
NAME    BAL     R15,XCALLDC
        DC      CL6'NAME',H'0'
        DC      AL4(DNAME-XTART)
```

The routine XCALLDC is in FUNCTION. It assumes that the COMPLEX*8 routine has only one argument. The argument is checked to make sure that it is COMPLEX*8 and is defined. The values of the real and imaginary parts of the argument are placed in consecutive double-word locations which have been initialized to zero.

R1 is set up to point to an argument list which indicates the new location of the argument and has COMPLEX*16 as its mode and the mode of the function being called. Control is then passed to the routine DNAME as though it had been called directly from a WATFOR programme.

(h) NAME FENTZ

FENTZ is used as the entry sequence for a level-0 routine. The code generated is:

```
NAME    BAL     R15,START+10(R3)
        USING   *,R15
        DC      CL6'NAME'
```

The result of the BAL instruction is simply to branch around the name leaving R15 set up as a base register (which also points to the name of the routine), since upon entry R3 contains A(NAME-START).

(i) CHCKA ADDR,MODE

CHCKA is used by level-0 routines to check the mode of an argument in a list. ADDR indicates the base displacement address of the argument in the calling sequence. MODE is the mode expected for the argument.

The code generated compares the argument in the list with a simple variable of mode MODE. If not equal, error message SR-4 is issued and execution is terminated.

(j) CHCKL ADDR,MODE

CHCKL is used by level-0 functions to make sure the argument in the list at ADDR is the last argument of a function with mode MODE. If this is not so, error message SR-7 or SR-2 is issued and execution is terminated.

(k) CHCKLSR ADDR

CHCKLSR is used by level-0 subroutines to make sure the argument in the list at ADDR is the last argument of a subroutine. If this is not so, error message SR-7 or SR-2 is issued and execution is terminated.

(l) CHCKDEF ADDR,MODE

CHCKDEF is used by level-0 routines to check if arguments are defined. The LMODE[1]bytes located at ADDR are compared with X'8080...80'. If equal, the variable was undefined, so message UV-5 is issued and execution is terminated for RUN = CHECK.
CHCKDEF2, CHCKDEF3 and CHCKDEF4 are variations of CHCKDEF.

(m) FOMIT NAME

FOMIT is used to omit routines which are not required by the installation. If the function NAME is in the list of functions to be omitted (set up by the FUNCOMIT macro - see Implementation Guide), the instruction 'ORG NAME' is generated.

5.1.3. THE 'IN-LINE' FUNCTIONS

The functions handled by most FORTRAN compilers as in-line functions are actual routines in WATFOR. They are all level-0 functions.

As well as performing the desired operations, they check the number, mode and definition of the arguments.

---

1. LMODE. The mode concatenated with L (L integer 4 is equated to 4)

Following is a description of the library functions added or modified by the WATFOR group.

1. AMAXO, AMINO

AMAXO and AMINO are real*4 functions having two or more integer*4 arguments. A branch instruction is set to branch not low for AMAXO or branch not high for AMINO. The same routine is then used for both functions to find the maximum or minimum of the arguments. The result is placed in R1 and the routine XFLOAT10 is used to float the integer, placing the result in F0. Control is then returned to the calling programme.

2. AMAX1, AMIN1

AMAX1 and AMIN1 are real*4 functions having two or more real*4 arguments. A branch instruction is set to branch not low for AMAX1 or branch not high for AMIN1. The same routine is then used for both functions to find the maximum or minimum of the arguments. Control is then returned to the calling programme.

3. DMAX1, DMIN1

DMAX1 and DMIN1 are real*8 functions having two or more real*8 arguments. The operation is the same as for AMAX1 and AMIN1 with real*4 instructions changed to real*8 instructions.

4. MAXO, MINO

MAXO and MINO are integer*4 functions with two or more integer*4 arguments. The operation is the same as for AMAX1 and AMIN1 with real*4 instructions changed to integer*4 instructions.

5. MAX1, MIN1

MAX1 and MIN1 are integer*4 functions with two or more real*4 arguments. A branch instruction is set to branch not low for MAX1 and branch not high for MIN1. The same routine is then used for both functions to find the maximum or minimum. The fix routine in the function INT is then used to fix the result in R0. Control is then returned to the calling programme.

6. MOD

MOD is an integer*4 function with two integer*4 arguments.

If ARG1 is positive the value of the function is the remainder resulting when ARG1 is divided by ARG2.

If ARG1 is negative the value of the function is ARG2 + the remainder of ARG1/ARG2.

7. AMOD

AMOD is a real*4 function with two real*4 arguments.

The value of the function is:

ARG1 - [ARG1/ARG2]*ARG2 where [X] is the integer portion of X.

8. DMOD

DMOD is a real*8 function with two real*8 arguments. The operation of DMOD is the same as that of AMOD except that all instructions are done in double precision.

9. INT, IFIX

INT and IFIX are integer*4 functions with one real*4 argument. (They are in fact the same function.)

If $|ARG|$ is in the range $0 \leq |ARG| \leq 2^{31} - 1$ the value of the function is the sign of ARG times the largest integer $\leq |ARG|$.

If $|ARG|$ is $\geq 2^{31}$, the value of the function is the remainder of $|ARG|/2^{31}$, times the sign of ARG.

10. HFIX

HFIX is an integer*2 function with one real*4 argument. If $|IFIX(ARG)|$ is $> 2^{15} - 1$, the value of HFIX is unpredictable. Otherwise

HFIX (ARG) = IFIX (ARG)

11. IDINT

IDINT is an integer*4 function with one real*8 argument.

If $|ARG|$ is in the range of $0 \leq |ARG| \leq 2^{31} - 1$ the value of the function is the sign of ARG times the largest integer $\leq |ARG|$.

If $2^{31} \leq |ARG| \leq 2^{56}$ the value of the function is the remainder of $|ARG|/2^{31}$ times the sign of ARG.

If $|ARG| \geq 2^{56}$ the value of the function is the sign of arg times the positive integer formed by the last 31 bits of ARG.

12. AINT

AINT is a real*4 function with one real*4 argument.

The value of the function is equal to the sign of ARG times the largest integer $\leq |ARG|$.

13. FLOAT

FLOAT is a real*4 function with one integer*4 argument.

The value of the function is the floating-point representation of the argument.

14. DFLOAT

DFLOAT is a real*8 function with one integer*4 argument. The value of the function is the real*8 representation of the argument.

15. DIM

DIM is a real*4 function with two real*4 arguments. The value of the function is ARG1 - ARG2 if ARG1 > ARG2, and zero if ARG1 $\leq$ ARG2.

16. IDIM

IDIM is an integer*4 function with two integer*4 arguments. The value of the function is ARG1 - ARG2 if ARG1 > ARG2, and zero if ARG1 $\leq$ ARG2.

17. SIGN

SIGN is a real*4 function with two real*4 arguments. The value of the function is the sign of ARG2 times $|ARG1|$.

18. DSIGN

DSIGN is a real*8 function with two real*8 arguments. The value of the function is the sign of ARG2 times $|ARG1|$.

19. ISIGN

ISIGN is an integer*4 function with two integer*4 arguments. The value of the function is the sign of ARG2 times $|ARG1|$.

20. IABS, ABS, DABS

IABS is an integer*4 function with one integer*4 argument. The value of the function is $|ARG|$. ABS and DABS are similar functions for real*4 and real*8.

21. SNGL

SNGL is a real*4 function with one real*8 argument. The value of the function is the real*4 representation of the argument.

22. DBLE

DBLE is a real*8 function with one real*4 argument. The value of the function is the real*8 representation of the argument.

23. CMPLX, DCMPLX

CMPLX is a complex*8 function with two real*4 arguments. The value of the function is the complex number whose real part is ARG1 and whose imaginary part is ARG2.

DCMPLX is a similar complex*16 function with real*8 arguments.

24. REAL

REAL is a real*4 function with one complex*8 argument. The value of the function is the real part of the argument.

25. AIMAG

AIMAG is a real*4 function with one complex*8 argument. The value of the function is the imaginary part of the argument.

26. CONJG, DCONJG

CONJG is a complex*8 function with one complex*8 argument. The value of the function is the complex conjugate of the argument. DCONJG is a similar complex*16 function.

27. Sense Light Routines SLITE, SLITET

The subroutine subprogrammes SLITE, SLITET are provided for setting and testing the states of 4 pseudo-sense lights which are implemented as bit switches in storage at the label SLITES.

Subroutine SLITE is entered with the standard WATFOR entry macro FENTZ followed by checks for a single defined integer argument. If these tests pass, the value of the argument, j say, is screened for range $0 \leq j \leq 4$ with terminating error LI-2 for failure. If j equals zero, all sense lights are turned 'off' by setting all bit switches to 0; otherwise the jth light is turned 'on' by setting the corresponding bit switch to 1. SLITE then returns to the calling routine via R14.

Subroutine SLITET is entered via FENTZ. Checks are performed for two integer arguments of which the first, say j, is also tested for

being defined. Then j is tested for range $1 \leq j \leq 4$ with terminating error LI-2 given if not. The jth bit of SLITES is tested: if 0 (light j 'off') the value of the 2nd argument of SLITET is set to 2; if 1 (light j 'on'), set 2nd argument to 1 and jth bit to 0 (turn light j 'off') SLITET returns to caller via R14.

Registers R1, R2, R3, R15 are used, without saving or restoring, by SLITE, SLITET.

28. Bit-manipulating Functions EOR, OR, AND, COMPL

These real-valued function subprogrammes were provided in WATFOR for compatability (except of course for word size) of programmes converted from University of Waterloo's 7040 operations. They provide for bit-wise 'exclusive or', 'or', 'and', inversion respectively of word size arguments of any type. Functions EOR, OR, AND require two arguments; COMPL requires one.

EOR, OR, AND are each entered via FENTZ macro and at each entry point, checking is performed to insure that only two arguments are present in the calling sequence. (This test is done at each entry point to insure R15 points to the routine name should there be an error message to print.) The opcode of an instruction in a section of coding common to the three routines is then initialized before the common coding is jointed.

This common coding then checks that both arguments are defined before performing the appropriate bit modifications. This is done by loading the 1st argument into R0 and performing a machine 'and', 'or' or 'exclusive or' as initialized above, (N,O,X) of the 2nd argument. Floating register F0 is then loaded from R0 since the functions are real. Return is taken via R14.

COMPL is entered via the FENTZ macro and checks are performed for a single, defined argument which is loaded into R0. The bits of R0 are inverted by exclusive oring with a full word of 1's and R0 is transferred to F0 since COMPL is real. Return is via R14.

These routines destroy the contents of registers R0, R1, R2, R3, R15.

29. Execution-terminating Routine EXIT

Subroutine subprogramme EXIT is entered via the FENTZ macro and a check is performed that no arguments are included in the call. Execution is terminated by branching to XSTOP. Thus, for WATFOR, a call to EXIT functions exactly like a simple STOP statement.

30. Interrupt Count Routine TRAPS

The original intent of WATFOR was to terminate execution of the compiled programme on the 1st of a floating point overflow, underflow

or fixed or floating divide exception since these are usually programme errors. This exceptions are noted by issuing a SPIE macro, just before execution commences, to return to routine XRUPT for processing.

Later it was decided that, to accommodate more experienced programmers, more interrupts of the type mentioned above could be allowed by letting the programmer specify the maximum of each type he would tolerate. Thus XRUPT decrements a count for each interrupt processed and terminates execution when the first such count reaches zero. The routine TRAPS is used to modify these count fields to other than the default 1 set by the job initializer in 'MAIN'. A call to TRAPS appears as follows

CALL TRAPS (i, j, k, ℓ, m)

where the arguments have the following significance:

i - number of fixed-point overflows to be allowed
  - initializes field XFXOFLOW (This interrupt is masked off in standard WATFOR and hence this argument is, in essence, ignored.)
j - number of exponent overflows to be allowed
  - initializes field XEXOFLOW
k - number of exponent underflows to be allowed
  - initializes field XEXUFLOW
ℓ - number of fixed divide exceptions to be allowed
  - initializes field XFXDVCNT
m - number of floating divide exceptions to be allowed
  - initializes field XFLDVCNT

TRAPS may be used with 1 to 5 arguments and only those present will be used for initialization. TRAPS may be called and subsequently recalled from any part of a programme.

Since it was assumed that TRAPS would be used by only experienced programmers, no checking of arguments is performed. Each is treated as an integer value made positive if negative and taken as 1 if zero if necessary.

TRAPS destroys the previous contents of registers R1, R2, R3, R4, R15.

31. Interrupt Indicator Routines DVCHK, OVERFL

These routines are provided to retrieve the status of pseudo-divide check and overflow indicators implemented as byte fields XDVCHKSW, XOVRFLSW. These indicators are initialized to 'off' (X'02') by the job initializer in MAIN, are set by the interrupt processor XRUPT when exponent overflows, underflows or divide checks occur and are turned off by DVCHK, OVERFL respectively.

XRUPT sets these fields as follows:

XDVCHKSW - set to X'01' for fixed divide exception
- set to X'01' for floating divide exception
XOVRFLSW - set to X'01' for exponent overflow
- set to X'03' for exponent underflow.

DVCHK and OVERFL share coding since each operates in the same way. Each entry point merely establishes a pointer register to the 'indicator' to be processed. The current setting of the indicator is transferred (expanded to a full word) to the argument of the routine and the indicator is turned 'off'. Return is via R14. Registers R0, R1, R2, R3 are modified.

5.2 Execution Time Format

This is described in section 4.7.

5.3. FRIOSCAN

5.3.1. Introduction

The main function of the FRIOSCAN routine in WATFOR is to convert input data into a fixed format suitable for conversion to internal machine representation. FRIOSCAN processes logical, integer, real, complex and hexadecimal variables in free or format I/O. Invalid data characters or invalid data formats are caught in FRIOSCAN and result in termination of the object programme. Characters punched on a 26 or 29 keypunch are made equivalent.

FRIOSCAN is closely linked to another WATFOR routine FORMCONV. FORMCONV sets up certain registers to describe the input data to be converted and then passes control to FRIOSCAN. When FRIOSCAN converts the input data into the required format, control is passed back to FORMCONV where the data is converted into internal machine representation. There are 3 basic steps in FRIOSCAN:

1. Initialization of switches, storage areas and special table.
2. SCAN of input characters using the TRT (translate and test) instruction and special table to select routines to process the input characters.
3. Routines chosen in step 2 check for invalid data format, store the input data in the required format and then either reset the special table to continue scan step 2 or if at the end of the field, return to FORMCONV.

A flow chart (see Figure 5.3.1.) and a more detailed description of these 3 basic steps follows.

Flowchart of Frioscan

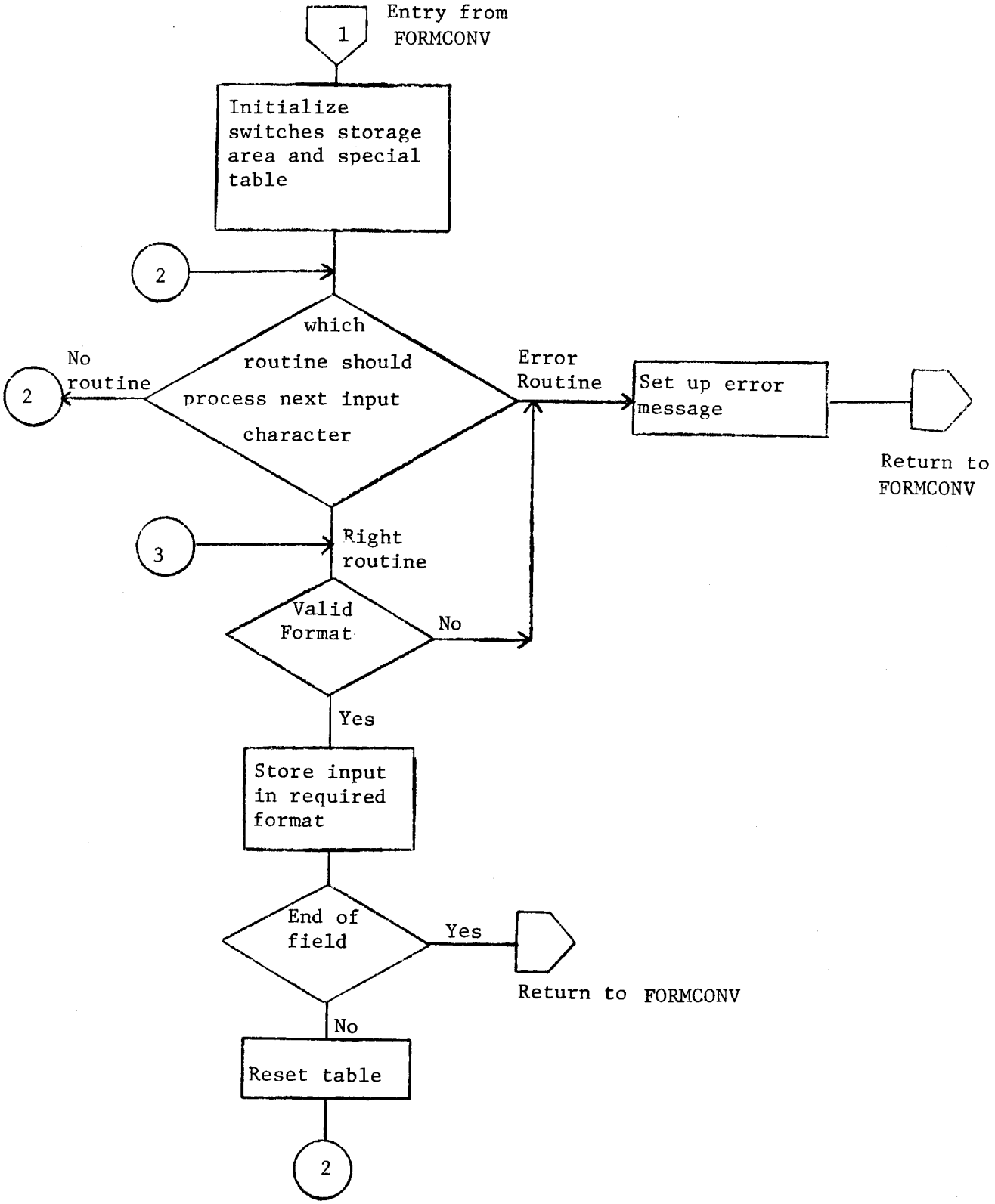


Figure 5.3.1.

5.3.2. INITIALIZATION

As mentioned above the routine FORMCONV sets up registers to describe the input data which FRIOSCAN is to convert. These registers and their purposes are described in table 5.3.1.

REGISTER	PURPOSE
REG 1	pointer to next input character
REG 3	length of input field
REG 6	indicator as to whether variable is in free or format I/O
REG 7	indicator of type of variable

Table 5.3.1.

FORMCONV sets the starting address of the input field in register 1 and its length in register 3, so that the first step in the initialization process is to calculate the end address of the field and store it in a free register (register 4).

Next the 256 byte table 'FRIOTABL' is initialized according to the type of variable being scanned so that when the TRT instruction is executed the proper routines will process that variable. If the variable is in free I/O the table is again altered.

A 2 byte switch 'FRIODUPL' (see table 5.3.2.) is used to store the duplication factor. It is always initialized to 1 and altered only if necessary for a hexadecimal variable.

A 1 byte switch 'FRBUFEND' (see table 5.3.2.) is the name of a 16 byte storage area used to pass the converted input data to FORMCONV. It is initialized to zero.

'FRIOTABL' is altered at this point so that 26 and 29 keypunch characters are considered equivalent.

5.3.3. SCAN OF INPUT FIELD

The length in bytes from the end of the input field to the character being scanned is calculated. The table 'FRIOTABL' has had every byte in it set to an appropriate value. If a non-zero byte is found in the table for a given character using the TRT instruction, this byte is used to select a given routine. If a zero byte is found the TRT instruction uses the next input character to obtain a routine. If no field is found in a card, a buffer is requested in free I/O or the field is assumed to be all zeros in format I/O. If an invalid character is found, an error message is set up and control is passed back to FORMCONV where the object programme is terminated.

5.3.4. Routines to PROCESS THE INPUT DATA

These routines check for invalid data format and store the input data in a fixed format. This format consists of a store area for the actual numerical value of the variable. Various switches and counters used are explained in table 5.3.2.

NAME	POSSIBLE VALUES	PURPOSE
FRIOTEMP	ZL16'0'	- to clear area for new data
	up to 16 decimal digits	- represents numeric value of data passed to FORMCONV
FRIOEXP	H'0'	- to clear area for new data
	2 byte binary number	- represents binary value of exponent if present.
FRIOBLNK	H'0'	- to clear area for new data
	up to 2 decimal digits	- contains count of number of leading blanks or zeros.
	X'80'	- upper bit set on if a decimal point is found.
FRIOVSGN	X'00'	- indicates positive variable.
	X'80'	- indicates negative variable.
FRIOESGN	X'00'	- indicates no exponent found.
	X'40'	- indicates presence of exponent.
	X'80'	- indicates presence of a negative exponent.
FRBUFEND	X'00'	- set to zero during initialization.
	X'FF'	- indicates end of buffer in free I/O.

Table 5.3.2.

The card format of the different types of variables is shown in table 5.3.3 where d represents a decimal digit and any symbol in square brackets may be omitted.

Type of Variable	Incoming data format
Integer	[±] ddd
Real	[±] dd [.] ddd [$\begin{smallmatrix}E\\D\end{smallmatrix}$][±][dd]
Complex in Free I/O	([±]dd[.]ddd[$\begin{smallmatrix}E\\D\end{smallmatrix}$][±][dd],[±]dd[.]dd[$\begin{smallmatrix}E\\D\end{smallmatrix}$][ ][dd]),
Logical	any string of characters [containing a T or F]
Hexadecimal	[d][Z]ddd (where Z is required in free I/O)

Table 5.3.3.

The 5 different variable types are stored in the following format by the routines in FRIOSCAN:

1. Integer Variable

The upper bit of 'FRIOVSGN' is set on if a minus sign is detected. The leftmost 16 digits are stored right justified in 'FRIOTEMP'.

2. Real Variable

The upper bit of 'FRIOVSGN' is set on if a minus sign is detected in the first position. The leftmost 16 digits before the exponent are stored right justified in 'FRIOTEMP'. The high order bit of 'FRIOBLNK' is set on if a decimal point is found. If no decimal point is found in free I/O, the decimal point is assumed after the last digit before the exponent part if present. Leading blanks or zeros are omitted in format I/O to allow FORMCONV to place the decimal point but a count of the leading blanks or zeros is stored in 'FRIOBLNK'.

If an E or D is found, the second bit of 'FRIOESGN' is set on. If the exponent sign is negative, the upper bit of 'FRIOESGN' is set on. The exponent is calculated as a binary value to normalize a number with a decimal point before the first digit and is stored in 'FRIOEXP'.

3. Logical Variable

If the variable is found to be true, the high order byte of 'FRIOTEMP' is set to the value X'FF' otherwise 'FRIOTEMP' is left at zero as set during the initialization step. If no T or F is found the variable is assumed to be false.

4. Complex Variable

Complex variables in format I/O are passed to FRIOSCAN as real variables. Complex variables in free I/O enter in the format shown in table 5.4.3. The comma must be at the end of the first field and there must be a right bracket at the end of the second field. The comma after the right bracket is ignored. The data is passed back to FORMCONV in the same format as 2 real variables.

5. Hexadecimal Variable

If a duplication factor is found 'FRIODUPL' is set to its value. The leftmost 16 digits are stored right justified in 'FRIOTEMP'.

5.4 FORMCONV

5.4.1. Introduction

FORMCONV is the formatting and conversion routine for the execution phase of the compiler. It is called by routines in the STARTA to perform output conversion and formatting and input conversion. The routine calls the FRIOSCAN (5.3) routine to handle the input scanning of data cards during the processing of READ statements and the FIOCS (5.5) routine to perform input and output operations.

WATFOR allows non-formatted input/output statements for the 'reader' and 'printer'. This allows the user who is learning the FORTRAN language to run and test programmes without having to learn the concept of the FORMAT statement. It is also very useful for the experienced programmer when he is debugging a programme. To avoid confusion this type of I/O will be referred to as free I/O to distinguish it from the normal un-formatted (binary) I/O. Binary I/O requires no conversion and hence is not mentioned in the following discussion but is presented in section 5.4.8.

FORMCONV is called by the routine XIOINIT to do initialization for processing a read or write operation or by XSIMPELT, XSUBSELT or XARRAY to do an actual input or output formatting and conversion operation.

For initialization the routines called are respectively:

OUTBCDI - for an OUTPUT-FORMATTED operation
INBCDI - for an INPUT-FORMATTED operation
OUTFREEI - for a FREE OUTPUT operation
INFREEI - for a FREE INPUT operation

The formatting and conversion routines are respectively:

OUTBCD+4	- for FORMATTED	OUTPUT
INBCD+4	for FORMATTED	INPUT
OUTFREE+4	for FREE	OUTPUT
INFREE+4	for FREE	INPUT

FORMCONV's base register R15 is assumed to have been loaded with the correct value by the calling routine and all of the above-mentioned routines begin by loading R13 with the save area pointer which also addresses the second half of the programme.

As has been explained in section 4.7. the format specifications of the form nSw.d have been broken down into a Format Stack entry of the form

AL1(n),AL1(w),AL1(d),AL1(code for S)

Throughout the routine FORMCONV

n	is referred to as	FIELDCNT
w	is referred to as	LFIELD
d	is referred to as	NDEC

The relative address of the routine for X is referred to as ROUTNUM.

The routines that process an I/O list element have the following characteristics in common.

1. First check the end of I/O list switch (ENDSW) to see if the end of list has been signalled.
2. Check the variable type and give an error message (FM-5) if the format type does not match the variable type.
3. Go to the LENCHECK routine to see that the field will fit in the output buffer.

The formatted output routines perform all the above functions. The free input and output routines perform item 3 only, while the formatted input routine performs items 1 and 3.

Following is a description of the routines to process each type of format specification for the four types of I/O. Several service routines are used for the various codes and are described in section 5.4.6.

5.4.2. FORMATTED OUTPUT

Control is first passed by XIOINIT to OUTBCDI with

R1 = A(format stack - START).
R2 = address of output buffer.
R3 = length of output buffer.

R1 is stored in FORMPTR, R2 is stored in ABUFFER (same as XBUFFER) and in BPOINT and R3 is stored in LBUFFER.

The count in the duplication factor FIELDCNT is zeroed. The instruction SUPEXEC1 is initialized to select a routine from the table ORTNTBLE in the common routine for OUTBCD and INBCD to process the present specification. NBDOT is zeroed and OUTBCDI returns control to the object code.

For each I/O LIST item control is passed to OUTBCD+4 by a sequence, for example

```
BAL     R14,XSIMPELT
DC      AL1(code),AL3(address of variable-start)
```

OUTBCD saves the code in CODEBYTE and examines the FIELDCNT to determine if the format specification is the same as the last time.

If not FORMPTR is fetched and incremented and the FORMAT stack entry broken out. Using the fourth byte of the stack entry a code is selected which gets the relative address of the appropriate routine from ORTNTBLE. This value is saved in ROUTNUM in case of future duplication factors. The routine is then entered.

This section describes the routines to process the various format specifications for formatted output. Some of the routines will also be used for formatted input.

(a) STFRBRK - FIRST LEVEL LEFT BRACKET PROCESSOR (CODE=X'08')

The format stack entry for 'n(' is of the form

```
DC  AL1(n,0,0),X'08'
```

where n, which has been saved in FIELDCNT, is moved to FRBRCNT, the first level bracket count.

(b) BOOTOFF - INVALID FORMAT PROCESSOR (CODE=X'0C')

This is the code for an invalid format and causes an ERROR FM-5 to be issued.

(c) SCALESET - SCALING FACTOR PROCESSOR (CODE=X'10')

The format stack entry for 'nP' is

```
DC H'n',AL1(0),X'10'
```

The first half word of the stack entry is moved into NBDOT.

(d) PRFROUT - OUTPUT F FORMAT PROCESSOR (CODE=X'14')

1. Do initialization and checking and mark as F format (PRINMSK).
2. Get relative address of variable stored by OUTBCD and convert it to an absolute address.
3. If the type is double precision go to PRDPF. Otherwise go to SPCON to do single precision conversion. The number is sent down in TEMP and results come back in R7 and EXP such that

$$TEMP = R7*10^{EXP}$$

4. Convert R7 to decimal, unpack and put the address of first significant digit in R5.
5. Calculate the number of digits before the decimal point (Exp + Length of Integer from R7 and Scaling Factor) and put it in R3.
6. Put the length of output field (LFIELD) in R6.
7. Set exit address from COMPRSET as the cleanup routine (SUPCLEAN).
8. If G format go to PRGROUT1, otherwise go to COMPRSET.

(e) PRDPF - Handles double precision F

1. Put the double precision number in floating register 0.
2. Go to the double precision conversion routine (DPCON) with

$$FR0 = INTEGER*10^{EXP}$$

where INTEGER is in R2 and R3.

3. Go to CONVDP to convert the number to unpacked decimal and return the address of first significant character in R5.
4. Calculate the number of digits before the decimal point (EXP + Length of integer from R2 and R3 + scaling factor) and put it in R3.
5. Proceed to step 6 above.

(f) PRDROUT - OUTPUT D FORMAT PROCESSOR (CODE=X'1C')

1. Do initialization and checking and mark as D format (PRINMSK).
2. Convert the relative address to an absolute address.

3. Put the double precision number in floating register 0 and go to the double precision conversion routine (DPCON).

$$FR0 = INTEGER*10^{EXP}$$

where INTEGER is in R2 and R3.

4. Go to CONVDP to set the number of digits before the decimal point equal to the scaling factor (NBDOT) and put it in R3.
5. Set length of field in R6 to be total length of field (LFIELD) minus four.
6. Call COMPRSET to put number in buffer and insert the '0'.
7. Set R3 to end of unpacked double precision number.
8. Use the value in R3 and the value in R5 (Start of Digits), along with EXP and NBDOT to evaluate the exponent.
9. Convert the exponent to decimal, unpack and insert in last two digits of field.
10. Insert a minus sign if necessary and go to cleanup (SUPCLEAN).

(g) PREROUT - OUTPUT E FORMAT PROCESSOR (CODE=X'18')

1. Do initialization and checking and mark as E format (PRINMSK).
2. Convert the relative address to an absolute address.
3. Move the single precision number into TEMP and go to single precision conversion routine (SPCON) where

$$TEMP = R7*10^{EXP}$$

4. Place the number of digits before the decimal (NBDOT) in R3 and convert R7 to decimal, unpack and put address of the first significant character in R5.
5. Set the length of field in R6 to be total length of field (LFIELD) minus four.
6. Call COMPRSET to put the number in the buffer, and insert E.
7. Set R3 to end of the unpacked single precision number and go to step 3 for double precision.

(h) PRIROUT - OUTPUT I FORMAT PROCESSOR (CODE=X'20')

1. Do initialization and checking.
2. Convert the relative address to an absolute address.
3. Place the integer in R0 using 'LH' for integer*2 and 'L' for integer*4 and convert R0 to decimal. Unpack and locate the first significant digit.
4. Calculate the length of the number minus one and put it in R5 and calculate the relative position within the field where the sign will go. If the number doesn't fit in the field go to the asterisk generator (BLOTOUT).
5. Insert the sign if necessary and move in the digits. Update the buffer pointer and go to cleanup (SUPCLEAN).

(i) TABULATE - T FORMAT PROCESSOR (CODE=X'24')

The buffer pointer (BPOINT) is set to the value of the start of the buffer (ABUFFER) + LFIELD = 1 and RECLEN is set to LFIELD - 1 where LFIELD is the value of n in 'Tn'.

(j) PRAROUT - OUTPUT A FORMAT PROCESSOR (CODE=X'28')

1. Do initialization and checking and convert the relative address to an absolute address.
2. Get the address of the next buffer position (BPOINT) in R4 and get the length of variable in R3.
3. Output MIN characters into the buffer where

   MIN = MIN(LFIELD, number of characters in variable).

4. Update the buffer pointer and go to cleanup (SUPCLEAN).

(k) PRLROUT - OUTPUT L FORMAT PROCESSOR (CODE=X'2C')

1. Do initialization and checking and get the address of the last position in the field.
2. Convert the relative address to absolute address and insert 'T' or 'F' depending on variable.

(ℓ) SPACERT - X FORMAT PROCESSOR (CODE=X'30')

1. Call LENCHECK routine to check if there is enough room in the buffer and increment buffer pointer (BPOINT) by LFIELD if there is.
2. Go to OUTBCD1 to pick up the next entry.

(m) PRHROUT - H FORMAT PROCESSOR (CODE=X'34')

The format stack entry for a Hollerith H is as follows:

```
DC      AL1(0,n,0), X'34'
DC      CLn'Hollerith'
DS      OF
```

1. Call the LENCHECK routine and place the length of the Hollerith minus one in R2.
2. Put buffer pointer (BPOINT) in R3 and use the routine ROUTCODE to index a table of displacements with the value 0 for output and 6 for input.
3. Execute a MVC instruction to move the information in or out and round the length up to a full word.
4. Update the format stack pointer by the length of the Hollerith.
5. Return to OUTBCD processor.

(n) ENDFRBRK - FIRST LEVEL RIGHT BRACKET PROCESSOR (CODE=X'38')

The format stack entry for a first level right bracket is of the form

DC AL2(*-Address of start of format loop),AL1(0),X'38'

1. Get the first level bracket count (FRBRCNT) and decrement by one. If the count is zero return to OUTBCD1 to pick up next sequential stack entry.
2. Otherwise store the first level bracket count. Using the first half-word of the stack entry, calculate the address of the stack entry following the corresponding 'n(' stack entry.
3. Return to the point in OUTBCD just after where the stack pointer is updated.

(o) NEWLINE (NEWCARD) - / PROCESSOR (CODE=X'3C')

1. Set the exit address from OUTREC as return to OUTBCD1 and go to OUTREC.

(p) PRGROUT - OUTPUT G FORMAT PROCESSOR (CODE=X'40')

1. Check for end of I/O list.
2. If the type is integer, go to the entry point in the integer routine (PRIROUT1).
3. If type is logical go to entry point in the logical routine (PRIROUT1).
4. If type is real or complex mark as G FORMAT (PRINMSK) and go to F FORMAT routine at entry point PRFROUT1. Control will be returned to the G FORMAT routine at point PRGROUT1 when the PRINMSK is tested by the F FORMAT routine.
5. A check is made to see if the magnitude of the number is greater than or equal to zero and less than or equal to NDEC which comes from the 's' of nGw.s and is the number of significant digits to be generated. If this check fails then a branch is taken to the E ROUTINE at PREROUT1 + 4 for single precision or to the D ROUTINE at PRDROUT1 + 4 for double precision.
6. If the check is successful then NDEC is saved in NDEC1 and NDEC is set to the number of digits after the decimal point (number of significant digits requested 's' minus magnitude of number).
7. The PRINMSK is set to F FORMAT and the routine COMPRSET is called to output the number, NDEC is restored and the buffer pointer is updated. Go to CLEANUP (SUPCLEAN).

(q) ENDSTAT (ENDSTATI) - END OF FORMAT PROCESSOR (CODE=X'44')

The format stack entry for the end of format right bracket is

DC AL2(*-Address of return stack entry),AL1(0),X'44'

1. The ENDSW variable is checked.
2. If both bit 6 and bit 7 are off then the end of the I/O list has not been reached. Bit 6 is turned on and then the entry is treated as a first or second level right bracket by the routine COMBRK1 which is entered after the OUTREC routine has been called to output the record.
3. If the end of the I/O list has been reached (bit 7 of ENDSW = 1) then GO TO ENDROUT to finish up and return to the object code.
4. Otherwise bit 6 of ENDSW = 1 and the error message FM-7, non-terminating format is issued, since this is the second time through ENDSTAT with no I/O list items processed.

(r) PRZROUT - OUTPUT Z FORMAT PROCESSOR (CODE=X'48')

1. Do initialization and checking and convert the relative address to an absolute address.
2. Get the buffer address (BPOINT) in R4 and using the CODEBYTE, put the variable's length in R6 and double it (number of characters to be output).
3. Set R3 to end of unpacked hexadecimal number to be plus one and put the length of the field (LFIELD) in R7.
4. If field length (LFIELD) is less than or equal to the number of characters available, then go to PRZROUT1 and output the last (R7 = LFIELD) characters of the number.
5. Otherwise increment the buffer pointer (R4) by the number of blanks to be output (LFIELD (in R7) - number of characters (in R6)).
6. Set R7 to number of characters (in R6), and go to PRZROUT1 and output the last (R7 = number of characters) characters of the number.
7. Update buffer pointer (BPOINT) and go to cleanup (SUPCLEAN).

ENDSCBRK - SECOND LEVEL RIGHT BRACKET PROCESSOR (CODE=X'4C')

The format stack entry is

DC AL2(*-start of format loop),AL1(0),X'38'

This routine is the same as the first level right bracket routine except that it is the second level bracket count (SCBRCNT) that is tested.

STSCBRK - SECOND LEVEL LEFT BRACKET PROCESSOR (CODE=X'50')

The format stack entry for a second level left bracket 'n(' is

```
DC AL1(N,0,0),X'50'
```

N, which has been saved in FIELDCNT is moved to the second level bracket count (SCBRCNT) and return to OUTBCD1.

5.4.3. FORMATTED INPUT

Initialization

Control is first passed by XIOINIT to INBCDI with

R1 = A(format stack-start)
R2 = Address of input buffer
R3 = Length of input buffer

R1 is stored in FORMPTR, R2 is stored in ABUFFER and BPOINT and R3 is stored in LBUFFER.

1. The duplication factor (FIELDCNT) is zeroed.
2. The routine code (ROUTCODE) is set to 2.
3. The record length (RECLEN) is zeroed.
4. The instruction SUPEXEC1 is initialized to select a routine from the table INTNTBLE in the common routine OUTBCD and INBCD.
5. The pointer to the input routine is initialized to the address of FRIOSCAN.
6. The scaling factor is zeroed.
7. The routine BUFFIN (BUFFINIT) is called.
8. Return to object code.

For each input list item control is passed to INBCD + 4.

RDEROUT - INPUT E FORMAT PROCESSOR (CODE=X'18')

1. Do initialization and checking and put the code for real(8) in R7 for INFRIO.
2. Call INFRIO to get number from buffer (see common subroutines section) and check if a decimal point was found (bit 0 of FRIOTEMP + 18 = 1).
3. If no decimal point was found, call the routine IMPLDOT to adjust the exponent using the format specification.
4. Call INSPCON to convert the number in TEMP and EXP to a floating point. The number returned in floating register

0 satisfies

$$FRO = TEMP*10^{EXP}$$

with the sign of the number determined by NEGSW.

5. Put the relative address of the variable in R3.
6. FRO is stored in the variable pointed to by R3 and is also saved in TEMP.
7. Go to cleanup (INCLEAN).

RDDROUT - INPUT I FORMAT PROCESSOR (CODE=X'20')

1. Do initialization and checking and put the code for integer(4) in R7 for INFRIO.
2. Call INFRIO to get number from buffer. The unpacked decimal integer is sent back right justified in TEMP and is packed into TEMP2.
3. If the number is negative (bit 0 of NEGSW = 1) then set the sign of the packed number to minus. If the number is too small, issue error KO-8 (integer out of range), otherwise proceed to step 5.
4. If the number is positive and larger than a full word issue error KO-8.
5. Convert the number to binary and put the relative address of the variable in R3.
6. The value is stored in the variable pointed to by R3 and is also save in TEMP.
7. Go to cleanup (INCLEAN).

RDAROUT - INPUT A FORMAT PROCESSOR (CODE X'28')

1. Do initialization and checking and convert the relative address to an absolute address.
2. Get buffer pointer and the length of variable and if the variable length is greater than the field length move in LFIELD characters.
3. If not, move variable length characters from buffer into variable and go to cleanup (INCLEAN).
4. Fill out to end of variable with (VARIABLE LENGTH - LFIELD) blanks, and go to cleanup (INCLEAN).

RDLROUT - INPUT L FORMAT PROCESSOR (CODE=X'2C')

1. Do initialization and checking and put the code for logical (0) in R7 for FRIOSCAN.
2. Call INFRIO to get number from the buffer. A logical variable is sent back in first byte of TEMP.
3. Convert the relative address to an absolute address and move the variable from TEMP into storage.
4. Go to cleanup (INCLEAN).

RDGROUT - INPUT G FORMAT PROCESSOR (CODE=X'40')

1. Do initialization and checking and get the code byte and double it.
2. Use the doubled code byte to get the routine address from the table IRTFRIO (same as free input routines) and go to the routine.

RDZROUT - INPUT Z FORMAT PROCESSOR (CODE=X'48')

1. Do initialization and checking and put the code for hexadecimal(16) in R7 for INFRIO.
2. Call INFRIO to get number from buffer. The most significant 32 hexadecimal digits are returned (right-justified) in TEMP.
3. Convert the relative address to an absolute address and get variable length and the address of the end of the variable (i.e. TEMP ± 16).
4. Locate start of variable to be moved and move the variable from TEMP into storage.
5. Go to cleanup (INCLEAN).

5.4.4. Free Output

OUTFREEI - Control is first passed by XIOINIT to OUTFREEI for initialization with

R2 = Address of the output buffer
R3 = Length of the output buffer

1. Set up R13 to point to the save area.
2. Call the buffer initialization routine (BUFFINIT).
3. Leave the control character in the first position and update the pointers.
4. Set the routine code (ROUTCODE) to 1 to indicate free output.
5. Zero the scaling factor, restore registers and return to object code.

OUTFREE - For each I/O list item control is passed to OUTFREE + 4. For end of I/O list, control is passed to OUTFREE + 0, which branches to ENDFREEO. This routine outputs the last record and then returns to the object code. For each I/O list item:

1. Set up R13 and save the relative address of the variable.
2. Get the code byte of variable and only keep the lower 3 bits which indicate the type and length.
3. Put the code byte in R3 double it and use the double code byte to select the routine address.
4. Double R3 again and use R3 as an index to get the LFIELD and NDEC specifications from a table of free output format specifications (FRIOSPEC).
5. Go to the routine.

Note: In all cases the routine which is called is an entry point in the corresponding formatted output routine which bypasses the type and end of I/O list checking.

5.4.5. Free Input

INFREEI - This is the initialization routine for free input. Control is first passed to INFREEI by XIOINIT with

R2 = Address of the input buffer.
R3 = Length of the input buffer.

1. Set up R13 and call the buffer initialization routine (BUFFIN).
2. Save the record length in LFIELD to be passed later to FRIOSCAN.
3. Initialize the pointer to the input routine to the address of FRIOSCAN and set the routine code (ROUTCODE) to 3 to indicate free input.
4. Zero the duplication factor, zero the scaling factor, restore the registers and return to the object code.

INFREE - For each I/O list item control is passed to INFREE+4. For end of I/O list control is passed to INFREE + 0 which branches to ENDFREEI, which returns to the object code. For input list items:

1. Set up R13 and save the relative variable address.
2. Get the code byte of variable and only keep the lower 3 bits which indicate type and length.
3. Get the duplication factor and decrement by one and test for positive.
4. If not positive, double the code byte and use to index a table of free input routines (IRTFRIO).
5. Go to selected routine which is an entry point in the corresponding formatted input routine.

DUPROUT - This routine handles duplication factors in free input.

1. Get the length of the variable and decrement by one for MVC instructions.
2. Record the updated duplication factor and convert the relative address to an absolute address.
3. Is variable complex?
4. If not, move variable from TEMP (where it has been stored just for this case) into storage and return to the object code.
5. If so, move first half of complex number from COMSAVE (where it was stored by complex routine just for this case) into storage.
6. Increment the address by the variable length and move in the second half from TEMP.
7. Return to the object code.

5.4.6. COMMON SUBROUTINES AND UTILITY ROUTINES

(a) LENCHECK

This subroutine gets the record length (RECLEN) and increments it by the field length (LFIELD) and compares it to the buffer length (LBUFFER). If the field will not fit, then the free record out routine (FREERECO) is entered for free output. Otherwise the error message (IO-3, output record too long) is issued.

(b) OUTREC (INREC)

This subroutine is the interface to the FORTRAN input/output subroutine (FIOCS). A code required by FIOCS is selected using the ROUTCODE as an index. This code (which equals 2 for output, and 1 for input), is stored in the line after the call to FIOCS. The buffer initialization routine (BUFFINIT) is then entered to record the buffer address and length and initialize the record length (RECLEN).

These are the cleanup routines for an I/O list entry. They perform the following operations:

(c) SUPCLEAN

1. Store the buffer pointer R4 in BPOINT.
2. Check the variable type for complex and branch to the complex routine COMPRTN if type is complex.
3. Turn off ENDSW (bit 6) to indicate that an I/O list item has been processed.
4. Return to object code.

(d) INCLEAN

1. Add field length (LFIELD) to buffer pointer (BPOINT).
2. Go to SUPCLEAN.

(e) FREERECO

1. Save the return address and go to OUTREC to output current record.
2. Initialize BPOINT, ABUFFER, LBUFFER and RECLEAN.

(f) COMPRTN

1. Change the variable type from complex to real (CODEBYTE).
2. Update address of variable by length of real part (4 - for complex*8,8-for complex*16).
3. Save the VALUE of TEMP in COMSAVE in case of free input.

4. For formatted I/O return to COMPRET, which is an entry point in OUTBCD. This makes the second half of a complex number look like a second call to OUTBCD.
5. For free I/O separate into input and output and go to the appropriate routine according to single or double precision. This looks like a call from INFREE or OUTFREE.

(g) ENDROUT

1. For output (bit 6 of ROUTCODE + 1 = 0) call the OUTREC routine to output the last record. Then return to the object code.

(h) BLOTOUT

1. This routine is called when a number, to be output, will not fit in the space left for it.
2. The field pointed to by R4 and of length LFIELD is filled with asterisks.
3. Go to cleanup (SUPCLEAN).

(i) DPCON

This subroutine takes the double precision floating point number, say F, in floating register 0 and expresses it in the form

$$F = \text{INTEGER}_{16} * 10^{EXP}$$

where EXP is a half-word decimal exponent and INTEGER is in the range $16^{13} \leq \text{INTEGER} < 16^{15}$ unless F is 0 in which case INTEGER equals 0. INTEGER is returned in the combined register pair R2 and R3. A floating point number, say F, has the form

± 7 bits 56 bits

where $.I = \frac{I}{16^{14}}$ (I is a fourteen hexadecimal digit integer with the first digit non-zero). This means .I is in the range

$$\frac{1}{16} \leq .I < 1$$

If any two numbers of the form .I are divided, say $.I_j/.I_k$

the answer is of the form

$$\frac{.I_j}{.I_k} = .I_\ell * 16^{\delta} \text{ where}$$

$\delta = 0$ if $.I_j < .I_k$ or $\delta = 1$ if $.I_j \geq .I_h$

Divide here is a /360 floating point division with the implied truncation.

The seven digit exponent field is excess 64. i.e. an exponent of 65 would mean the .I part of the number would be multiplied by 16^1.

DPCON splits this 7 bit exponent field into two parts, a 3 bit lower field j and a 4 bit upper field i.

Thus the exponent can be represented as $8i + j$.

Thus

$$F = .I_1 * 16^{8i + j - 64}$$

or $$F = .I_1 * 16^{8(i - 8)} * 16_j$$

or $$F = \frac{.I_1 * 16^{14}}{16^{-8(i - 8)} * 16^{-j} * 16^{14}}$$

DPCON makes use of two tables indexed by i (represented by (DTR1)) and an integer q_1 (represented by DTR1A) which express $16^{-8(i - 8)}$ in the form:

$$16^{-8(i - 8)} = .I_2 * 10^{-q_1} \text{ for } i = 0 \text{ to } 15.$$

Thus F now equals

$$F = \frac{.I_1 * 16^{14}}{.I_2 * 10^{-q_1} * 16^{-(j - 14)}}$$

and carrying through the division

$$F = \frac{.I_3 * 16^{\delta_1} * 16^{14}}{10^{-q_1} * 16^{-(j - 14)}} \quad \text{where } \delta_1 = 0 \text{ or } 1$$

$$= \frac{.I_3 * 16^{14}}{10^{-q_1} * 16^{-(j + \delta_1 - 14)}}$$

We now employ two more tables of constants $.I_4$ - (DTR2) and integral q_2 - (DTR2A) indexed by $(j + \delta_1)^4$ such that:

$$16^{-(j + \delta_1 - 14)} = .I_4 * 10^{-q_2}$$

for $(j + \delta_1) = 0$ to 8.

Now $$F = \frac{.I_3 * 16^{14}}{.I^4 * 10^{-q_2} * 10^{-q_1}}$$

$$= \frac{.I_5 * 16^{\delta_2} * 16^{14}}{10^{-q_1} * 10^{-q_2}} \quad \text{where } \delta_2 = 0 \text{ or } 1$$

$$= (I_5 * 16^{\delta_2}) * 10^{q_1 + q_2}$$

which is the product of a 14 or 15 hexadecimal digit integer and some integral power of 10 which is our desired result.

The evaluation of the constant q_i consists of expressing 16^n in the form $.I * 10^{-q}$

If $C + m = \log_{10}(16^n)$ where C is the integral characteristic and m the mantissa we have

$$10^{C+m} = 16^n$$

or $$10^{(C + 1) + (m - 1)} = 16^n$$

or $$10^{C + 1} * .I = 16^n$$

where $.I = 10^{m-1}$

Since $0 \leq m < 1$

$$\frac{1}{16} < \frac{1}{10} \leq 10^{m-1} < 1$$

and hence 10^{m-1} is of the required form .I

$$q = -(C + 1)$$

(j) SPCON

This subroutine takes the single precision floating point number in the variable TEMP and expresses it in the form

TEMP = INTEGER * 10 ** EXP

where EXP is a half word decimal exponent and INTEGER is in the range $(2^{31})/10 < \text{INTEGER} < 2^{31}$.

(k) INDPCON

This subroutine does the reverse of DPCON. It takes as input the 16-character unpacked decimal number in TEMP, a decimal exponent in EXP and a sign indicator in NEGSW. The 16-character number in TEMP is considered to be a 16-decimal digit integer with value such that

INTEGER * 10 ** EXP

has the same value as the number on the input card modified by any scaling factors that apply. This routine generates the equivalent floating-point number and returns it in floating register zero.

1. The number is packed in such a manner so as to generate a fourteen-hexadecimal digit integer in the range 16^{13} to $16^{14}-1$ unless the number is zero.
2. This is then floated by concatenating X'4E' onto the front to produce the floating value of the integer part.
3. The sign is then inserted.
4. If the exponent is less than -64, then FRO is divided by 10 ** (64) and the exponent incremented by 64.
5. The exponent is split into two indices, one consisting of the last four bits of the exponent which is used to index a table of the form 10 ** N for N from 0 to -15, which is divided into FRO, the other consisting of the first three bits plus a constant which is used to index a table of the form 10 ** 8N for N from -4 to +4.

(ℓ) INSPCON

This subroutine is virtually the same as INDPCON except that a single-precision number is returned in FRO.

(m) COMPRSET

This subroutine rounds F type numbers and takes as input:

1. R5 - a pointer to the first non-zero character of an unpacked decimal number.
2. R3 - the number of decimal digits to occur before the decimal point (can be negative).
3. R6 - the length of the output field.
4. NPLACE - the maximum number of significant digits that can be output.
5. NDEC - the number of digits after the decimal point.
6. NEGSW - the sign of the number (i.e. X'80' for negative, X'00' for positive).
7. BPOINT - the address of the next buffer position to be filled.

It outputs the rounded number into the buffer with sign, (R3) digits before the decimal point, and NDEC decimal places after the decimal point.

THE ALGORITHM

ROUNDING

1. The number of significant digits requested is calculated (R3 + NDEC).
2. The relative address of the first significant digit not to be used is calculated and if it is not positive and less than the number of significant digits available, then rounding is bypassed.
3. If this digit is .GE. 5, then the string is scanned backwards looking for the first digit .NE. 9 and this digit is incremented by 1. All intervening 9's are set to 0.
4. A check is made to see if the final address is less than the original starting address (happens when 0999 goes to 1000). If so, the starting address is updated, and for F-type format only, the number of digits before the decimal is incremented by one.

INSERTING THE NUMBER

1. Find the address where dot will go.
2. R3 contains address relative to dot where first significant digit will go. Save R3 in R2 and then set R3 = MIN (1, R3) which is address relative to dot where first significant digit or leading zero will go, i.e. take note of fact that 1/2 is output as 0.5.

3. If a negative number then go back one space and insert minus sign.
4. Find the address where first character is go to +1, and save its value of R7.
5. Initialize the remainder of the field to zero's.
6. Find the address of where the first significant digit will go and copy into that address the minimum of the number of significant digits available and the number of digits required to complete the field.
7. Shift the field starting at the address computed in step 4 and with the length computed in step 2 to the left by one place.
8. Insert the decimal point at the address computed in step 1.
9. Return to caller.

(o) INFRIO

This subroutine is the interface to the card scan routine (FRIOSCAN) which picks up a number from an input card, and puts it into a standard format. The first time FRIOSCAN is called, it enters at entry point FRIOSCAN whose address has been put into the variable FRIOROUT by the initialization routine. Thereafter FRIOROUT is updated by storing the contents of R5, which is returned by FRIOSCAN to inform FORMCONV where it should return the next time. FORMCONV sends to FRIOSCAN the following information.

R1 - pointing to the START of the field to be scanned.
R3 - containing the length of the field to be scanned minus one
R6 - contains a code indicating FORMAT(4) OR FREE(0)
R7 - contains a code describing the type of entry to be picked up.

0 - LOGICAL
4 - INTEGER
8 - REAL
12 - COMPLEX
16 - HEXADECIMAL

FRIOSCAN returns to FORMCONV with the following information stored in FRIOTEMP.

First 16 bytes

FRIOTEMP

- for integer the character representation of the integer right justified in the field.
- for REAL and COMPLEX a 16 character decimal integer.
- for logical, the first byte is set to X'FF' for FALSE and X'00' for TRUE.
- for hexadecimal the most significant 32 hex digits packed

and right justified in the field.

FRIOTEMP+16

- for REAL and COMPLEX a half-word exponent whose value is such that the integer above multiplied by 10** exponent equals the number read in.

FRIOTEMP+18

If bit 0 = 1 then a decimal point was found and
if bit 0 = 0 then it is the number of blanks from the start of the number to the first non-blank character.

FRIOTEMP+20

- a byte indicating the sign

X'80' - for negative
X'00' - for positive

FRIOTEMP+21

- a switch for the exponent

BIT 0 - on if exponent is negative
BIT 1 - on if exponent found

FRIOTEMP+22

- duplication factor in free I/O.

The normal return from FRIOSCAN is a B 8(R5). A BR R15 is the error return which transfers to the ERRFRIO routine which stores the last character from R0 in an FM-error message which is then issued.

The return B 4(R5) is for free hexadecimal format (not implemented yet).

INFRIO then loads R4 with the address of FRIOTEMP and moves the first 16 bytes of FRIOTEMP into TEMP, gets the exponent from FRIOTEMP+16 and modifies it with the scaling factor (NBDOT) if an exponent was not specified (Bit 1 of FRIOTEMP+21 = 0). The sign is moved from FRIOTEMP+20 to NEGSW and the duplication factor to DUPFAC. The routine then returns to the caller.

(p) CONVDP

1. Set the number of significant digits available (NPLACE) to 16.
2. Convert the 14-hexadecimal digit integer in R2 and R3 to decimal in two steps and unpack.
3. Put the address of the first significant character in R5.

(q) IMPLDOT

Routine to adjust exponent according to format specification when to decimal point is found in the input field.

1. Get the exponent and increment by length of the field.
2. Subtract the number of decimal places and subtract the number of blanks before the first digit.
3. Record the modified exponent and return.

5.4.7. IMPORTANT VARIABLES AND SWITCHES

FORMPTR - (the address of the next format stack entry - START).
- (initialized to the start of the format stack - START) by the OUTBCDI and INBCDI processors.

FIELDCNT - duplication factor n on the specification nSw.d.
LFIELD - the field width of the specification nSw.d i.e. (w).
NDEC - the number of decimal digits after the decimal place. i.e. (d).
ROUTNUM - address of routine for code S.
ABUFFER - the start of the output buffer.
BPOINT - the next position in the output buffer to be filled.
LBUFFER - the length of the current buffer.
RECLEN - the number of bytes output in the current record so far.
ROUTCODE - code for current routine.

0 - for BCD OUTPUT
1 - for FREE OUTPUT
2 - for BCD INPUT
3 - for FREE INPUT

NBDOT - the number of decimal digits to come before the decimal point.
CODEBYTE - code to define variable type

0 - for LOGICAL*4
1 - for LOGICAL*1
2 - for INTEGER*4
3 - for INTEGER*2
4 - for REAL*4
5 - for REAL*8
6 - for COMPLEX*8
7 - for COMPLEX*16

FRBRCNT - the current value of the first level bracket count.
SCBRCNT - the current value of the second level bracket count.

ENDSW - Bit 0 - Bit 5 - always zero
- Bit 6 - used to detect non-terminating formats
  - tested by end of format routine.

  If on, ERROR FM-7, non-terminating format is issued.
  If not on, then turned on.

  - it is turned off by initialization and by the processing of an I/O list item.

- Bit 7 - End of I/O list reached.
  Tested by I/O list format processors and end of format stack processor
  Turned on by XSIMPELT etc. by branching to OUTBCD etc. instead of OUTBCD+4.

NPLACE - the maximum number of significant digits capable of being generated for a given number.

Set to 7 for single-precision.
Set to 16 for double-precision.

PRINMSK - marks a field type as E or D, F or G

= X'00' for E or D
= X'01' for G format
= X'02' for F format

NEGSW - SIGN INDICATOR FOR INPUT AND OUTPUT NUMBERS

- X'80' - FOR NEGATIVE
- X'00' - FOR POSITIVE

5.4.8. Non-formatted (Binary) I/O INBINI, OUTBINI, INBIN, OUTBIN

These routines perform the binary I/O operations. The routines are basically a copy of the routines in IBCOM (FORTRAN H routine).

Again as in the rest of FORMCONV three entry points are used (set by XIOINIT in STARTA) for both read and write.

The entries INBINI and OUTBINI are the initialization entries and merely save buffer pointers for future use.

XIOSWN - Input (X'00') vs. Output (X'0F') indicator
R6 - record count
R7 - work register
R8 - address of the end of the buffer
R9 - address of the next available word of the buffer (R10+4 to start)
R10 - address of the start of the buffer.

These registers are initialized and stored in XKEEP.

The entry points INBIN+4 and OUTBIN+4 are used to fetch a variable or place a variable in the buffer. Using the mode and type byte of the variable and the switch XIOSWN the variable is moved to or from the buffer after checking that we are still within the buffer range. If we are over the end of the buffer and the operation is write, the length of the record is stored in the first word (green word) of the record. A call to FIOCS causes the record to be written and control returns to move in the next variable. If the operation is read a test is made on the 'green word' to see if anything follows and the appropriate read is performed. The pointers are updated and control returns to the object code.

The entry points INBIN and OUTBIN are used to process the end of the list of variables. If the operation is read, the rest of the logical record is read (if not already read) and control returns to the object code. On write the green word (the actual number of physical records) is stored and the final record is written.

5.5 FIOCS

This routine is used by the compiler for all its I/O at both compile and execution time. The routine FIOCS is a copy of the FIOCS routine used by the IBM H-level compiler. Several changes and additions have been made to do more error checking and page and line control.

5.5.1. Error Messages

The method of handling error messages has been changed to make it more suitable for WATFOR. A one byte error code is inserted in location PARAMS. FIOCS has previously inserted the address of the data set reference number (DSRN) in PARAMS. Hence the return sequence in case of an error in FIOCS is

```
        L       R15,=V(MYIBCOM)
        CNOP    2,4
        BALR    14,15
PARAMS  DC      X'code',AL3(DSRN)
```

where MYIBCOM is the address of the I/O error processor in STARTA (Section 3.13.). The error code is a numeric value from 0 - 9 corresponding to WATFOR'S'UN' errors.

5.5.2. Switches added

CURUNIT	The current unit number is saved here.
INHIBIT	Set off unless a control card has been read from the 'input' unit at which time it is turned on. This switch is used only at execution time.
CTYPE	Branch/NOP switch. Set to Branch unless current unit is 'input' and we are reading at execution time.
PCONTROL	The control card character ($).
PPRINT	Equated to the print unit (e.g. 6)
PREADC	Equated to the read unit (e.g. 5)
PPUNCH	Equated to the punch unit (e.g. 7)

5.5.3. Line and Page Control (PRITE)

After checking that the unit is the printer, register 15 is set up to point to a set of constants in MAIN. These constants contain the following:

```
PLINES     DS   F   lines per page allowed
PLNCOUNT   DS   F   lines so far on the current page
PPAGES     DS   F   pages allowed
PPGCOUNT   DS   F   pages so far
           DS   F   not used
PXCARD     DS   C   was the punch used?
```

Register 14 is set to point to the output buffer. The control character is checked for one of the types (+, blank, 0, -, 1) and if any other is present it is set to blank. The appropriate line and/or page counts are increased. If the page count becomes too large and WATFOR is in the execution phase, the appropriate error return is taken.

5.5.4. Entry Coding Added

WATFOR uses FIOCS for both its compile and execution time I/O and hence several changes were required upon entry to FIOCS.

(a) Compile Time

The two switches INHIBIT and CTYPE are initialized. Control then transfers in the same manner as the IBM version with the exception of the "initialize entry". In this case the unit number is saved first and in the case of output, line and page control coding is performed.

(b) Execution-Time

Before going to the various routines some checking is performed.

(i) Initialize Entry (PINIT)

The unit number is saved and several checks are made to see if the unit is being used incorrectly.
e.g. 1 (If the input unit is 5, the statement WRITE(5, 300)A would cause an error message to be issued.)
e.g. 2 (If we already have taken a return of the type END= as the result of an end-of-file on the input unit the switch INHIBIT is used so that no further input can occur from the 'input' unit.)

(ii) Input (FRITE)

If the unit is the 'input' unit again test INHIBIT to see if a read is allowed.

(iii) Output (PRITE)

The switch CTYPE is set off. If the unit is the output unit a check is made for a valid control character and line and page counting are performed.

(iv) Control (PCNTRL)

If input, output, or punch is referred to, issue an error message.

e.g. REWIND 5

(v) Execution Time

For the case of input from the 'input' unit a check is made for the control character ($).

6 MAIN

6.1. Introduction

This routine's main function is supervisory. It controls flow between the compile and execution phase of the compiler. It also performs the necessary job accounting, initializing of constants and switches, opening and closing of data sets and timing of jobs. Figure 6.2. describes the main flow of the routine. This chapter also includes a description of the library subprogramme processor MLIBR.

In order to provide addressability for the communication regions MAIN uses several routines in the STARTA and COMMR as 'links' to the next required step. (Figure 6.1 describes the flow of these routines)

XSTART

This routine is used just previous to the start of execution. It issues the execution SPIE macro to get control of interrupts 1-7,9-13,15 It then links to assign values defined in DATA statements (if present). Finally the object code (M/PROG) is called to execute the users programme.

XSTOP

A STOP or CALL EXIT statement causes a transfer to the routine XSTOP which resets register 12 and 13 and calls CSTOP.

STOP

This routine is used when a batch is complete and the compiler wishes to return control to the caller (O.S.?). The value in register 13 and the caller's registers are restored and control returns via register 14.

START

This is the entry point to the compiler. The registers are saved and the save areas (WATFOR's and the caller's) are linked and control now transfers to CSTART (in COMMR).

CSTART

This routine merely sets up register 10 for COMMR and transfers to MAINP (in MAIN).

CSTOP

XSTOP transfers to CSTOP where register 10 is set up and control transfers to MSTOP (in MAIN).

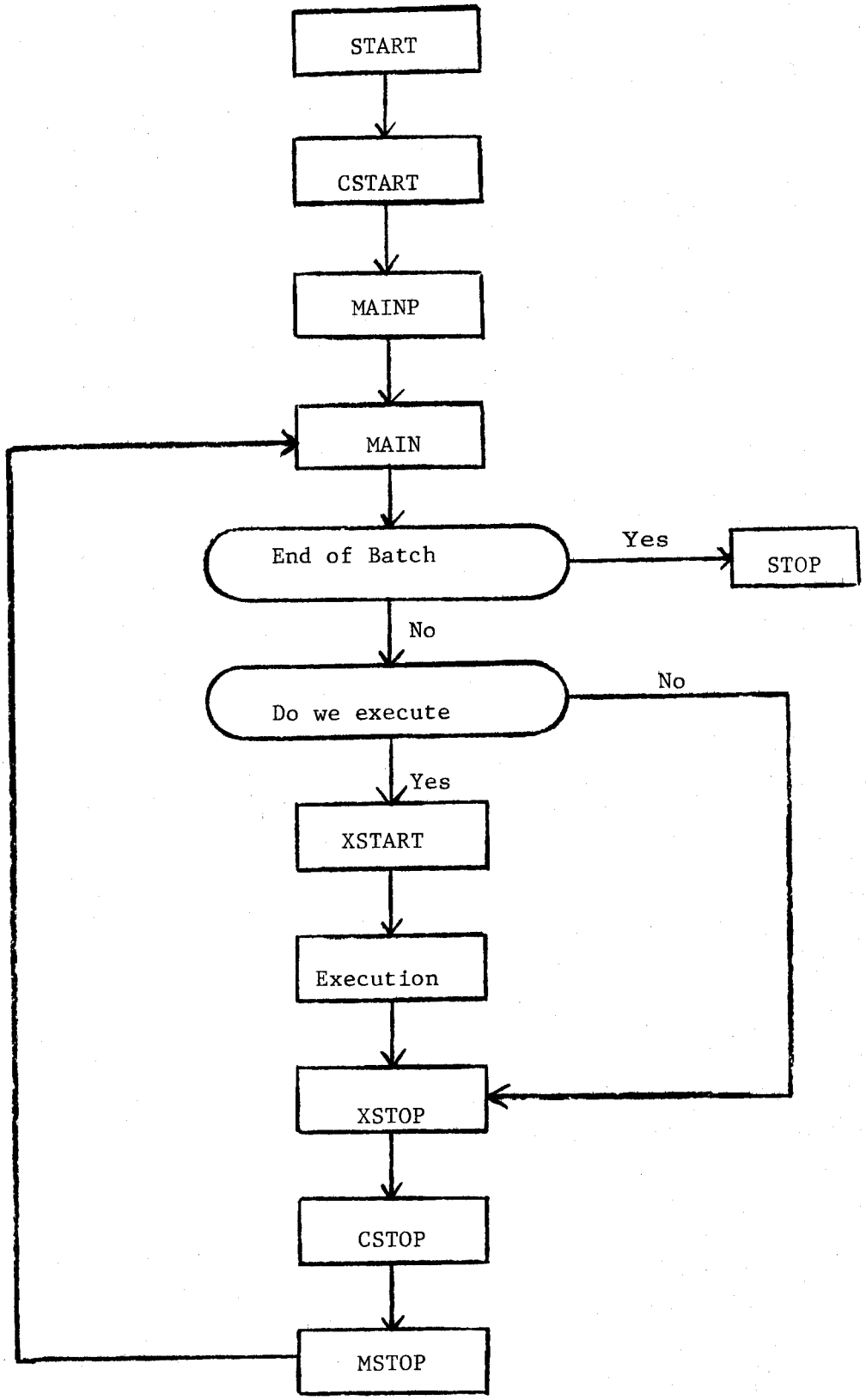

START
CSTART
MAINP
MAIN
End of Batch
Yes
STOP
No
Do we execute
No
Yes
XSTART
Execution
XSTOP
CSTOP
MSTOP

Figure 6.1.

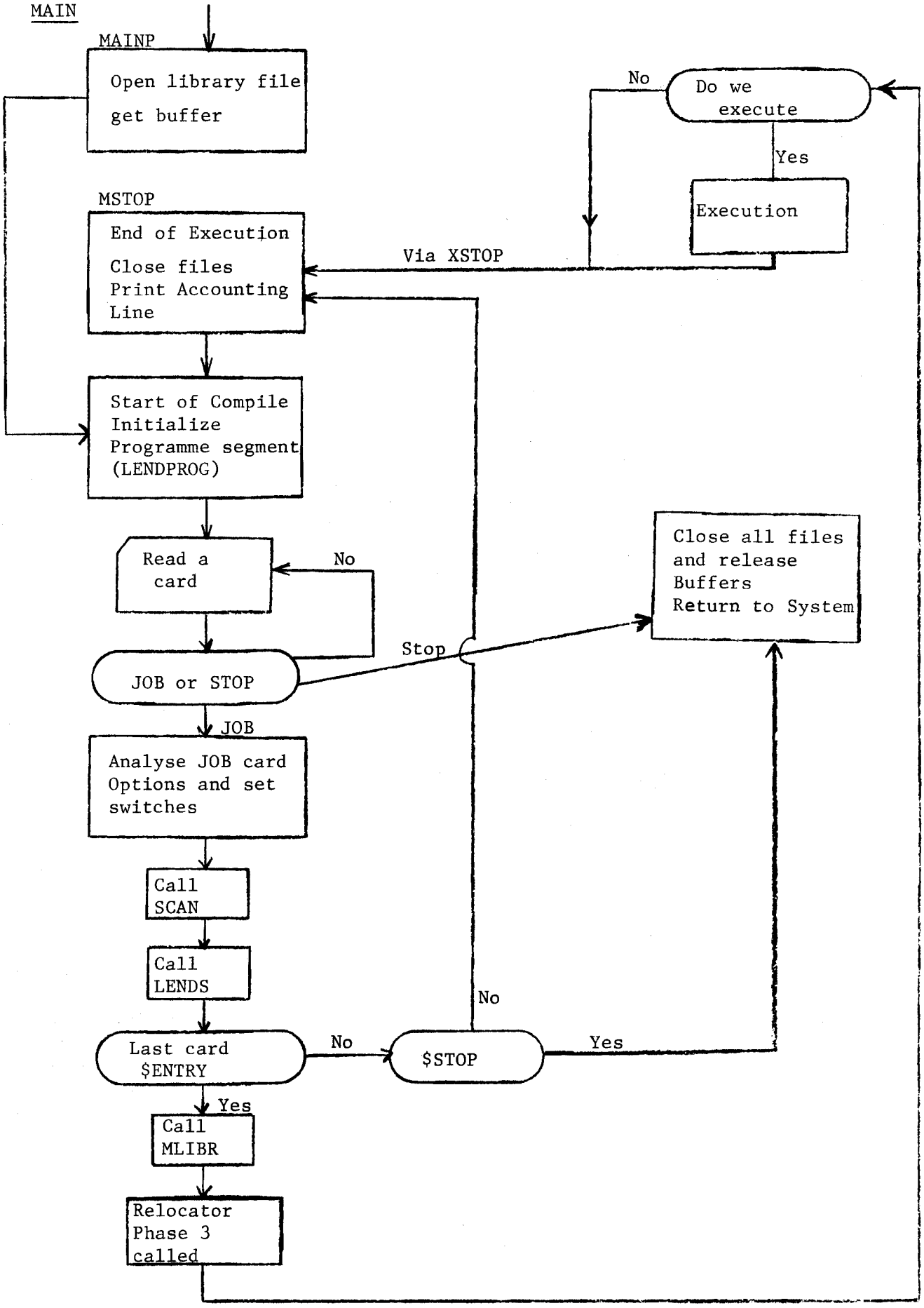

MAIN
MAINP
Open library file
get buffer
No
Do we execute
Yes
Execution
MSTOP
End of Execution
Close files
Print Accounting Line
Via XSTOP
Start of Compile
Initialize
Programme segment
(LENDPROG)
Read a card
No
Close all files and release Buffers Return to System
Stop
JOB or STOP
JOB
Analyse JOB card
Options and set switches
Call SCAN
Call LENDS
No
Last card $ENTRY
No
$STOP
Yes
Yes
Call MLIBR
Relocator Phase 3 called

Figure 6.2.

6.2 Initial Batch Entry (MAINP)

This routine is used only at the beginning of the first WATFOR job. It checks if the user or installation has inserted a DD card for the library data set and opens the data set if present. The required buffer is obtained by issuing a "GETMAIN" where the length is obtained from the library (WATLIB) DCB. Pointers are set up to point to the beginning and end of the buffer (CUFF & CBUFFE).

6.3 End of Batch (MAIN60)

When a $STOP card is encountered by MAIN the 'I have got a card switch' (CIHGACRD) is turned on and control transfers to MSTOP (to print the accounting for the last job compiled and/or executed). Control then transfers to MAIN60 where the $STOP card is printed. The library data set is closed and the buffer is freed. All other data sets are closed by issuing a call to FIOCS. We now transfer to STOP in STARTA which in turn returns to the system.

6.4 Start of Compilation (MAIN)

This portion of coding can be broken into three main parts:

1. Initialization for the complete programme.
2. Job Card analysis.
3. Compilation of the programme.

Following this depending on several error conditions the user programme may or may not execute.

1. Initialization:

   Most of the switches and constants have been described in other parts of the manual. This portion of code in addition to the routine LENDPROG performs the initialization.

   (i) The standard O.S. STIMER and TTIMER macros are used by WATFOR to determine compilation and execution time. A STIMER is issued to indicate start of compilation using the value in MTIMER1.

   (ii) A SPIE macro is used to ignore compiler interrupts 8, 10, and 14.

   (iii) The following constants and switches and registers are initialized:

XERRSWT	error switch - off
XEXECSW	execution switch - off
Reg's 5 & 6	point to object code and symbol table
CMOSWTCH	memory overflow - off
XISNRTN+1	reset timer overflow switch - off
CSRSWTCH	main-line programme entry - off
XISN	statement counter set to zero

Control also transfers to LENDPROG (Section 4.2.13.) for more initialization.

2. Job Card Analysis:

After checking that the last job did not read the $JOB (if it did, don't read another card) read cards until a $JOB card is encountered. Before analysing the possible options the following switches and constants (installation defaults) are set:

CMODESWT	X'00' for BCDIC
	X'06' for EBCDC
CUNDEFSW	undefined variable switch
	X'00' for RUN=NOCHECK
	X'0F' for RUN=CHECK
	X'FF' for RUN=FREE
XNOPDEFZ+1	X'00' for RUN=FREE or CHECK
XNOPDEFN+1	X'80' for RUN=NOCHECK

The following constants are set to one

XEXUFLOW	number of floating point underflows
XEXOFLOW	number of floating point overflows
XFXOFLOW	number of fixed point overflows
XFXDVCNT	number of fixed divide checks
XFLDVCNT	number of floating divide checks
XDVCHKSW	divide check indicator set X'02'
XOVRFLSW	overflow/underflow switch set X'02'

The above settings may be changed by information obtained from the $JOB card or by the programmer by the appropriate library function call (e.g. DVCHK).

Counters for the number of lines per page allowed (XLNCOUNT), number of pages allowed (XPAGES) and the execution time (XTIME) of the user programme are also set to default values (obtained from OPTIONS).

The $JOB card options are now analysed. The present parameters allow the user to specify:

1.	Keypunch used	(KP = 26 or 29)
2.	Lines per page	(LINES = number)
3.	Pages for a job	(PAGES = number)
4.	Maximum execution time	(TIME = number)
5.	Run type	RUN = FREE, CHECK, NOCHECK

These options are described in detail in the WATFOR Implementation Manual.

The coding and logic to process the above parameters are relatively straight forward and will not be described in detail. Several points, however, should be mentioned:

1. If at any point an error is encountered an error message is issued. The parameter in error and any parameters not processed as yet will not be processed. Hence the default values will be assumed.
2. The routine FRIOSCAN (section 5.4.) is used to collect and convert constants used on the $JOB card.

3. Compilation of the Programme:

Another card is now read. If it is a $STOP card control will transfer to the end of batch processor;if not, the CIHGACRD switch is turned on and control transfers to the routine SCAN which maintains control until a $JOB, $ENTRY or a $STOP card is encountered at which time it returns to MAIN. If the 'END' statement was not the last statement processed (use CSRSWTCH) go to the routine LENDS and generate the appropriate action. A check is now made to determine what type of control card caused the return from SCAN. If it is not $ENTRY control transfers to MXSTOP where the end of compile time is determined. We can now possibly proceed with the next job or finish the batch. If it is $ENTRY, the card is printed and a test is made to see if a programme has been compiled (if CGBEG is zero - no programme). Control now transfers to MLIBR (section 6.6.) to process unresolved subroutines from the library. The third phase of the relocator is now invoked to set up the necessary arrays. Checks are now made to determine if the programme and associated data areas are too large and if this is true, the programme will not be executed.

The error switch (XERRSWT) indicates if any fatal errors that might inhibit execution were encountered in the programme. A TTIMER is issued to obtain the compile time. An STIMER is issued to set the maximum execution time and we are now ready to execute the programme. At the end of execution control will transfer to XSTOP.

6.5 End of Execution (MSTOP)

This routine is used at the end of execution of a job, and at the end of compile if the programme did not execute. A TTIMER is issued if the job went into execution to obtain the execution time.

Closing of data sets

The routine FIOCS as used is not serially re-useable or re-enterant. Hence, if files were closed a new copy of FIOCS would be required for each job in the batch. More important is that this would require extra unnecessary time in a batch environment and hence the following approach has been taken.

Each unit is checked to see if it has been opened (i.e. used by a programme). If not no action is taken. If yes, then a check is made if the unit is the "input" or "output" and again if yes no action is taken since the compiler uses these units for its output and input. If the punch unit was used, two blank cards are punched to clear the buffers. This is done by simulating the statement

```
      PUNCHn
n     FORMAT(//)
```

If the unit is a disk or tape utility and it has been used a REWIND is issued on the unit.

Printing The Accounting Line

The compile and execution time are now determined and stored in the output line. The object code space, data areas and free area space are computed and stored. The line is now printed.

6.6 FORTRAN and User Library Subprogramme (MLIBR)

WATFOR allows the user two types of library subprogrammes The standard FORTRAN library functions are available. These are all core resident (section 5.1.) and merely require that the address be resolved. It is also permissable to have a FORTRAN source library (WATLIB) available for the user. In this case WATFOR will compile the subprogramme obtained from the library. The routine MLIBR maintains control over this processing.

MLIBR sets the switch CSUBRDS to indicate that card images will come from the library and saves the present card image in MXCARD.

Using the LLIST of the symbol table B2 is tested if the entry is resolved and if it is the next entry is checked. If not then the 'name' is checked against a table of available FORTRAN library functions. This table contains the name, type and address of the function. If the 'name' is found a check is made to see if the types correspond using B1 of the symbol table entry and the function table type. (An SR-2 error message is given if the types do not correspond.) The address of the function routine is placed in the symbol table and control returns to check the next entry in the LLIST.

If the entry is not found in the function list, a 'FIND' is issued in the source library data set. If the 'name' is not found an SR-2 error message is issued. If the 'name' is found the 'FIND' macro has set up the WATLIB DCB to point to the required member.

Since the CREAD routine can use blocked input the current record pointer (CPOINT) is set to point to end of the buffer (CBUFFE) to force a read. CMODESWT (keypunch mode) is set to X'0C' to specify that the keypunch mode will be the same as the user's. Control now transfers to SCAN and subsequently the statement processors to compile the subprogramme. On return the next item in the LLIST is checked.

When the end of LLIST is encountered the original card image is restored, the read switch reset and control returns to MAIN. Note that if library subprogrammes call other library subprogrammes the 'names' would have been added to the end of the LLIST and hence would be resolved as they are encountered in the LLIST.

Chapter 7

Debugging the Compiler

Certain debugging facilities have been built into the compiler to make our work easier. These facilities would only be used by an installation if it wishes to make major changes or additions to the compiler. It should be noted that all WATFOR's debugging facilities are removed (assembled out) when the compiler is released. This is accomplished by setting certain parameters in the 'OPTIONS' deck.

Option		Release	Debug
&WATTYPE	SETC	'DISTR'	'TEST'
&TRACE	SETC	'OMIT'	'USE'
&SNAPS	SETC	'OMIT'	'USE'

Note: If an installation wishes to include this facility the entire compiler must be reassembled.

Three major debugging aids are used and these are now described.

1. SNAPSHOT

The first aid involves a snapshot (dump) in hexadecimal format. This allows the various processors to display information that they have produced at compile time. For a particular FORTRAN statement this would include a dump of the stack, the ISN code generated (if statement is executable), any object code generated by the processor as well as any symbol table modifications or additions. Finally, at the end of the programme segment the complete object code generated plus the symbol table for the segment is displayed. The relocator is invoked at the end of programme segment to assign storage and fill in B/D addresses. The object code and symbol table are displayed again following this process.

A routine wishing to display information uses the CSNAP macro described below:

```
CSNAP  IDENTIFIER,AREA,n
```

IDENTIFIER - a BCD name to identify the output (max. length 8 characters)
AREA - the address in the programme where the snapshot is to start
n - the number of output bytes.

e.g. CSNAP STACK,0(R9),8

would output

STACK 00080101C1C2C3C4 (a possible first entry in stack).

Each of the processors and routines has inserted CSNAP's to display information pertinent to the module.

If the user wishes to display this data when running a test programme he merely punches the letter 'S' in column 7 of the $JOB card for the programme.

2. TRACE

The second facility involves a /360 assembler trace programme. This programme provides an instruction-by-instruction trace displaying the location of the instruction, the instruction itself, and the contents of the registers and data areas affected by the instruction. (The manual Trace /360 available from the Computing Centre describes the features of this programme.) Placing 'S' in column 7 of the $JOB card initiates the trace routine for the object code generated and any associated routines subject to the restrictions discussed below.

Several problems arose after a period of time while using the Trace programme.

1. The volume of paper generated by the trace was often unnecessary.
2. No convenient facility was available to trace at compile time.
3. Usually a particular statement in the FORTRAN programme was the cause of the error. Hence, we wished to trace particular statements rather than the whole programme.
4. If a bug occurred in a particular processor at a low level (i.e. called by many levels of routines) again the volume of paper generated was prohibitive.
5. Tracing the I/O routine for each or any statement became ridiculous.

The above problems were solved in the following ways:

1. Six new statements, recognized by SCAN, were added to WATFOR. These statements could be inserted in the FORTRAN programme for test purposes.

ONCOMPILE[1]	turn on trace at compile time.
OFFCOMPILE	turn off trace at compile time.
STARTRACE	turn on trace at execution time.
ENDTRACE	turn off trace at execution time.
ONSNAPS	turn on snapshot information at compile time.
OFFSNAPS	turn off snapshot information at compile time.

e.g.

```
STARTTRACE
ONSNAPS
ONCOMPILE
A = B*C + SIN(Z)
PRINT,A
OFFCOMPILE
ENDTRACE
OFFSNAPS
```

would trace both execution and compile time as well as displaying any snapshots generated for the two FORTRAN statements. Hence, the tester can limit his output as well as debugging both compile and execution time routines.

A second scheme was initiated to prevent tracing of debugged routines as well as the I/O routine FIOCS. This scheme requires a special assembler language routine (RELFORM) to be included as the initial entry point to WATFOR. The routine uses two methods to inhibit tracing. The first involves the order of the Linkedit INCLUDE cards. All modules "included" after the include card for RELFORM are not traced.

```
e.g.  INCLUDE   OBJLIB(MAIN)   }  These routines are
      INCLUDE   OBJLIB(SPECS)  }  traced
      INCLUDE   OBJLIB(RELFORM)
      INCLUDE   OBJLIB(FIOCS)  }  These routines are
      INCLUDE   OBJLIB(SCAN)   }  not traced
```

This method allows us to eliminate tracing by processor routines. However, we also did not want to trace certain debugged routines (LOOKUP, OUTPUT, CONSTANT COLLECTOR etc.) located in the communications regions. Hence a table of areas not to be traced is set up in RELFORM. Finally RELFORM transfers control to the compiler.

---

1. In each case only the first four letters need be used.

These methods of selective tracing have allowed us to determine bugs in difficult cases.

3. TEST DECKS

In order to test a compiler a large number of test decks are required. Since one of WATFOR's main design philosophies is to produce good error diagnostics, the test decks should test most of the possible errors that a programmer would make. We have at present approximately 1000 test decks the majority of which test error conditions. These have been collected for the past several years. They include decks used to test the 7040 WATFOR compiler as well as decks generated by our group. Finally, any installations reporting bugs supply us with more test programmes. After the reported bug has been corrected, the programme is added to our test library.

Chapter 8

Conclusion

When writing a programme as involved as WATFOR new ideas, better methods, and faster routines are often suggested by the compiler writing team as well as by our users. However, in our modern computer environment it seems that while ideas should be easy to implement, it always takes longer than anticipated. (In fact, as someone has stated if a change or addition is implemented requiring more than two instructions it should be considered a "major" change to the programme.)

For these reasons the WATFOR group decided in April 1967 to stop all new development on the compiler and release it for use at our installation and others with the requirement that we would attempt to correct any bugs. All new developments were grouped together and basically a new project was started to implement these ideas into WATFOR as a new version of the compiler (Version 1). To avoid confusion we decided to name the new project WATFIV.[1]

Following is a list of new features and changes included as part of WATFIV.

1. Several features of FORTRAN as defined by C28-6515 had not been implemented in April 1966 when we released WATFOR. These features listed below have been included in WATFIV

   (i) Direct Access I/O
   (ii) The NAMELIST statement

2. The SHARE FORTRAN Project proposed that IBM adopt the type CHARACTER as an additional variable type (See SSD 164 C-4653). We have included the CHARACTER variable type as defined by SHARE in WATFIV.
3. A facility has been included to permit an installation to display "English error messages" instead of WATFOR's normal scheme of issuing a coded error message. This of course will require more core to store the messages. For users who do not have the extra core the old scheme may still be used.
4. Since most installation changes to WATFOR involve accounting and JOB card analysis, "handles" have been placed in WATFIV for an installation accounting routine. This routine will be a separate module to be maintained by the installation. This means that our updates shouldn't conflict or cause problems with installation modifications in this area.

---

1. At the time WATFIV was chosen for seemingly obvious reasons (i.e. 5 (FIV) is one greater than 4 (FOR)). However since that time someone has pointed out that WATFIV could stand for WATerloo Fortran IV.

5. We have had many requests regarding WATFOR's dependency on the operating system. Installations often wish to convert WATFOR to run under a different operating system (e.g. DOS). WATFIV will isolate most of system dependent functions to one particular routine so that conversion will be easier. (This will not include the I/O routines FIOCS or DIOCS (Direct Access I/O routine).)
6. The structure of the compiler has been re-organized so that assembly times will not be as long. This basically involves including the extended START area (STARTB) as part of MAIN.

   Also STARTA and COMMR have been changed to be used as DSECTS when they are assembled with any of the statement processors routines. This means that object decks are smaller and hence less space is required for the WATFIV object data set (WATFOR.OBJLIB).
7. An attempt is being made to reduce compile time.

   (i) Most of the algorithms and routines were re-examined for efficiency as well as for how much space they required.[2]
   (ii) The LOOKUP routine for variables has been made faster by introducing a pseudo-hashing scheme.

8. An attempt is being made to compile larger programmes for the same size area. This is accomplished by performing a clean-up on the symbol table for each programme segment by only retaining the GVLIST (Global Variable List). Hence symbol table space is recovered at the end of each programme segment, allowing the user to compile larger programmes as long as he used subprogrammes. The clean-up takes time and hence compile-time is slightly degradated but this should be compensated for by the faster lookups.
9. The object code for DO-loops and subscripting has been improved slightly to give faster execution times.
10. Several new 'OPTIONS' have been added to allow the installation more flexibility.
11. Other suggestions and ideas are being considered and may be implemented if time allows.

---

2. Paul Cress gave us an added incentive to reduce the size of our routines by offering one beer for each 100 bytes of code we could remove. (He still hasn't paid off.)

APPENDIX A - Subprogramme Linkage Conventions in WATFOR

The purpose of this appendix is to explain in detail subprogramme linkage conventions used in WATFOR for those installations who may wish to add their own functions and subroutines to the core-resident library to make them available to users in the way SIN, EXP etc., presently are.

A.1 Subprogramme Calling Sequence

The calling sequence generated by WATFOR may be illustrated by the following example.

Suppose subroutine or function 'rtn' is referenced by a CALL or function reference in a Fortran Statement e.g.

```
(i)  CALL rtn (arg1,arg2,arg3, ..., argn)
(ii) Y = rtn (arg1,arg2,arg3, ..., argn)
```

The calling sequence generated by WATFOR for either is essentially

```
                CNOP    0,4
                L       R3,=V(rtn-START)
                LA      R14,retaddr
                BAL     R1,START(R3)
                DC      AL1(c1),AL3(loc1)
                DC      AL1(c2),AL3(loc2)
Argument list   :
                DC      AL1(cn),AL3(locn)
                DC      AL1(cn+1),AL3(junk)
retaddr         EQU     *
```

where c_i represents a code byte that contains information about the i^{th} element of the argument list and loc_i contains addressing information for argi.

In addition, R13 of the calling programme points to an 18 word OS-type save area.

Thus, on entry to the called programme

- R3+R12 points to the entry point 'rtn' (since R12 always contains A(START) throughout execution of WATFOR).
- R1 points to the argument list located on a fullword boundary in the calling programme.

- R14 contains the return address.
- R13 points to a save area for register storage in the calling programme.

Upon return, the calling programme expects that:

- the result of a function subprogramme (as in (ii) above) is returned in
  - R0 if logical or integer
  - F0 if real
  - (F0,F2) if complex
- its base registers R5-R13 are restored before returning.
- the low-order half of floating register F6 will contain zero as it did on entry. (WATFOR uses F6 to 'double' single precision numbers. e.g. to accomplish D1 = X + D2 where D1,D2 are REAL*8, X REAL*4 WATFOR generates

```
LD      F0,D2
LE      F6,X
ADR     F0,F6
STD     F0,D1    .)
```

In short if a subprogramme uses F6 it should zero it before returning.

The 6 possibilities for code byte c_i and associated meaning of loc_i are:

(i) Simple quantity: c_i = | 0 | 0 | 1 | 0 | k | m | m | m |

- mmm is the 'type' of the quantity as in B1 of the symbol table

  i.e. 000 for LOGICAL*4
  001 for LOGICAL*1
  010 for INTEGER*4
  011 for INTEGER*2
  100 for REAL*4
  101 for REAL*8
  110 for COMPLEX*8
  111 for COMPLEX*16

- k is 1 if the quantity is a constant, temporary, DO-parameter or ASSIGNed variable (i.e. should not be changed by the called subprogramme).
- in this case AL3(loc_i) = AL3(arg_i-START)

(ii) Array name X: c_i = | 0 | 1 | n | n | n | m | m | m |

- mmm is the type of the array (as in (i))
- nnn is the number of dimensions of X
- AL3(loc_i) = AL3(.X-START) where .X is the name of the 'dot routine' for array X (see below).

(iii) Subroutine name R: c_i = | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

- AL3(loc_i) = AL3(addr-START) where addr is a full word containing A(R-START)

(iv) Function or EXTERNALed name F: c_i = | 0 | 0 | 0 | 1 | 1 | m | m | m |

- mmm is the type of the function (as in (i))
- AL3(loc_i) = AL3(addr-START) where addr is a full word containing A(F-START)

(v) Statement Number Argument (multiple return): &n

c_i = | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

- AL3(loc_i) = AL3(addr-START) where addr is the address of the statement labelled n in the calling programme.

(vi) Argument List Terminator

There are 2 cases:- if the called programme is to be a subroutine,

c_i = | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

- if the called programme is to be a function of type mmm,

c_i = | 1 | 0 | 0 | 1 | 1 | m | m | m |

Note that the accompanying adcon contains no useful information.

A.2 Subscripting and the use of the Dot Routine

All array references in WATFOR are done via the so-called 'dot routine' which is constructed by the compiler for each array that is declared in the programme. This dot routine contains information pertinent to the array (e.g. its subscript ranges, address in memory) and is used for indexing into the array and for checking for out of range subscripts.

The format of a dot routine for an array, say X, declared in the programme in which it is defined (i.e. not a dummy parameter) is:

```
        CNOP    2,4        }  Not present if
        DC      CL6'X'     }  RUN = NOCHECK
        CNOP    0,4
.X      BAL     R15,xrtn
        DC      AL1(f),AL3(basaddr-START)
        DC      AL1(s),AL3(number of bytes in array)
        DC      A(d1)      }
        DC      A(d2)      }
        :                  }  only n present
        :                  }
        DC      A(dn)      }
```

Here
- n is the number of dimensions of X
- xrtn is XA1 if n = 1 or XAN if n > 1 (XA1, XAN are in STARTA)
- f = 4n-4
- s = $\log_2 \ell$ where ℓ is the length of an array element, i.e. s is 0,1,2,3,4 for 1, 2, 4, 8, 16 byte elements.
- basaddr is the base address of X i.e. the 'location' of X(0,0, ..., 0) and is calculated as

$$A(X(1,1,\ldots,1)) - \ell \sum_{j=1}^{n} \prod_{i=0}^{j-1} d_i \qquad (d_o \equiv 1)$$

- d_i are the actual dimensions declared.

The dot routine constructed for an array name which is a dummy parameter in a subprogramme entry list has the format:

```
        CNOP    0,4        }  Not present if
        DC      CL6'X'     }  RUN = NOCHECK
        CNOP    2,4
.X      BALR    R15,0
        L       R3,8(,R15)
        B       START(R3)
        DC      A(*-*)
```

where the last adcon is filled in by WATFOR's entry sequence routine XENT (cf. 'prologue') with the address of the dot routine for the actual

argument obtained from the call. (See coding generated by LINKR, section 4.2.)

The dot routines are contained in the programme segment local data area whereas the array storage is in the 'array area'. The use of the dot routine for subscripting is shown by the following example which indicates that WATFOR handles subscripting much like a subroutine call. The object code for statement Y = X(I,3,R) is (RUN = NOCHECK)

```
                    CNOP    0,4
                    BAL     R14,.X
                  ┌ DC      AL1(X'4C'),AL3(I-START)
Subscript List    { DC      AL1(X'4C'),AL3(=3-START)
                  └ DC      AL1(X'0C'),AL3(R-START)
                    LE      F0,START(R3)
                    STE     F0,Y
```

Note that the result of the call to .X is that the address (relative to START) of the element in X, specified by the subscripts, is returned in R3.

In general, an array reference for an array with n dimensions is made by the code:

```
                    CNOP    0,4
                    BAL     R14,dotrtn
                  ⎧ DC      AL1(t1),AL3(s1-START)
                  ⎪ DC      AL1(t2),AL3(s2-START)
Subscript List    ⎨ .
                  ⎪ :
                  ⎪ DC      AL(tn),AL3(sn-START)
                  ⎩
```

where s_i is the name of the actual subscript and t_i is a code which describes its type (used by XA1 and XAN) as follows:

INTEGER*4 subscript has t_i = X'4C' (=AL1(XSSI-XSSD)

INTEGER*2 subscript has t_i = X'3A' (=AL1(XSSH-XSSD)

REAL*4 subscript has t_i = X'0C' (=AL1(XSSR-XSSD)

REAL*8 subscript has t_i = X'00' (=AL1(XSSD-XSSD)

Note that calling the dot routine of a dummy array merely passes control up one level to the dot routine of the calling programme, continuing until the dot routine of the array is reached in the programme in which the array was declared. (This in essence means that WATFOR ignores, at execution time, the dimensions declared for dummy arrays and

is the reason why object-time dimensions don't work exactly as described on page 93 of C28-6515-5).

Because a subprogramme has no readily available method of finding the address of the true dot routine or the address of the array itself, the routine X1STELT (in STARTA) was written. The inputs to X1STELT are:

- 4 times the number of dimensions of the array in R0
- A(dotrtn-START) in R3

Call X1STELT by BAL R1,X1STELT (X1STELT returns on R1). The outputs are:

- R15 contains the address +4 of the true dot routine
- R2 contains the total length in bytes of the array
- R3 contains the address (relative to START) of the first element of the array.

As can be seen there are several ways by which a hand written subprogramme can reference arrays.

A.3 Adding Subprogrammes to WATFOR

There are several steps which should be followed.

1. Code the subprogramme to use the conventions described above. Declare each entry point with an ENTRY statement.
2. If the subprogramme uses routines in STARTA, for convenience place the coding in source module FUNCTION since csect STARTA is copied (assembler COPY feature) in that module. (There may be some difficulty if many new entry points are added to FUNCTION since the assembler allows a maximum of 100. It is a fairly simple matter, however, to create another source module which uses the basic structure of FUNCTION.) Reassemble FUNCTION.
3. If the step 2 is not necessary or convenient, assemble the routine separately.
4. To make the new routine known to WATFOR update the list of known in-core routines in source module MAIN, at label FBEG (sequence number 00005880) using the FLIST or SLIST macro for a function or subroutine subprogramme respectively. This must be done for each entry point should there be multiple entry points to the routine. (The list is not ordered.) Reassemble MAIN.
5. Re-linkedit WATFOR, including any new object modules.

A problem exists if an assembly language subroutine calls a Fortran subprogramme since WATFOR will not be able to discover that it is necessary to compile this routine.

Moreover a missing entry point will occur in the linkedit of the compiler in step 5 above if the routine is EXTRNed. For example consider the following programme set up

Presented by Programmer

```
$JOB
  .
  .
  .
CALL ASPRG(A,B)   ASPRG is an incore routine added to
  .               WATFOR as described above.  ASPRG
  .               contains a call to FPROG
  .
END
$ENTRY
```

On direct access library WATLIB

```
SUBROUTINE FPROG(X,Y,Z)
  .
  .
  .
RETURN
  .
  .
  .
END
```

There are several ways of getting around this problem fairly simply. Force WATFOR to know that FPROG is required by the programme by EXTERNALing it or including a CALL to it which will not be executed. Then include the name of FPROG in the call to ASPRG or have an entry point in ASPRG which is called with the name of FPROG as argument to initialize a linkage adcon in ASPROG.

For example

```
$JOB
  .
  .
  .
EXTERNAL FPROG
  .
  .
  .
CALL ASPRG(A,B,FPROG)
  .
  .
  .
END
$ENTRY
```

or

```
$JOB
  .
  .
  .
  GO TO 2
  CALL FPROG
2 CONTINUE
  .
  .
  .
  CALL ASPROG(A,B,FPROG)
  .
  .
  .
  END
  $ENTRY
```

A.4 Notes and Cautions

1. WATFOR assumes that R12 always contains A(START). Thus coding such as B START(R3) is really the same as B 0(R3,R12) and installation written subprogrammes can refer to START by including the statement USING START,12 since, if the routine is called by a WATFOR-compiled programme, this will be true. (START is EXTRNed as XTART in deck MAIN.)
2. The assembly language subprogramme may simulate a multiple return, e.g. RETURN I, by searching down the calling programmes argument list for the I^{th} statement number argument (taking care not to run off the end of the argument list). Obtain the address of the statement number and

   (i) store it in the R14 position of the calling programme's save area and do a normal return

   or (ii) with the address of the statement number in, say R14, reload the calling programmes base registers R5-R13 and return on R14.

   For example, suppose R1 points to the argument list entry of the I^{th} statement number argument, in case (ii):

```
L  R14,START(R1)     Load A(stat. num-START)
L  R13,4(,R13)       Restore calling save area pointer
LM R5,R11,40(R13)    Restore calling registers
B  START(R14)   or { LA  R14,START(R14) }  Return
                   { BR  R14            }
```

3. WATFOR does not provide automatic function typing, i.e. the programmer must declare library routines

   e.g. REAL*8 DEXP

   Thus installations must advise their users that any functions added to WATFOR must be declared if the implicit typing rule will not supply the proper function type.
4. Logical values are treated in WATFOR in the following way:-

   - the 1^{st} byte only of the variable contains the logical value, i.e. the low-order 3 bytes of a LOGICAL*4 variable are essentially ignored by WATFOR in logical operations.
   - .TRUE. is X'FF'
   - .FALSE. is X'00'

5. A hollerith constant in a CALL statement is treated as a REAL*4 vector, i.e. a dot routine as described above is created for each hollerith constant. It should be noted that WATFOR right pads hollerith constants with blanks to a multiple of four bytes; e.g. for CALL NAME ('ABCDE'), 'ABCDE' is treated exactly like a real vector with 2 elements. (This was done for compatability with 7040 WATFOR.)

6. There are a number of useful macros available which may be used by assembly subprogrammes. See section 5.1.2. of this manual or the coding in deck FUNCTION.
7. WATFOR handles specification interrupts at execution time by the routine XRUPT in STARTA. Boundary alignment of operands is treated here and this routine uses R14 as a work register. Thus user routines should be cautious on problems that might involve improper operand alignment or other specification interrupts.
8. Note that there is a difference in WATFOR's treatment of the following statements. (A is a vector):

```
CALL RTN(A(1))
CALL RTN(A)
```

   In the former, A(1) is the name of a single simple quantity; in the latter, A is the name of a collection of quantities. The calling sequences generated by WATFOR reflect this distinction. (See Section A.1 above.)
9. Register usage:

   (i) R0,R1,R2,R3,R4,R14,R15,F0,F2,F4, may be used by the called subprogramme without restoring them upon return. However if subscripting is done within the routine using the dot routine method, it should be noted that R2,R3, R4,R15,F4 are used by the subscripting routines XA1, XAN.
   (ii) In addition X1STELT uses R0 and R1.
   (iii) Be careful of F6 and R12.
   (iv) The calling routine (if compiled by WATFOR) uses R11 as a programme base register and the value in R11 points to the ISN (halfword) of the statement which contains the call. R5-R10 cover the local data area of the calling routine.

10. Assembly language routines can very easily clobber the compiler if arguments are used improperly.
11. Installations should be careful to keep their entry and return sequence coding as separate as possible from the body of the routine being added since some of the conventions mentioned in this Appendix will be changed in Version 1 of WATFOR.