JM6/Tech. Lib.

NASA CR-
147797

HAL/S-FC & HAL/S-360
Compiler System
Program Description

IR-182-1

13 May 1976

(NASA-CR-147797) HAL/S-FC AND HAL/S-360 COMPILER SYSTEM PROGRAM DESCRIPTION (Intermetrics, Inc.) 839 p HC $21.25 N76-27884
CSCL 09B Unclas
G3/60 45760

Prepared
by
Intermetrics, Inc.

National Aeronautics and Space Administration
LYNDON B. JOHNSON SPACE CENTER
Houston, Texas

HAL/S-FC & HAL/S-360
Compiler System
Program Description

IR-182-1

13 May 1976

Prepared
by
Intermetrics, Inc.

National Aeronautics and Space Administration
LYNDON B. JOHNSON SPACE CENTER
Houston, Texas

FOREWORD

This document was prepared for IBM Federal Systems Division, Houston, Texas, under purchase order #479270-Z-44 Alteration 2.

The HAL/S-FC and HAL/S-360 Compiler System -- Program Description was prepared by the staff of Intermetrics, Inc. Technical direction by Dr. Bruce Knobe, typescript by Valerie Cripps.

1.0 INTRODUCTION

1.1 Scope of Document

This document supplies information necessary for maintaining the HAL/S-360 and HAL/S-FC compilers. It is intended as a companion to the source listings. A large portion of the required material can be found in the Intermetrics' documents:

HAL/S-FC Compiler System Specification, IR-95-3.

HAL/S-360 Compiler System Specification, IR-60-3.

HAL/S-FC Compiler System Functional Specification, IR-59-4.

HAL/S-360 Compiler System Functional Specification, PDRL # IM004.

and in the IBM Federal Systems Division documents:

Interface Control Document: HAL/FCOS, Revision 3.

Interface Control Document: HAL/SDL, Revision 5.

In order to eliminate the problem of maintaining multiple up-to-date copies of the same information, matieral available in the above documents is in general not duplicated here.

Familiarity with the above documents is presumed throughout this document. References to the above documents appear in appropriate places and occasionally short sections have been reproduced here for convenience or clarity of presentation.

This manual is for the HAL/S-360[1] and HAL/S-FC[2] compilers and their associated run time facilities which implement the full HAL/S language[3]. The compilers are designed to operate "stand-alone" on any compatible IBM 360/370 computer and within the Software Development Laboratory (SDL) at NASA/JSC, Houston, Texas.

[1] HAL/S-360 User's Manual, 10 May 1976, IR-58-13.
[2] HAL/S-FC User's Manual, 10 May, 1976, IR-83-8.
[3] HAL/S Language Specification, 24 November 1975, IR-61-7.

Table of Contents

Table of Contents (Cont'd.)

Table of Contents (Cont'd.)

Table of Contents (Cont'd.)

Table of Contents (Cont'd.)

Table of Contents (Cont'd.)

Table of Contents (Cont'd.)

1.0 INTRODUCTION

1.1 Scope of Document

This document supplies information necessary for maintaining the HAL/S-360 and HAL/S-FC compilers. It is intended as a companion to the source listings. A large portion of the required material can be found in the Intermetrics' documents:

HAL/S-FC Compiler System Specification, IR-95-3.

HAL/S-360 Compiler System Specification, IR-60-3.

HAL/S-FC Compiler System Functional Specification, IR-59-4.

HAL/S-360 Compiler System Functional Specification, PDRL # IM004.

and in the IBM Federal Systems Division documents:

Interface Control Document: HAL/FCOS, Revision 3.

Interface Control Document: HAL/SDL, Revision 5.

In order to eliminate the problem of maintaining multiple up-to-date copies of the same information, matieral available in the above documents is in general not duplicated here.

Familiarity with the above documents is presumed throughout this document. References to the above documents appear in appropriate places and occasionally short sections have been reproduced here for convenience or clarity of presentation.

This manual is for the HAL/S-360[1] and HAL/S-FC[2] compilers and their associated run time facilities which implement the full HAL/S language[3]. The compilers are designed to operate "stand-alone" on any compatible IBM 360/370 computer and within the Software Development Laboratory (SDL) at NASA/JSC, Houston, Texas.

[1] HAL/S-360 User's Manual, 10 May 1976, IR-58-13.
[2] HAL/S-FC User's Manual, 10 May, 1976, IR-83-8.
[3] HAL/S Language Specification, 24 November 1975, IR-61-7.

1.2 Outline of the Document

The HAL/S compiler system consists of:

1) A sub-monitor, coded in assembly language which interfaces the rest of the compiler to its operating environment. The rest of the compiler is written in XPL[1].

2) Phase 1 of the compiler which performs lexical, syntactic, and semantic analysis passing the accumulated information along to subsequent phases. Phase 1 also produces an annotated source listing.

3) Phase 1.5 of the compiler which performs machine independent optimizations.

4) Phase 2 of the compiler which performs code generation and assembly for either the IBM 360 (HAL/S-360) or IBM AP-101 (HAL/S-FC).

5) Phase 3 of the compiler which generates a set of simulation tables to aid in run time verification.

6) HALLINK which augments the function of the normal linkage editor.

7) A comprehensive run-time library which provides an extensive set of mathematical, conversion, and language support routines.

[1] McKeeman, Horning, and Wortman, A Compiler Generator, Prentice-Hall, Englewood Cliffs, N.J. (1970).

Section 2 provides an overview of the compiler and the run time environment it expects.

Section 3 provides a detailed description of the data structures used by more than one phase.

Section 4 provides a detailed description of the data and subroutines in Phase 1.

Section 5 provides a detailed description of the data and sub-routines in Phase 2 of HAL/S-FC and where necessary, a second description for the HAL/S-360 routine.

Section 6 provides a complete discussion (data and procedures) of Phase 1.5 - the optimization pass.

Section 7 discusses the libraries.

Section 8 discusses HALLINK.

Section 9 provides details for the sub-monitor including flow diagrams.

Section 10 discusses the real time simulation facility available in HAL/S-360.

Section 11 discusses the macro libraries used for writing AP-101 or 360 assembly language programs compatible with HAL/S compiler generated code.

Section 12 deals with the routines available for accessing the SDF tables produced by Phase 3.

Section 13 explains those features which Intermetrics added to the standard 360 XPL implementation.

This document was compiled over a long period of time. Material was acquired from many different people and several internal documents. Because of these factors, the level of detail varies greatly. An attempt was made to define a reasonable level of documentation, the level depending on the importance and complexity of the thing to be documented. When more detailed material already existed, however, it was included.

1.3 Status of Document

This document plus the documents mentioned in Sections 1.1 and 1.2 plus the source code comprise the complete maintenance documentation for the HAL/S-FC and HAL/S-360 compilers. This publication documents release 10 of the HAL/S-FC compiler and release 14 of the HAL/S-360 compiler.

2.0 OVERVIEW OF THE HAL/S SYSTEM

2.1 Once Over Lightly

HAL/S is a large sophisticated language and its implementation on the AP-101 and 360 computers produce very high quality translations. It is no surprise, therefore, that the compiler is a large multi-pass design. The overall compiler can be broken into four phases:

Phase 1 inputs the source language and does a syntactic and semantic analysis generating the source listing, a file of instructions in an internal format (HALMAT) and a collection of tables to be used in subsequent phases.

Phase 1.5 massages the code produced by Phase 1, performing machine independent optimization.

Phase 2 inputs the HALMAT produced by Phase 1 and outputs machine language object modules in a form suitable for the OS-360 or FCOS linkage editor.

Phase 3 produces the SDF tables.

The four phases described above are written in XPL, a language specifically designed for compiler implementation. It is essential that the reader be familiar with most of the contents of the book, "A Compiler Generator", by McKeeman, Horning and Wortman, which describes the XPL compiler writing system. The XPL compiler (XCOM) requires more sophisticated interaction with the operating system than that provided in the XPL language; thus, the compiler (written in XPL) is augmented by a sub-monitor (written in assembly language). The HAL/S compiler has a substantially larger but conceptually similar sub-monitor. Thus, the compiler itself is built of four phases written in XPL plus a sub-monitor written in assembly language.

In addition to the compiler, there is a large library containing all the routines that can be explicitly called by the source language programmer plus a large collection of routines for implementing various facilities of the language (e.g. matrix operations, I/O, etc.). These routines are written in the assembly language of the target machine.

Certain information only becomes available at the link step of a job. Since the OS 360 linkage editor is not capable of performing all the functions required, it is augmented by HALLINK; this step is not required on the flight computer where the FCOS linkage editor is more closely aligned with the HAL/S compiler's object modules.

HAL/S has substantial facilities for doing real time programming. These facilities are intended for use on the flight computer where they are supported by the operating system. In order to allow testing of such programs using HAL/S-360, a real time executive has been produced to simulate flight computer real time in the HAL/S-360 environment.

A considerable quantity of assembly language has been written to interface with HAL/S object code (e.g. the libraries). To facilitiate this process, a library of macros has been produced for the AP-101 and another for the 360.

The above material constitutes the complete HAL/S system. In addition to that system, we also describe some changes made in the XPL language to facilitate construction of the HAL/S compiler.

2.2 A Firm Foundation

As described in Section 2.1, the HAL/S compiler is made up of separate modules, each module performing a distinct function in the compilation process. The relationships of the various modules in the compiler to each other and to the compiler environment are shown in Figures 1 and 2. The five modules of the compiler (sub-monitor,Phase 1, Phase 1.5, Phase 2, Phase 3) are described in more detail in the following sub-sections.

2.2.1 The Sub-monitor

The sub-monitor is the controlling module in a compilation. It performs all sequencing and control operations.

The sequencing function of the sub-monitor directs the compilation by deciding which of the other modules are in the computer memory. The sub-monitor makes use of overlay techniques to make maximum utility of available memory. The sub-monitor supervises loading and execution of the other modules and passes any required information between the modules.

The control function of the sub-monitor handles all interfaces between secondary modules in memory and the operating system under which the entire compiler runs. These interface functions include all Input/Output operations, all memory management, and all special requests to the operating system such as time-of-day information.

The sub-monitor is written in OS/360 Basic Assembler Language.

2.2.2 Phase 1

The basic design cf phase 1 started with the XCOM design. The scanner routine has been replaced by a much more complicated routine to handle the multi-line format that HAL/S supports and an entire new module, the output writer, was added to produce the indented, annotated, multiline HAL/S source listing. The MSP parser has been replaced by a LALR parser. Notice that since both MSP and LALR parsers reduce the handle, the rest of the compiler does not care which parser is being used. Anybody working on the parser should first familiarize himself with the work of DeRemer ("Simple LR(h) Grammars", Comm. ACM 1971) and Lalonde (CSRG Report #2, University of Toronto).

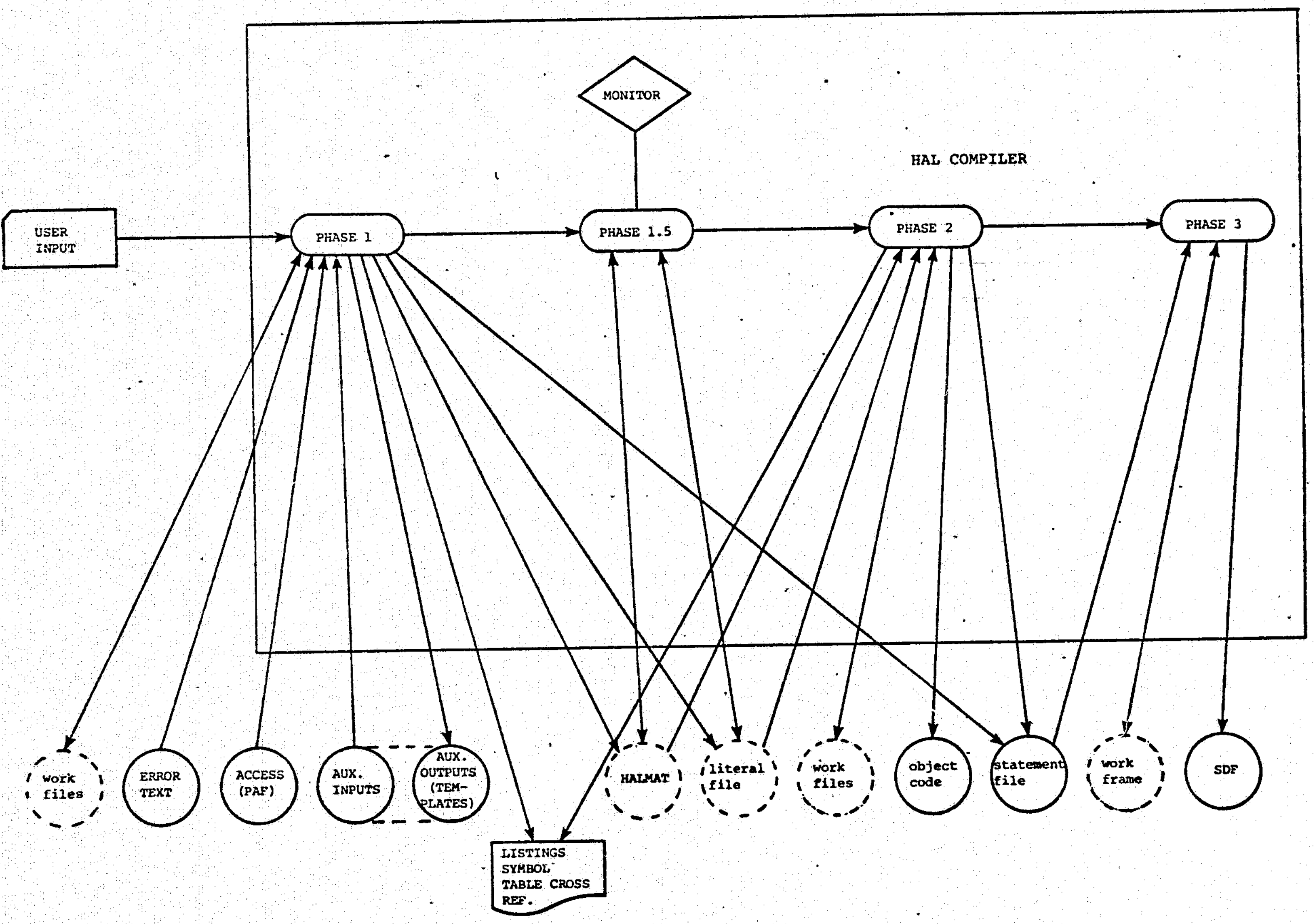

MONITOR
HAL COMPILER
USER INPUT
PHASE 1
PHASE 1.5
PHASE 2
PHASE 3
work files
ERROR TEXT
ACCESS (PAF)
AUX. INPUTS
AUX. OUTPUTS (TEM-PLATES)
LISTINGS SYMBOL TABLE CROSS REF.
HALMAT
literal file
work files
object code
statement file
work frame
SDF

Figure 1

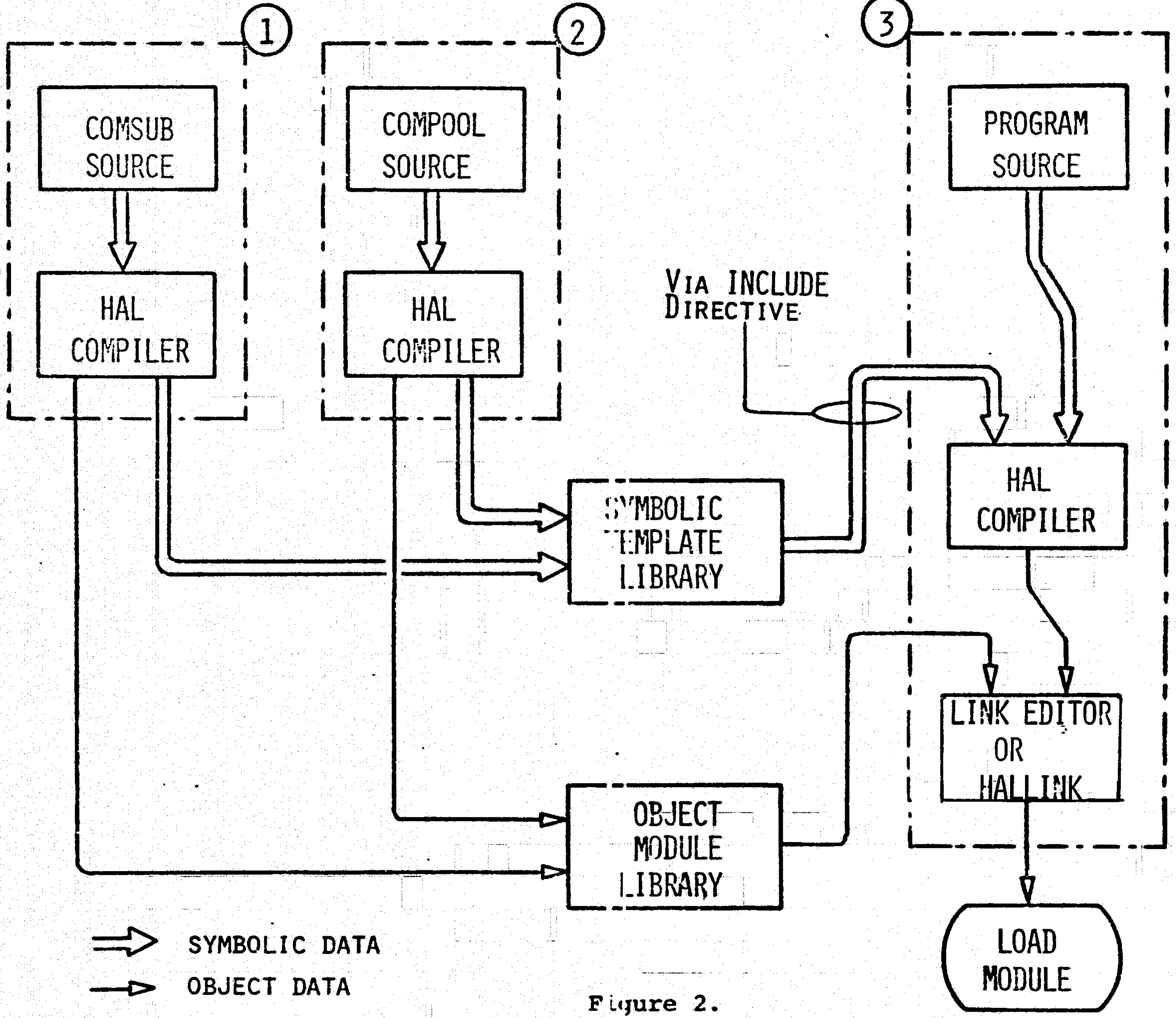


Figure 2.

HAL COMPILATION SYSTEM

Phase 1 performs all syntactic and semantic analysis of the user's HAL/S source statements. This analysis is driven by a parsing system which generates a complete parse of the input. The parsing algorithm detects and identifies all syntax errors in the source statements and makes information generated as a result of the parse available to other sections of Phase 1.

Phase 1 is responsible for the identification of all compiler directives and for the proper implementation of the facility which allows separate compilation of COMPOOLs, COMSUBs, and PROGRAMs.

This separate compilation facility is illustrated in Figure 2. The boxes labeled 1 through 3 each identify a separate Unit of Compilation. A Unit of Compilation is the minimum element of the HAL/S language which may be compiled separately.

Units labeled 1 and 2 illustrate the system which is implemented by the compiler to allow separate compilation of COMPOOLs and external PROCEDUREs and FUNCTIONs (COMSUBs). This system allows the compiler to perform complete static verification of all data types and formal parameters even in PROGRAMs (Unit 3) which reference separately compiled Units. This system is implemented by producing a symbolic template for each Unit 1 or Unit 2 compilation as well as any object code. When a PROGRAM makes reference to one of these separate Units, the symbolic template must be INCLUDE'd (identified by an INCLUDE compiler directive) by the programmer. Phase 1 automatically generates these templates whenever a Unit of Compilation of type 1 or 2 is compiled. The templates are compatible with standard INCLUDE library formats.

Phase 1 is also responsible for production of the source listing and the symbol table/cross reference table listing. Phase 1, written in the XPL language, consists of four distinct parts:

1. The Scanner

2. The Syntax Analyzer

3. The Semantic Analysis Routines

4. The Listing Synthesizer

Figure 3 illustrates the organization of Phase 1 in more detail.

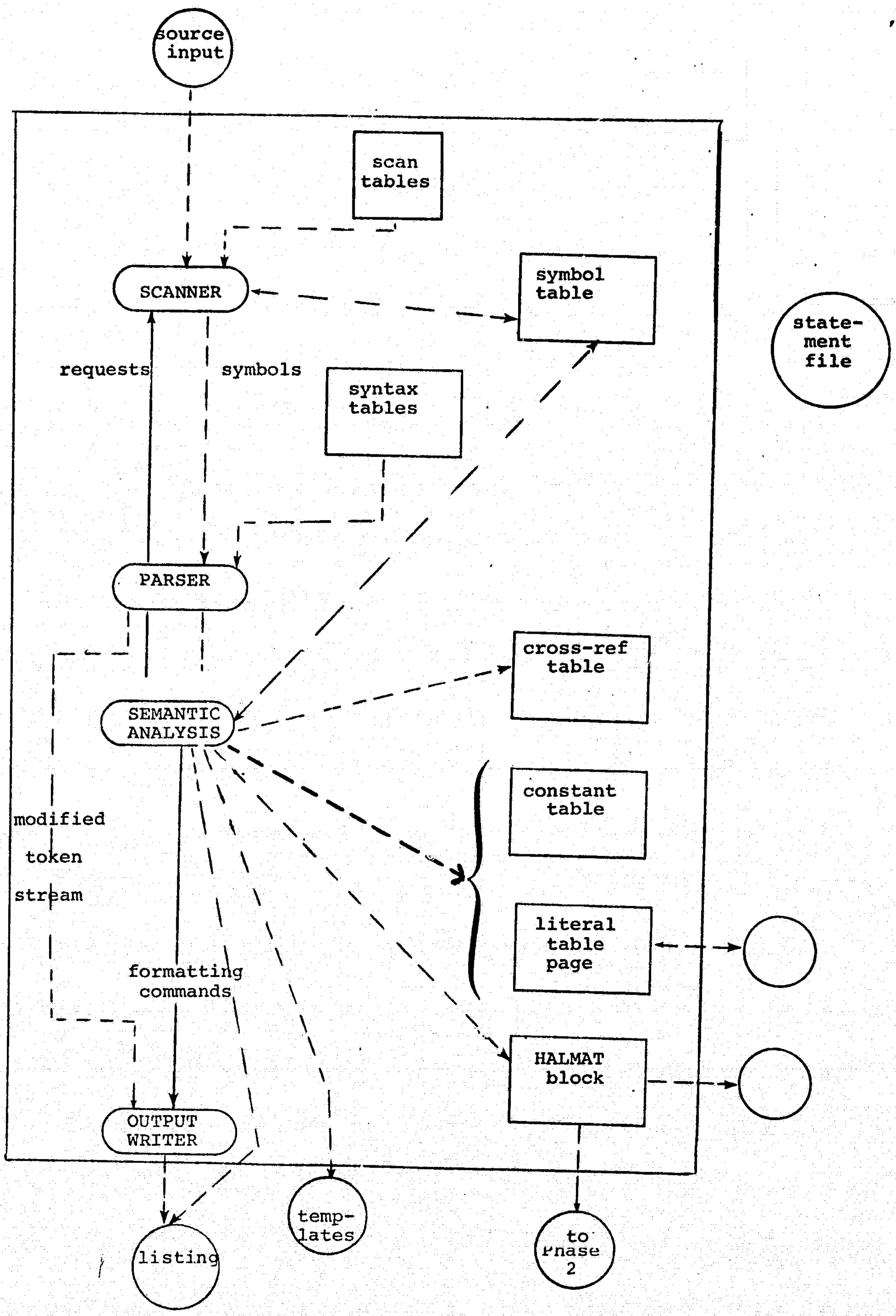


Figure 3: Phase 1 Organization

The Scanner. The Scanner is sometimes called the Lexical Analyzer. It scans the sequence of characters that comprise the source input (letters, digits, punctuation, spaces) and generates a stream of tokens which are meaningful symbols to the Syntax Analyzer, (e.g. reserved words, identifiers, literals, and other terminals). It discards the semantically irrelevant text and handles embedded comments. The proper interpretation of multi-line input is done in the Scanner.

Each symbol is converted to an internal "token" in a simplified format so that the analyzer is presented with a stream of uniform symbols. This permits the rest of the compiler to operate in an efficient manner using fixed length numerically-formatted data instead of variable length character strings. The Scanner is called upon by the Syntax Analayzer as needed to deliver the next token from the input stream.

The requirement for a scanner module rather than the much simpler standard XPL scanner is generated by the multi-line HAL/S input format and the more complicated grammer. The HAL/S source statements are originally entered into the compiler in the form of card images. The text of the statements occupies columns 2-80.

Column 1 is reserved for defining the type of the individual card as follows:

'C' in column 1 indicates a comment card. The contents of the card will be ignored by the compiler.

'D' in column 1 indicates a compiler directive card. Compiler directives inform the compiler of user requests for specific compilation features.

'M' in column 1 indicates the main line of a HAL/S statement. Columns 2-80 of the card may contain HAL/S statement text.

'E' in column 1 indicates the exponent line of a HAL/S statement. Columns 2-80 of the card may contain HAL/S statement text. These cards may only occur in association with an 'M' card.

'S' in column 1 indicates the subscript line of a HAL/S statement. Columns 2-80 of the card may contain HAL/S statement text. These cards may only occur in association with an 'M' card.

'ƀ' blank in column 1 will be treated by the compiler as if it were an 'M'.

All other characters occuring in column 1 are treated as errors. Such illegal characters will cause the card on which they occur to be treated as a comment card. The compiler also flags any illegal sequence of cards as an error. The HAL/S compilers accept user input in single line or multi-line form as described in the HAL/S Language Specification.

The scanner reads source statements from either the normal source (SYSIN) or from an INCLUDE library. An include library contains auxiliary source inputs that may be called in by user requests. The source to be included may be either user-written source statements or template data generated by the compiler for COMPOOLS or COMSUBs. The INCLUDE library takes the form of a partitioned data set. An individual member of the data set is the minimum data which can be INCLUDE'd.

In addition to its principal input function of reading source programs, the scanner has a secondary function of reading the Program Access File (PAF). This file contains information used by the compiler to assign ACCESS rights to individual users. The structure of the data set is a partitioned organization with each member specifying the ACCESS rights for one Program Identification Name (PIN).

The scanner also has an output function. Since the primary source listing is completely reformatted by the compiler, an optional secondary source listing may be requested which lists the original card images as they were input to the system.

The Syntax Analyzer. The Syntax Analyzer decomposes the input stream as delivered by the scanner to determine if it is legal according to the grammar of the language. Once the parser verifies the syntactic correctness of the input, control is passed to the appropriate semantic analysis routine.

The parse is conducted using the table-driven algorithm of DeRemer and Lalonde.

The Semantic Analysis Routines. Once a complete syntactic check has been performed and the format identified, a semantic routine is invoked. Given the particular construct and access to the compiler tables, the analysis routine checks for semantic correctness and then interprets the meaning. The result of this interpretation is some action taken by the compiler to properly implement the language construct in question. This action may range from adding information to the symbol table to generating some intermediate code language elements (HALMAT). The HALMAT is a machine independent representation of the program being compiled. It is used to drive the code generation process. The HALMAT is further discussed under the topic of internal compiler data transfer.

In addition to its principal analytic function, the semantic analysis phase also adds useful information to the source listing. Specifically:

a) Block Summaries. At the close of each PROCEDURE, TASK, PROGRAM, FUNCTION, or UPDATE block, a summary of interactions between the block being closed and the outer scope in which the block is nested. The information includes both variable and block references (e.g. a block summary for a PROCEDURE lists all variables used in that PROCEDURE and any code blocks referenced by that PROCEDURE).

b) Program Layout. At the close of any PROGRAM, a summary of all blocks contained within the PROGRAM. This summary lists the name and type of each block and will indicate by indentation, the nesting relationships which exist between the blocks.

The semantic analysis module is also responsible for producing templates for COMPOOLs and external procedure COMSUBs. Whenever a COMPOOL or COMSUB is compiled, the HAL/S compiler produces a symbolic template of the compiled module. Refer to Figure 2 for a graphic representation of the compilation process. The templates generated in this manner serve to define all interfaces between the COMPOOL and COMSUB's and the HAL/S programs in which they are used. The templates are generated to be compatible with the INCLUDE library.

On recompilation of a COMPOOL or COMSUB a mechanism is provided to generate a new template only when the old template needs to be changed.

The Output Writer. At appropriate points in the analysis, the Output Writer is given control. This routine generates the fully annotated primary source listing by synthesizing the source statements. The synthesis is driven by the tables and other data generated during syntactic and semantic analysis.

The requirement for an output writer module rather that the simpler existing XPL system is generated by the format of the HAL/S primary source listing. This listing provides standard, automatic annotation to enhance the readability of the HAL/S source code. It allows each programmer to enter his programs in free-form input consistent with his own coding preferences. The compiler edits the input during compilation into a standard listing form so that all program listings observe the same coding rules.

Although original HAL/S source input is in the form of card images, the compiler treats the input as a continuous stream of information. Elements of the source listing are generated statement-by-statement, regardless of the original input form.

The editing performed by the compiler includes expansion of any single line HAL/S input into full multi-line form, the addition of annotation marks (overpunches, structure and array brackets), and the logical indenting of statements.

The annotation generated by the compiler is in the form of marks supplied to indicate the type or organization of individual symbols. The marks are generated as follows:

Overpunches - Variables of type vector, matrix, character, bit, or structure appear in the listing with a characteristic mark above the variable name as in $\overset{*}{M}$ for a matrix. The marks are:

* for matrix,

- for vector,

, for character,

. for bit or boolean,

+ for structure.

Brackets — Variables which have dimensioned array or structure organization are enclosed in brackets:

[A] for arrays,

{S} for structures.

Bracketing occurs in addition to overpunching.

Underlining — All REPLACE variables are underlined when they appear in the listing,

e.g. REPLACE A BY "B";

$C = \underline{A} + D;$

Statement indentation is done to highlight the logical construction of the program. In general, the more deeply a statement is indented, the deeper it is in the logical construction of the program. The indentation performs alignment of associated statements (e.g. END and CLOSE statements are indented identically as their respective DO or PROCEDURE statements.)

The primary source listing identifies each HAL/S statement with a statement number. The listing also identifies program blocks by listing the name of the block in which a statement occurs in the right margin associated with that statement.

Cleanups. In addition to the four major modules described above, phase 1 also has a collection of cleanup routines which append additional material to the listing. In particular, they produce:

a) Symbol Table Listing. A display of the complete symbol table generated during the compilation. The table is sorted alphabetically and identifies each user-defined symbol by name. The table identifies all attributes of the symbols, such as type, array/structure size, matrix/vector size, character string length, precision, etc.

b) Cross Reference Table Listing. In the Symbol Table Listing, a display of the complete cross reference map for each symbol defined. This table indicates, by number, the statements in which individual symbols appeared in the compilation. In addition, the listing indicates the type of reference made to the symbol by distinguishing between assignment, simple reference, and use as a subscript. Also, the cross reference listing summarizes total usage of variables (e.g. if a variable is declared, but never used, the listing will indicate this condition). If the usage summary indicates that a variable is referenced but never assigned a value, the compiler will flag this condition as an error.

c) Replace Text Listing. For each variable defined to be a REPLACE variable, the compiler lists the text that was substituted for the variable.

d) Error Message Summary. When compilation errors were detected, the compiler already inserted an error message in the primary source listing at the point of detection. At the end of the primary source listing, a summary of errors is printed indicating which statements in the compilation received such error messages.

2.2.3 Phase 1.5

Phase 1.5 attempts machine independent optimizations on the HALMAT. Since an understanding of Phase 1.5 is not necessary for the rest of the compiler, it is treated as a separate topic after the discussion of phase 2. At present, phase 1.5 eliminates common subexpressions, folds constants, eliminates unnecessary matrix transpose operations and reduces the strength of some operators. Long term plans call for a substantial extension of these facilities. Before doing any work on phase 1.5, the Intermetrics Report, Common Subexpression Recognition, IR #127-1 (7 July 1975) should be carefully studied.

2.2.4 Phase 2

By the end of phase 1.5, an optimized machine independent representation of the program exists in the form of HALMAT plus tables. Phase 1 and 1.5 are identical for the FC and 360 compilers. Phase 2 translates the HALMAT into object modules using a three pass design.

Pass 1 allocates storage for data and translates to a second intermediate code resembling 360 machine language. Pass 2 resolves all forward address references and compactifies the code by eliminating unnecessary base register loads on the 360 and using short form addressing on the AP-101. Pass 3 produces object modules for either the 360 or AP-101.

Phase 2 for the AP-101 is an adaptation of phase 2 for the 360; consequently, the two programs have the same overall design and many routines are identical or differ only in minor details. A major part of phase 2 deals with keeping track of register contents, storing intermediate values, etc. This part is essentially identical. The code dealing with compactification is substantially different.

The Code Generation Phase acquires all necessary data from previous phases and uses that data to direct the generation of object code for the target computer.

Phase 2 produces, on request, a formatted mnemonic listing of object code produced. In addition, Phase 2 must supply proper object code interfaces to the runtime system.

Phase 2 contains four distinct sections operating in three passes:

1. Declared Storage Allocation	Pass 1
2. Initial Code Generation	
3. Code Compaction	Pass 2
4. Object Module Creation.	Pass 3

Figure 4 illustrates the organization of Phase 2 in more detail. Phase 2 is written in the XPL language.

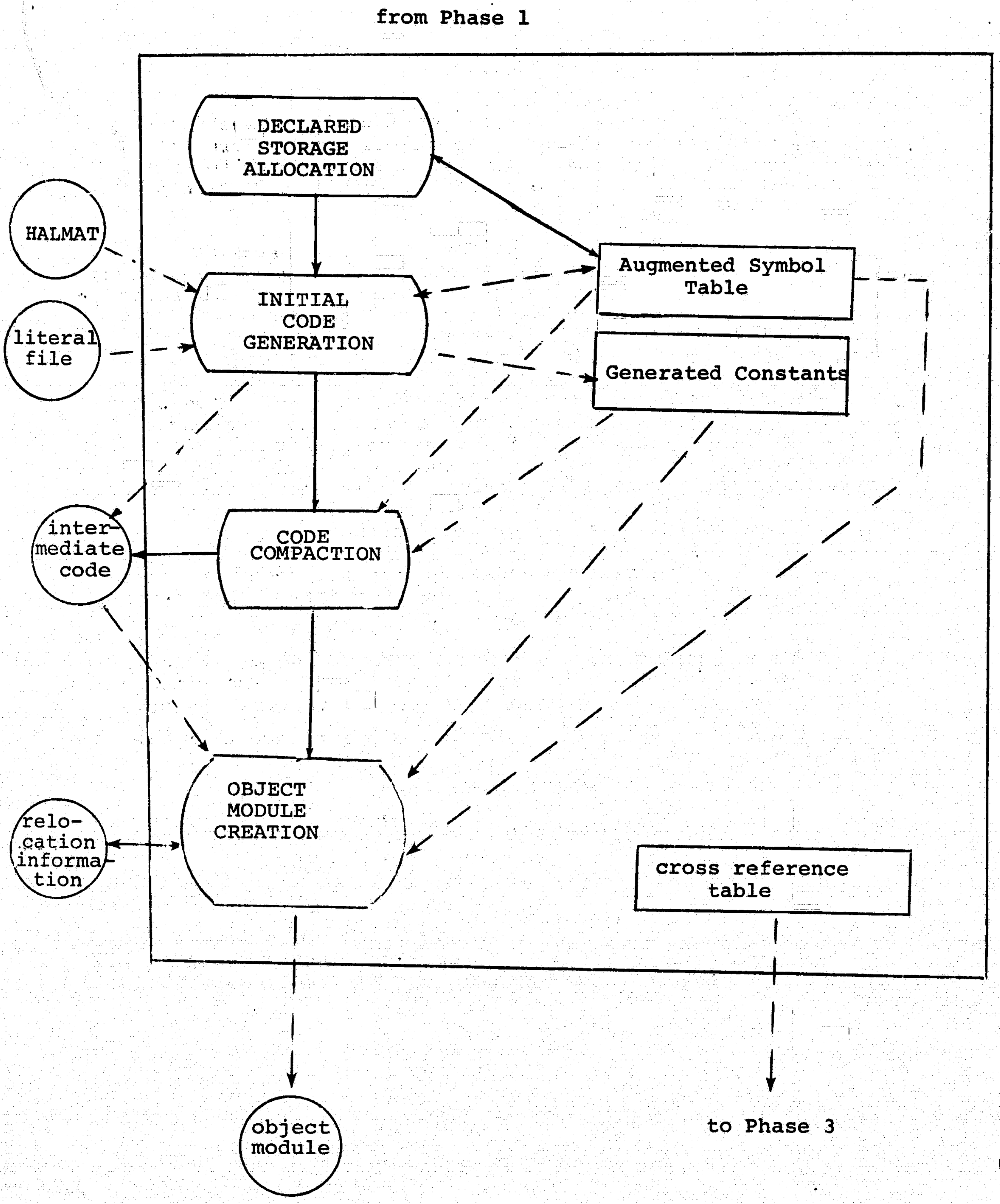


Figure 4: Phase 2 Organization

Pass 1

Declared Storage Allocation. Using symbol table information generated by Phase 1, this module (INITIALISE) allocates the necessary memory data explicitly declared by the user. The assignment of storage is done in a manner to best take advantage of word alignment and frequency of use. Base registers are assigned to data at this time.

Initial Code Generation. This module (GENERATE), translates the HALMAT from Phase 1 into a second intermediate code resembling an extension of 360 machine language. Register allocation, loads and stores, etc. are all determined at this point.

During this pass, local machine dependent optimizations are performed to reduce the amount of code generated. Each time a variable is to be forced into a register, a check is made to determine if the variable has been previously loaded or still exists in the register which last assigned the variable. If so, the register version, rather than the storage copy, is used for the associated arithmetic operation. This scheme also works for indexed variables. Also, constant terms involved in additive operations are carried at compile time until they must be incorporated into the variable part of the expression. Thus,

```
J = 8 + ((K + 3) - 2) + 4;
```

is compiled as if the statement were:

```
J = K + 13;
```

Operations which are cummutative are commuted if:

1. the right-hand operand is in a register,
2. the right-hand operand is a literal which can be loaded by an immediate instruction.

Included in the Code Generation is the building of the list of generated constants. This data is originally obtained from the Literal File, which contains the constants in a generalized internal form. The generated constants are specific to the context in which they are to be used; (e.g. generate an integer constant rather than a floating point constant). The last operation in pass 1 is outputting the generated constants onto the intermediate code file using GENERATE_CONSTANTS.

Pass 2

Code Compaction. This pass (OBJECT_CONDENSER) operates both on generated object data and generated object instructions.

The generated constants are output starting with those requiring the largest boundary alignment being emitted first. This compresses the literal pool to its smallest possible size.

During initial code generation, all branches to unknown labels (i.e. any forward references) generate an instruction sequence to reach any possible destination. The compaction process attempts to reduce this to a short instruction on the AP-101 and to eliminate the base register load on the 360.

This section will also compute the actual length of code and the data in each control section.

Pass 3

Object Module Creation. This pass (OBJECT_GENERATOR) transforms the internally coded instructions and data into standard FCOS or OS/360 object module format. This includes generation of:

a) ESD cards for each control section.

b) SYM card for SYMBOLS defined in program.

c) TXT cards for code and initial data.

d) RLD cards for necessary address constants.

e) END card for each PROGRAM.

f) Object Code Listing. On request, this module will also produce a formatted, mnemonic listing of object code produced by the code generation Phase. This listing identifies basic machine instructions by their standard assembler language mnemonics. References to data and to program addresses are identified by symbol reference. Corresponding HAL/S data names are indicated in the listing. The assembler code listing shows generated instructions on a statement by statement basis, following the same order as the HAL/S source statement (i.e. nesting of HAL/S code blocks which produce separate CSECT's will cause the assembler code listing to display the generated CSECTs in a nested manner). The individual lines in the assembler code listing are compatible in format with the absolute listing function of the link editor.

2.2.5 Internal Data Transfer

Communication between Phases of the HAL/S compiler occurs in two ways: 1) via direct, in-memory tables (i.e. common areas) and 2) via data stored on direct access I/O devices by one Phase and retrieved for use by another Phase.

Figure 1 shows the data relationships that exist in the compiler. The relationships to be discussed in this section are those involved in inter-Phase communication. Data transfer is in one direction only; i.e. since phases operate in sequence and not concurrently or iteratively, data can only flow from earlier to later phases.

Monitor/Phase Data Relationships

The Monitor does not participate in the actual generation or retrieval of any inter-Phase data. It acts only as a central channel for managing I/O operations on such data, or as an overlay supervisor in the handling of memory-resident common data. The Monitor may receive data from individual Phases in the form of completion codes indicating whether the compilation sequence is to continue.

Phase 1/Phase 1.5/Phase 2 Data Relationships

The interface between Phase 1 and 1.5 and Phase 2 has been designed in the most target-machine-independent manner possible. The degree to which this machine-independence has been achieved has determined the ease with which the code generator (Phase 2) can be modularly replaced. Since Phases 1 and 1.5 are identical for both the 360 and AP-101 compilers, the design has been completely successful.

Phase 1 passes information to Phase 1.5 and Phase 2 via both in-memory tables and external files. The data passed via a common memory area includes all symbol table and cross reference table information. These tables contain complete descriptions of all user-defined symbols and the HAL/S statements in which they are used. Since this table data is tied to HAL/S source code it is in a machine-independent form. Additional data passed in memory includes status information, special request information, error condition data detected in Phase 1, and some literal data information.

Data is passed from Phase 1 via two files on I/O devices. One file contains representations of all numeric literal data encountered by Phase 1 during the compilation. The literal data is in an internal, coded form which allows Phase 2 to produce object code literals in the proper target machine format.

The second I/O file contains a description of the compiled HAL/S program in an intermediate language form known as HALMAT. HALMAT is defined in the HAL/S-360 Compiler System Specification. The HALMAT for a given compilation describes the HAL/S source program in an elemental, operation-by-operation form. All HAL/S statements are represented as groups of operations. The operations consist of an operator (e.g. vector add) and operands upon which the designated operation is requested. The operands may be, for example, simple data items (e.g. simply indicating a particular symbol table entry) or results of previous operations (e.g. references to previous HALMAT operations which produced some intermediate result). The principal job of phase 1.5 is to replace sequences of HALMAT instructions by a reference to some previous HALMAT instruction which has already computed the result. Thus, the interposition of phase 1.5 between phases 1 and 2 has no effect on the data flow between them. Phase 1.5 is a transparent but distorting window. The HALMAT language itself describes only HAL/S constructs and refers only to the tables generated by Phase 1. It therefore is independent of the target machine's object code format. The form and organization of the HALMAT, however, permits an orderly, operation-by-operation generation of target code by Phase 2.

Data Passed to the Table Generation Phase

Information generated in Phase I and modified by Phase II is passed to Phase 3 via both in-memory tables and an external file. Symbol table and cross-reference information, augmented by relative address information from the code generator is passed in the common memory area.

The external file passed to the table generator contains information concerning the individual HAL source statements as scanned by Phase 1 and translated into object code in Phase 2. The file contains information to identify and locatė in the generated code each executable source statement with regards to type, symbolic references, and modified variables, Each of these features refer to the source code so that table generation is independent of the target machine's object code.

2.2.6 XPL and The Translator Writing System

The HAL/S compilers have been implemented using the XPL Translator Writing System (TWS), as the primary tool. The TWS is a program or a set of programs comprising a tool to assist in the writing of translator-compilers, interpreters, assemblers, etc. Its usefulness is derived from its ability to supply uniform functional modules for standard functions such as text scanning, and to automate the production of language-dependent portions of the compiler. The problem of correct syntax analysis is solved by using a scheme in which all parsing of input is driven by automatically generated tables. The tables are produced from an explicit specification of the language grammar. This produces a more complete, thoroughly checked compiler, and yet one that lends itself easily to modifications and changes.

The use of the XPL TWS has had its major influence in Phase 1 of the compiler where the syntax analysis is performed. Figure 5 illustrates the use of the XPL system in the generation of Phase 1 of HAL/S. The Grammar Analyzer is an independent program whose purpose is to accept a description of a grammar, analyze it for ambiguities, and produce a set of parsing tables. The parsing tables become a part of the syntax analysis routines in the compiler. Table look-up procedures to access the analyzer-generated tables are part of the XPL system. Thus, a correct parse of sentences in HAL/S is guaranteed by this separation of parse rules from semantic processing rules. The semantic processing routines and other utility functions form the remainder of Phase 1.

Certain aspects of the XPL language have had a significant effect on the HAL compiler and should be kept in mind.

- XPL procedure parameters are passed by value; thus it is impossible to return a value through a parameter.

- XPL does not allow arrays as procedure parameters; thus a very large amount of material must be global.

- XPL does no type checking, a value is TRUE if its low order bit is 1; TRUE=1 and FALSE=0 when used as arithmetic quantities.

- XPL does not check that a call passes the correct number of actual parameters.

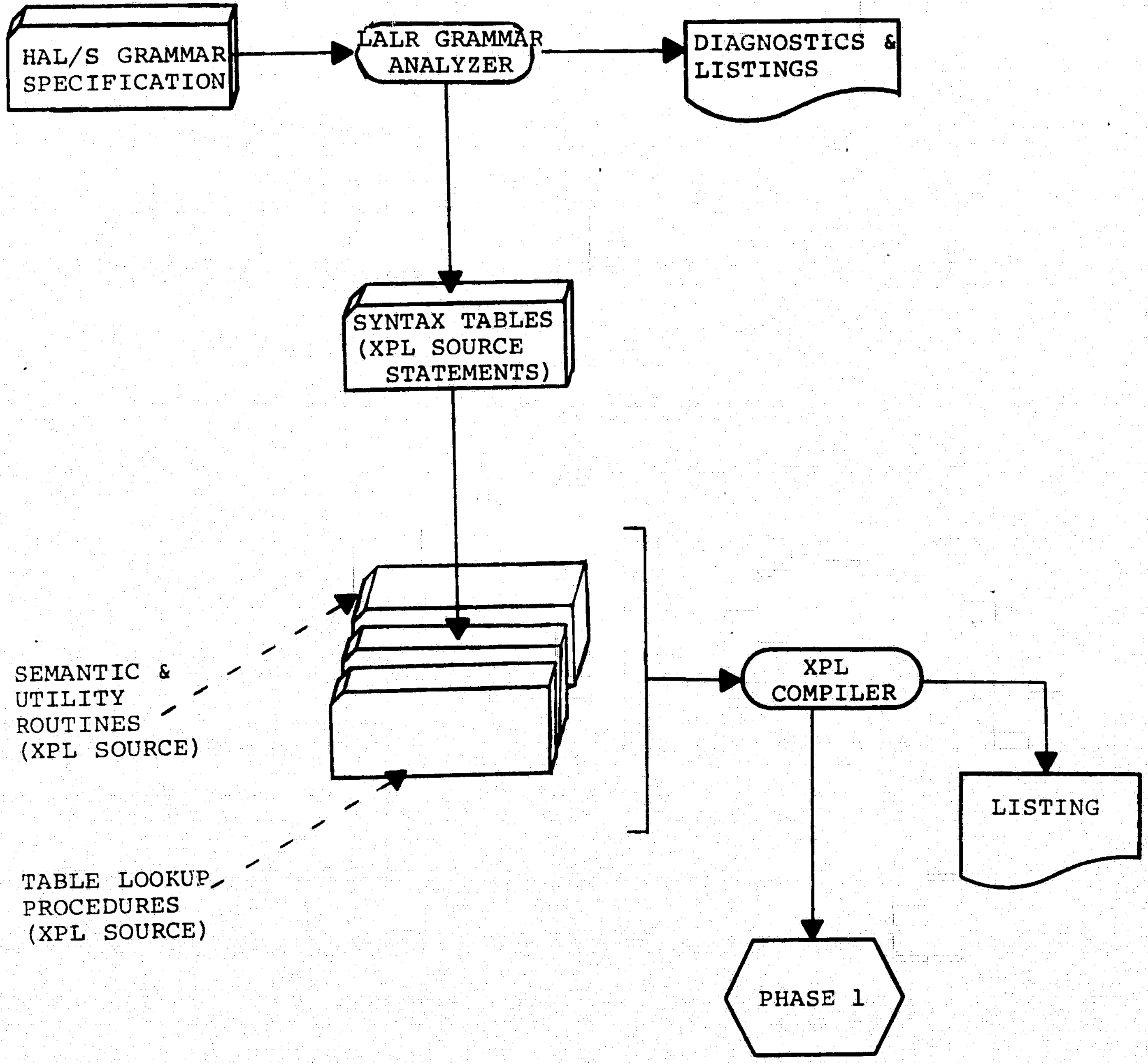

HAL/S GRAMMAR SPECIFICATION
LALR GRAMMAR ANALYZER
DIAGNOSTICS & LISTINGS
SYNTAX TABLES (XPL SOURCE STATEMENTS)
SEMANTIC & UTILITY ROUTINES (XPL SOURCE)
TABLE LOOKUP PROCEDURES (XPL SOURCE)
XPL COMPILER
LISTING
PHASE 1

Figure 5.

Using the XPL TWS to Implement Phase 1

Certain language/implementation details about string manipulations in XPL are important to an understanding of the HAL compiler. XPL maintains an area for string storage. This area is accessed via descriptors; that is, the direct value of a character string variable is a descriptor, not a string. The code A=B copies the B descriptor into A, not the B string. This makes for a large saving in space. There are pitfalls. When using BYTE in an assignment context, the string itself is modified, thus,

```
B = 'XYZ''
A = B;
BYTE(A,1) = BYTE('V');
```

will change the sole copy of the string XYZ to VYZ, changing both A and B! SUBSTR is fairly innocent, but it never checks its arguments -- this can lead to some very strange effects when the argument is invalid.

If BYTE is to be used to assign to a string it is essential to force a new string (not a new descriptor) into existence. Concatenating something onto an existing string will have this effect unless the string is null in which case an optimizer will victimize you.

2.2.7 Debugging Aids

If a D (compiler directive) card has EB or EBUG as its first token, a ¢ or H debugging directive is expected. The legal directives are:

¢0	Interlist HALMAT in the primary listing
¢1	Stop processing at the end of Phase 1
¢2	Stop processing at the end of Phase 2
¢3	Turn on Phase 1 identifier trace
¢4	Turn on Phase 1 token trace
¢5	Print HALMAT from Phase 2 (as reordered)
¢6	Print intermediate code listing from Phase 2
¢7	Print Phase 1 symbol table after next HAL source statement and turn off option
¢8	Print Phase 1 production trace
¢9	Print Phase 2 diagnostic information
¢A	In Phase 1 ABEND <u>NOW</u>
¢B	Print Phase 1 HALMAT by block. This will reflect any reorderings performed after the ¢0 printing.
¢C	Print Phase 1 state trace
¢D	Turn on standard Phase 1 listing
¢E	Print literal table from Phase 1
¢F	Set to expand symbol table printing

All debugging information is printed in the primary source listing.

If T is a toggle as defined above, ¢T + turns on the option, ¢T - turns off the option, ¢T inverts the current sense of the toggle.

The ¢ toggles are primarily useful in Phase 1 because the toggles are flipped when the DEBUG card is read. In order to provide similar facilities to Phase 1.5 and 2, the H(n) option is available. If H(n) appears on a DEBUG card, the number, n, will be inserted in the next HALMAT SMRK instruction issued.

$0 \leq n \leq 127$ is reserved for Phase 1.5 (see Sec. 6.8)

$128 \leq n \leq 255$ is reserved for Phase 2.

200 - off HALMAT, assembler code, stack trace

201 - on HALMAT, assembler code, stack trace

202 - off HALMAT, assembler code

203 - on HALMAT, assembler code

204 - invert register trace

205 - invert HALMAT

206 - invert assembler code

207 - invert binary code

208 - invert subscript trace

209 - invert stack trace

When an option is selected to print HALMAT, the format is:

operator words -- OP(N), T, P

operand words -- D(Q), T1, T2

3.0 COMMON DATA STRUCTURES

The phases of the HAL/S compiler communicate in two ways: via the HALMAT file and via commonly used data structures. The format of the HALMAT file is described in the HAL/S-360 Compiler System Specification, Appendix A. This chapter provides a detailed description of those data structures used for inter-phase communication.

3.1 Literal Table

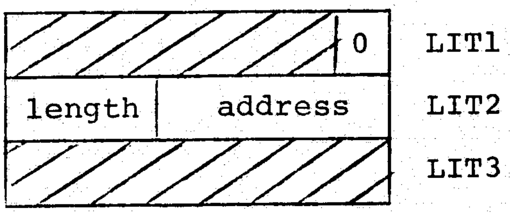
The HAL/S Literal Table is used to convey literal information from Phase 1 to subsequent compiler phases. Certain single valued variables declared as CONSTANT also use the literal table.

There are three parallel arrays used to specify literals: LIT1, LIT2, and LIT3. Not all literals need be in memory at the same time. An intermediate file is used to pass literal information. The LIT1, LIT2, and LIT3 arrays are stored next to each other and their commulative size is the size of one I/O block. Thus, one FILE statement serves to transfer all three arrays. The LIT qualifier on a HALMAT operand indicates that the operand is to be retrieved from the literal table.

There are only three types of literal entries: 1) character, 2) arithmetic, 3) bit. Each has a different format on the literal file. Each type may undergo transformation during the code generation process, thus eliminating the emission of unnecessary code for literal conversions.

Character Literals

Format



The length specified is one less than the actual length of the string, consistent with XPL descriptor notation. The address refers to an entry in the array LIT_CHAR, which is a BIT(8) array whose size is determined by the LITCHARS compiler option. If over LITCHARS bytes of character literal information is encountered in a HAL/S program, the compilation is abandoned (LIT_CHAR cannot be kept on an intermediate file).

Arithmetic Literals

Format

▨	1	LIT1
double precision		LIT2
floating point #		LIT3

This is the most general form of numeric literal. The code generator transforms the number to single precision or integer as required by the context in which the literal appears. If LIT2 = "FF000000", then the number was found invalid by Phase 1.

Bit Literals

Format

▨	2	LIT1
Bit Pattern		LIT2
length		LIT3

The first word contains up to 32 bits of information, as required, to specify the bit literal. The length field specifies the bit count as determined by the source input. It is always a multiple of 4 for hexadecimal. For decimal literals only, the length represents the number of significant bits in the literal value. For all others, the length reflects the number of characters in the string specifying the literal, including lending zeros.

CURLBLK is the number of the page of the literal file currently in memory.

LIT_TOP is the index of the last entry in the literal table.

LITLIM is the highest literal index number in the page currently in memory.

LITMAX is the number of pages in the literal table.

LITORG is the lowest literal index number in the page currently in memory.

LIT_CHAR_FREE is the number of character positions still available in LIT_CHAR.

LIT_CHAR_AD is the address of the next available character in LIT_CHAR.

3.2 Symbol Table

The HAL/S symbol table consists of a large group of parallel arrays of length SYT_SIZE (can be set with JCL option SYMBOLS) plus a small group of arrays augmenting the parallel ones, which describes all the properties of declared variables and labels. The symbol table is created by Phase 1 of the HAL/S compiler and augmented by Phase 2. It is available in the COMMON communication area for use by subsequent phases of the compiler. The names of the arrays and their associated bit widths are listed below. A detailed explanation of the contents of each array follows.

Created by Phase 1
and Passed to All Subsequent Phases

EXT_ARRAY	(16)
SYT_NAME	CHARACTER
SYT_ADDR	(32)
SYT_CLASS	(8)
SYT_TYPE	(8)
SYT_DIMS	(16)
SYT_ARRAY	(16)
SYT_FLAGS	(32)
SYT_LOCK#	(8)
SYT_NEST	(8)
SYT_SCOPE	(8)
SYT_LINK1	(16)
SYT_LINK2	(16)
SYT_PTR	(16)
SYT_XREF	(16)
SYT_LABEL	literally SYT_LINK2
VAR_LENGTH	identical to SYT_DIMS
XREF	(32)

Created and Used Only by Phase 1

SYT_HASHLINK	(16)
SYT_HASHSTART	(16)

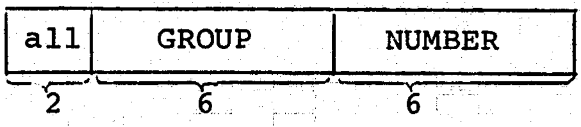
Created by Phase 2

SYT_SORT	(16)	Used only in Phase 2
SYT_BASE	(16)	
SYT_DISP	(16)	
SYT_PARM	(16)	
SYT_CONST	(32)	
SYT_LEVEL	(16)	
EXTENT	(32)	Passed to Phase 3

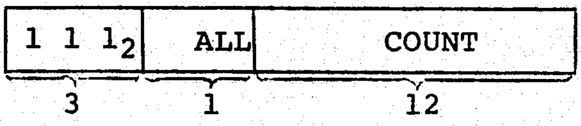
EXT_ARRAY

For dimensioned variables, each SYT_ARRAY entry points to an entry in EXT_ARRAY which contains information about the entry's arrayness. EXT_ARRAY contains the number (n) of array dimensions specified. The following n entries contain the actual array sizes. For * size arrays, the array size is specified as a negative pointer back to the symbol table entry. These entries are entered starting from 0 and EXT_ARRAY_PTR points to the last entry.

For block names, EXT_ARRAY contains an entry for each unique error referenced in an ON ERROR or OFF ERROR statement. The form of the entry is:



where NUMBER is "3F" if the entry is for the entire group and the entry is "3FFF" if it is for all errors. These entries are entered starting from the end of the array and moving down towards 0. ON_ERROR_PTR points to the last (i.e. lowest) entry. If the block is still being processed, SYT_ARRAY is a negative pointer to the first EXT_ARRAY entry for the block. When the block is closed, SYT_ARRAY becomes:



where COUNT is the number of EXT_ARRAY entries, and ALL is 1 if there was an entry for all errors. After transforming SYT_ARRAY, the counted EXT_ARRAY entries are discarded.

EXTENT

This array contains the number of halfwords necessary to hold the entire data item unless the item has * arrayness. If the item has * arrayness, EXTENT contains the width of one copy.

NDECSY

Points to the last entry in the symbol table in Phase 1.

SYT_ADDR

The relative location of the declared variable. For block labels, it is the relative location of the block header within the program data area. For formal parameters and AUTOMATIC variables of a function or procedure, SYT_ADDR is the relative location of the variable within the runtime stack frame of the procedure. For structure template nodes, it is the relative location of the node from the beginning of the template. For major structure template, the STRUC_SIZE

SYT_ARRAY

The SYT_ARRAY array is used for any data type which can exhibit arrayness or copiness. For arrays, see EXT_ARRAY. If SYT_ARRAY is zero, no arrayness is present. For structure copies, a positive value indicates the number of copies; a negative number indicates * size copiness, and points back to the symbol entry.

For block names, see EXT_ARRAY.

SYT_BASE

The base register used for addressing the declared variable. If SYT_BASE is negative, the register is virtual and code must be generated to load a real register instead.

INITIALISE uses the space to hold the size of the data item; for aggregates, the size of a single element; for structures, the size of the largest element. The size information is required for setting up proper boundary alignments when assigning storage addresses.

SYT_CLASS

The SYT_CLASS array is used to classify a symbol into major categories (cf. SYT_TYPE). These classifications are used to determine which type of token must be generated by the scanner to properly compile the statement. The classifications are:

Name	Value	Classification
VAR_CLASS	1	Variable name
LABEL_CLASS	2	Label name
FUNC_CLASS	3	Function name
REPL_ARG_CLASS	5	Replace argument
REPL_CLASS	6	Replace macro name
TEMPLATE_CLASS	7	Structure template variables
TPL_LAB_CLASS	8	Structure template label
TPL_FUNC_CLASS	9	Structure template function

SYT_CONST

1) When addressing aggregate data, the HAL compiler computes addresses relative to the 0th element because this generates the most efficient code. Since all HAL subscripts start at 1, the address of a variable is the address of its 1st element. Thus, the base address for subscripting is:

$$\text{address}(\text{variable}_0) = \text{address}(\text{variable}_1) - \text{constant.}$$

SYT_CONST is this constant.

For simple variables and single copy structures, SYT_CONST is 0.

2) For update labels, this indicates the lock group numbers involved in the block.

SYT_DIMS

1) The SYT_DIMS array is interpreted as follows for each name type:

BIT - Bit width

CHARACTER - Maximum character length

MATRIX -

row size	column size
8	8

VECTOR - Vector length

STRUCTURE TEMPLATE - There is not static information in SYT_DIMS for the root node of a structure template. When analyzing operations between two structures it is sometimes necessary to perform a structure walk. This walk may reach a node of type Q-structure. In that case, SYT_DIMS(Q) contains a negative pointer back to the containing structure's node for operand 1 and SYT_LINK2(Q) contains the equivalent for operand 0.

A node of type structure template, which has no descendants (i.e. SYT_LINK1=0) must be of type Q-structure for some Q. In this case, SYT_DIMS points to Q's template.

STRUCTURE VARIABLE - Pointer to the symbol table entry for the template.

STMT_LABEL - 0 - defined only
1 - unlabelled update block
2 - labelled update block
3 - reached by GO TO
4-7 - unreachable by GO TO (IF labels)

MACRO - Number of parameters.

2) For arrayed character formal parameters, SYT_DIMS is a negative pointer to the symbol table entry.

SYT_DISP

The displacement used for generating base-displacement addresses for accessing the data items. For an aggregate data item, it is the displacement necessary to generate the actual address minus SYT_CONST, i.e. the address of the 0th item.

In INITIALIZE, 0 indicates program data area;
≠0 then value is scope# = csect# of item.

For structure templates, the number of extra bytes required to achieve the same alignment as the beginning of the node.

SYT_FLAGS

SYT_FLAGS contains many descriptive flags used by Phase 1 to determine conflicting declarative attributes for symbols. The following list of flag entries is used by the subsequent compiler phases:

Name	Value	Attribute Tested by the Flag
ACCESS_FLAG	"00010000"	ACCESS protected
ALDENSE_FLAGS	"0000000C"	ALIGNED_FLAG or DENSE_FLAG
ALIGNED_FLAG	"00000008"	Item is declared ALIGNED
ARRAY_FLAG	"00002000"	Item is an array
ASSIGN_FLAG	"00000020"	Entry is a formal parameter requiring an assign parameter
ASSIGN_OR_NAME	"10000020"	NAME_FLAG or ASSIGN_FLAG
ASSIGN_PARM	"00000020"	Same as ASSIGN_FLAG
AUTO_FLAG	"00000100"	Entry requires automatic initialization
AUTSTAT_FLAGS	"00000300"	AUTO_FLAG or STATIC_FLAG
CONSTANT_FLAG	"00001000"	Entry has the CONSTANT attribute
DEFAULT_ATTR	"00800208"	Attributes for implicit declarations
DEFINED_BLOCK	"10100000"	NAME_FLAG or EXTERNAL_FLAG
DEFINED_LABEL	"00000060"	Label reference is resolvable
DENSE_FLAG	"00000004"	Entry is subject to dense allocation rules
DOUBLE_FLAG	"00400000"	Use double precision
DUMMY_FLAG	"01000000"	Formal parameter of a procedure or function which had no declaration
DUPL_FLAG	"04000000"	Duplicate name in structure template
ENDSCOPE_FLAG	"00004000"	Indicates end of COMPOOL list
EVIL_FLAGS	"00200000"	Structure template not properly completed
EXCLUSIVE_FLAG	"00080000"	Procedure or function is to have exclusive usage

Name	Value	Attribute Tested by the Flag
EXTERNAL_FLAG	"00100000"	Block name is not part of the compilation unit
IGNORE_FLAG	"01000000"	Routine INITIALISE ignores this
IMP_DECL	"00000010"	Symbol implicitly declared
IMPL_T_FLAG	"00040000"	Is used with a transpose operation
INIT_CONST	"00001800"	CONST_FLAG or INIT_FLAG
INIT_FLAG	"00000800"	¬INIT_CONST
INP_OR_CONST	"00001400"	INPUT_PARM or CONSTANT_FLAG
INPUT_PARM	"00800208"	Variable is a formal parameter of input type
LATCH_FLAG	"00020000"	Event variable entry has the LATCHED attribute
LATCHED_FLAG		See LATCH_FLAG.
LOCK_BITS	"00000001"	Entry is a member of a lock group indicated by SYT_LOCK#
LOCK_FLAG		See LOCK_BITS
MISC_NAME_FLAG	"40000000"	The structure contains a name variable somewhere in it
NAME_FLAG	"10000000"	Entry has the NAME attribute
NONHAL_FLAG	"02000000"	Procedure or function uses non-HAL linkage conventions
PARM_FLAGS	"00000420"	Entry is a parameter
PM_FLAGS	"00C20080"	Flags which must match for assign by reference
POINTER_FLAG	"80000000"	Entry is a formal parameter passed by reference
POINTER_OR_NAME	"90000000"	Entry is a formal parameter or has the NAME attribute
READ_ACCESS_FLAG	"20000000"	Read only
REENTRANT_FLAG	"00000002"	Procedure or function is REENTRANT
REMOTE_FLAG	"00000080"	Entry has the REMOTE attribute
RIGID_FLAG	"04000000"	Entry has RIGID atribute
SD_FLAGS	"00C00000"	SINGLE_FLAG or DOUBLE_FLAG
SINGLE_FLAG	"00800000"	Use single precision
SM_FLAGS	"10C2008C"	Flags which must match on structure terminals
STATIC_FLAG	"00000200"	Item is declared STATIC
TEMPORARY_FLAG	"08000000"	Entry is a DO group temporary.

SYT_HASHLINK

See SYT_HASHSTART

SYT_HASHSTART

The symbol table is accessed via a hash function. SYT_HASHSTART is an independent array whose elements point to the first entry in the symbol table with a particular hash code. Entries with the same hash code are linked using SYT_HASHLINK which is one of the parallel SYT arrays.

SYT_LABEL: literally 'SYT_LINK2'

A statement number generated by Phase 2 for every entry in the symbol table of label class (cf. GETSTATNO).

SYT_LEVEL

1) A pointer to the symbol table entry for another variable in the same block. SYT_LEVEL provides a linked list of all the variables declared in a block. The entry for the block's name is the beginning of the list. This entry is pointed to by PROC_LINK (scope# (block)).

2) Used to form a linked list of all structure template names. STRUCT_START points to the list's beginning.

3) For formal parameters with * arrayness or character size, SYT_LEVEL indicates the presence of zero, one, of both of these features by value of 0, 1, 2, respectively. This is the number of words of storage necessary to pass the information.

4) INITIALISE saves NDECSY of the node here for later use by ALLOCATE_TEMPLATE before use 2).

SYT_LINK1

1) For structure templates: See SYT_LINK2.

2) Used to form a linked list of all procedures and functions using non-HAL linkage conventions. XPROGLINK points to the beginning of this list.

3) Used to form a chain of all tasks. SYT_LINK1 of the main program points to the beginning of this list.

4) If the entry is the label of an exclusive block, it is a number identifying the block.

5) Used to form a chain of all REMOTE variables. FIRSTREMOTE points to the beginning of this chain.

6) Used to form a linked list of all external labels. ENTRYPOINT points to the beginning of this list.

7) For REPLACE names points to beginning of <text> in MACRO_TEXT.

8) Used to form a list of TEMPORARY variables.

9) For labels in phase 1, -DO_LEVEL at the point of declaration of the label.

SYT_LINK2

Labels

Phase 1 uses this entry to back chain label definitions. The beginning of the list (i.e. the last label) is in SYT_LINK2(0). Phase 2 uses the name SYT_LABEL (see that entry for definition).

Structure Templates

The symbol table format for a structure template consists of a linked list to define ordering, using the companion arrays SYT_LINK1 and SYT_LINK2.

A structure walk begins with a major structure pointing to a template name via SYT_DIMS, as described earlier. The tree walk, if performed properly, will begin and end at the same template reference point. The following general rules apply to structure walks:

1) SYT_LINK2 generally points to the next terminal symbol or node point at the same level number as the current symbol (i.e. its right brother); SYT_LINK2 is usually zero for the structure name entry, however see SYT_DIMS for structure templates.

2) If SYT_LINK1 of an entry is non-zero, the entry is a node (i.e. not a terminal) and SYT_LINK1 points to its first descendant.

3) If SYT_LINK2 of an entry is negative, it indicates the last item in a minor node, and the absolute value of SYT_LINK2 refers to the minor node point (i.e. its father); the structure walk proceeds from SYT_LINK2 of the minor node.

Example:

	SYT #	SYT_LINK1	SYT_LINK2
STRUCTURE A:	1	2	-
1 B,	2	3	5
2 C,	3	0	4
2 D,	4	0	-2
1 E,	5	6	-1
2 F,	6	7	9
3 G,	7	0	8
3 H,	8	0	-6
2 J;	9	0	-5

SYT_LOCK#

If SYT_FLAGS indicates that the variable is a member of a lock group, SYT_LOCK# indicates the lock group number.

For templates of external units (e.g. compools, comsubs, etc.) SYT_LOCK# is the version number of the template.

For the root node of a structure template SYT_LOCK#="80".

SYT_NAME

The actual name of the variable.

SYT_NEST

SYT_NEST indicates the nest level at which a variable or label is defined. It is useful for determining proper name scoping.

SYT_PARM

1) If the entry is a formal parameter, this is the register in which it will be passed. If there are insufficient register SYT_PARM is negative.

2) If the entry is a task, this is a number identifying the task.

3) If the entry is a function, 0 indicates the function requires an area for returning a result; -1 indicates that the result will be returned in a register.

SYT_PTR

For block names, SYT_PTR points to the first declared symbol in the block. If the block has arguments, SYT_PTR is quarantee to point to the first argument in the list.

For unqualified structures, SYT_PTR of the template name refers to the corresponding major structure name.

For REPLACE names, the MACRO_INDEX .

For CONSTANTs, a negative pointer to the literal table.

For labels, SYT_PTR links together all labels for the same statement.

SYT_SCOPE

SYT_SCOPE uniquely identifies the block in which a variable or label appears. A number is assigned to each block as it is defined.

SYT_SIZE

The size of the symbol table as determined from the JCL SYMBOLS option.

SYT_SORT

Array used for sorting the symbol table entries. An entry has the form:

scope #	symbol table pointer

16

SYT_TYPE

The SYT_TYPE array gives a more detailed description of the symbol, and is meaningful in the context of the associated SYT_CLASS. The following is a list of the allowable types and their associated reference number:

Name$_1$/Name$_2$	Phase 1 Value	Phase 2 Value	Description
BIT_TYPE/BITS	1	1 or 9	Bit string
CHAR_TYPE/CHAR	2		Character string
MAT_TYPE/MATRIX	3	3 or 11	Matrix data
VEC_TYPE/VECTOR	4	4 or 12	Vector data
SCALAR_TYPE/SCALAR	5	5 or 13	Scalar data
INT_TYPE/INTEGER	6	6 or 14	Integer data
BORC_TYPE	7		Bit or Character string -- used for built-in functions which allow more than one type of argument

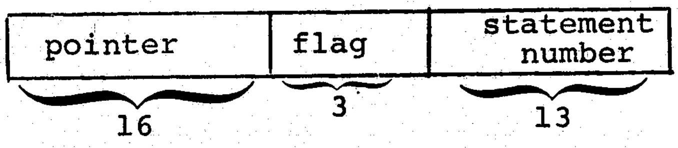
$Name_1/Name_2$	Phase 1 Value	Phase 2 Value	Description
IORS_TYPE	8		Integer or Scalar data (see BORC)
Event_Type/EVENT	9	17	Event variable
MAJ_STRUC/STRUCTURE	10	16	Major structure or structure node
ANY_TYPE	11		A number greater than all real data types
TEMPL_NAME	62		Structure template name
ANY_LABEL	64		Not an actual type, but used to distinguish labels from other types
STMT_LABEL	66		Statement label
UNSPEC_LABEL	67		Used by Phase 1 to classify labels until enough information is available to subclassify them
IND_CALL_LABEL	69		See description of procedure labels below
PROC_LABEL	71		See description of procedure labels below
TASK_LABEL	72		Task label
PROG_LABEL	73		Program label
COMPOOL_LABEL	74		Compool label
EQUATE_LABEL	75		Name is an external name defined by an EQUATE declaration

PROCEDURE LABELS create a difficulty unlike any other HAL/S name. If a procedure is declared in a given scope and called in the same scope, there is no complication; however, the declaration may appear after the call. Thus, if a procedure is declared in an outer scope, at the point of call it is not yet known whether the outer scope declaration is the correct one. To handle this problem, a new symbol table entry is made for the procedure at the point of call with SYT_TYPE =IND_CALL_LABEL and SYT_PTR pointing to the previous entry for the name. If a new definition for the name is encountered, the chain is traced back to the proper NEST level and pointed at the new declaration by procedure SET_LABEL_TYPE. The label on a procedure is therefore of type PROC_LABEL and all and only those calls which definitely call a specific declaration point directly to that entry.

Phase 2 uses SYT_TYPE to distinguish between single and double precision by ORing in a bit in the "8" position. This requires renumbering EVENT and STRUCTURE to values that do not conflict with the double precision convention. The complete set of phase 2 names can be found in Section 3.3.8 ("operand types and properties").

SYT_XREF

References to variables are accumulated in array XREF. An XREF entry is in the form:



Where pointer points to the next entry for the same variable; flag indicates a declaration, assignment, reference, or subscript usage; statement number is the statement number of the usage.

The list is maintained in the order of occurrence so references later on the list are at higher statement numbers. Multiple references to the same variable in the same statement may set more than one bit in the flag but do not generate multiple entries in the list.

SYT_XREF for a variable points to the beginning of the list. SYT_XREF(SYTSIZE) is the STMT_NUM of the line opening the block.

XREF_LIM is the size of XREF table as determined by JCL parameter XREFSIZE.

XREF_FULL is set when the XREF table overflows so that the overflow error message will be issued only once.

XREF_INDEX points to the last entry in XREF.

XREF_ASSIGN is a mask for an assignment usage.

XREF_REF is a mask for a reference usage.

XREF_SUBSCR is a mask for a subscript usage.

XREF_MASK is a mask for the statement number section.

SYTSIZE

Same as SYT_SIZE.

VAR_LENGTH

Identical to SYT_DIMS.

XREF

See SYT_XREF.

3.3 The COMMunication and VALS Arrays

The array COMM is reserved for inter-phase communcation. Most of the COMM array is unused. The defined portion is:

COMM	
0	LIT_CHAR_ADDR
1	LIT_CHAR_LEFT
2	LIT_TOP
3	STMT_NUM
4	FL_NO_MAX
5	MAX_SCOPE#
6	TOGGLE
7	OPTION_BITS
10	SYT_MAX
20	OBJECT_MACHINE
21	OBJECT_INSTRUCTIONS
22	WALKBACK_LOOPS

COMM(7) = OPTION_BITS

Hex	JCL parm field name	P1	P2		P3	P1.5
			360/FC			
00000001	DUMP					
00000002	LISTING2	✓				
00000004	LIST		✓	✓		
00000008	TRACE		✓	✓	✓	
00000010	X0 NO TEMP	✓				
00000020	X1 NO CSE					✓
00000040	X2 NO VM		✓	✓		
00000080	X3 CSE WATCH					✓
00000100	X4 360 - 0 TIMES FC - F8 COMP		✓	✓		
00000200	X5 CSE TRACE					✓
00000400	ZCON			✓		
00000800	TABLES	✓	✓	✓		
00001000	TABDMP				✓	
00002000	X9					
00004000	XA 360 - Extra Data FC - ABSLIST		✓	✓		
00008000	TABLST				✓	
00010000	PARSE	✓				
00020000	LSTALL		✓	✓		
00040000	FCDATA		✓			
00080000	SRN	✓		✓	✓	
00100000	ADDRS	✓		✓	✓	
00200000	LFXI			✓		
00400000	DECK		✓	✓		
00800000	SDL	✓		✓	✓	
01000000	X6 Print Phase 1.5 statistics					✓
02000000	SCAL			✓		
04000000	MICROCODE			✓		
08000000	XB				✓	
10000000	XC				✓	
20000000	XD					
40000000	XE					
80000000	XF					

VALS

VALS is a collection of parameters for the compiler. The address of VALS is in the 4th word of the sub-monitor's communication area; therefore, VALS must be initialized by:

```
TMP = MONITOR(13)

COREWORD(ADDR(VALS)) = COREWORD(TMP+16)
```

The VALS array contains:

0	title
1	linect
2	payls
3	symbols
4	macrosize
5	litstrings
6	compunit
7	xrefsize
8	cardtype
9	labelsize
10	data sector

4.0 PHASE I

Phase I of the HAL/S compilers is a classical syntax directed compiler whose input is HAL/S source code and output is the intermediate code HALMAT. The description of such a compiler is naturally broken up into:

4.1 The Parser

4.2 The Scanner

4.3 The Output Writer

4.4 The Semantic Routines

In general, the data is described in the subsections; however, some items are used in many places so Section 4.5 defines all the global names used in Phase I.

4.1 The Parser

Phase 1 is a classical syntax directed compiler. Thus, the parser has the responsibility of overall logical control. It calls the scanner (Section 4.2) to input tokens, the output writer (Section 4.3) to print the listing, and the semantic routine (Section 4.4) to generate code. In this compiler, the parser is LARL(1), the parse routine is COMPILATION_LOOP and like most bottom up parsers, the semantic routine is called just before reducing the stack. The code generated is HALMAT, an intermediate code which is translated to machine code by Phase 2.

4.1.1 Global Variables Used by the Parser

#PRODUCE_NAME(production number)	
	The left side of the production.
APPLY1(I)	Enter APPLY1 by current state and search for match with state before stacking production. If match found, APPLY2(I) is the new state.
APPLY2	See APPLY1.
BCD	See SCAN.
CHARACTER_STRING	See global definitions -- TOKEN.
CONTEXT	See SCAN.
FIXF	Stack of FIXINGs, indexed by SP.
FIXING	See SCAN.
FIXL	Stack of SYT_INDEXs, indexed by SP.
FIXV	Stack of VALUEs, indexed by SP.
IMPLIED_TYPE	See SCAN.

INDEX1(state) Points to the beginning of the entries for state in READ1, APPLY1, and LOOK1. It is the new STATE for null productions.

INDEX2(state) Points to the end of state's entries in READ1. When doing reduction, the number of items in the production's right side.

LOOK Holds the old state when a new state is computed by a look ahead.

LOOK_STACK Is where LOOKs are stacked -- indexed by SP.

LOOK1(I) Enter by state, search for match with look ahead token. If match found, LOOK2(I) is the new state.

LOOK2 See LOOK1.

MAXL# See STATE.

MAXP# See STATE.

MAXR# See STATE.

MP See SP.

MPP1 See SP.

NO_LOOK_AHEAD_DONE Is true if the parser has not buffered one token ahead by doing a look ahead.

PARSE_STACK Stack of grammatical items, terminal or non-terminal -- indexed by SP.

READ1 An array of tokens, indexed by INDEX1 and INDEX2. READ1 is entered by STATE and searched for TOKEN; when a match is found, the associated READ2 entry is the new STATE. If no match is found, there is a syntax error.

READ2 See READ1.

REDUCTIONS — Total number of reductions made by parser.

REPLACE_TEXT — See global definitions -- TOKEN.

RESERVED_WORD — See SCAN.

SEMI_COLON — See global definitions -- TOKEN.

SP — Is the stack pointer for the top of the parser's stacks; MP is set to the index of the left-most symbol of a production when doing a reduction; MPP1 ≡ MP+1. After a reduction, naturally SP is set to MP.

STATE — An integer used to encode the current state of the parser. This is used to index into the rest of the parser tables.

If 0 ≤ STATE ≤ MAXR#, it is a read state.

If MAXR# < STATE ≤ MAXL#, it is a lookahead state.

If MAXL# < STATE ≤ MAXP#, it is a read a null state.

If MAXP# < STATE, it is a reduce state.

STATE_NAME(state) — Is the token associated with this state.

STATE_STACK — Is the controlling stack of the parser. This is where STATEs are stacked -- indexed by SP.

STMT_END_FLAG — See global definitions -- GRAMMAR_FLAGS.

STMT_PTR — See global definitions -- GRAMMAR_FLAGS.

SUBSCRIPT_LEVEL — Incremented for each $, decremented at the end of the subscript.

SYT_INDEX — See SCAN.

TEMPORARY_IMPLIED — See SCAN.

VALUE — See SCAN.

VAR — This is where BCDs are stacked -- indexed by S..

VOCAB_INDEX See procedure SCAN -- identifiers.

4.1.2 Procedures of the Parser

```
COMPILATION_LOOP -- 1542300
ADD_TO_STACK     -- 1543400
```

COMPILATION_LOOP is the main program of the parser.

At any given moment, the parser is in some state. Depending on the state, the parser will either:

1. Read the next token and stack the current state using ADD_TO_STACK. Then compute a new state based on the old state and the new token. This is the only place that syntactic errors are discovered.

2. Reduce the top states on the stack, call SYNTHESIZE to perform the semantic analysis associated with the production and compute a new state based on the new top of STATE_STACK and the current state.

3. Look ahead one symbol and change state depending on the current state and the next symbol.

4. Read a null token, push the state stack and change state.

Possibilities 1 and 2 are the real heart of the parser, 3 and 4 enable a clean bookkeeping algorithm. Figure 4.1 is an example of the parser at work.

```
scanner: <LABEL> = "SIMPLE"
scanner: ":"
                        reduction number 304 -- <LABEL DEFINITION> ::= <LABEL>
scanner: "PROGRAM"
                        reduction number 305 -- <LABEL EXTERNAL> ::= <LABEL DEFINITION>
                        reduction number 307 -- <BLOCK STMT HEAD> ::= <LABEL EXTERNAL> PROGRAM
scanner: ";"
                        reduction number 301 -- <BLOCK STMT TOP> ::= <BLOCK STMT HEAD>
                        reduction number 298 -- <BLOCK STMT> ::= <BLOCK STMT TOP> ;


  source line was:  SIMPLE:
  source line was: PROGRAM;


scanner: "DECLARE"
scanner: <IDENTIFIER> = "A"
scanner: ";"
                        reduction number 358 -- <NAME ID> ::= <IDENTIFIER>
                        reduction number 356 -- <DECLARATION> ::= <NAME ID>
                        reduction number 342 -- <DECLARATION LIST> ::= <DECLARATION>
                        reduction number 340 -- <DECLARE BODY> ::= <DECLARATION LIST>
                        reduction number 339 -- <DECLARE STATEMENT> ::= DECLARE <DECLARE BODY>


  source line was:    DECLARE A;


                        reduction number 329 -- <DECLARE ELEMENT> ::= <DECLARE STATEMENT>
                        reduction number 345 -- <DECLARE GROUP> ::= <DECLARE ELEMENT>
scanner: <ARITH ID> = "A"
                        reduction number 291 -- <BLOCK BODY> ::= <DECLARE GROUP>
                        reduction number 222 -- <PREFIX> ::=
scanner: "="
                        reduction number 230 -- <SUBSCRIPT> ::=
                        reduction number 216 -- <ARITH VAR> ::= <PREFIX> <ARITH ID> <SUBSCRIPT>
                        reduction number 193 -- <VARIABLE> ::= <ARITH VAR>
scanner: <ARITH ID> = "A"
                        reduction number 248 -- <=1> ::= =
                        reduction number 222 -- <PREFIX> ::=
scanner: "+"
                        reduction number 230 -- <SUBSCRIPT> ::=
                        reduction number 216 -- <ARITH VAR> ::= <PREFIX> <ARITH ID> <SUBSCRIPT>
                        reduction number 27  -- <PRIMARY> ::= <ARITH VAR>
                        reduction number 15  -- <FACTOR> ::= <PRIMARY>
                        reduction number 11  -- <PRODUCT> ::= <FACTOR>
```

Figure 4.1 Example of Parser - Scanner Action

```
                    reduction number 9    -- <TERM> ::= <PRODUCT>
                    reduction number 4    -- <ARITH EXP> ::= <TERM>
scanner: <SIMPLE NUMBER> = "1"
                    reduction number 424 -- <NUMBER> ::= <SIMPLE NUMBER>
                    reduction number 19   -- <PRE PRIMARY> ::= <NUMBER>
scanner: ";"
                    reduction number 31   -- <PRIMARY> ::= <PRE PRIMARY>
                    reduction number 15   -- <FACTOR> ::= <PRIMARY>
                    reduction number 11   -- <PRODUCT> ::= <FACTOR>
                    reduction number 9    -- <TERM> ::= <PRODUCT>
                    reduction number 7    -- <ARITH EXP> ::= <ARITH EXP> + <TERM>
                    reduction number 181 -- <EXPRESSION> ::= <ARITH EXP>
                    reduction number 136 -- <ASSIGNMENT> ::= <VARIABLE> <=1> <EXPRESSION>
                    reduction number 41   -- <BASIC STATEMENT> ::= <ASSIGNMENT>
                    reduction number 36   -- <STATEMENT> ::= <BASIC STATEMENT>

   source line was:     A = A + 1;


                    reduction number 38   -- <ANY STATEMENT> ::= <STATEMENT>
                    reduction number 292 -- <BLOCK BODY> ::= <BLOCK BODY> <ANY STATEMENT>
scanner: "CLOSE"
scanner: <LABEL> = "SIMPLE"
                    reduction number 427 -- <CLOSING> ::= CLOSE <LABEL>
scanner: ";"
                    reduction number 289 -- <BLOCK DEFINITION> ::= <BLOCK STMT> <BLOCK BODY> <CLOSING>

   source line was: CLOSE SIMPLE;


                    reduction number 2    -- <COMPILE LIST> ::= <BLOCK DEFINITION>
scanner: "_|_"
                    reduction number 1    -- <COMPILATION> ::= <COMPILE LIST> _|_
```

NOTES:

Notice that the first time (in the DECLARE statment) the scanner sees "A", it returns an <IDENTIFIER>; however, all subsequent times it returns an <ARITH ID>.

Source lines appear at the point that the output writer would write them.

RECOVER -- 1534500
STACK_DUMP -- 1087300
SAVE_DUMP -- 280600

RECOVER is called by COMPILATION_LOOP when a syntactic error is discovered. Its job is to throw away enough of the parser's stacks and of the input stream to enable the parser to start working again.

Call STACK_DUMP to dump the current STATE_STACK. STACK_DUMP formats up lines and calls SAVE_DUMP to insert them in SAVE_STACK_DUMP for eventual printing by the output writer.

Advance the input stream to a semicolon or _|_.

Reset principle global flags to default status.

Pop elements off the STATE_STACK until CHECK_TOKEN indicates that STATE_STACK is compatible with TOKEN. Dump the reduced stack, output all the skipped material via the output writer and then pick up in COMPILATION_LOOP.

CHECK_TOKEN -- 1529700

This routine is called by RECOVER to check whether the current stack top, NSTATE, or pre-look ahead state, NLOOK, is compatible with the next token, NTOKEN. It returns 0 if not compatible or a new STATE number if okay.

For a read state, NTOKEN must appear in the appropriate part of INDEX2.

For a reduce state, do the reduction and try the reduced state.

For a look ahead state, search for a look ahead match and if found, do the reduction and continue checking.

```
EMIT_EXTERNAL -- 764600
EX_WRITE      -- 765300
```

EMIT_EXTERNAL is called by COMPILATION_LOOP to format up templates and output them via EX_WRITE.

At any given moment it is in one of five states determined by EXTERNALIZE. EXTERNALIZE is set by SYNTHESIZE which also calls EMIT_EXTERNAL to change its state.

0 - Not doing anything.

1 - Format templates -- be careful to handle macro texts properly (see MACRO_TEXT in SCAN).

2 - Clean up and set EXTERNALIZE to zero.

3 - Initialize and set EXTERNALIZE to one.

4 - Temporarily not doing anyting.

4.2 The Scanner

The scanner provides the input interface between the compiler and the world. The rest of the compiler deals with tokens and strings assembled by the scanner. The rest of the compiler deals with 1-dimensional format regardless of the input. The rest of the compiler deals with a single input stream regardless of include statements and macro expansions.

The scanner is divided into two parts. STREAM gets the next character and SCAN assembles characters into tokens. Since symbol table information is necessary to determine the token type, SCAN contains the symbol table routine -- IDENTIFY. Since some character strings are not delivered to the parser, they must be delivered directly to the output writer; thus, SCAN contains the routines for saving tokens. Since compiler directives and access rights are not part of the grammar, SCAN contains routines for handling these concepts.

4.2.1 SCAN

SCAN receives characters from STREAM and returns tokens to the parser. All symbol table searches are made here, macro expansions are processed here, a considerable amount of macro definition work is done here. The principle interfaces to the parser are TOKEN which is set to the internal code for the syntactic item read and SYT_INDEX which transmits additional information for semantic processing.

Notice that each call to SCAN returns a token; consequently, macro expansions must be done on the fly.

4.2.1.1 Local Variables of SCAN.

CHAR_ALREADY_SCANNED
Contains character which SCAN read after a "/" during look-ahead for comments; =0 if empty.

CHAR_NEEDED
Switch off when a character has been obtained from STREAM and has not yet been used.

DEC_POINT
Switch ON if decimal point has already been found in current numeric token.

DONT_ENTER

ESCAPE_LEVEL
Count of escape characters prefixed to NEXT_CHAR.

EXP_BEGIN
Index in INTERNAL_BCD of first character of exponent in current numeric token.

EXP_DIGITS
Length of exponent in characters.

EXP_SIGN
Sign (+ or -) of exponent of current numeric token.

FLAG
In IDENTIFY, used to accumulate flags for SYT_FLAGS.

I
In IDENTIFY, the symbol table index of the identifier.

INTERNAL_BCD
Copy of BCD used within SCAN.

L
In IDENTIFY, the length of the identifier.

OVERPUNCH_ALREADY_SCANNED
See CHAR_ALREADY_SCANNED.

SEARCH_NEEDED
SCAN attempts to position the input after all embedded comments before returning a token. If it is not successful, then SEARCH_NEEDED is set so that it will search for comments the next time it is entered.

SIG_DIGITS
Number of significatn digits in current numeric token.

4.2.1.2 Global Variables Referenced by SCAN.

ADDR_FIXED_LIMIT Address of a location containing, in floating format, the largest numeric literal allowed by HAL/S. See DW.

ADDR_FIXER Address of a location containing an increment to be used while checking a literal against fixed limits. See DW.

ADDR_VALUE Address of a location used to store the value of a numeric literal in full floating format. See DW.

ARITH_FUNC_TOKEN See global definitions -- TOKEN.

ARITH_TOKEN See global definitions -- TOKEN.

ASSIGN_PARM See symbol table -- SYT_FLAGS

BASE_PARM_LEVEL See STREAM.

BCD Character string of current item being assembled by SCAN.

BI_INDEX Similar to V_INDEX but for the names of built-in functions.

BI_INFO Indexing by built-in number gives word of information:

type (see SYT_TYPE)	# of args	pointer to BI_ARG_TYPE
32 25	24 17	16 9 8 1

For more detail, see SYNTHESIZE.

BI_NAME(J) Is the character string containing the name of the J th built-in function.

BIT_FUNC_TOKEN See global definitions -- TOKEN.

BIT_TOKEN See global definitions -- TOKEN.

BIT_TYPE See symbol table -- SYT_TYPE.

BLANK_COUNT See STREAM.

BUILDING_TEMPLATE	See SYNTHESIZE.
C	See O-W.
CHAR_FUNC_TOKEN	See global definitions -- TOKEN.
CHAR_OP (0 or 1)	Translates from 0 or 1 escapes to equivalent over punch escape character.
CHAR_TOKEN	See global definitions -- TOKEN.
CHAR_TYPE	See global definitions -- SYT_TYPE.
CHARACTER_STRING	See global definitions -- TOKEN.
CHARTYPE	See STREAM.
COMMA	See global definitions -- TOKEN.
COMMENT_COUNT	See O-W.
CONCATENATE	See global definitions -- TOKEN.

CONTEXT

The type of identifiers is determined by the scanner. Since the proper symbol table lookup depends on the context in which the identifier appeared, this context must be known to the scanner. EXPRESSION_CONTEXT means that compile time constants are expected for dimension information. DECLARE_CONTEXT means that the identifier is being declared for the current scope. PARM_CONTEXT means that the identifier is a formal parameter which is not yet declared in this scope, but will be. ASSIGN_CONTEXT is a special case of PARM_CONTEXT for assign parameters of procedures.

REPL_CONTEXT indicates that a REPLACE definition is being processed and so a macro name that otherwise would be "previously defined" can be defined. Once the macro name has been defined, we switch to REPLACE_PARM_CONTEXT which allows formal parameter names to conflict with anything except other formal parameters of the same macro.

Since a new CONTEXT is often started by a reserved word, SET_CONTEXT gives the appropriate context for each reserved word.

There are some other flags which augment CONTEXT. TEMPLATE_IMPLIED augments DECLARE_CONTEXT indicating that the token name is a template name (i.e. either a declaration of a template or of a structure variable). LABEL_IMPLIED indicates that a look ahead has found a colon and the context implies that the : is a label delimiter.

CPD_NUMBER — See global definitions -- TOKEN.

DECLARE_CONTEXT — See CONTEXT

DEF_BIT_LENGTH
DEF_CHAR_LENGTH
DEF_MAT_LENGTH
DEF_VEC_LENGTH — Default lengths for implicit declarations of variables.

DEFAULT_ATTR — See symbol table -- SYT_FLAGS.

DEFAULT_TYPE — Identical to SCALAR_TYPE. See symbol table -- SYT_FLAGS.

DEFINED_LABEL — See symbol table -- SYT_FLAGS

DONT_SET_WAIT — See PRINTING_ENABLED.

DUPL_FLAG — See symbol table -- SYT_FLAGS.

DW — 56 byte area reserved for floating point and literal operations; the area is needed because the operations are performed by MONITOR calls.

byte offset					index
0					0 ← DW_AD
4					1
24					6 ← ADDR_VALUE
32	4E	00	00	00	8 ← ADDR_FIXER
	0				
40	48	7F	FF	FF	10 ← ADDR_FIXED_LIMIT
	FF	FF	FF	FF	
48	40	7F	FF	FF	12 ← ADDR_ROUNDER
	FF	FF	FF	FF	

EOFILE	See global definitions -- TOKEN.
ESCAPE	The escape character.
EVENT_TOKEN	See global definitions -- TOKEN.
EVIL_FLAG	See symbol table -- SYT_FLAGS.
EXP_OVERFLOW	Switch ON if a character representation could not be converted to floating point number.
EXP_TYPE	Exponent indicator on current numeric token; 'E', 'H', 'B' allowed.
EXPONENTIATE	See global definitions -- TOKEN.
EXPRESSION_CONTEXT	See CONTEXT.
FACTORING	See SYNTHESIZE.
FIRST_FREE	See MACRO_TEXT.
FIRST_TIME	See STREAM.
FIRST_TIME_PARM	See STREAM.
FOUND_CENT	On if macro substitution markers (i.e. ¢name¢) were found while scanning the macro parameter.
GROUP_NEEDED	See STREAM.
ID_TOKEN	See global definitions -- TOKEN.
IDENT_COUNT	Total number of calls to IDENTIFY, for compilation statistics.
IMP_DECL	See symbol table -- SYT_FLAGS.
IMPLICIT_T	Switch ON if token may be the matrix transpose symbol 'T'.

IMPLIED_TYPE — Token type, as implied by presence of overpunch.

INACTIVE_FLAG — See symbol table -- SYT_FLAGS.

IND_CALL_LAB — See symbol table -- SYT_TYPE.

INPUT_PARM — See symbol table -- SYT_FLAGS.

INT_TYPE — See symbol table -- SYT_TYPE.

KIN — Index in symbol table of a structure element underneath the structure indexed by QUALIFICATION.

LAB_TOKEN — See global definitions -- TOKEN.

LABEL_CLASS — See symbol table -- SYT_CLASS.

LABEL_IMPLIED — See CONTEXT.

LEFT_PAREN — See global definitions -- TOKEN.

LETTER_OR_DIGIT — See STREAM.

LEVEL — See global definitions -- TOKEN.

LOOKUP_ONLY — Switch ON if IDENTIFY should only search the symbol table without creating a token.

M_BLANK_COUNT (macro_expan_level)

Is the BLANK_COUNT after reading the complete macro invocation.

M_CENT — See STREAM.

M_P (macro_expan_level) — Is the saved value of MACRO_POINT for this level.

M_PRINT (macro_expan_level)

Is the saved value of PRINTING_ENABLED for this level.

M_TOKENS (macro_expan_level)
Equals number of tokens created while expanding this macro.

MACRO_ARG_COUNT
The number of formal arguments so far encountered in REPLACE definition.

MACRO_ARG_FLAG
See global definitions -- GRAMMAR_FLAGS.

MACRO_CALL_PARM_TABLE
Contains the character strings for the values of the actual parameters of all currently expanding REPLACEs. Outer REPLACEs are lower in the table and the left-most parameter is lower than the right-most.

MACRO_EXPAN_LEVEL
Nesting depth of macro expansion.

MACRO_EXPAN_STACK (macro_expan_level)
Equals symbol table entry for REPLACE name.

MACRO_FOUND
On if REPLACE name has been found and requires expansion.

MACRO_NAME
REPLACE name being defined.

MACRO_POINT
Pointer to current point in <text> of current macro in MACRO_TEXT.

MACRO_TEXT
The <text> part of a REPLACE statement is stored in MACRO_TEXT by SCAN. START_POINT points to the beginning of the current <text>, T_INDEX points to the next character position, and FIRST_FREE points to the beginning of the next <text>. Pairs of " marks have been replaced by single " marks. Multiple blanks have been replaced by "EE" followed by BLANK_COUNT. The <text> is ended by an "EF".

MAJ_STRUC
See symbol table -- SYT_TYPE.

MAT_TYPE
See symbol table -- SYT_TYPE.

NAME_HASH
See STREAM.

NAMING See SYNTHESIZE.

NDECSY See Symbol Table.

NEW_MEL See OLD_MEL.

NEXT_CHAR The next character from STREAM.

NO_ARG_ARITH_FUNC See global definitions -- TOKEN

NO_ARG_BIT_FUNC See global definitions -- TOKEN.

NO_ARG_CHAR_FUNC See global definitions -- TOKEN.

NO_ARG_STRUCT_FUNC See global definitions -- TOKEN.

NONHAL_FLAG See symbol table -- SYT_FLAGS.

NUM_OF_PARM See STREAM.

NUMBER See global definitions -- TOKEN.

OLD_MEL Saved value of MACRO_EXPAN_LEVEL to enable detection of an exit from a macro expansion.

OLD_MP Saved value of MACRO_POINT -- enables some look ahead in the text.

OLD_PEL Similar to OLD_MEL for PARM_EXPAN_LEVEL.

OLD_TOPS Saved value of TOP_OF_PARM_STACK.

OUTER_REF Used to collect uses of scoped in variables for printing by BLOCK_SUMMARY. An entry has the form:

flag	symbol
3	13

where flag is as in XREF and symbol is a pointer to the symbol table entry for the referenced variable. OUTER_REF_INDEX points to the last entry in OUTER_REF and OUTER_REF_LIM is the size of OUTER_REF. OUTER_REF_PTR(nest) has the form:

switch	pointer
1	15

where pointer points to the first OUTER_REF entry for level nest and switch is set after printing the overflow message to inhibit multiple printing of the message.

OUTER_REF_INDEX
OUTER_REF_LIM
OUTER_REF_PTR
See OUTER_REF.

OVER_PUNCH
See STREAM.

OVER_PUNCH_TYPE
If OVER_PUNCH_TYPE(I) = char then an over punch of char implies that the identifier is of type I.

P_CENT
See STREAM.

PARM_CONTEXT
See CONTEXT.

PARM_COUNT
Number of parameters in stack examined by PARM_FOUND. TOP_OF_PARM_STACK - PARM_COUNT gives the stack offset of the current macro's parameters.

PARM_EXPAN_LEVEL	See STREAM.
PARM_REPLACE_PTR	See STREAM.
PARM_STACK_PTR	See STREAM.
PASS	Used for saving value of PRINTING_ENABLED during macro expansion.
PC_LIMIT	Length of longest %macro name.
PCNAME	String containing names of %macros, left-justified in 16-character fields.
PERCENT_MACRO	See global definitions -- TOKEN.
PRINT_FLAG	See global definitions -- GRAMMAR_FLAGS.
PRINTING_ENABLED	A token is ultimately printed if PRINT_FLAG is on in GRAMMAR_FLAGS. This decision is made by an AND of PRINTING_ENABLED (general context) and SUPPRESS_THIS_TOKEN_ONLY (local). When changing PRINTING_ENABLED it is possible to delay its effect for one word by setting WAIT. WAIT is set when exiting a macro expansion. If the expansion generated no tokens, setting WAIT is inhibited by DONT_SET_WAIT.
PROC_LABEL	See symbol table -- SYT_TYPE.
PROCMARK	Index into symbol table - everything below it was declared in other (outer) procedure blocks.
QUALIFICATION	See Section 4.4.
RECOVERING	See O-W.
REF_ID_LOC	See Structures and Templates.
REPL_ARG_CLASS	See symbol table -- SYT_CLASS.
REPL_CLASS	See symbol table -- SYT_CLASS.
REPL_CONTEXT	See CONTEXT.
REPLACE_PARM_CONTEXT	See CONTEXT.

REPLACE_TEXT	See global definitions -- TOKEN.
RESERVED_LIMIT	Length of longest reserved word.
RESERVED_WORD	Switch ON if current token is a HAL/S reserved word.
RESTORE	Used to save the value of PRINTING_ENABLED during macro expansion.
RT_PAREN	See global definitions -- TOKEN.
SAVE_BLANK_COUNT	When SCAN is searching for a non-blank (on macro exit this can be a problem) SAVE_BLANK_COUNT is used to save the last BLANK_COUNT.
SAVE_COMMENT	See O-W.
SAVE_NEXT_CHAR	See STREAM.
SAVE_OVER_PUNCH	See STREAM.
SAVE_PE	Saved value of PRINTING_ENABLED used to make printing decisions at the end of macro or macro parameter expansions.
SCALAR_TYPE	See symbol table -- SYT_TYPE.
SCAN_COUNT	Total number of TOKENs SCANned -- for compilation statistics.
SET_CONTEXT	See CONTEXT.
SOME_BCD	Contains the substring of BCD up to the point where ¢name¢ was discovered.
SQUEEZING SRN SRN_COUNT SRN_PRESENT	See O-W.
START_POINT	See MACRO_TEXT.
STMT_LABEL	See symbol table -- SYT_TYPE.
STMT_PTR	See GRAMMAR_FLAGS.
STRING_OVERFLOW	Switch ON if character literal is too long.

STRUC_TOKEN	See global definitions -- TOKEN.
STRUCT_FUNC_TOKEN	See global definitions -- TOKEN.
STRUCT_TEMPLATE	See global definitions -- TOKEN.
STRUCTURE_WORD	See global definitions -- TOKEN.
SUPPRESS_THIS_TOKEN_ONLY	See PRINTING_ENABLED.
SYT_INDEX	For literals, its absolute index in the literal tables; for built-ins, the index of built-in functions in BI_INFO; for %macros, the internal number of the macro; for other identifiers, a symbol table pointer. SYT_INDEX is zeroed at SCAN_START.
T_INDEX	See MACRO_TEXT.
TASK_LABEL	See symbol table -- SYT_TYPE.
TEMP_STRING	Used to accumulate character strings in analyzing macro calls.
TEMPL_NAME	See symbol table -- SYT_TYPE.
TEMPLATE_CLASS	See symbol table -- SYT_CLASS.
TEMPLATE_IMPLIED	See CONTEXT.
TEMPORARY	See global definitions -- TOKEN.
TEMPORARY_FLAG	See symbol table -- SYT_FLAGS.
TEMPORARY_IMPLIED	Switch ON if TEMPORARY keyword has been read in this statement.
TOKEN	The type of the current token. A value of -1 indicates a REPLACE name. For definition of other values, see global variables.
TOKEN_FLAGS	See global definitions -- GRAMMAR_FLAGS.
TOP_OF_PARM_STACK	Points to the top of the MACRO_CALL_PARM_TABLE. Parameter lists being scanned are built immediately above this point.

TRANS_IN (Char)	Is a two byte translation table for char. The right byte is the single escape translation and the left byte is the double escape translation.
TX (Char)	Is the internal TOKEN code for the special character char.
UNSPEC_LABEL	See symbol table -- SYT_TYPE.
V_INDEX	See procedure SCAN -- identifiers.
VALID_00_CHAR	Input character that can be escaped to give "00".
VALID_00_OP	Overpunch required to translate VALID_00_CHAR to "00".
VALUE	Numerical value of current token (if token is numeric).
VAR_CLASS	See symbol table -- SYT_CLASS.
VAR_LENGTH	See symbol table -- identical to SYT_DIMS.
VEC_TYPE	See symbol table -- SYT_TYPE.
VOCAB_INDEX	See procedure SCAN -- identifiers.
WAIT	See PRINTING_ENABLED.
XREF_REF	See symbol table -- SYT_XREF.
X1	1 blank.

SCAN	-- 577700
CALL_SCAN	-- 967400
BUILD_BCD	-- 579000
BUILD_INTERNAL_BCD	-- 580000
ID_LOOP	-- 606200
CHAR_OP_CHECK	-- 578708
BUILT_COMMENT	-- 755800

SCAN is called from three places: INITIALIZATION, RECOVER and COMPILATION_LOOP. The call from INITIALIZATION is executed once and gets everything primed, the other calls are all routed through CALL_SCAN and are genuine requests for another token. The purpose of interposing CALL_SCAN is to allow clean handling of some diagnostic printing. SCAN calls STREAM to get characters one by one in NEXT_CHAR. It puts them together in BCD until it finds a delimiter and then determines the TOKEN type of BCD. TOKEN SYT_INDEX, and BCD are SCAN's principal interfaces to the outside world.

The global structure of the routine is a DO CASE on the type of the first character of the next token. Each case in turn accumulates the rest of the token and builds BCD and an internal version via BUILD_BCD and BUILD_INTERNAL_BCD. In addition, it may issue error messages based on the context; for instance, if the first character is a digit, the token must be a number which can contain only characters from a given set and may be delimited only by characters from some second set.

Since all macro and macro parameter expansions are handled at the scanner level, a large number of items may be read before a syntactic token is obtained; thus, the routine may very well execute several cycles of "pick up word; set up to expand word; go back to the beginning".

After accumulating a token but before actually returning it, SCAN looks to see if the next thing in the input stream is an embedded comment. If it is, the comment is accumulated one character at a time using BUILD_COMMENT to save the characters in SAVE_COMMENT. Finally, the token is returned.

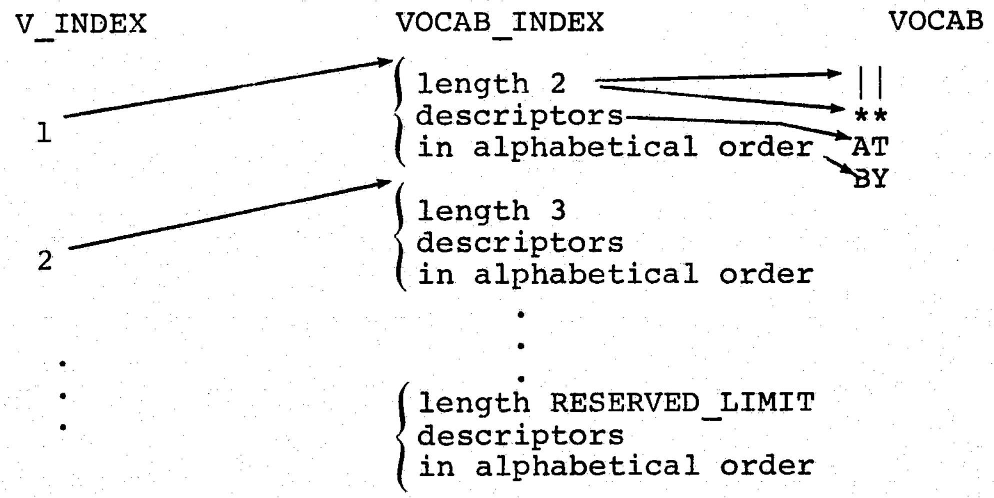
Details of the Central DO CASE

1 - numbers

Accumulate the entire number including exponent if any. Convert number to 360 floating point, check it for range and enter it in the literal table via PREP_LITERAL.

2 - identifiers

This is where most of the work starts. First, use ID_LOOP to accumulate the identifier and set IMPLIED_TYPE if there is an overpunch. Then search the list of reserved words for the identifier. The tables are organized like this:



The reason for the explicitly hand crafted descriptors is to prevent overflow of the limited size descriptor table. If the identifier is a reserved word, set up the CONTEXT (see data description) and return.

If the identifier is not a reserved word, it may be a macro parameter. Check this via PARM_FOUND and if it is, expand the parameter. Notice that the parameter never generates a token itself.

If the identifier is not a macro parameter, then look it up in the symbol table via IDENTIFY but do not return it yet -- maybe its a macro call. If it is not a macro name then return TOKEN as set up by IDENTIFY; otherwise, set up the macro expansion via PUSH_MACRO and then start taking characters from the expansion. Notice that the macro call itself does not actually generate a token.

4 - period

If next character is a digit, build a decimal fraction in the normal way; otherwise, return dot product TOKEN.

5 - character literal

Build the string by concatenating characters. Be careful to:

- expand multiple blanks
- check for ' ' and replace it by '
- translate escaped characters using CHAR_OP_CHECK.

Return a character string TOKEN.

7 - | or ||

Return either an OR or a CAT TOKEN.

8 - * or **

Return either a cross product or exponentiate TOKEN.

9 - "FE" = end of file

Return an end of file TOKEN.

10 - Special Characters Treated as Blanks

Simulate blank and reenter SCAN.

11 - " = REPLACE Text

Insert the text in MACRO_TEXT. Be careful to:

- replace " " by "

- encode BLANK_COUNT for multiple blanks

- insert "EE" end of macro character.

Return a replace text TOKEN.

12 - %macros

Accumulate entire name; return index of name in SYT_INDEX and return percent macro TOKEN.

13 - REPLACE macro call

This code is reached if the first character is a "¢" or if a ¢ was found while scanning an identifier in case 2. In the former situation, after accumulating and setting up for the expansion of the macro or parameter name, the code simply starts from the beginning of the scanner. In the latter situation, the code must set up for expansion and then return back to finish accumulating the identifier it was originally working on. Notice that if the source is:

```
¢macro_name(args)¢
```

then the second ¢ is not read by this code. It is checked by PARAMETER_PROCESSING and skipped by PUSH_MACRO.

PUSH_MACRO -- 625200

PUSH_MACRO is called to handle a macro call.

Push symbol table entry for macro onto MACRO_EXPAN_STACK, push the macro name onto STMT_STACK via SAVE_TOKEN, set up NUM_OF_PARM so that number of actual parameters can be compared with the number of formal parameters. Read in the actual parameters via PARAMETER_PROCESSING.

PARAMETER_PROCESSING -- 580900

PARAMETER_PROCESSING is called by PUSH_MACRO after finding a macro name to build a list of the actual macro parameters in MACRO_CALL_PARM_TABLE. The parameters are entered into STMT_STACK via SAVE_TOKEN. The bulk of the routine simply updates pointers and counters described in the data description section. Notice that although PARAMETER_PROCESSING reads a lot of information, it does not actually generate any tokens but simply prepares for a macro expansion.

PARM_FOUND -- 615700

PARM_FOUND is called for each non-reserved word identifier to check if it is a formal parameter of a macro being expanded. The symbol table entries for the formal parameters are immediately after the entry for the macro; thus, PARM_FOUND need only loop comparing BCD to SYT_NAME. If a match is found, it is stacked in the parameter stack and TRUE is returned; otherwise, FALSE is returned.

IDENTIFY builds the symbol table and searches it for identifiers. In principal, this should be a triviality; however, the mass of detail and the requirement of performing IDENTIFY at SCAN time makes things substantially more complex.

IDENTIFY receives two arguments. BCD is the character string to be looked up. CENT_IDENTIFY is true if the name was enclosed in "¢" signs. It returns values in SYT_INDEX and TOKEN.

To look up a name in the symbol table, compute NAME_HASH = HASH(name). NAME_HASH is an index into the hash table HASHSTART, thus, if I = HASHSTART(NAME_HASH), then I points to a symbol table entry with the given hash code. Symbol table entries with the same hash code are linked via their SYT_HASHLINK fields; thus, if entry I is not the right one, try I = SYT_HASHLINK(I). If the link is zero, there are no more entries for that hash code.

Before looking up a name in the symbol table, if it is a template name, prefix it with a blank; if it is an EQUATE name, prefix it with a @; try looking it up in the table of built-in function names.

The universe of names is divided into two parts, those that are already in the table and those that are not.

Name Already in Table

If the name is a macro name then either set up to expand it or simulate a "name not found" to permit a new declaration for the macro name.

It would be nice now to simply return the symbol table pointer but the actual actions required depend on the context in which the identifier appears (cf. CONTEXT).

For the run of the mill situation:

- variables -- set TOKEN appropriately.
- labels -- set TOKEN, create cross reference, check legality.
- functions -- check legality, set TOKEN appropriately.

- templates -- notice that all qualifier names in a structure reference are template names. Search the descendants of the node currently reached (as indicated by QUALIFICATION). If the name is there, move QUALIFICATION to this entry; otherwise, move through hash link for an alternative symbol table entry to try.

In EXPRESSION_CONTEXT, process like run of the mill.

After a GO TO, if the name is not local or not a label, create a new entry; otherwise, check legality.

After a CALL, if the name is not local, create a local entry of type IND_CALL_LAB pointing to the non-local entry. Check for legality.

After SCHEDULE, process normally.

In DECLARE_CONTEXT, if the existing entry is from an outer scope, make a new one. If in the middle of constructing a template, set to indicate that the name already exists (which is legal in a structure qualifier) and go pick up in the hash links.

Name Not Already in Table

Once again, the appropriate action depends on the CONTEXT.

For the ordinary case; labels are detected by spotting the colon and defining them to be of type UNSPEC_LABEL (see SYT_FLAGS); a T ought to be a transpose operator; everything else is a use of an undeclared name (this is not DECLARE_CONTEXT) and is therefore illegal so print an error message and default type it.

Only declared names may appear in EXPRESSION_CONTEXT.

After a GO TO, create an entry for a label that will be defined later.

After a CALL, create an entry for a procedure name which will be defined later.

After a SCHEDULE, create an entry for a task name which will be defined later.

In DECLARE_CONTEXT, create an entry and return it unless the name was previously located in which case just return the previous entry.

After REPLACE, make an entry for a macro name and switch CONTEXT to expect formal parameters.

SAVE_TOKEN -- 399700
OUTPUT_STACK_RELOCATE -- 400500

The source listing is ultimately printed by the output writer. The output writer is invoked only when appropriate "new E/M/S group" points are reached; thus, the material to be printed must be saved somewhere in the interim. The saving operation is performed by SAVE_TOKEN which is called by the parser whenever it receives a token. Since macro calls are invisible to the parser, they are transmitted directly to SAVE_TOKEN from SCAN.

SAVE_TOKEN receives the token code in TOKEN, the character string in CHAR, and the type (i.e. SYT_TYPE) in TYPE. It puts the type in TOKEN_FLAGS. If the item is not a reserved word it saves the character string in SAVE_BCD and a pointer to SAVE_BCD in TOKEN_FLAGS. The token is saved in STMT_STACK. GRAMMAR_FLAGS is set to indicate whether or not to print the item.

There are two things that can overflow. STMT_PTR can get too large or BCD_PTR can get too large. If either happens, OUTPUT_STACK_RELOCATE is called to force some printing and then a relocation of all unprinted material down in the stack.

ENTER	--	556200
SET_XREF	--	552300
ENTER_XREF	--	549400
SET_OUTER_REF	--	547800
COMPRESS_OUTER_REF	--	533000

ENTER receives a name and class for an identifier and creates a symbol table entry for it. The hash table is modified to point to this symbol table entry first and the identifier usage is entered in the cross reference table via SET_XREF. Notice that if the entry is a formal parameter of a macro, it is entered after the current entry in the hash link if possible.

SET_XREF receives a symbol table pointer (LOC), an XREF flag (FLAG), and a second XREF flag (FLAG2). SET_XREF builts a new (or adds to an existing) XREF entry and connects it to the appropriate linked XREF list via ENTER_XREF. If the variable is declared in an enclosing scope, SET_OUTER_REF is called to make FLAG2 entry in OUTER_REF. Notice that unless told otherwise, a subscript usage will be converted to a reference usage for SET_OUTER_REF. If the OUTER_REF array overflows, SET_OUTER_REF will in turn call COMPRESS_OUTER_REF to compress out duplicate entries in OUTER_REF.

SAVE_LITERAL	--	569800
PREP_LITERAL	--	574100
GET_LITERAL	--	175900

SAVE_LITERAL adds literals to the literal table (see Section 3.1). Before performing any manipulations on the paged part of the table, it uses GET_LITERAL to load the proper page and convert the absolute literal table index to an index relative to the current page. SAVE_LITERAL returns the absolute literal table index for the literal saved.

When dealing with character strings, INLINE code is necessary because it is necessary to copy the character strings to LIT_CHAR. The obvious XPL code would copy only the descriptor.

Notice that at this level, multiple instances of a literal generate multiple copies in the literal table. Phase II will generate only one copy of each desired literal.

PREP_LITERAL takes a floating point number fresh from creation by a MONITOR(10) call, checks it for proper limits, enters it in the literal table via SAVE_LITERAL and sets SYT_INDEX to the absolute index of the literal.

4.2.2 STREAM

STREAM is the character level half of the scanner. It actually reads the input, processes compiler directives, and passes to SCAN a single linear stream of characters.

4.2.2.1 Variables of STREAM.

ARROW

Displacement, in number of lines, of the current character relative to the last character transmitted; used to detect flying exponents and to regenerate parentheses around E or S groups.

ARROW_FLAG

When returning created characters, the information about the next real character is saved in SAVE_BLANK_COUNT1, SAVE_NEXT_CHAR1, and SAVE_OVER_PUNCH1. ARROW_FLAG indicates that this information should be restored and used before moving to the next character.

BLANKS

Blank field, 44 characters long.

CP

Card pointer - index of character being scanned on current card.

E_BLANKS

E_IND indicates blank compression internal to E_STACK. E_BLANKS indicates blank compression at the end. That is, there were E_BLANKS blanks compressed off the end of E_STACK. E_BLANKS can be: -1--E_STACK ends with non-blank; 0--E_STACK ends with a single blank; >0--blanks were compressed off.

E_COUNT

Number of E-lines in current group.

E_IND — If E_STACK(point) is blank, then E_IND(point) blanks were compressed out; otherwise, when E_STACK(point) was copied from E_LINE(index), E_IND(point) was copied from E_INDICATOR(index). S_IND is reached just like S_INDICATOR.

E_INDICATOR — See procedure COMP.

E_LINE — See procedure COMP.

E_STACK — Holds complete exponent ready for transmission - strings of blanks have been compressed using E_IND. If no non-blank characters were found in the exponent BUILD_XSCRIPTS set E_STACK to null.

EP — 0 - Index of last character in E_STACK.
1 - Index of last character in S_STACK.

IND_SHIFT — Literally 7 -- used to create references to S_array name(sub) by writing E_array name(sub + 2^{IND_SHIFT}).

INDEX — Index of next non-blank character in M line.

INPUT_PAD — Special M-line card generated at EOF = [M /**/ @ @ ' @ @]. The /**/ terminates any open comments; the @ is an EOF mark and the ' closes any open quotes.

M_BLANKS — See E_BLANKS.

M_LINE — The actual character string of the M line.

POINTER — When returning characters from an exponent or subscript, POINTER points to the next character in E_STACK or S_STACK.

PREV_CARD — Card type of previous input line - used to check EMS sequencing via ORDER_OK.

RETURN_CHAR — See TYPE_CHAR.

RETURNING_E	Switch ON if in the process of returning characters from E_line, initially false.
RETURNING_M	See RETURNING_E, initially true.
RETURNING_S	See RETURNING_E, initially false.
S_BLANKS	See E_BLANKS.
S_COUNT	Number of S lines in current group (see procedure COMP).
S_IND(i)	E_IND(i + 2^{IND_SHIFT}) but used for subscripts.
S_INDICATOR	See procedure COMP.
S_LINE	See procedure COMP.
S_STACK(i)	E_STACK(i + 2^{IND_SHIFT}) but used for subscripts.
SAVE_BLANK_COUNT1	See ARROW_FLAG.
SAVE_NEXT_CHAR1	See ARROW_FLAG.
SAVE_OVER_PUNCH1	See ARROW_FLAG.
SP	EP(1), but used for subscripts.
TYPE_CHAR	When reading multi-line input, STREAM simulates linear input by adding subscript, superscript, and parenthesis characters. Whenever the line level changes, the necessary characters are inserted in TYPE_CHAR and returned on successive calls to STREAM. Since sometimes the same TYPE_CHAR appears several times in succession, RETURN_CHAR is used to hold a repeat count.

4.2.2.2 Global Variables Referenced by STREAM.

ACCESS_FLAGS
See symbol table -- SYT_FLAGS.

ACCESS_FOUND
Switch ON if any ACCESS attributes have been coded in this compilation.

BASE_PARM_LEVEL(macro_expan_level)
The value of PARM_EXPAN_LEVEL on entry to this macro. When PARM_EXPAN_LEVEL > BASE_PARM_LEVEL, parameter expansion is underway.

BLANK_COUNT
If STREAM finds a string of blanks, it returns only one in NEXT_CHAR and sets BLANK_COUNT to the number compacted out.

BLOCK_MODE
=0 before encountering the primary unit of compilation (after which 'D PROGRAM' cards are invalid). See SYNTHESIZE for more detail.

CARD_COUNT
Number of cards read from all input files.

CARD_TYPE
Indexing by hex card type (E, M or blank, S, C, or D) yields DO-CASE code (1, 2, 3, or 4, respectively).

CHARTYPE (byte)
Is the type of the associated character, 0= illegal, 1 = digit, 2 = alphabetic; ...

COMMENTING
Switch ON for every card read after the first one in a series; used to suppress double spacing on output.

CURRENT_CARD
Card image buffer, filled by READ_CARD from input file.

END_GROUP
Switch ON if CURRENT_CARD contains the beginning of a new EMS group -- set by ORDER_OK.

END_OF_INPUT
Switch ON if EOF read on input file.

ENDSCOPE_FLAG
See symbol table -- SYT_FLAGS.

FIRST_TIME (macro_expan_level)

True almost all the time. Set false after putting out created blank after macro expansion so that only one blank is created. M_CENT indicates that the macro call was in ¢ signs so that not even the first blank should be created.

FIRST_TIME_PARM (parm_expan_level)

Like FIRST_TIME but used for actual parameters.

GROUP_NEEDED — Switch ON if STREAM buffers have been exhausted and GET_GROUP must be called.

INCLUDE_END — On if just read END on INCLUDE file.

INCLUDE_COMPRESSED — Switch ON if current include file is in compressed format.

INCLUDE_LIST — Switch ON if include file is being listed at all (default is ON - turned OFF by 'D INCLUDE ... NOLIST' card option).

INCLUDE_LIST2 — Switch ON if include file is being printed on secondary listing (cf. INCLUDE_LIST).

INCLUDE_MSG — Name of current include file - used in messages, set by PROCESS_COMMENT.

INCLUDE_OFFSET — Absolute position with respect to input stream of first include card -- the relative position of the current card within the include file can be calculated from CARD_COUNT-INCLUDE_OFFSET. When reading from primary file, INCLUDE_OFFSET is set up to subtract out the sum of all previous include files; thus giving the relative position within the primary file.

INCLUDE_OPENED — Switch ON if include file open.

INCLUDING — On if reading from INCLUDE file.

INITIAL_INCLUDE_RECORD

Switch ON if first record of include file is already in CURRENT_CARD.

INPUT_DEV — Current source file (0=SYSIN, 4=include file).

INPUT_REC — Input buffer for DECOMPRESS, (0) SYSIN, (1) include file.

IODEV See SYNTHESIZE.

LETTER_OR_DIGITS (character)
Is true if and only if character belongs to the set {A-Z, a-z, _, 0-9}. When reading ACCESS files, $ is temporarily added to the set.

LISTING2 Switch ON if secondary (unformatted) listing is being produced.

LRECL Length of records in INPUT_REC.

M_BLANK_COUNT See SCAN.

M_CENT See FIRST_TIME.

M_P See SCAN.

MACRO_CALL_PARM_TABLE See SCAN.

MACRO_EXPAN_LEVEL Current depth of macro expansion nesting -- indexes macro processing stacks.

MACRO_FOUND ON if a macro name has been identified and needs expansion.

MACRO_POINT See SCAN.

MACRO_TEXT See SCAN.

NAME_HASH Hased code for a name - used to index SYT_HASHLINK.

NEW_LEVEL Relative to line number of line containing the current character. Value is 0 for M line, 1 for line above M line, -1 for line below M line, etc.

NEXT Index of last line in SAVE_GROUP.

NEXT_CHAR This is the principal interface between STREAM and SCAN. The next character as a bit(8) is delivered here. See also BLANK_COUNT.

NONBLANK_FOUND Switch ON if STACK found a non-blank character when stacking sub/super script.

NUM_OF_PARM (macro_expan_level)

Is the number of parameters required for that macro.

OLD_LEVEL Level of last character transmitted (cf. NEW_LEVEL).

OVER_PUNCH If ≠ 0, character is punched directly over NEXT_CHAR (i.e., on E line).

P_CENT Like M_CENT only used for actual parameters.

PARM_EXPAN_LEVEL When expanding REPLACE parameters, this indexes the stacks required by the observation that actual parameters may in turn contain parameters from calling macros which must be expanded in line.

PARM_REPLACE_PTR (parm_expan_level)

Is a pointer to the next character in MACRO_CALL_PARM_TABLE (PARM_STACK_PTR) to be passed by STREAM.

PARM_STACK_PTR (parm_expan_level)

Is the actual parameter being expanded.

PROGRAM_ID Name of access control file, from 'D PROGRAM' card.

READ_ACCESS_FLAG See symbol table -- SYT_FLAGS.

SAVE_CARD Copy of CURRENT_CARD made by READ_CARD, stored by SAVE_INPUT for secondary listing.

SAVE_GROUP	Stack of lines to be printed on LISTING2 file, collected by SAVE_INPUT, printed by OUTPUT_GROUP.
SAVE_NEXT_CHAR	Most recent value of NEXT_CHAR; saved here while macro processing goes on.
SAVE_OVER_PUNCH	See SAVE_NEXT_CHAR.
STARS	Field of 5 stars - used in listing messages.
TEXT_LIMIT	Number of columns reserved for HAL/S text on input card - everything to the right is put into SRN.
TOO_MANY_LINES	Switch ON if SAVE_GROUP is full.
TOP_OF_PARM_STACK	See SCAN.
X1	Blank fields of 1,4,70 and 8 characters.
X4	Blank fields of 1,4,70 and 8 characters.
X70	Blank fields of 1,4,70 and 8 characters.
X8	Blank fields of 1,4,70 and 8 characters.

4.2.2.3 Procedures of STREAM.

STREAM	--	350000
STACK_RETURN_CHAR	--	384500

The routine is essentially broken into two independent parts, the first part delivers characters from REPLACE expansions and the second part delivers characters from source lines.

When expanding macros it is possible to be nested inside several macro expansions and several parameter expansions -- the necessary detail is part 1.

When handling source lines, characters are created to simulate the linear input format -- created but undelivered characters are saved in TYPE_CHAR using STACK_RETURN_CHAR. Characters can come from the M line, the S line, or the E line. After trying them in turn, get some more input via BUILD_XSCRIPTS.

BUILD_XSCRIPTS	--	408400
STACK	--	405300
CHOP	--	403900

BUILD_XSCRIPTS advances to the next non-blank in the M line, accumulating a compressed exponent in E_STACK and compressed subscript string in S_STACK.

STACK is called from BUILD_XSCPTS with argument 0 for exponent and 1 for subscript. STACK appends the character to the appropriate S or E STACK unless it is a multiple blank in which case it just counts it.

CHOP advances to the next character position.

GET_GROUP -- 395200

The principal function of GET_GROUP is to read in a single E/M/S group and linearize it for easier handling by the rest of STREAM.

COMP(0) is called to handle E lines.

COMP(1) is called to handle S lines.

The M line is simpler and so is handled in line.

PROCESS_COMMENT is called to handle comments.

The linearized exponents and subscripts are described in COMP; the M line is already linear. The three lines are returned in E_LINE, M_LINE, and S_LINE.

OUTPUT_GROUP -- 191800

OUTPUT_GROUP is called to print the previous group as saved by SAVE_INPUT on the secondary listing. It is usually called by GET_GROUP but is also called once by PRINT_SUMMARY to clean up at the end.

COMP	--	392200
SCAN_CARD	--	389800
READ_CARD	--	385700
SAVE_INPUT	--	187800
NEXT_RECORD	--	343800
ORDER_OK	--	345500

Notice that the declarations for E_INDICATOR and S_INDICATOR are such that they will be allocated contiguously; thus, when subscripting E_INDICATOR with values greater than 127, the S_INDICATOR is set.

E_INDICATOR	S_INDICATOR
0-127	128-256

The computations POINT=SHL(TYPE, IND_SHIFT) E_INDICATOR(CP+POINT)=... have this effect since TYPE=0 for E lines and 1 for S lines implies that POINT will be 0 for E lines and 128 for S lines.

A similar procedure is followed for the E_LINE/S_LINE pair and the ECOUNT/SCOUNT pair:

```
E_LINE = ELINE(0) .........
S_LINE = ELINE(1) .........
```

All exponent lines are linearly compacted into E_LINE and all subscript lines are linearly compacted into S_LINE. E_INDICATOR and S_INDICATOR contain the line number of the line originally containing the character where the highest of N exponent lines is numbered N and the number is decremented down to 1 for the line immediately above the M line. The first subscript line is numbered 1 and this number is incremented for each succeeding subscript line.

SCAN_CARD is called by COMP to set up E_LINE, E_INDICATOR, S_LINE, and S_INDICATOR, and issues error messages for illegally overlapping characters.

READ_CARD is called by COMP to obtain the next input card via NEXT_RECORD; to manage EOF indicators; to save the source lines for the secondary listing via SAVE_INPUT and to count cards.

COMP itself keeps track of a change in the type of the cards, checks their order via ORDER_OK, and switches the exponent line numbers from 1...N to N...1.

COMP is called by GET_GROUP with TYPE=0 for E lines and TYPE=1 for S lines.

PROCESS_COMMENT	-- 356500
PRINT_COMMENT	-- 357200
D_TOKEN	-- 354900

PROCESS_COMMENT is called by GET_GROUP to hande C or D cards. C cards are scanned for ¢ toggles which are set, reset or inverted as requested. D cards are scanned for directives using D_TOKEN to get the next token from the card.

The entire processing of D card directives is performed here including the opening of an INCLUDE file and the processing of PROGRAM directives via INTERPRET_ACCESS_FILE.

Comments and directives are printed on the secondary listing via PRINT_COMMENT.

INTERPRET_ACCESS_FILE	-- 316800
ADVANCE_CP	-- 323100
NEXT_TOKEN	-- 324800
ACCESS_ERROR	-- 317600
LOOKUP	-- 321600
RESET_ACCESS_FLAG	-- 320700

INTERPRET_ACCESS_FLAG is called by PROCESS_COMMENT when a PROGRAM directive is processed. INTERPRET_ACCESS_FILE reads and processes the access file (unit 6).

ADVANCE_CP is used to increment the Card Position by 1, reading a new card when necessary and finally setting EOF_FLAG. The function NEXT_TOKEN reads the input out of S(CP) using ADVANCE_CP and builds tokens returning either 0 and a token in A_TOKEN or a delimiter number.

The file is read and errors are reported using ACCESS_ERROR which takes an error message number and a character string arguments to be printed. When an identifier is read, it is located in the symbol table using the function LOOKUP which takes an identifier as an argument and returns a symbol table pointer or -1. When a symbol to be accessed is located, it's access protection is turned off using RESET_ACCESS_FLAG. Notice that the symbol table is built so that entries for a single COMPOOL reside in successive slots enabling the easy traversal of all entries of a COMPOOL.

4.3 The Output Writer

Phase 1 generates the primary source listing. This listing is indented, underlined, overlined, bracketed, and in several other ways reformatted. The items to be printed are stored in the statement stack (see data description of GRAMMAR_FLAGS). They are actually printed when a new line point (e.g. end of statement) occurs or when the statement stack overflows. It is the sole responsibility of the output writer to lay out and print the entire primary source listing.

4.3.1 Local Variables of the Output Writer

BUILD_E
BUILD_E_IND
BUILD_E_UND
BUILD_M

See BUILD_S.

BUILD_S

The output writer constructs an entire E/M/S group before printing it. All the subscript lines are positioned in BUILD_S, exponent lines in BUILD_E, and the M line in BUILD_M. Since the subscript and exponent lines are multi-line items, the line number for each character of BUILD_S is indicated in BUILD_S_IND -- similarly BUILD_E_IND. Any character may require underlining -- this is indicated in BUILD_S_UND, BUILD_E_UND, and M_UNDERSCORE. If M_UNDERSCORE is not empty then M_UNDERSOCRE_NEEDED is true. The next character position in each line is indicated by S_PTR, E_PTR, and M_PTR. These pointers are updated properly to keep in step. In particular, on calls to EXPAND, M_PTR will always be at least as large as E_PTR and S_PTR.

BUILD_S_IND
BUILD_S_UND

See BUILD_S.

E_CHAR_PTR	See SAVE_S_C.
E_CHAR_PTR_MAX	See SAVE_S_C.
E_LEVEL	Index of E line currently being built or referenced.
E_PTR	See BUILD_S.
ERRORCODE	Code of current error message to be printed, extracted from SAVE_ERROR_MESSAGE - used as key for retrieving canned message from file #5.
EXP_END	See SUB_END.
EXP_START	See SUB_START.
FIND_ONLY	Switch ON if MATCH is not to zero out the parentheses it finds.
IMBEDDING	Switch ON if error message includes some optional text to be inserted into canned message (variable ident., etc.).
INCLUDE_COUNT	From SRN_COUNT(2) - substitute SRN during include file, incremented from SRN on 1st include card.
LABEL_END	If there are any labels, points to colon on last label; otherwise, LABEL_START-1.
LABEL_START	Index of first item to print -- if there are any labels, they start here.
LINE_FULL	Switch ON if EXPAND should be called to dump the buffers.
M_CHAR_PTR M_CHAR_PTR_MAX	See SAVE_S_C. M_CHAR_PTR is also used to index REPLACE text in MACRO_TEXT when printing REPLACE definitions.
M_PTR	See BUILD_S.

M_UNDERSCORE	See BUILD_S.
M_UNDERSCORE_NEEDED	See BUILD_S.
MACRO_WRITTEN	Switch ON if a macro name was written out anywhere in the statement.
MAX_E_LEVEL	Number of E-lines required to print part of statement scanned so far.
MAX_S_LEVEL	Number of S_lines required to print part of statement scanned so far.
NEXT_CC	Carriage control character for next E/M/S group.
PRNTERRWARN	Switch ON if error overflow warning has never been printed - turned off so message only printed once.
PTR	Index of token being currently processed.
PTR_END	Parameter #2 - index of last token in statement STMT_STACK.
PTR_START	Parameter #1 - index of first token in statement STMT_STACK.
S_CHAR_PTR	See SAVE_S_C.
S_CHAR_PTR_MAX	See SAVE_S_C.
S_LEVEL	Index of S-line being built or referenced.
S_PTR	See BUILD_S.
SAVE_E_C	See SAVE_S_C.
SAVE_MAX_E_LEVEL SAVE_MAX_S_LEVEL	On statements that will not fit on one line, these save the original values of MAX_E_LEVEL and MAX_S_LEVEL, so continuation lines will be in the same format -- used to restore their values after call to EXPAND clears them.

SAVE_S_C — Character strings in HAL are limited to 255 characters; however, when single quotes are expanded to double quotes in ATTACH, the string can grow to more than twice that length. The array SAVE_S_C is used to save the 1, 2, or 3 character strings necessary for a character string in the subscript and SAVE_E_C does the same for exponents. S_CHAR_PTR_MAX is the number of characters in the SAVE_S_C array and S_CHAR_PTR is the current character. Notice that the low order eight bits (i.e. 0-255) is a byte count and the next two bits select the array component. E_CHAR_PTR_MAX and E_CHAR_PTR perform the same functions for SAVE_E_C. A similar procedure is followed for the M line, using M_CHAR_PTR_MAX and M_CHAR_PTR but there is no SAVE_M_C because the M line can be taken directly out of C -- the string returned by ATTACH.

SDL_INFO — First 6 characters are SRN of current statement, next 2 are record revision indicators (only present if SDL_OPTION is ON), last 8 is change authorization from file #5.

SEVERITY — Of error, as retrieved from file #5.

SPACE_NEEDED — Set by ATTACH to number of blanks required in front of the token it just returned. It is always either 0 or 1.

SUB_END — Index in statement stack of last token of subscript.

SUB_START — Index in statement stack of first subscript token.

(SUB_END, SUB_START) Subscript runs between these two -- both are vectors, with one entry for each possible level --- indexed by S_LEVEL. See GRAMMAR_FLAGS.

UNDER_LINE — Buffer for underscore that will overprint E or S-line for macro indication.

UNDERLINING — Switch ON if UNDER_LINE contains anything to be printed.

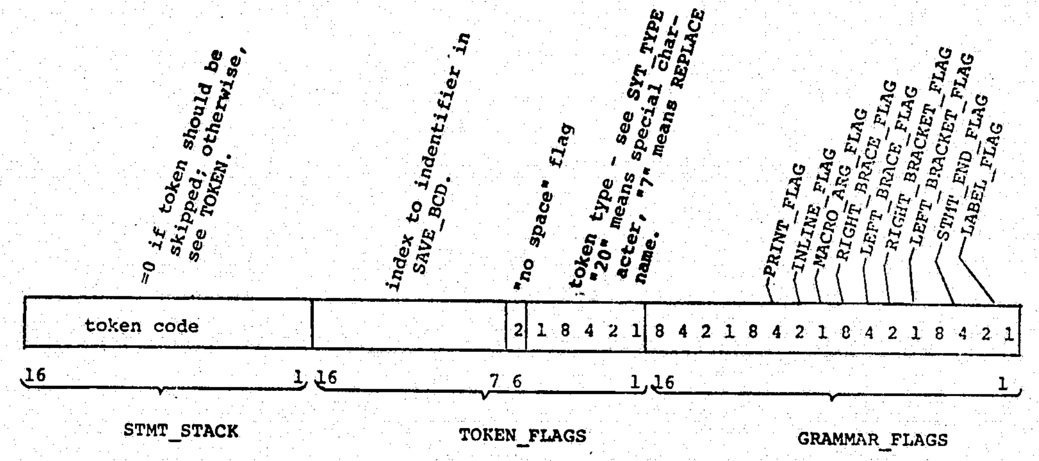
4.3.2 Global Variables Referenced by the Output Writer

BCD_PTR	See GRAMMAR_FLAGS.
C	Temporary character string vector - ATTACH returns token names here.
CHAR_OP	The overpunches used in character literals to cause translation to alternate character set - corresponds to prefix of ¢ or ¢¢.
CHARACTER_STRING	See global definitions -- TOKEN.
COMMENT_COUNT	Number of characters of comments associated with this statement (limit is 255).
COMPILING	Switch ON while compilation is continuing normally -- turned OFF to indicate fatal error -- execution will be halted in COMPILATION_LOOP.
CURRENT_SCOPE	Name of the block actually being read by STREAM.
DOLLAR	See global definitions -- TOKEN.
DOT_TOKEN	See global definitions -- TOKEN.
DOUBLE	Carriage control character to cause double-spacing.
DUMP_MACRO_LIST	Set by MACRO_TEXT_DUMP when a printing of the REPLACE texts rather than the current line is required from the output writer.
ERROR_COUNT	Number of errors accumulated during compilation.
ESCAPE	Escape character for I/O of non-HAL/S characters.
EXPONENTIATE	See global definitions -- TOKEN.
FUNC_FLAG	See GRAMMAR_FLAGS.

GRAMMAR_FLAGS

The statement stack is used to store up a source statement before printing. The stack is built of three parallel arrays as indicated in the diagram. STMT_PTR points to the top-most entry in the stack. Notice that the actual character strings are stored in SAVE_BCD. TOKEN_FLAGS simply contains an index into SAVE_BCD. BCD_PTR points to the last entry in SAVE_BCD. In the general case, some of the material in the stack has been printed and LAST_WRITE points to the first unprinted item.

A Statement Stack Item:



In order to associate items in the parser's stack with their entries in the statement stack, the parser maintains STACK_PTR entries. STACK_PTR (parser stack pointer) points to the element's entry in the statement stack.

GRAMMAR_FLAGS values

0042	ATTR_BEGIN_FLAG	
0428	FUNC_FLAG	Token is a function call.
0577	INLINE_FLAG	Token is an inline function.
0671	LABEL_FLAG	Token is a label.
0687	LEFT_BRACE_FLAG	Preceed token by '{' on output.
0688	LEFT_BRACKET_FLAG	Preceed token by '[' on output.
0786	MACRO_ARG_FLAG	Token is an argument to a macro.
0976	PRINT_FLAG	Token should be printed.
0978	PRINT_FLAG_OFF	¬PRINT_FLAG -- Used to turn off PRINT_FLAG.

1047	RIGHT_BRACE_FLAG	Append "}" after token on output.
1048	RIGHT_BRACKET_FLAG	Append "]" after token on output.
1160	STMT_END_FLAG	Final token in statement.

INCLUDE_CHAR	Character printed on the listing next to the statement number if the source was read from an include file - otherwise blank.
INCLUDE_END	Switch ON if just read EOF on include file.
INCLUDING	Switch ON if reading from include file.
INDENT_LEVEL	Column number of current left margin indention.
INFORMATION	Information to be printed with SAVE_SCOPE to the right of the source statement (DO CASE numbers, etc.).
INLINE_FLAG	See GRAMMAR_FLAGS.
INLINE_INDENT	Column number for indention of current inline function.
INLINE_INDENT_RESET	Used to restore INDENT_LEVEL to value it had before interruption by inline function.
LABEL_COUNT	Number of labels on current statement (each is two tokens -- label and :).
LABEL_FLAG	See GRAMMAR_FLAGS.
LAST	The number of errors in the current statement.
LAST_SPACE	Usually the value of post spacing on last token - may be altered in special cases.
LAST_WRITE LEFT_BRACE_FLAG LEFT_BRACKET_FLAG	See GRAMMAR_FLAGS.
LEFT_PAREN	See global definitions -- TOKEN.
LINE_LIM	Number of lines in listing page as read from JCL LIST = option.
LINE_MAX	This is usually LINE_LIM -- it is set to 0 to force a page eject.

MAC_NUM	Symbol table pointer for the last REPLACE name defined.
MACRO_ARG_FLAG	See GRAMMAR_FLAGS.
MACRO_INDEX	The number of REPLACE texts that have been defined in this compilation unit.
MACRO_TEXT	See SCAN.
MAJ_STRUC	See symbol table -- SYT_FLAGS.
MAX_SEVERITY	Maximum SEVERITY of errors found so far in program.
OUT_PREV_ERROR	Statement number where last error message was printed.
OVER_PUNCH_TYPE(token)	Is the overpunch character to apply (bit ".", char",", vector ".", structure "+", matrix "*").
PAD1	Blank field the width of the statement number info on the M-line - used to pad S and E lines on the left.
PAD2	As PAD1, plus space for line type and VBAR - used to pad underscore lines on the left.
PAGE	Carriage control character to cause page eject.
PAGE_THROWN	Switch ON if page eject just done - used to reduce multiple paging to a single eject.
PLUS	Carriage control character to enable overprinting of underscore characters.
PREVIOUS_ERROR	Set to STMT_NUM at the time an error is detected and used to set OUT_PREV_ERROR.
PRINT_FLAG	See GRAMMAR_FLAGS.
PRINT_FLAG_OFF	See GRAMMAR_FLAGS.

RECOVERING
Set by RECOVER - overrides PRINT_FLAG_OFF to force printing of all output stacks.

REPLACE_TEXT
See global definitions -- TOKEN.

RIGHT_BRACE_FLAG
See GRAMMAR_FLAGS.

RIGHT_BRACKET_FLAG
See GRAMMAR_FLAGS.

RT_PAREN
See global definitions -- TOKEN.

SAVE_BCD
See GRAMMAR_FLAGS.

SAVE_COMMENT
Text of comment to be printed with this statement.

SAVE_ERROR_MESSAGE
Stack of error messages for this statement. Each entry is a character string containing an eight character code followed optionally by text to be imbedded.

SAVE_LINE_#(I)
Is the number of the line containing the I^{th} error.

SAVE_SCOPE
Name of the block to which the current statement belongs. Required because CURRENT_SCOPE may be updated before printing some material accumulated in the older scope.

SAVE_SEVERITY(I)
Is the SEVERITY of the I^{th} error message.

SAVE_STACK_DUMP
Array of formatted lines corresponding to dump of parse stack.

SCALAR_TYPE
See symbol table -- SYT_TYPE.

SDL_OPTION
Switch ON if printing extra SDL info (SRN, change authorization field, record revision indicator) on listing; OFF if NOSDL option specified.

SPACE_FLAGS(token)
Specifies the pre and post spacing for token. The pre-spacing is the high order four bits and the post-spacing the low order four bits. Since spacing is done one way on the M line and a different way on E and S lines,

SPACE_FLAGS(token + number of tokens)

is the spacing for E and S lines.

Pre/Post Codes:

0 - always wants a space, if not overridden by the other token

1 - only want a space if the other token wants one too

2 - never wants a space

3 - always gets a space

SQUEEZING
Switch is set by SAVE_TOKEN when it needs more space to save the current item. In this case, the output writer should write out the minimum amount of material (one E/M/S group) and return. The switch is cleared by OUTPUT_WRITER.

SRN
Statement reference number and additional SDL info, obtained from source card to the right of the text area (TEXT_LIMIT).

SRN_COUNT
M-card count when reading from include file - indexed in such a way as to be the card number of the current token.

SRN_PRESENT
Switch ON if SRN is being read from input cards.

STACK_DUMP_PTR
Index of last item in SAVE_STACK_DUMP - = -1 if empty.

STACK_DUMPED
Switch ON if STACK_DUMP and SAVE_DUMP have just filled SAVE_STACK_DUMP.

STACK_PTR
See GRAMMAR_FLAGS.

STATEMENT_SEVERITY
Maximum SEVERITY of errors in this statement.

STMT_END_FLAG
See GRAMMAR_FLAGS.

STMT_NUM
Line number of current statement.

STMT_PTR
See GRAMMAR_FLAGS.

STMT_STACK
See GRAMMAR_FLAGS.

STRUC_TOKEN
See global definitions -- TOKEN.

SYT_LINK1
See symbol table.

TOKEN_FLAGS See GRAMMAR_FLAGS.

TOO_MANY_ERRORS Switch ON if error stack was filled up - some messages may not have been recorded.

TRANS_OUT(char) Yields a 16 bit description of char's printable form. The low order byte is the character to print. If TRANS_OUT is zero, print char itself; otherwise the high order byte indicates the number of escapes (0 → 1 escape, 1 → 2 escapes).

TX(special character) Is the TOKEN code for the character.

VBAR A vertical bar, "|", used to delimit the listing margins.

VOCAB_INDEX See procedure SCAN -- identifiers.

WAS_HERE Used only by PRINT_TEXT to print 2 double quote marks for each embedded one.

X1 Blank field of length 1.

X70 Blank field of length 70.

4.3.3 Procedures of the Output Writer

OUTPUT_WRITER -- 291000
PRINT_TEXT -- 377500

OUTPUT_WRITER is the entry point and central control of the output writer module. It assembles and prints E/M/S groups followed by error messages.

Set up LABEL_START and LABEL_END.

Calculate positioning of subscripts. Use MATCH to find and eliminate parentheses around subscripts, then if the subscript is a subscripted expression, restore the parenthesis. If the subscript is subscripted, find the end of the lowest subscript.

Do the same thing for superscripts.

Now that an entire subscript has been located, divide it up for multi-line printing using SUB_START (S_LEVEL) and SUB_END (S_LEVEL). The actual character string to be printed is built in BUILD_S with associated indicators in BUILD_S_IND and BUILD_S_UND. The character strings to be printed including spacing and braces and brackets, are computed by ATTACH. S_LEVEL is incremented for each $ and decremented when the end of a subscript is reached.

Do the same thing for superscripts.

BUILD_M is set up in a similar manner without the difficulties of multi-line format. Notice that the text of a REPLACE statement must appear on the M line and thus presents a problem only here. PRINT_TEXT is used to print the macro text in a straightforward manner. When printing labels, un-indent far enough so that the label ends just before the indentation point.

If the label will not fit, un-indent to the left margin and print the labels on a separate line.

After everything has been built and overflowing lines have been printed, print the current buffer and clean up all the hanging indicators for the next time around.

If there were any error messages pending, print them. Notice that the error message text must be read in from an auxiliary file and imbedded text must be inserted instead of "??".

If DUMP_MACRO_LIST is set, then the output writer simply prints all the REPLACE_TEXTS. It starts off at the beginning of all the texts, PRINT_TEXT prints a single text advancing M_CHAR_PTR to the end of the text. Then increment M_CHAR_PTR one more position and PRINT_TEXT again.

ATTACH -- 295400
ADD -- 299300

ATTACH is called by OUTPUT_WRITER to compute the character string for an item to be printed. ATTACH must compute the character string, the pre-spacing, the enclosing brackets or braces, the display character for non-HAL characters, and the expansion of embedded single quotes in character strings. Since the spacing in exponent/subscript lines is different from the spacing of M lines, OFFSET is also delivered to allow proper lookup in the SPACE_FLAGS table.

Formatting character string tokens can be complicated (see data description of SAVE_S_C), so a separate procedure, ADD, is used to append a character to the existing substring.

MATCH -- 325800

When OUTPUT_WRITER is scanning subscripts and superscripts, it looks for the end of parenthesized sub/superscripts and eliminates the parenthesis. It then replaces the parenthesis if they are necessary. The search and elimination is performed by MATCH which takes as argument the index of the left paren and returns as value the index of the right paren. If FIND_ONLY is set, the elimination is suppressed.

CHECK_FOR_FUNC -- 329500
SKIP_REPL -- 328500

If a sub/superscript is not parenthesized, then OUTPUT_WRITER locates the end of it via CHECK_FOR_FUNC. This searches for the end of a function call (possibly subscripted with nested calls), skips macros via SKIP_REPL, and locates the end of qualified structure names. CHECK_FOR_FUNC receives a starting point as argument and returns the location of the end as value.

EXPAND -- 305700
COMMENT_BRACKET -- 306900

EXPAND is called by OUTPUT_WRITER to actually print an E/M/S group which has been formatted in BUILD_E, BUILD_M, and BUILD_S. If the group contains an end of statement then EXPAND will add to it any accumulated comments. Comments are printed in the M line if they fit. If the comment will not fit and the statement is short, it is printed on the M and S lines; if the statement is long, it is printed after the statement. Comments are inserted into the output string by COMMENT_BRACKET which takes a string and a position within the string and modifies the argument string using the BYTE pseudo-function.

4.4 The Semantic Routines

The HAL/S compilers handle semantics in a very standard manner. Immediately before performing a reduction, the parser calls SYNTHESIZE. SYNTHESIZE is an enormous CASE statement on the production number.

We have broken up the entire grammar into six sections. The individual productions are covered as follows:

1-3	4.4.7
4-32	4.4.5
33-81	4.4.6
82-135	4.4.5
136-176	4.4.6
177-178	4.4.5
179-180	4.4.6
181-192	4.4.5
193-205	4.4.4
206-208	4.4.5
209-249	4.4.4
250-272	4.4.5
273-288	4.4.6
289-292	4.4.7
293-328	4.4.2
329-425	4.4.3
426-428	4.4.7
429-449	4.4.6

When working through semantic routines of this nature, it is important to figure out the reduction sequence. We include here the complete reduction sequence for a meaningless program which has a large collection of constructs in it.

```
              C|  DECLARATION OF A PROGRAM

---------------------------------------------------------------------
                              scanner returns token number 98

                              scanner returns token number 16

reduction 304

                              scanner returns token number 107

reduction 305
reduction 307

                              scanner returns token number 10

reduction 301
reduction 298

              M| SIMPLE:
              M| PROGRAM;

              C|  DECLARATION WITH IMPLIED TYPE

---------------------------------------------------------------------
                              scanner returns token number 103
                              scanner returns token number 131
                              scanner returns token number 10

reduction 358
reduction 356
reduction 342
reduction 340
reduction 339

              M|    DECLARE A;

reduction 329
reduction 345

              C|  STANDARD FORM DECLARATION

---------------------------------------------------------------------
                              scanner returns token number 103
                              scanner returns token number 131
                              scanner returns token number 105

reduction 358
reduction 385

                              scanner returns token number 10
```

```
reduction 382
reduction 376
reduction 370
reduction 362
reduction 357
reduction 342
reduction 340
reduction 339

                  M|    DECLARE B INTEGER;

reduction 329
reduction 346

                  C|  DECLARATION WITH FACTORED TYPE

-----------------------------------------------------------------

                              scanner returns token number 103
                              scanner returns token number 105

reduction 385

                              scanner returns token number 14

reduction 382
reduction 376
reduction 370
reduction 362

                              scanner returns token number 131
                              scanner returns token number 10

reduction 358
reduction 356
reduction 342
reduction 341
reduction 339

                  M|    DECLARE INTEGER, C;

reduction 329
reduction 346

                  C|  AN EQUATE DECLARATION

-----------------------------------------------------------------

                              scanner returns token number 82
                              scanner returns token number 112
                              scanner returns token number 131
                              scanner returns token number 30
                              scanner returns token number 126

reduction 222

                              scanner returns token number 10
```

```
reduction 230
reduction 216
reduction 193
reduction 332

                M|    EQUATE EXTERNAL X TO A;

reduction 346

                C|  A STRUCTURE DECLARATION

-----------------------------------------------------------------------

                          scanner returns token number 123
                          scanner returns token number 131
                          scanner returns token number 16
                          scanner returns token number 99

reduction 348

                M|    STRUCTURE Q:

-----------------------------------------------------------------------

                          scanner returns token number 131
                          scanner returns token number 14

reduction 358
reduction 356

                          scanner returns token number 99

reduction 350

                M|      1 Q1,

-----------------------------------------------------------------------

                          READ TOKEN 131
                          READ TOKEN 14

reduction 358
reduction 356

                          READ TOKEN 99

reduction 350

                M|        2 Q2,

-----------------------------------------------------------------------

                          READ TOKEN 131
                          READ TOKEN 10

reduction 358
reduction 356
reduction 351
reduction 347

              M|         2 Q3;

reduction 331
reduction 346

              C|  DECLARE A SIMPLE STRUCTURE VARIABLE

-------------------------------------------------------------------

                              READ TOKEN 103
                              READ TOKEN 131
                              READ TOKEN 139

reduction 358

                              READ TOKEN 12
                              READ TOKEN 123
                              READ TOKEN 10

reduction 353
reduction 352
reduction 373
reduction 370
reduction 362
reduction 357
reduction 342
reduction 340
reduction 339

              M|     DECLARE QQ Q-STRUCTURE;

reduction 329
reduction 346

              C|  DECLARE A STRUCTURE VARIABLE WITH COPIES

-------------------------------------------------------------------

                              READ TOKEN 103
                              READ TOKEN 131
                              READ TOKEN 139

reduction 358

                              READ TOKEN 12
                              READ TOKEN 123
                              READ TOKEN 3

reduction 355

                              READ TOKEN 99

reduction 425
reduction 19

                             READ TOKEN 9

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 391
reduction 354
reduction 352
reduction 373

                             READ TOKEN 10

reduction 370
reduction 362
reduction 357
reduction 342
reduction 340
reduction 339

              M|     DECLARE Q_COPIES Q-STRUCTURE(3);

reduction 329
reduction 346

              C|   DECLARE A ONE DIMENSIONAL ARRAY

-----------------------------------------------------------------

                             READ TOKEN 103
                             READ TOKEN 131
                             READ TOKEN 64

reduction 358

                             READ TOKEN 3

reduction 368

                             READ TOKEN 99

reduction 425
reduction 19

                             READ TOKEN 9

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 391
reduction 363
```

```
                                        READ TOKEN 10

reduction 361
reduction 357
reduction 342
reduction 340
reduction 339

                  M|    DECLARE ONE ARRAY(5);

reduction 329
reduction 346

                  C|  DECLARE A TWO DIMENSIONAL ARRAY

-------------------------------------------------------------
                                        READ TOKEN 103
                                        READ TOKEN 131
                                        READ TOKEN 64

reduction 358

                                        READ TOKEN 3

reduction 368

                                        READ TOKEN 136

reduction 424
reduction 19

                                        READ TOKEN 14

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 391
reduction 369

                                        READ TOKEN 99

reduction 425
reduction 19

                                        READ TOKEN 9

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 391
reduction 363
```

```
                                    READ TOKEN 10

reduction 361
reduction 357
reduction 342
reduction 340
reduction 339

                 M|    DECLARE TWO ARRAY(5, 5);

reduction 329
reduction 346

                 C|  A NO ARGUMENT FUNCTION DECLARATION

----------------------------------------------------------

                                    READ TOKEN 98

reduction 291

                                    READ TOKEN 16

reduction 304

                                    READ TOKEN 113

reduction 305
reduction 316

                                    READ TOKEN 89

reduction 386

                                    READ TOKEN 10

reduction 382
reduction 376
reduction 319
reduction 313
reduction 301
reduction 298

                 M| FUNC:
                 M|    FUNCTION SCALAR;

----------------------------------------------------------

                                    READ TOKEN 88

reduction 290

                                    READ TOKEN 136

reduction 424
reduction 19
```

```
                              READ TOKEN 10

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 53
reduction 36

              M|       RETURN 1;

reduction 38
reduction 292

                              READ TOKEN 65
                              READ TOKEN 98

reduction 427

                              READ TOKEN 10

reduction 289

              M|    CLOSE FUNC;

reduction 39
reduction 292

              C|  JUST A LABEL

-------------------------------------------------------------
                              READ TOKEN 98
                              READ TOKEN 16

reduction 304

                              READ TOKEN 10

reduction 47
reduction 40
reduction 36

              M| LBL:
              M|    ;

reduction 38
reduction 292

              C|  A SIMPLE ARITHMETIC EXPRESSION

-------------------------------------------------------------
                              READ TOKEN 126
```

```
reduction 222

                              READ TOKEN 19

reduction 230
reduction 216
reduction 193

                              READ TOKEN 126

reduction 248
reduction 222

                              READ TOKEN 4

reduction 230
reduction 216
reduction 27
reduction 15
reduction 11
reduction 9
reduction 4

                              READ TOKEN 136

reduction 424
reduction 19

                              READ TOKEN 10

reduction 31
reduction 15
reduction 11
reduction 9
reduction 7
reduction 181
reduction 136
reduction 41
reduction 36

                  M|    A = A + 1;

reduction 38
reduction 292

                  C|  A ONE DIMENSIONAL SUBSCRIPT

------------------------------------------------------------

                              READ TOKEN 126

reduction 222

                              READ TOKEN 7

reduction 249
```

```
                         READ TOKEN 136

reduction 424
reduction 228
reduction 216
reduction 193

                         READ TOKEN 19
                         READ TOKEN 126

reduction 248
reduction 222

                         READ TOKEN 7

reduction 249

                         READ TOKEN 136

reduction 424
reduction 228
reduction 216
reduction 27

                         READ TOKEN 10

reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 136
reduction 41
reduction 36

            M|   ONE = ONE ;
            S|     1     1

reduction 38
reduction 292

            C|  A TWO DIMENSIONAL SUBSCRIPT

----------------------------------------------------

                         READ TOKEN 126

reduction 222

                         READ TOKEN 7

reduction 249

                         READ TOKEN 3
                         READ TOKEN 136

reduction 231
```

```
          reduction 424
          reduction 19

                                   READ TOKEN 14

          reduction 31
          reduction 15
          reduction 11
          reduction 9
          reduction 4
          reduction 243
          reduction 238
          reduction 237
          reduction 235

                                   READ TOKEN 99

          reduction 425
          reduction 19

                                   READ TOKEN 9

          reduction 31
          reduction 15
          reduction 11
          reduction 9
          reduction 4
          reduction 243
          reduction 238
          reduction 237
          reduction 226
          reduction 216
          reduction 193

                                   READ TOKEN 19
                                   READ TOKEN 126

          reduction 248
          reduction 222

                                   READ TOKEN 7

          reduction 249

                                   READ TOKEN 3
                                   READ TOKEN 136

          reduction 231
          reduction 424
          reduction 19

                                   READ TOKEN 14

          reduction 31
          reduction 15
          reduction 11
```

```
reduction 9
reduction 4
reduction 243
reduction 238
reduction 237
reduction 235

                              READ TOKEN 99

reduction 425
reduction 19

                              READ TOKEN 9

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 243
reduction 238
reduction 237
reduction 226
reduction 216
reduction 27

                              READ TOKEN 10

reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 136
reduction 41
reduction 36

              M|    TWO    = TWO    ;
              S|       1,1      1,1

reduction 38
reduction 292

              C|  A SIMPLE QUALIFIED STRUCTURE REFERENCE

------------------------------------------------------------------

                              READ TOKEN 135

reduction 220

                              READ TOKEN 1
                              READ TOKEN 135

reduction 221

                              READ TOKEN 19
```

```
reduction 230
reduction 215
reduction 194

                             READ TOKEN 135

reduction 248
reduction 220

                             READ TOKEN 1
                             READ TOKEN 135

reduction 221

                             READ TOKEN 10

reduction 230
reduction 215
reduction 186
reduction 184
reduction 136
reduction 41
reduction 36

              E|       +       +
              M|    QQ.Q1 = QQ.Q1;

reduction 38
reduction 292

              C|  A SUBSCRIPTED STRUCTURE REFERENCE

-----------------------------------------------------------

                             READ TOKEN 135

reduction 220

                             READ TOKEN 7

reduction 249

                             READ TOKEN 136

reduction 424
reduction 228
reduction 215
reduction 194

                             READ TOKEN 19
                             READ TOKEN 135

reduction 248
reduction 220

                             READ TOKEN 7
```

```
reduction 249

                                  READ TOKEN 99

reduction 425
reduction 228
reduction 215
reduction 186
reduction 184
reduction 136

                                  READ TOKEN 10

reduction 41
reduction 36

                     E|        +              +
                     M|    Q_COPIES = Q_COPIES ;
                     S|              1              2

reduction 38
reduction 292

                     C|  A SUBSCRIPTED MINOR STRUCTURE REFERENCE

----------------------------------------------------------------
                                  READ TOKEN 135

reduction 220

                                  READ TOKEN 1
                                  READ TOKEN 135

reduction 221

                                  READ TOKEN 7

reduction 249

                                  READ TOKEN 136

reduction 424
reduction 228
reduction 215
reduction 194

                                  READ TOKEN 19
                                  READ TOKEN 135

reduction 248
reduction 220

                                  READ TOKEN 1
                                  READ TOKEN 135

reduction 221
```

```
                              READ TOKEN 7

reduction 249

                              READ TOKEN 99

reduction 425
reduction 228
reduction 215
reduction 186
reduction 184
reduction 136

                              READ TOKEN 10

reduction 41
reduction 36

              E|            +             +
              M|    Q_COPIES.Q1 = Q_COPIES.Q1 ;
              S|              1             1

reduction 38
reduction 292

              C|  A BUILT-IN FUNCTION CALL

--------------------------------------------------------------

                              READ TOKEN 126

reduction 222

                              READ TOKEN 19

reduction 230
reduction 216
reduction 193

                              READ TOKEN 130

reduction 248
reduction 21

                              READ TOKEN 3
                              READ TOKEN 126

reduction 222

                              READ TOKEN 9

reduction 230
reduction 216
reduction 27
reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 191
reduction 177
reduction 28

                              READ TOKEN 10

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 136
reduction 41
reduction 36

              M|    A = SIN(A);

reduction 38
reduction 292

              C|  DEFINE A TWO ARGUMENT FUNCTION

------------------------------------------------------------
                              READ TOKEN 98
                              READ TOKEN 16

reduction 304

                              READ TOKEN 113

reduction 305
reduction 316

                              READ TOKEN 3

reduction 325

                              READ TOKEN 131
                              READ TOKEN 14

reduction 326

                              READ TOKEN 131
                              READ TOKEN 9

reduction 324

                              READ TOKEN 89

reduction 386

                              READ TOKEN 10
```

```
reduction 382
reduction 376
reduction 320
reduction 313
reduction 301
reduction 298

              M| FUNC2:
              M|    FUNCTION(ARG1, ARG2) SCALAR;

----------------------------------------------------------

                              READ TOKEN 103
                              READ TOKEN 89

reduction 386

                              READ TOKEN 14

reduction 382
reduction 376
reduction 370
reduction 362

                              READ TOKEN 131
                              READ TOKEN 14

reduction 358
reduction 356
reduction 342
reduction 344

              M|       DECLARE SCALAR,

----------------------------------------------------------

                              READ TOKEN 131
                              READ TOKEN 10

reduction 358
reduction 356
reduction 343
reduction 341
reduction 339

              M|                   ARG1, ARG2;

reduction 329
reduction 345

                              READ TOKEN 88

reduction 291

                              READ TOKEN 126

reduction 222
```

```
                              READ TOKEN 4

reduction 230
reduction 216
reduction 27
reduction 15
reduction 11
reduction 9
reduction 4

                              READ TOKEN 126

reduction 222

                              READ TOKEN 10

reduction 230
reduction 216
reduction 27
reduction 15
reduction 11
reduction 9
reduction 7
reduction 181
reduction 53
reduction 36

               M|       RETURN ARG1 + ARG2;

reduction 38
reduction 292

                              READ TOKEN 65
                              READ TOKEN 98

reduction 427

                              READ TOKEN 10

reduction 289

               M|    CLOSE FUNC2;

reduction 39
reduction 292

               C|  CALL A TWO ARGUMENT FUNCTION

--------------------------------------------------------

                              READ TOKEN 126

reduction 222

                              READ TOKEN 19

reduction 230
reduction 216
reduction 193

                              READ TOKEN 130

reduction 248
reduction 21

                              READ TOKEN 3
                              READ TOKEN 126

reduction 222

                              READ TOKEN 14

reduction 230
reduction 216
reduction 27
reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 191
reduction 177

                              READ TOKEN 126

reduction 222

                              READ TOKEN 9

reduction 230
reduction 216
reduction 27
reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 191
reduction 178
reduction 28

                              READ TOKEN 10

reduction 31
reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 136
reduction 41
reduction 36
```

```
                M|     A = FUNC2(A, A);

reduction 38
reduction 292

                C|  CALL A NO ARGUMENT FUNCTION

-------------------------------------------------
                               READ TOKEN 126

reduction 222

                               READ TOKEN 19

reduction 230
reduction 216
reduction 193

                               READ TOKEN 141

reduction 248
reduction 222

                               READ TOKEN 10

reduction 230
reduction 211
reduction 29
reduction 15
reduction 11
reduction 9
reduction 4
reduction 181
reduction 136
reduction 41
reduction 36

                M|     A = FUNC;

reduction 38
reduction 292

                C|  CLOSE A PROGRAM

-------------------------------------------------
                               READ TOKEN 65
                               READ TOKEN 98

reduction 427

                               READ TOKEN 10

reduction 289
```

```
              M| CLOSE SIMPLE;
reduction 2
                              READ TOKEN 31
reduction 1
```

4.4.1 Global Variables Accessed by the Semantic Routines

ACCESS_FLAG   See symbol table -- SYT_FLAGS.

ACCESS_FOUND   See STREAM.

ALT_PCARG#(i)   If the number of arguments in a % macro does not match PCARG#(i), use ALT_PCARG#(i).

ARRAY_SUB_COUNT   LITERALLY VAL_P(PTR(MP)) initialized to -1. Reset to SUB_COUNT - STRUCTURE_SUB_COUNT on finding a ":" in a subscript.

ARRAYNESS_STACK   See VAR_ARRAYNESS.

AS_PTR   See VAR_ARRAYNESS.

ASSIGN_ARG_LIST   True when processing %COPY to inhibit lack group checking.

ASSIGN_CONTEXT   See CONTEXT in SCAN.

ASSIGN_TYPE   Specifies possible legal type transformation.

	11	10	9	8	7	6	5	4	3	2	1	0
0-null	0	0	0	0	0	0	0	0	0	0	0	0
1-bit	0	0	0	0	0	0	0	0	0	0	1	0
2-char	0	0	0	0	0	1	1	0	0	1	0	1
3-mat	0	0	0	0	0	0	0	0	1	0	0	1
4-vec	0	0	0	0	0	0	0	1	0	0	0	1
5-seq	0	0	0	0	0	1	1	0	0	0	0	1
6-int	0	0	0	0	0	1	1	0	0	0	0	1
7-borc	0	0	0	0	0	0	0	0	0	0	0	0
8-iors	0	0	0	0	0	1	1	0	0	0	0	0
9-event	0	0	0	0	0	0	0	0	0	0	0	0
10-struc	0	1	0	0	0	0	0	0	0	0	0	0
11-any	0	1	1	1	1	1	1	1	1	1	1	0

ATOM#_FAULT   See NEXT_ATOM#.

ATOM#_LIM   See NEXT_ATOM#.

ATTOMS   See NEXT_ATOM#.

ATTR_FOUND

Is turned off after finding the first <declaration> of a <declaration list>. It is turned on if SAVE_TOKEN forces an output writer call and after the second <declaration>. It is used by SYNTHESIZE to make the output writer line up declarations properly.

ATTR_LOC

Is set to point to the name (in the statement stack) being declared unless it is a template declaration -- it is reset by SAVE_TOKEN if the statement stack overflows forcing an output writer call.

ATTR_MASK

See ATTRIBUTES.

ATTRIBUTES

This is SYT_FLAGS kind of information for an identifier being declared. When an attribute is found it becomes illegal to specify that attribute again and conceivably several others (e.g. DOUBLE outlaws DOUBLE and SINGLE). The illegal attributes are accumulated in ATTR_MASK.

BI_ARG_TYPE (bi_info)

Specifies the type of argument required. Notice that anything that can be converted to this type is acceptable; consequently, ASSIGN_TYPE(BI_ARG_TYPE) is the thing to use in tests.

BI_FLAGS

	TR	SQ	i/c	number
	1	1	1	4

if i/c = 1, then function has special processing in i/c context and number selects the special processing.
if SQ=1, argument must be square.
if TR=1, result has dimensions of transpose of argument.

BI_FUNC_FLAG

On when handling built-in function in initial/constant context.

BI_INFO

result type	number of args	BI_ARG_TYPE pointer
8	8	16

If the function takes more than one argument, pointer+1 points to entry for second argument, etc. BI_INFO(0) is a copy of BI_INFO (current function).

BI_XREF(loc) For loc > 0, serves the same function for built-in functions that XREF serves for other names. BI_XREF(0) is set true when a cross reference is built.

BIT_LENGTH Length of bit string specification being processed.

BLOCK_MODE(nest) Type of block at nesting level nest.

4 - PROG_MODE ≡ program;
3 - CMPL_MODE ≡ compool;
5 - TASK_MODE ≡ task;
6 - UPDATE_MODE ≡ update block;
2 - FUNC_MODE ≡ function declaration;
1 - PROC_MODE ≡ procedure declaration;
7 - INLINE_MODE ≡ inline function.

BLOCK_SYTREF(nest) Symbol table pointer for name of block at nesting level nest.

BUILDING_TEMPLATE On when building template from a structure statement.

CHAR_LENGTH Length of character string specification being processed.

CLASS Identifier being declared is:

0 - none of the below,
1 - procedure, program, task,
2 - function.

CLOSE_BCD The name of the identifier to be removed from the hash table.

CMPL_MODE See BLOCK_MODE.

CONTEXT See SCAN.

CUR_IC_BLK See IC_LINE.

CURRENT_ARRAYNESS See VAR_ARRAYNESS.

CURRENT_ATOM See NEXT_ATOM#.

CURRENT_SCOPE The name associated with the current block.

DEF_BIT_LENGTH
DEF_CHAR_LENGTH
DEF_MAT_LENGTH
DEF_VEC_LENGTH

} Default values for the length if the declaration contains an illegal or unspecified value.

DELAY_CONTEXT_CHECK On when processing the arguments of a % macro or NAME pseudo-function.

DO_CHAIN See DO_LEVEL.

DO_INIT A flag indicating whether accumulated initialization should be transformed to HALMAT.

DO_INX See DO_LEVEL.

DO_LEVEL Since DO groups can be nested, the compiler must maintain a stack for all "active" DO groups. The stack is indexed by DO_LEVEL.

DO_LOC is the flow number of the instruction following the end of the DO group. DO_LOC+1 is the flow number of the repeat point. DO_LOC(0) counts the number of DO groups encountered after the DO stack overflowed so that proper processing can be restored at the right time.

```
DO_INX =  0  DO;
          1  for discrete DO FOR;
          2  DO CASE;
          3  DO WHILE/UNTIL.
```

DO_CHAIN = symbol table pointer for first temporary declared in the group. The rest are linked by the SYT_LINK field.

DO_PARSE = Points to the parse stack position immediately below the DO keyword.

DO_LOC See DO_LEVEL.

DO_PARSE See DO_LEVEL.

EXT_P See PTR_TOP.

EXTERNAL Set to 1 on finding definition of external level and reset to proper mode (e.g. PROC_MODE, CMPL_MODE) when the rest of the information is acquired.

FACTOR_FOUND See FACTORING.

FACTORED_IC_FND On if an initial/constant value was encountered while FACTORING.

FACTORED_TYPE Any TYPE information accumulated while FACTORING is copied here. Notice that this has the same pseudo-array structure as TYPE.

FACTORING When processing a DECLARE statement, anything found before an identifier is a factored attribute. FACTORING is on until the identifier is encountered. FACTOR_FOUND is on if a factored attribute is actually found.

FCN_ARG(fcn_lv) The number of arguments encountered for the function. -1 for declared but not yet defined functions. -2 for non-HAL functions.

FCN_LOC(fcn_lv)

FCN_MODE	Value
0	symbol table pointer
1	bi-info pointer
2	shaper number
3	shaper number
4	bi-info pointer

FCN_LV Since function calls may be nested, a stack is required to save partially examined function calls. FCN_LV is the stack pointer -- it is 0 for procedures and I/O.

FCN_MODE(fcn_lv) 0 - procedure, I/O, user function
1 - normal built-in function
2 - arith shaping function
3 - string shaping function
4 - list function

FIRST_STMT — Line number of first statement of block.

FIX_DIM — The size of the dimension just subscripted
(e.g. 3 AT 1 would yield FIX_DIM = 3,
14 would yield FIX_DIM = 1,
4 TO 8 would yield FIX_DIM = 5).

FIXF — Parser stack initialized to FIXING by parser.

- for <statement>
  <basic statement>
  <any statement>
  <other statement>
  a pointer to the previous label on the same statement.

FIXL — Parser stack initialized to SYT_INDEX by parser and usually maintained as a symbol table pointer.

- for <minor attribute> something to incorporate into ATTR_MASK.
- for <prec spec> something to incorporate into ATTR_MASK.
- for <double qual name head> the TYPE.
- for <repeat head> IC_LINE at the time of the reduction.
- for <qual struct> symbol table pointer for template.
- for <# expression> :

  1 - just a #
  2 - # + <term>
  3 - # - <term>

- for <subscript>:

  "1" bit on for real subscript,
  off for null subscript.

  "2" bit on for user defined function,
  off otherwise.

- for <arith conv>:

  0 → MATRIX
  1 → VECTOR
  2 → SCALAR
  3 → INTEGER

- for <bit const head> the value of the repetition factor.

- for <FOR KEY> symbol table pointer in FOR TEMPORARY, otherwise, 0.

- for <while key> and <stopping>:

  0 for WHILE,
  1 for UNTIL.

- for <terminator> HALMAT CANC or TERM.

FIXV

Parser stack initialized to VALUE by parser.

- for <struct stmt head> the current value of level .

- for <minor attribute> something to incorporate into ATTRIBUTES.

- for <prec spec> something to incorporate into ATTRIBUTES.

- for <doubly qual name head> the first dimension of a matrix.

- for <repeat head> the number of elements affected.

- for <prefix>:

  0 - dummy prefix.
  1 - real prefix (i.e. qualified structure reference)

- for <qual struct> symbol tabel pointer for major structure.

- for <for key> a pointer to the DFOR instruction.

- for <iteration body> a pointer to the last AFOR issued.

- for <terminator>:

  TERMINATE "E000"
  CANCEL "A000"

- for <file exp> the device number.

FL_NO

Whenever the compiler wants to refer to a point in the HALMAT it generates an internal label called a flow number. When the appropriate point in the HALMAT is reached, the flow number is defined by an LBL HALMAT operator. FL_NO is simply incremented each time to generate unique flow numbers. It is stacked in lots of places (e.g. the DO stack).

FUNC_MODE

See BLOCK_MODE.

HALMAT_BLOCK

See NEXT_ATOM#.

HALMAT_FILE

See NEXT_ATOM#.

IC_FILE

See IC_LINE.

IC_FND

≡ TYPE(. . .) on if an i/c has been found.

IC_FORM

See IC_LINE.

IC_FOUND

0 - no initialization pending.
1 - factored initialization pending.
3 - non-factored initialization pending.

IC_LEN

See IC_LINE.

IC_LIM

See IC_LINE.

IC_LINE

The i/c que is stored as the paged file, IC_FILE. The current page CUR_IC_BLK, resides in IC_VAL which contains the lines IC_ORG ≤ line number < IC_LIM. IC_MAX is the largest value attained by CUR_IC_BLK.

IC_FORM is the form of the i/c que entry.

IC_FORM = 1 - entry is an <arith exp> for a repeat count
2 - entry is a constant for an i/c value
3 - entry is made after all value entries in a <repeated constant> and is used to generate the ELRI.

When IC_FORM(i) = 2,

IC_LEN(i) is the PSEUDO_FORM of the entry.

IC_TYPE(i) is the PSEUDO_TYPE of the entry.

IC_VAL(i) is NUM_ELEMENTS at the time the entry was made.

IC_LOC(i) is a literal table pointer.

When IC_FORM(i) = 1,

IC_LEN(i) is number of values affected by this repetition count.

IC_TYPE(i)

IC_VAL(i) is a nesting number used by Phase 2 to check matching SLRI, ELRI operations.

IC_LOC(i) is the repeat count.

IC_LOC
See IC_LINE.

IC_MAX
See IC_LINE.

IC_ORG
See IC_LINE.

IC_PTR
At the beginning of processing an i/c "statement", an indirect stack entry is created to describe the rest of the list. IC_PTR points to that entry.

IC_PTR1
Value of IC_PTR in factored case.

IC_PTR2
Value of IC_PTR in non-factored case.

IC_TYPE
See IC_LINE.

IC_VAL
See IC_LINE.

ICQ
When doing initialization ICQ takes on the value of IC_PTR1 or IC_PTR2 -- which ever is appropriate.

ID_LOC
Symbol table pointer for name being declared.

ILL_ATTR(type)
Is a SYT_FLAGS style bit string of attributes illegal for that type.

ILL_CLASS_ATTR(class)
Is a SYT_FLAGS style mask of attributes illegal for that class.

ILL_EQUATE_ATTR
Is a SYT_FLAGS style mask of attributes illegal for EQUATE.

ILL_INIT_ATTR Same for initiaization.

ILL_LATCHED_ATTR Same for latched event.

ILL_MINOR_STRUC A SYT_FLAGS style mask for attributes illegal for a minor structure node.

ILL_NAME_ATTR Same for NAME operation.

ILL_TEMPL_ATTR Same for templates.

ILL_TEMPORARY_ATTR Same for temporaries.

ILL_TERM_ATTR(name) A SYT_FLAGS style mask for attributes illegal for a structure terminal node with or without name attribute.

IMPLIED_UPDATE_LABEL Counts the number of unlabelled update blocks. Used to generate unique labels for those blocks.

IND_LINK Points to the last subscript entry processed by REDUCE_SUBSCRIPT.

INDENT_INCR The indentation increment.

INDENT_LEVEL See Output Writer.

INIT_EMISSION On if some initialization has been issued.

INLINE_LABEL Incremented by 1 for each inline function processed.

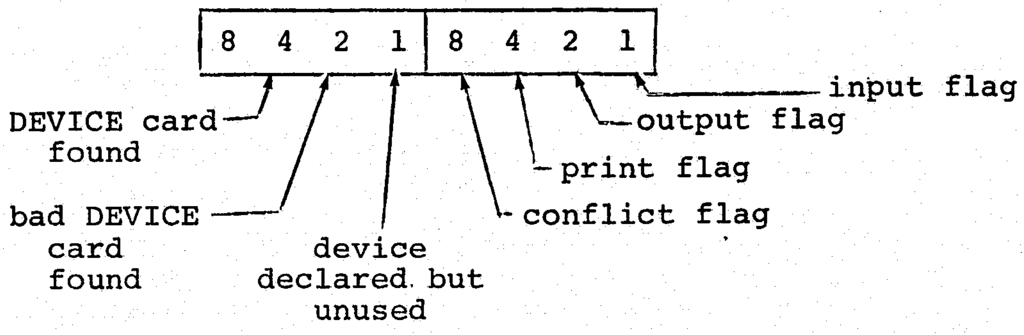
INLINE_LEVEL Incremented on entering inline function decremented on leaving it; consequently, it should be 0 or 1.

INLINE_NAME The name of the inline function being processed.

INX See PTR_TOP.

IODEV

Indexing by device number (0-9) yields the device's characteristics in the form of an eight bit descriptor.



conflict ≡ DEVICE Says print but READ or READALL was found.

LABEL_COUNT

Total number of labels declared so far.

LAST_POP#

See NEXT_ATOM#.

LOC_P

See PTR_TOP.

LOCK#

The value of <constant> in the LOCK(<constant>) declaration. "FF" indicates the value was illegal.

MAT_LENGTH

dim 1	dim 2
8	8

where the current matrix declaration is for dimensions dim 1, and dim 2.

MAX_PTR_TOP

See PTR_TOP.

MAX_SCOPE#

When entering a new scope, a new SCOPE# must be generated. Since SCOPE# can decrease when exiting a scope, MAX_SCOPE# = maximum value achieved by SCOPE# is required.

MAXNEST

Maximum value of NEST.

MISC_NAME_FLAG   See symbol table -- SYT_FLAGS.

N_DIM   The number of dimensions in a declared array.

NDECSY   See symbol table.

NEST   Every time a scope is entered, NEST is incremented; every time a scope is exited, NEST is decremented; thus, NEST is the number of enclosing scopes.

NEXT_ATOM#   The HALMAT is kept on a paged file, HALMAT_FILE. The current block, number HALMAT_BLOCK, is stored in ATOMS. NEXT_ATOM# points to the next available location in ATOMS; LAST_POP# is the NEXT_ATOM# value for the last HALMAT operator word. CURRENT_ATOM is a word to be inserted in the HALMAT file. ATOM#_FAULT is used to control HALMAT_OUT. If it is -1, clear out the whole buffer; otherwise, output that part of the buffer up to, but not including, ATOM#_FAULT.

NEXT_SUB   Pointer to the indirect stack entry for the next subscript item to process.

NONHAL   The value of <level> in NONHAL(<level>).

NUM_ELEMENTS   Number of elements to set in an initial list.

NUM_STACKS   Number of i/c que entries for an initial list.

ON_ERROR_PTR   See symbol table -- EXT_ARRAY.

OUTER_REF_INDEX   See OUTER_REF in SCAN.

OUTER_REF_PTR   See OUTER_REF in SCAN.

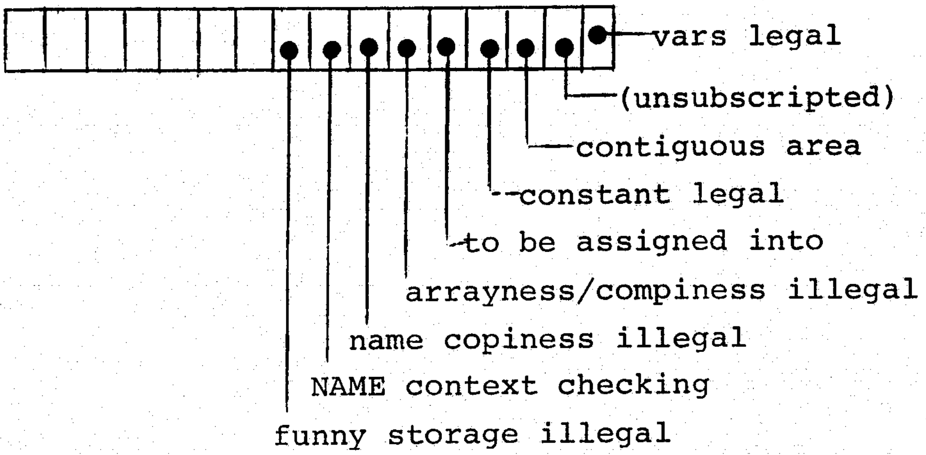
PARM_CONTEXT   See CONTEXT in SCAN.

PARMS_PRESENT   Number of formal parameters encountered.

PARMS_WATCH   Expecting formal parameters.

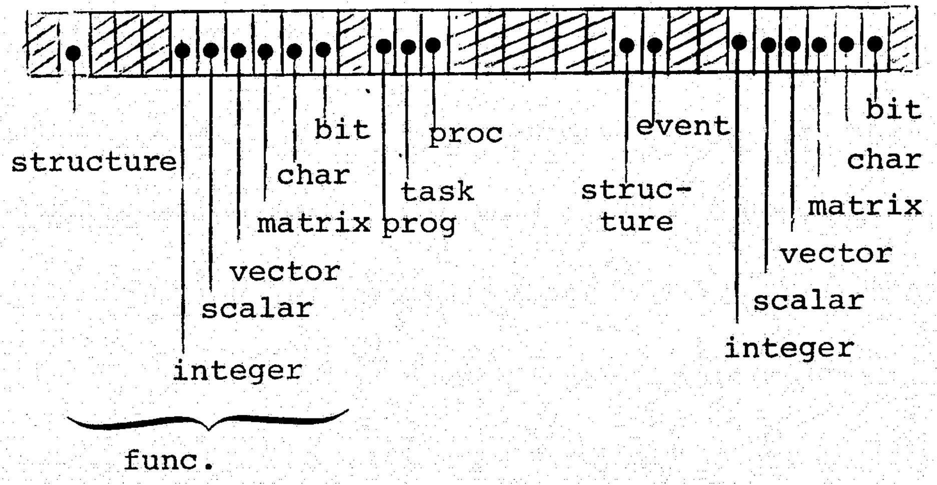
PCARG#(i) The number of arguments expected for the i^{th} % macro.

PCARGBITS Restrictions on % macro arguments.



PCARGOFF(i) A pointer (for the i^{th} % macro) to the beginning of the list of descriptors in PCARGBITS and PCARGTYPE.

PCARGTYPE Legality indicators for % macro arguments.



PCCOPY_INDEX — The index in PC... of %COPY.

PROCMARK — Pointer to the first symbol table entry for the current block.

PROCMARK_STACK(nest) — The PROCMARK for the enclosing nesting level.

PROG_MODE — See BLOCK_MODE.

PROGRAM_LAYOUT — Contains the symbol table pointer for the block name of the associated block.

PROGRAM_LAYOUT_INDEX — Points to the last entry in PROGRAM_LAYOUT.

PSEUDO_FORM — See PTR_TOP.

PSEUDO_LENGTH — See PTR_TOP.

PSEUDO_TYPE — See PTR_TOP.

PTR — See PTR_TOP.

PTR_TOP — When the semantics of an item require more space than is available in the parser's direct stack, space is allocated in the indirect stack and the parser's PTR stack entry is set to point to the entry. PTR_TOP points to the top of the indirect stack and MAX_PTR_TOP is the greatest value achieved by PTR_TOP.

EXT_P — INDENT_LEVEL before entering inline function.

INX — STMT_NUM before entering inline function.

VAL_P masks are:

"1" item has arrayness
"2" item has copiness
"4" major structure
"8" array or component subscripting
"10" subscripting illegal for assign parameter or name, i.e. subscript is not a numeric index; character or bit component subscript; vector and matrix not scalar; subscript not removing array copies; name(?) → arrayed subscript; subscript removed some but not all copiness

"20" item contains subscripting
"40" item contains precision modifier
"80" item is SUBBIT(something)
"100" item is NAME(something) or NULL
"200" name attribute
"400" null
"800" name(?) nesting warning
"1000" leaf node is a template name
"2000" some but not all conditions for status "10" have been found, be on the look-out

- for <init/const head> PSEUDO_TYPE:

  0 no * in declaration,
  1 * in declaration.

  LOC_P = number of elements affected

  VAL_P = number of GVRs used

  PSEUDO_LENGTH = length of list including this item

  PSEUDO_FORM = 0

  INX = value of IC_LINE at the beginning of the list

- for <repeat head> INX = number of elements specified in repeat count.

- for <arith exp> Of form XLIT, LOC_P = literal table pointer.

- for values associated with IC_PTR or ICQ
  See <init/const head> above.

- for <constant>, <bit constant>

  ... a literal table pointer.

- for <...var>:

  PSEUDO_TYPE = SYT_TYPE of id

  PSEUDO_LENGTH = VAR_LENGTH of id

  PSEUDO_FORM = SYT
  LOC_P = symbol table pointer of id

  or PSEUDO_FORM = XPT
  LOC_P = HALMAT pointer

  EXT_P = STACK_PTR of id

  INX = NEXT_ATOM# value for first operand of TSUB is one was issued.

- for <prefix>
  or <qual struct>:

All of the entries for the qualifiers are immediately above the entry for <prefix>. INX≠0 means there is another entry. LOC_P of each such entry is a symbol table pointer for the qualifier PSEUDO_TYPE = MAJ_STRUC, EXT_P = STACK_PTR.

- for <subscript>
  or <$>:

LOC_P contains the value for numeric subscript.

VAL_P, see ARRAY_SUB_COUNT.

PSEUDO_LENGTH, see STRUCTURE_SUB_COUNT.

INX, see SUB_COUNT.

- for <sub> and its constituents :

INX =	0	*	type
	1	sub exp	type
	2	TO	type
	3	AT	type

LOC_P = value if <sub> is a number,
        VAC pointer if it is computed.

VAL_P =	0	no #
	1	just a #
	2	# + <term>
	3	# - <term>

The INX entry is transformed by REDUCE_SUBSCRIPT so that the low order bit indicates partitioning down to a single element, 4 indicates array subscripting 8 indicates structure subscripting.

PSEUDO_LENGTH links together entries for the parts (i.e. array, structure, ...) of an entire subscript. PSEUDO_LENGTH(0) points to the beginning of the list.

- for <list expression>:

Same as <expression> except that INX = <arith exp> in <list expression>s of the form <arith exp> # <expression>.

- for <bit prim>:

INX = 0 not an event
1 event found with REFER_LOC > 0
2 event found with REFER_LOC <= 0

- for <qualifier>:

PSEUDO_FORM = 1 SINGLE
2 DOUBLE

- for <bit qualifier>:

	radix
PSEUDO_LENGTH = 1	- BIN
2	- DEC
3	- OCT
4	- HEX

- for <while clause>:

INX = 0 for WHILE
1 for UNTIL

- for <for list>:

PTR = 0 discrete for
1 DO FOR TO
2 DO FOR TO BY

- for <any statement>:

PTR = 1 for <statement> and update block
0 otherwise

- for <terminate list>:

EXT_P = length of list

- for <label var> or REFER_LOC

INX =

bits	mean
"1"	AT
"2"	IN
"3"	ON
"4"	priority specified
"8"	DEPENDENT_specified
"10"	REPEAT
"20"	REPEAT EVERY
"30"	REPEAT_AFTER
"40"	UNTIL <arith exp>
"80"	WHILE <bit exp>
"C0"	UNTIL <bit exp>

- for <read key> or <write key>:

```
INX = 0 - READ
      1 - READALL
      2 - WRITE
```

- for <block body> :

```
PTR = 0 - just declarations
      1 - at least one statement
```

QUALIFICATION

When reading a qualified structure name (e.g. A,B,C) a separate call is made to IDENTIFY for each name. QUALIFICATION is reset each time that the symbol table entry for a node name is found, so that when searching for C, we find the C hanging from B which is hanging from A rather than some other C. QUALIFICATION is zero when not reading a qualified structure name.

REF_ID_LOC

When building a structure template, a pointer to the symbol table entry for the root node.

REFER_LOC

For WAIT -- 1,
for SCHEDULE -- indirect stack pointer
for program or task.

REL_OP

The kind of <relational op> 0 - =, 1 - NOT=, 2 - <, 3 - >, 4 - <=, 4 - NOT >, 5 - >=, 5 - NOT <.

RIGID_FLAG

See symbol table -- SYT_FLAGS.

S_ARRAY(i)

The size of the i^{th} dimension of the current identifier being declared. -1 means * arrayness.

SAVE_SCOPE

The name associated with the current block.

SCOPE#

Each naming scope requires a unique identifier to resolve the problems of nested declarations. This is SCOPE# and it is saved in SYT_SCOPE for each variable.

SCOPE#_STACK(nest)

The SCOPE# associated with the enclosing nest level.

SRN_COUNT_MARK

Saves SRN_COUNT(2) while processing inline functions.

SRN_MASK — Saves SRN(2) while processing inline functions.

STAB_MARK — Saves STAB_STACKTOP while processing inline functions.

STAB_STACK — Is a stack of information for generating simulation information. STAB_STACKTOP points to the topmost entry.

label entry =

101	symbol table pointer
3	13

STAB_STACKTOP — See STAB_STACK.

STACK_PTR — See GRAMMAR_FLAGS.

STARRED_DIMS — Number of dimensions specifying * arrayness in current declaration.

STRUC_DIM — The copiness of a structure declaration. -1 implies * copiness.

STRUC_PTR — When processing a structure declaration, this is a pointer to the symbol table entry for the template of the structure.

STRUCTURE_SUB_COUNT — LITERALLY PSEUDO_LENGTH(PTR(MP)) initialized to -1. SUB_COUNT is copied here on finding a ";" in a subscript. If no structure subscript is found before a ":" in the subscript, it is reset to 0.

SUB_COUNT — LITERALLY INX(PTR(MP)). The number of <sub>s encountered in the entire subscript.

SUB_SEEN — 0 - no <sub> encountered,
1 - <sub> encountered in current group,
2 - <sub> encountered in previous group but not yet in current group.

SUBSCRIPT_LEVEL

Zero for unsubscripted item, increased by one for each level of subscripting.

SYT_SCOPE

See symtol table and SCOPE#.

TASK_MODE

See BLOCK_MODE.

TEMP3

0 - radix was DEC,
1 - was BIN,
2 - radix was DEC, converted in production 259,
3 - radix was OCT,
4 - radix was HEX.

TYPE

TYPE(0) is the type just read from an attribute list. Notice that TYPE(1) ≡ BIT_LENGTH... .

UPDATE_BLOCK_LEVEL

Incremented on entering update block, decremented on leaving. Since update blocks may not be nested, it should always be zero or one.

UPDATE_MODE

See BLOCK_MODE.

VAL_P

See PTR_TOP.

VAR

Initially the name associated with an element on the parse stack. For blocks, it is replaced by CURRENT_SCOPE at the time the block is entered.

VAR_ARRAYNESS(i)

i = 0 - the number of subscripts possible.

$1 \leq i \leq$ VAR_ARRAYNESS(0) - the maximum for the i^{th} subscript.

After all subscripts have been processed, any residual arrayness is copied to CURRENT_ARRAYNESS. If CURRENT_ARRAYNESS≠0, then the residual arrayness must match.

CURRENT_ARRAYNESS must often be stacked on ARRAYNESS_STACK, (e.g. when evaluating a function argument). This is done by SAVE_ARRAYNESS. The stacking is done upside down. That is, stack CURRENT_ADDRESS(CURRENT_ARRAYNESS)... until finally, stack CURRENT_ARRAYNESS(0) on the top. AS_PTR points to the topmost entry in ARRAYNESS_STACK.

VEC_LENGTH	Length for vector declaration being processed.
XCDEF	Compool indicator.
XFDEF	Function indicator.
XMDEF	Program indicator.
XPDEC	Procedure indicator.
XTDEF	Task indicator.
XUDEF	Update block indicator.

4.4.2 <block stmt> and <... inline def>

As can be seen from the grammar fragment below, the <block stmt> is the opening of the block. This is where new scopes are entered, procedure, function, ..., names are defined, etc.

```
  1  <compilation>  ::= <compile list> _|_
  2  <compile list> ::= <block definition>
 39  <any statement> ::= <block definition>
289  <block definition> ::= <block stmt> <block body> <closing>
```

Although the inline functions appear in another part of the garmmar, they are most naturally treated here.

This section deals with productions 293-328

```
293  <ARITH INLINE DEF>  ::=  FUNCTION <ARITH SPEC>:
294                         |  FUNCTION;

295  <BIT INLINE DEF> ::=  FUNCTION <BIT SPEC> ;

296  <CHAR INLINE DEF> ::=  FUNCTION <CHAR SPEC> ;

297  <STRUC INLINE DEF> ::=  FUNCTION <STRUC SPEC> ;
```

```
298   <BLOCK STMT> ::= <BLOCK STMT TOP> ;

299   <BLOCK STMT TOP> ::= <BLOCK STMT TOP> ACCESS
300                      | <BLOCK STMT TOP> RIGID
301                      | <BLOCK STMT HEAD>
302                      | <BLOCK STMT HEAD> EXCLUSIVE
303                      | <BLOCK STMT HEAD> REENTRANT

304   <LABEL DEFINITION> ::= <LABEL> :

305   <LABEL EXTERNAL> ::= <LABEL DEFINITION>
306                      | <LABEL DEFINITION> EXTERNAL

307   <BLOCK STMT HEAD> ::= <LABEL EXTERNAL> PROGRAM
308                       | <LABEL EXTERNAL> COMPOOL
309                       | <LABEL DEFINITION> TASK
310                       | <LABEL DEFINITION> UPDATE
311                       | UPDATE
312                       | <FUNCTION NAME>
313                       | <FUNCTION NAME> <FUNC STMT BODY>
314                       | <PROCEDURE NAME>
315                       | <PROCEDURE NAME> <PROC STMT BODY>

316   <FUNCTION NAME> ::= <LABEL EXTERNAL> FUNCTION

317   <PROCEDURE NAME> ::= <LABEL EXTERNAL> PROCEDURE

318   <FUNC STMT BODY> ::= <PARAMETER LIST>
319                      | <TYPE SPEC>
320                      | <PARAMETER LIST> <TYPE SPEC>

321   <PROC STMT BODY> ::= <PARAMETER LIST>
322                      | <ASSIGN LIST>
323                      | <PARAMETER LIST> <ASSIGN LIST>

324   <PARAMETER LIST> ::= <PARAMETER HEAD> <IDENTIFIER> )

325   <PARAMETER HEAD> ::= (
326                      | <PARAMETER HEAD> <IDENTIFIER> ,

327   <ASSIGN LIST> ::= <ASSIGN> <PARAMETER LIST>

328   <ASSIGN> ::= ASSIGN
```

Productions 293-297

```
<arith inline def> ::=  FUNCTION <arith spec> ;
                        FUNCTION;
<bit inline def>   ::=  FUNCTION <bit spec> ;
<char inline def>  ::=  FUNCTION <char spec>;
<struc inline def> ::=  FUNCTION <struc spec>;
```

Set TEMP to the size of the function result, 0 for integer or scalar.

Build an indirect stack entry for the result. Save the various simulation and SDL information until the inline is finished.

Augment the name of the function with a unique number and make a symbol table entry for it.

Issue an IDEF instruction. SAVE_ARRAYNESS.

Finish normal procedure processing by joining production 317.

Production 298 <block stmt> ::= <block stmt top>;

Clear out the listing buffers, turn off template generation, set to indent the rest of the block, and emit a HALMAT statement mark.

Production 299 <block stmt top> ::= <block stmt top> ACCESS

Set ACCESS_FLAG for the block's name.

Production 300 <block stmt top> ::= <block stmt top> RIGID

Set RIGID_FLAG for block's name.

Production 301 <block stmt top> ::= <block stmt head>

Nothing.

Production 302 <block stmt top> ::= <block stmt head> EXCLUSIVE

Set EXCLUSIVE_FLAG for the block's name.

Production 303 <block stmt top> ::= <block stmt head> REENTRANT

Set REENTRANT_FLAG for the block's name.

Production 304 <label definition> ::= <label>

Generate HALMAT to define the label. Set up the SYT_LINK1 and SYT_LINK2 entries. Count the label. If a simulation was requested, stack the label's symbol table entry via STAB_LAB. Set the LABEL_FLAG in the label's GRAMMAR_FLAGS entry. Make a cross reference entry.

Production 305 <label external> ::= <label definition>

Do nothing.

Production 306 <label external> ::= <label definition> EXTERNAL

Set external flag in SYT_FLAGS. Temporarily turn off acquisition of simulation information.

Production 307 <block stmt head> ::= <label external> PROGRAM

Set to no parameters. Insert PROG_LABEL as SYT_TYPE and check for consistency using SET_LABEL_TYPE.

Most of the time, control will proceed to DUPLICATE_BLOCK (including from compools, tasks, and update blocks) where BLOCK_MODE is usually zero. Here, we initialize for the new block, set up EXTERNALIZE and call EMIT_EXTERNAL to start template production. Finally, we join all other control flow paths which can enter a new scope at NEW_SCOPE in production 317.

Production 308 <block stmt head> ::= <label external> COMPOOL

Set for one parameter. Define SYT_TYPE of the label via SET_LABEL_TYPE. Join production 307.

Production 309 <block stmt head> ::= <label definition> TASK

Define SYT_TYPE of label via SET_LABEL_TYPE. Set LATCHED_FLAG for task name so it will behave like a latched event. Join production 307.

Production 310 <block stmt head> ::= <label definition> UPDATE

Set for labeled update block and backspace over HALMAT which defined the label. Unlabeled update blocks join here. Define label to be normal statement label via SET_LABEL_TYPE. Join all scope defining statements at NEW_SCOPE.

Production 311 <block stmt head> ::= UPDATE

Generate a label and simulate an UPDATE statement with that label. Join labelled update blocks.

Production 312 <block stmt head> ::= <function name>

Check the type for legality and fill in scalar if not specified. If it is numeric and the necessary attributes have not been specified, fill in the default.

If the function was not defined earlier, SET_SYT_ENTRIES; otherwise, check that this declaration agrees with the earlier one.

Clear out the TYPE information, set FACTORING and clear DO_INIT in preparation for handling the declaration part of the function.

Production 313 <block stmt head> ::= <function name>
<function stmt body>

Same as production 312.

Production 314 <block stmt head> ::= <procedure name>

Turn off PARMS_WATCH. Everything else has already been done in <procedure name>.

Production 315 <block stmt head> ::= <procedure name>
<procedure stmt body>

Everything has already been done.

Production 316 <function name> ::= <label external> FUNCTION

Set ID_LOC to symbol table entry for label. Fill in symbol table entry. Join <procedure name> in production 317.

Production 317 <procedure name> ::= <label external> PROCEDURE

Define label as procedure using SET_LABEL_TYPE. <function name> joins in here from production 316. Set for no parameters seen, turn on PARMS_WATCH and join up with everything else at DUPLICATE_BLOCK. Later, everything comes back down here at NEW_SCOPE.

Clear the SYT_LINK1 entry and remove the name from the SYT_LINK2 list of labels. Back up the HALMAT to eliminate the label definition. Notice that this has already been done for update blocks.

Issue HALMAT to define the label. <arith inline def> joins here from production 293. Initialize all the descriptors for this new nest level (see data descriptions for their meanings) and stack the old ones. If the block is an inline function; save listing information, set up special listing format for an inline and emit a HALMAT statement mask.

Production 318 - 323

Productions 313 and 315 require a <func stmt body> and <proc stmt body> respectively. These items are purely syntactic; however, there are semantics associated with their constituents covered by productions 324 - 328 and 372 - 377. The latter group handles the <type spec> on function declarations and is discussed in the section on declarations.

Production 324 <parameter list> ::= <parameter head> <identifier>

Count the last parameter.

Production 325 <parameter head> ::= (

Just get prepared for 326.

Production 326 <parameter head> ::= <parameter head> <identifier>,

Count all the parameters except the last.

Production 327 <assign list> ::= <assign> <parameter list>

Nothing.

Production 328 <Assign> ::= ASSIGN

Reset the context properly.

4.4.3 <declare group>

As can be seen from the grammar fragment below, the <declare group> is the declaration section of the <block definition>. This is where all new variables for the newly opened scope are defined.

```
  1 <compilation> ::= <compile list> _|_
  2 <compile list> ::= <block definition>
 39 <any statement>  ::= <block definition>
289 <block definition> ::= <block stmt><block body> <closing>
290 <block body>  ::=
291                    |<declare group>
292                    |<block body> <any statement>
```

This section deals with productions 329-425.

```
329    <DECLARE ELEMENT> ::= <DECLARE STATEMENT>
330                        | <REPLACE STMT> ;
331                        | <STRUCTURE STMT>
332                        | EQUATE EXTERNAL <IDENTIFIER> TO <VARIABLE> ;

333    <REPLACE STMT> ::= REPLACE <REPLACE HEAD> BY <TEXT>

334    <REPLACE HEAD> ::= <IDENTIFIER>
335                     | <IDENTIFIER> ( <ARG LIST> )

336    <ARG LIST> ::= <IDENTIFIER>
337                 | <ARG LIST> , <IDENTIFIER>
```

```
338   <TEMPORARY STMT> ::= TEMPORARY <DECLARE BODY> ;

339   <DECLARE STATEMENT> ::= DECLARE <DECLARE BODY> ;

340   <DECLARE BODY> ::= <DECLARATION LIST>
341                    | <ATTRIBUTES> , <DECLARATION LIST>

342   <DECLARATION LIST> ::= <DECLARATION>
343                        | <DCL LIST ,> <DECLARATION>

344   <DCL LIST ,> ::= <DECLARATION LIST> ,

345   <DECLARE GROUP> ::= <DECLARE ELEMENT>
346                     | <DECLARE GROUP> <DECLARE ELEMENT>

347   <STRUCTURE STMT> ::= STRUCTURE <STRUCT STMT HEAD> <STRUCT STMT TAIL>

348   <STRUCT STMT HEAD> ::= <IDENTIFIER> : <LEVEL>
349                        | <IDENTIFIER> <MINOR ATTR LIST> : <LEVEL>
350                        | <STRUCT STMT HEAD> <DECLARATION> , <LEVEL>

351   <STRUCT STMT TAIL> ::= <DECLARATION> ;

352   <STRUCT SPEC> ::= <STRUCT TEMPLATE> <STRUCT SPEC BODY>

353   <STRUCT SPEC BODY> ::= - STRUCTURE
354                        | <STRUCT SPEC HEAD> <LITERAL EXP OR *> )

355   <STRUCT SPEC HEAD> ::= - STRUCTURE (

356   <DECLARATION> ::= <NAME ID>
357                   | <NAME ID> <ATTRIBUTES>

358   <NAME ID> ::= <IDENTIFIER>
359               | <IDENTIFIER> NAME

360   <ATTRIBUTES> ::= <ARRAY SPEC> <TYPE & MINOR ATTR>
361                  | <ARRAY SPEC>
362                  | <TYPE & MINOR ATTR>

363   <ARRAY SPEC> ::= <ARRAY HEAD> <LITERAL EXP OR *> )
364                  | FUNCTION
365                  | PROCEDURE
366                  | PROGRAM
367                  | TASK

368   <ARRAY HEAD> ::= ARRAY (
369                  | <ARRAY HEAD> <LITERAL EXP OR *> ,

370   <TYPE & MINOR ATTR> ::= <TYPE SPEC>
371                         | <TYPE SPEC> <MINOR ATTR LIST>
372                         | <MINOR ATTR LIST>

373   <TYPE SPEC> ::= <STRUCT SPEC>
374                 | <BIT SPEC>
375                 | <CHAR SPEC>
376                 | <ARITH SPEC>
377                 | EVENT
```

```
378    <BIT SPEC> ::= BOOLEAN
379                 | BIT ( <LITERAL EXP OR *> )

380    <CHAR SPEC> ::= CHARACTER ( <LITERAL EXP OR *> )

381    <ARITH SPEC> ::= <PREC SPEC>
382                   | <SQ DQ NAME>
383                   | <SQ DQ NAME> <PREC SPEC>

384    <SQ DQ NAME> ::= <DOUBLY QUAL NAME HEAD> <LITERAL EXP OR *> )
385                   | INTEGER
386                   | SCALAR
387                   | VECTOR
388                   | MATRIX

389    <DOUBLY QUAL NAME HEAD> ::= VECTOR (
390                              | MATRIX ( <LITERAL EXP OR *> ,

391    <LITERAL EXP OR *> ::= <ARITH EXP>
392                         | *

393    <PREC SPEC> ::= SINGLE
394                  | DOUBLE

395    <MINOR ATTR LIST> ::= <MINOR ATTRIBUTE>
396                        | <MINOR ATTR LIST> <MINOR ATTRIBUTE>

397    <MINOR ATTRIBUTE> ::= STATIC
398                        | AUTOMATIC
399                        | DENSE
400                        | ALIGNED
401                        | ACCESS
402                        | LOCK ( <LITERAL EXP OR *> )
403                        | REMOTE
404                        | RIGID
405                        | <INIT/CONST HEAD> <REPEATED CONSTANT> )
406                        | <INIT/CONST HEAD> * )
407                        | LATCHED
408                        | NONHAL ( <LEVEL> )

409    <INIT/CONST HEAD> ::= INITIAL (
410                        | CONSTANT (
411                        | <INIT/CONST HEAD> <REPEATED CONSTANT> ,

412    <REPEATED CONSTANT> ::= <EXPRESSION>
413                          | <REPEAT HEAD> <VARIABLE>
414                          | <REPEAT HEAD> <CONSTANT>
415                          | <NESTED REPEAT HEAD> <REPEATED CONSTANT> )
416                          | <REPEAT HEAD>

417    <REPEAT HEAD> ::= <ARITH EXP> #

418    <NESTED REPEAT HEAD> ::= <REPEAT HEAD> (
419                           | <NESTED REPEAT HEAD> <REPEATED CONSTANT> ,

420    <CONSTANT> ::= <NUMBER>
421                 | <COMPOUND NUMBER>
422                 | <BIT CONST>
423                 | <CHAR CONST>

424    <NUMBER> ::= <SIMPLE NUMBER>
425               | <LEVEL>
```

As can be seen from productions 345 and 346, a <declare group> is simply a list of <declare element>s; thus, the interesting question is "what goes into a <declare element>?"

As usual, the highest level productions do a little bookkeeping.

Production 329 <declare element> ::= <declare statement>

Nothing.

Production 330 <declare element> ::= <replace statement>

Several productions come here to clean up. Clear output writer buffers and emit a statement mark.

Production 331 <declare element> ::= <structure statement>

Save the size and join 330.

Production 332 <declare element> ::= EQUATE EXTERNAL
<identifier> TO <variable>

The EQUATE EXTERNAL feature is inconsistent with the rest of the language; therefore, the whole mechanism which handles all the other declares is by-passed.

Set SYT_PTR of <identifier> to point to the <variable>. Check that the EQUATE is legal, generate HALMAT initialization to perform the equate. Drop any accumulated arrayness. Pop PTR_TOP down to before the EQUATE statement.

Production 333 <replace stmt> ::= REPLACE <replace head> BY <text>

The <text> is already in MACRO_TEXT. Just fill in the symbol table entry for the replace name and drop any context.

Production 334 and 335

<replace head> ::= <identifier>|<identifier> (<arg list>)

Drop the context.

```
Production 336 and 337  <arg list>  ::= <identifier>
                                       | <arg list>,<identifier>
```

Count the argument and build and cross reference entry.

```
Production 338  <temporary stmt>  ::=  TEMPORARY <declare body>
```

See production 339.

```
Production 339  <declare statement>  ::=  DECLARE <declare body>
```

This production basically cleans house.

Set to accumulate new factored attirubtes. Discard any i/c information that was used up in <declare body>. Diddle the output writer to make everything line up.

Production 340 and 341

```
<declare body> ::= <declaration list>
                 | <attributes>, <declaration list>
```

Drop accumulated factored attributes.

```
Production 342  <declaration list>  ::= <declaration>
```

Adjust for output writer.

```
Production 343  <declaration list>  ::= <decl list,> <declaration>
```

Nothing.

```
Production 344  <dcl list,>  ::= <declaration list>,
```

Call output writer in parts to make things line up nicely. Emit a statement mark if any initialization was done.

Production 345 and 346

```
<declare group> ::= <declare element>
                  | <declare group> <declare element>
```

Nothing.

Production 347

<structure stmt> ::= STRUCTURE <structure stmt head>
                                                <structure stmt tail>

Set FACTORING.

Move FIXL and FIXV stacks down to simulate status in 350 and then join 350.

Production 348, 349

<struct stmt head> ::= <identifier>:<level>
                     | <identifier> <minor attr list>:<level>

Turn on BUILDING_TEMPLATE. Initialize for a new template. Insert SYT_CLASS and SYT_TYPE for identifier . Clear out TYPE array. Clear out output writer buffers. Join 350.

Production 350

<struct stmt head> ::= <struct stmt head> <declaration>,<level>

By this point the structure template has been initialized in 348 or 349 and zero or more nodes have been accumulated by recursive application of this production.

If DUPL_FLAG is on, turn it off and walk the structure checking that the duplicate name is not in the same template.

If <level> is greater than the current one, then the node being processed is a minor structure, not a leaf. Increment the current level and check that the declaration of the minor structure node contained nothing illegal for such a node (e.g. it cannot have a type and it cannot have arrayness). Set the SYT_CLASS of the minor structure. Copy in ALDENSE and RIGID attributes from the root node. Update the symbol table entry via SET_SYT_ENTRIES. Stack the old containing node on the indirect stack, set that the containing node is the node being processed, set SYT_LINK1 to point to the next symbol table entry so that that entry will be the first son.

If <level> is less than or equal to the current one then we have just accumulated all the sons of a node. Fill in the SYT_LINK2 entry of all last sons as a negative pointer to the father. Notice that several subtrees may be terminated so a loop popping all entries off the indirect stack is necessary. The entry just finished is a leaf so it must be a data entry -- check it like any other data declaration. Update symbol table entry via SET_SYT_ENTRIES. If the new level number is greater than zero, build a link from its left brother; otherwise, clear out the template building variables. The latter condition is achieved by finding the closing ";", reducing to

<struct stmt tail>(351), reducing to <structure stmt>(347) and jumping to STRUCT_GOING_UP.

Production 351   <struc stmt tail> ::= <declaration> ;

Nothing -- see 350.

Production 352

<struct spect> ::= <struct template> <struct spec body>

Set STRUC_PTR to point to the symbol table entry of the template, generate cross reference.

Production 353   <struc spec body> ::= -structure

Simple case is just syntactic; diddle the output writer.

Production 354

<struc spec body> ::= <struc spec head> <literal exp or *>

Check dimension and set STRUC_DIM. Reset CONTEXT back to DECLARE_CONTEXT after handling <literal exp or *>.

Production 355   <struc spec head> ::= -STRUCTURE (

This is here only to allow diddling the output writer.

Productions 356 and 357 <declaration> ::= <name id>
| <name id> <attributes>

By this time we have accumulated an identifier and all of its attributes including i/c attributes. This is the place where all of the hanging flags actually get installed permanently.

Set that any any pending initialization should be issued.

CHECK_CONFLICTS is called to check conflicts between factored and non-factored attributes. The factored ones are then copied to the non-factored to produce a complete description of the name. CHECK_CONSISTENCY is called by CHECK_CONFLICTS to check that the attributes are self consistent.

If the name is a formal parameter, decrement the number of expected parameters and check that the attributes are legal for a parameter. If it is not a parameter and we are looking for parameter declarations -- error.

If the name is an event, call CHECK_EVENT_CONFLICTS to check that the other attriubtes are consistent with an event.

If the name is not NAME variable, then:

- NONHAL must be either a procedure or function and cannot be in a COMPOOL.

- Functions cannot be declared in a COMPOOL.

- The only CLASS 1 objects that can appear in a DECLARE are tasks. For tasks, we must be in the outermost nest of a program block.

- If there was an illegal initialization attempted on a non-CLASS 0 name, issue error message and set to not perform initialization.

- Only CLASS 0 variables can be temporary.

- Check consistency of attributes for TEMPORARY CLASS 0 variables.

If the name is a NAME variable, check that the other attributes are consistent with NAME.

If the name is a structure, call CHECK_STRUC_CONFLICTS:

- If the name is not of variable class, it must be qualified with no copies.

- If we have an unqualified structure: it must have a template in the current scope; there must not already be an unqualified structure for the template; the template must have no duplicate names; the template must not reference any other structure.

- If the template contains a name variable then the structure cannot be temporary and any other template referencing the template must inherit the property of containing a name variable.

The accumulated information about the variable is finally inserted in the symbol table using SET_SYT_ENTRIES (described separately). Notice that SET_SYT_ENTRIES in turn calls HALMAT_INIT_CONST to actually emit HALMAT initialization for the variable.

If the variable is TEMPORARY, then link it into the list of temporaries for the current do nesting level and issue.

Production 358 <name id> ::= <identifier>

Set ID_LOC to point to <identifier>.

Production 359 <name id> ::= <identifier> NAME

Set NAME_IMPLIED and point ID_LOC at <identifier>.

Productions 360 and 361

```
<attributes> ::= <array spec> <type & minor attributes>
               | <array spec>
```

Check that dimension specifications were legal and fall into 362.

Production 362 <attributes> ::= <type & minor attributes>

Check the declaration for consistency via CHECK_CONSISTENCY.

If FACTORING is on then the attributes are factored attributes so copy them and set FACTOR_FOUND. Similarly, for initial/constants.

Production 363 <array spec> ::= <array head> <literal exp or *>)

Reset CONTEXT to DECLARE_CONTEXT after <literal exp or *> and fall into production 369.

Production 364 <array spec> ::= FUNCTION

CLASS = 1.

```
Production 365, 366 or 367  <array spec>  ::=  PROCEDURE
                                             |  PROGRAM
                                             |  TASK
```

Set TYPE and CLASS appropriately.

Production 368 <array head> ::= ARRAY (

Prepare to accumulate dimensions by zeroing existinq values. Set FIXL(SP) and FIXV(SP) to ARRAY_FLAG for use in production 396. Join 396.

Production 369 <array head> ::= <array head> <literal exp or *>,

Save dimension in S_ARRAY.

Production 370 and 371

```
<type & minor attr> ::= <type spec>
                      | <type spec> <minor attr list>
```

Check for valid CLASS.

Production 372 <type & minor attr> ::= <minor attr list>

Nothing.

```
Production 373, 374, 375, 376, 377  <type spec>  ::= <struct spec>
                                                   | <bit spec>
                                                   | <char spec>
                                                   | <arith spec>
                                                   |  EVENT
```

Set TYPE if not already set.

Production 378 <bit spec> ::= BOOLEAN

Simulate BIT(1).

Production 379 <bit spec> ::= BIT (<literal exp or *>)

Restore CONTEXT to DECLARE_CONTEXT after <literal exp or *>. Set TYPE to BIT_TYPE and BIT_LENGTH to declared length.

Production 380 <char spec> ::= CHARACTER (<literal exp or *>)

See production 379.

Production 381 and 383 <arith spec> ::= <prec spec>
| <sq dq name> <prec spec>

Incorporate accumulated information into ATTR_MASK and ATTRIBUTES.

Production 384

<sq dq name> ::= <doubly qual name head> <literal exp or *>)

Restore CONTEXT to DECLARE_CONTEXT after <literal exp or *>. Set up VEC_LENGTH or MAT_LENGTH.

Production 385, 386, 387, 388 <sq dq name> ::= INTEGER
| SCALAR
| VECTOR
| MATRIX

Set TYPE appropriately and initialize length to default length.

Production 389 <doubly qual name head> ::= VECTOR (

Set up FIXL for production 384.

Production 390 <double qual name head> ::= MATRIX (<literal exp or *>,

Set up FIXL and FIXV for production 384.

Production 391 <literal exp or *> ::= <arith exp>

Drop any storage on the indirect stack accumulated by <arith exp>. Drop any arrayness accumulated. Put integer value of <arith exp> in FIXV. Notice that if the <arith exp> was not a compile time constant, 0 is returned. Negative constants will be detected elsewhere; however, -1 means "*" so it is transformed to the equally illegal value 0.

Production 392 <literal exp or *> ::= *

Set FIXV to -1.

Production 393 and 394 <prec spec> ::= SINGLE
| DOUBLE

Set up FIXL and FIXV for 381.

Production 395 and 396

```
<minor attr list> ::= <minor attribute>
                    | <minor attr list> <minor attribute>
```

Accumulate attributes in ATTRIBUTE and illegal attributes in ATTR_MASK.

Production 397, 398, 399, 400, 401, 403, 404, 407

```
<minor attribute>  ::= STATIC
                     | AUTOMATIC
                     | DENSE
                     | ALIGNED
                     | ACCESS
                     | REMOTE
                     | RIGID
                     | LATCHED
```

See FIXL and FIXV for 396.

Production 402 <minor attribute> ::= LOCK (<literal exp or *>)

Restore CONTEXT to DECLARE_CONTEXT after <literal exp or *>. Set LOCK# to the value of the literal expression and set up FIXL and FIXV for 396.

Production 405, 406

```
 <minor attribute> ::= <init/const head> <repeated constant>
                     | <init/const head>*
```

Set that there is or is not an *. Drop BI_FUNC_FLAG. Drop any implicit transposes. Restore CONTEXT to DECLARE_CONTEXT.

Fill in final data in indirect stack entry which describes i/c list. (The entry was built by 409). Save a pointer to this entry, it is the key to the whole i/c list.

If all this happened while processing a template declaration, throw out the whole thing since you cannot initialize a template.

Production 408 <minor attribute> ::= NONHAL (<level>)

Save <level> in NONHAL. Set up FIXV and FIXL for 396.

Production 409 and 410 <init/const head> ::= INITIAL (

```
                                              | CONSTANT (
```

Set up FIXV and FIXL for 396. Set BI_FUNC_FLAG. Get and initialize indirect stack entry which will describe the i/c list.

Production 411

```
  <init/const head> ::= <init/const head> <repeated constant>,
```

Everything done in <repeated constant> ::= ...

Production 412, 413, 414

```
      <repeated constant>  ::= <expression>
                             | <repeat head> <variable>
                             | <repeat head> <constant>
```

If initializing to the NAME of something, set bit in PSEDUO_TYPE and check that the usage of the NAME pseudo function was legal in initialization context.

Drop any arrayness. Build an i/c que entry, count the value as one more element affected, and count the i/c que entry.

If there was a repeat count, then build an i/c que entry for the repeat count. Since FIXV (<repeat head>) is the value of NUM_ELEMENTS at the beginning, NUM_ELEMENTS-FIXV is the number of elements affected by the repeat count. Multiplying that by the value of the repeat count and adding FIXV back in again yields the correct number of elements affected by the i/c list.

Finally, everything is in the i/c que so pop the indirect stack entries.

Production 415

<repeated constant> ::= <nested repeat head> <repeated constant>)

Join middle of 414.

Production 416 <repeated constant> ::= <repeat head>

Accumulate the number of elements to be skipped and then discard the i/c que entry and indirect stack entries for the <repeat head>.

Production 417 <repeat head> ::= <arith exp> #

Drop any arrayness. Build INX and FIXV entries. Build i/c que entry.

Production 418, 419

```
<nested repeat head> ::= <repeat head> (
                       | <nested repeat head> <repeated constant>,
```

Everything is done by 414 and 417.

Production 420, 421 <constant> ::= <number>

```
                                 | <compound number>
```

Create and initialize an indirect stack entry.

Production 422, 423 <constant> ::= <bit const>

```
                                 | <char const>
```

All the work was done by 266 or 271.

Production 424, 425 <number> ::= <simple number>

```
                               | <level>
```

Purely syntactic.

SET_SYT_ENTRIES --- 1047500
ENTER_DIMS --- 1043400

This routine is called to fill in information accumulated about an identifier. The information is in various global variables. ID_LOC points to the symbol table entry.

Fill in type. Check for consistency and set LOCK_FLAG.
Copy ATTRIBUTES to SYT_FLAGS.
Check * size on character strings.
Enter dimension information via ENTER_DIMS.
Check copyness for structures.
Make tasks and programs latched events.
Do any initialization.
Zero the TYPE array.

ENTER_DIMS sets SYT_ARRAY(ID_LOC) to point to an EXT_ARRAY entry that describes its dimensions. A new EXT_ARRAY entry is produced only if an appropriate one does not already exist.

HALMAT_INIT_CONST --- 1015200
HOW_TO_INIT_ARGS --- 1013200
ICQ_ARRAYNESS_OUTPUT --- 1002000

All initialization is initiated here.

If no initialization pending, just return.

If this is not a factored case, reset IC_LINE to return the i/c que space and reset PTR_TOP to return the indirect stack space.

If initialization was cancelled due to an error, return.

Call HOW_TO_INIT_ARGS to figure out relation between the variables to be initialized and the values found. The argument is the number of values in the list. The value returned is:

0 - there are fewer values than required

1 - just initialize with a single value

2 - number of values matches one element of an array or one copy of a structure

3 - number of elements exactly matches number of values

4 - number of values greater than number of elements

Case 0

Is legal only if there is an * in the value list -- then ICQ_OUTPUT handles the element-by-element initialization.

Case 1

If there was an *, everything is simple -- just call ICQ_OUTPUT to initialize one element. If there was not an *, scan through the i/c que until a i/c value is reached. Now issue HALMAT to do the initialization unless it is a constant element and a constant value.

Case 2

Output initialization for one array element. If there was no * in the value list call ICQ_ARRAYNESS_OUTPUT to issue:

- an ADLP or IDLP operator
- one operand for each dimension of arrayness
- a DLPE operator.

The ADLP or IDLP operator will be moved back by phase 1.5.

Case 3

Do element-by-element initialization using ICQ_OUTPUT.

Case 4

Same as 3, but issue error message first.

ICQ_CHECK_TYPE -- 1003900

Check that the type of the i/c value (received in first argument) is compatible with the type of the element to be initialized. Return HALMAT initialization operator of proper type. If second argument is false, use SCALAR_TYPE instead of actual type of element to be initialized when computing HALMAT operator.

ICQ_OUTPUT -- 1007200

This routine handles element-by-element initialization.

If the item to be initialized is a structure, issue:

0	2	EXTN	0	0
sym pointer	0	SYT	0	1
temp pointer	0	SYT	0	1
0	1	STRI	0	0
HALMAT pointer	0	XPT	0	1

The field HALMAT pointer should have an arrow coming out of it as shown in left margin.

If the item is simple, issue:

0	1	STRI	0	0
sym pointer	0	SYT	0	1

Having issued the initial list header code, we will now traverse the list in the i/c que issuing HALMAT for each queued value.

CT Counts the values in the list.

K Points to the current value.

CT_LIT Counts the number of successive initializations into consecutive locations.

If IC_FORM=2, this is a value to be used, not an indicator of some kind. If the previous element was also IC_FORM=2, there were fewer than 256 such, it was immediately before this one in the initial list, and the value was immediately before this one in the literal table, the short form can be used -- just count in CT_LIT. In the other case, we have to issue a sequence of HALMAT:

type	2	?INIT	0	0
NUM+ELEMENTS	0	OFF	0	1
literal pointer	0	form	0	1

where ? ≡ B, C, M, V, S, I, or T depending on the type of the item to be initialized.

If IC_FORM≠2, then this is an administrative entry. First go back and fill in the proper count in the second operand of the initialization operator. If IC_FORM=1, this is a repeat count -- issue the HALMAT:

nest level	2	SLRI	0	0
repeat count	0	IMD	0	1
number of items repeated	0	IMD	0	1

If IC_FORM=3, this is the end of a repeated sequence, just issue:

nest level	0	ELRI	0	0

where the nest levels are check for consistency by phase 2.

When the list of values is exhausted, fill in the proper count in the second argument of the last initialization operation and then terminate the initialization with an ETRI.

4.4.4 <variable>

This section deals with productions: 193 - 205 and 209 - 249.

```
193                    | <STRUCTURE VAR>
194                    | <BIT VAR>
195                    | <EVENT VAR>
196                    | <SUBBIT HEAD> <VARIABLE> )
197                    | <CHAR VAR>
198                    | <NAME KEY> ( <NAME VAR> )

199    <NAME VAR> ::= <VARIABLE>
200                    | <LABEL VAR>
201                    | <MODIFIED ARITH FUNC>
202                    | <MODIFIED BIT FUNC>
203                    | <MODIFIED CHAR FUNC>
204                    | <MODIFIED STRUCT FUNC>

205    <NAME EXP> ::= <NAME KEY> ( <NAME VAR> )



209    <LABEL VAR> ::= <PREFIX> <LABEL> <SUBSCRIPT>

210    <MODIFIED ARITH FUNC> ::= <PREFIX> <NO ARG ARITH FUNC> <SUBSCRIPT>

211    <MODIFIED BIT FUNC> ::= <PREFIX> <NO ARG BIT FUNC> <SUBSCRIPT>

212    <MODIFIED CHAR FUNC> ::= <PREFIX> <NO ARG CHAR FUNC> <SUBSCRIPT>

213    <MODIFIED STRUCT FUNC> ::= <PREFIX> <NO ARG STRUCT FUNC> <SUBSCRIPT>

214    <STRUCTURE VAR> ::= <QUAL STRUCT> <SUBSCRIPT>

215    <ARITH VAR> ::= <PREFIX> <ARITH ID> <SUBSCRIPT>

216    <CHAR VAR> ::= <PREFIX> <CHAR ID> <SUBSCRIPT>

217    <BIT VAR> ::= <PREFIX> <BIT ID> <SUBSCRIPT>

218    <EVENT VAR> ::= <PREFIX> <EVENT ID> <SUBSCRIPT>

219    <QUAL STRUCT> ::= <STRUCTURE ID>
220                     | <QUAL STRUCT> . <STRUCTURE ID>

221    <PREFIX> ::=
222                 | <QUAL STRUCT> .

223    <SUBBIT HEAD> ::= <SUBBIT KEY> <SUBSCRIPT> (

224    <SUBBIT KEY> ::= SUBBIT

225    <SUBSCRIPT> ::= <SUB HEAD> )
226                   | <QUALIFIER>
227                   | <$> <NUMBER>
228                   | <$> <ARITH VAR>
229                   |

230    <SUB START> ::= <$> (
231                   | <$> ( @ <PREC SPEC> ,
232                   | <SUB HEAD> ;
233                   | <SUB HEAD> :
234                   | <SUB HEAD> ,
235   <SUB HEAD> ::= <SUB START>
236                  | <SUB START> <SUB>

237   <SUB> ::= <SUB EXP>
238           | *
239           | <SUB RUN HEAD> <SUB EXP>
240           | <ARITH EXP> AT <SUB EXP>

241   <SUB RUN HEAD> ::= <SUB EXP> TO

242   <SUB EXP> ::= <ARITH EXP>
243               | <# EXPRESSION>

244   <# EXPRESSION> ::= #
245                    | <# EXPRESSION> + <TERM>
246                    | <# EXPRESSION> - <TERM>

247   <=1> ::= =

248   <$> ::= $

249   <AND> ::= &
```

```
Production 193, 194, 195, 198  <variable> ::= <arith var>
                                           |  <structure var>
                                           |  <bit var>
                                           |  <char var>
```

If possible, check that the <variable> is legal in an assign context and make a cross reference table entry all via CHECK_ASSIGN_CONTEXT.

```
Production 196  <variable> ::= <event var>
```

Make it look like a <bit var> of length 1.

```
Production 197  <variable> ::= <subbit head> <variable>)
```

Check against nested SUBBITs in an assignment context. Close out the SUBBIT via END_SUBBIT_FCN. Set SUBBIT bit in VAL_P.

```
Production 199  <variable> ::= <name key> (<name var>)
```

Call CHECK_NAMING to check that the argument of NAME was legal, generate cross references and CHECK_ASSIGN_CONTEXT. It also builds the indirect and direct stack entries for <variable> by copying the information from <name var>.

```
Production 200  <name var> ::= <variable>
```

Cannot have NAME(NAME(...)) or NAME(SUBBIT(...)). Set TEMP_SYN accordingly for CHECK_NAMING.

```
Production 201  <name var> ::= <label var>
```

Only tasks or programs allowed. Set TEMP_SYN for CHECK_NAMING.

```
Production 202, 203, 204, 205  <name var> ::= <modified arith func>
                                           |  <modified bit func>
                                           |  <modified char func>
                                           |  <modified struct func>
```

Set TEMP_SYN for CHECK_NAMING.

C-3

Production 209 <name key> ::= NAME

Set various context flags.

Production 210, 211, 212, 213, 214

```
<label var>  ::= <prefix> <label> <subscript>
<modified arith func>  ::= <prefix> <no arg arith func> <subscript>
<modified bit func>  ::=  <prefix> <no arg bit func> <subscript>
<modified char func   ::= <prefix> <no arg char func> <subscript>
<modified struc func   ::= <prefix> <no arg struct func> <subscript>
```

For non-built-in functions, fall into production 216.

For built-in functions there cannot be any subscripting. Set up TEMP_SYN for CHECK_NAMING. Copy the FIXL and VAR fields from the function to the modified function. Pop the indirect stack down to the modified function.

Production 215 <structure var> ::= <qual struct> <subscript>

Jump into production 216.

Production 216, 217, 218, 219

```
<arith var>  ::= <prefix> <arith id> <subscript>
<char var>  ::= <prefix> <char id> <subscript>
<bit var>  ::= <prefix> <bit id> <subscript>
<event var>  ::= <prefix> <event id> <subscript>
```

Hl points to the indirect stack entry for the <prefix>. This will become the indirect stack entry for the <...var>.

If the <prefix> was empty , copy the symbol table pointer, STACK_PTR and VAR entries from the id. If there was a real <prefix>, append the rest of the name to the prefix and diddle the output writer.

ATTACH_SUBSCRIPT is described immediately after this production. If we have structure subscripting issue the TSUB now.

```
maj_struc                        1   TSUB   0   0
sym_pointer for structure        0   form   0   1
```

Emit the rest of the subscripting information via EMIT_SUBSCRIPT and then repair the number_of_operands field in the TSUB. Change the <... var> into a VAC pointing to the TSUB.

Pop the indirect stack.

If we have a qualified structure, issue an EXTN operator followed by one operand for each level of qualification. Fill in the VAL_P entry for the qualified structure and fill in accumulated information into the EXTN operator.

At this point $H2 = \begin{cases} \text{points to EXTN operator is issed} \\ \text{-1 otherwise} \end{cases}$

If there is a chain of subscripts hanging, issue a DSUB to take care of them; issue the subscripts via EMIT_SUBSCRIPT, fill in the proper number of arguments in the DSUB, indicate that the whole subscripted item is a VAC.

ATTACH_SUBSCRIPT	--	955700
GET_ARRAYNESS	--	871100
ATTACH_SUB_STRUCTURE	--	953100
ATTACH_SUB_ARRAY	--	949300
ATTACH_SUB_COMPONENT	--	942800
MATCH_ARRAYNESS	--	887800
SLIP_SUBSCRIPT	--	941900
AST_STACKER	--	940400
REDUCE_SUBSCRIPT	--	932600
CHECK_SUBSCRIPT	--	922700
EMIT_SUBSCRIPT	--	962300

ATTACH_SUBSCRIPT

When this routine is entered:

INX(<subscript>) = SUB_COUNT
VAL_P(<subscript>) = ARRAY_SUB_COUNT
PSEUDO_LENGTH(<subscript>) = STRUCTURE_SUB_COUNT

PSEUDO_LENGTH(<prefix>) = VAR_LENGTH(id)
PSEUDO_TYPE(<prefix>) = SYT_TYPE(<id>)
FIXL(<prefix>) = symbol table pointer of id

GET_ARRAYNESS sets up the VAR_ARRAYNESS array and fills in arrayness, copiness and NAME bits in VAL_P.

In general, there will have been some subscripting so INX(INX) is usually positive. PTR(<subscript>) points to a descriptor of the entire subscript. Stacked immediately above this descriptor on the indirect stack is one entry for each sub , number, etc. NEXT_SUB will be incremented as parts of the subscript are handled so that it always points at the current part under examination.

If there was a structure subscript terminated by a ";", check it for validity via ATTACH_SUB_STRUCTURE. If there was an array subscript terminated by a ":", check it for validity via ATTACH_SUB_ARRAY.

If the item can have component subscripting: call ATTACH_SUB_STRUCTURE and ATTACH_SUB_ARRAY to simulate "*" subscripting. Then process the component subscript via ATTACH_SUB_COMPONENT.

- Otherwise, if the item has arrayness and did not have a ":" demarked array subscript, simulate an "*" structure subscript and process the remaining subscript as an array subscript.

- Otherwise, if the item has copiness and did not have a ";" demarked structure subscript, process the remaining subscript as a structure subscript.

Finally, call MATCH_ARRAYNESS to check that the residual arrayness matches the rest of the statement's residual arrayness and return false if the item before subscripting had no copiness, true, otherwise.

ATTACH_SUB_STRUCTURE and ATTACH_SUB_ARRAY handle structure and array subscripts respectively. They check that the number of subscripts is legal and then call AST_STACKER to simulate an "*" subscript or REDUCE_SUBSCRIPT for real subscripts. If too many subscripts are specified, SLIP_SUBSCRIPTS advances NEXT_SUB over them.

REDUCE_SUBSCRIPT receives three arguments:

- MODE
- SIZE = the size of the dimension being processed.
- FLAG = indicator for level of checking required on TO and AT partitions:

  0 - normal
  1 - even zero length permitted
  2 - must be greater than one

In addition, NEXT_SUB is pointed at the subscript entry just processed. The aim of the routine is to check the validity of the subscript, generate correct forms, types, etc., for the subscript and generate HALMAT for scalar/integer and #I expression calculations.

- For an "*" subscript, just set FIX_DIM to the size of the dimension being processed.
- For a simple index, CHECK_SUBSCRIPT, FIX_DIM=1.
- For T_1 TO T_2 CHECK_SUBSCRIPT for T_1 and T_2 and make sure partition is of an acceptable size.
- For T_1 AT T_2 a simplified version of TO.

Other important effects of REDUCE_SUBSCRIPT are the setting of IND_LINK to the last subscript processed and the linking of all entries for a given subscript via their PSEUDO_LENGTH fields.

CHECK_SUBSCRIPT fills in the proper PSEUDO_FORM and PSEUDO_TYPE for an entry. If runtime arithmetic is needed for #I expression or scalar to integer conversions the HALMAT is generated here.

ATTACH_SUB_COMPONENT handles component subscripting for character and bit strings, vectors and matrices. It basically does a REDUCE_SUBSCRIPT, fills in the proper bits in VAL_P and checks for proper number of subscripts.

Production 220   <qual struct> ::= <structure id>

Build an indirect stack entry for <qual struct> and fill in FIXL and FIXV.

Production 221   <qual struct> ::= <qual struct> <structure id>

Build an indirect stack entry for the qualifier. This is right on top of the previous entry so it needs no pointer to be accessed.

Diddle the output writer.

Production 222   <prefix> ::= <empty>

Build a dummy indirect stack entry.

Production 223   <prefix> ::= <qual struct>.

Diddle the output writer. Inherit all stack entries from <qual struct>.

Production 224   <subbit head> ::= <subbit key> <subscript> (

Copy indirect stack entry from <subscript> to subbit head .

If the <subscript> was non-empty, then check that there is exactly one component subscript.

Production 225   <subbit key> ::= SUBBIT

Set up for pretty output.

Production 226   <subscript> ::= <sub head>)

The <sub head> cannot be an empty subscript. Descrement SUBSCRIPT_LEVEL.

Production 227 <subscript> ::= <qualifier>

Zero all the counts. Notice that STRUCTURE_SUB_COUNT and ARRAY_SUB_COUNT are normally initialized to -1 not zero and that tests for negative are made in several places.

Production 228 <subscript> ::= <$> <number>

Build an indirect stack entry for subscript. Fill in form and type of stack entry for number .

Production 229 joins here.

Fill in INX and VAL_P entries for <number> or <arith_var>. Initialize the subscript counters (n.b. these are all LITERALLYs). Decrement SUBSCRIPT_LEVEL.

Production 229 <subscript> ::= <$> <arith var>

Guarantee that the <arith var> is either an integer or a scalar via IORS, generate a cross reference, and join 228.

Production 230 <subscript> ::= <empty>

Set FIXL to indicate that this is a dummy and join production 249.

Production 231 <substart> ::= <$> (

Initialize counters which describe number of various kinds of subscripts.

Production 232 <substart> ::= <$>(@ <prec spec> ,

Copy the precision into PSEUDO_FORM(<substart>) and join 231.

Production 233 <sub start> ::= <sub head>;

There has to be a <sub> preceding the ";" and there must not have been a preceding ";".

Production 234 <sub start> ::= <sub head>:

There has to be a <sub> preceding the ":" and there must not have been a preceding ":".

Production 235 <sub start> ::= <sub head>,

There has to be a <sub> preceding the ",".

Production 236 <sub head> ::= <sub start>

Reset SUB_SEEN so that checks on SUB_SEEN will show false but it still indicates the whole listing.

Production 237 <sub head> ::= <sub start> <sub>

Count the <sub>.

Production 238 <sub> ::= <sub exp>

Set INX to indicate <sub exp> type <sub>.

Production 239 <sub> ::= *

Build an indirect stack entry for <sub>.

Production 240 <sub> ::= <sub run head> <sub exp>

Set INX to indicate that <sub run head> and <sub exp> are parts of TO <sub>.

Production 241 <sub> ::= <arith exp> AT <sub exp>

Check that <arith exp> is an integer or a scalar. Set INX to indicate that <arith exp> and <sub exp> are parts of an AT<sub>. Copy PTR(<sub exp>) down one space in the stack so that the two top stack elements will point at the two <sub> constituents.

<u>Production 242</u> <sub run head> ::= <sub exp> TO

Nothing.

<u>Production 243</u> <sub exp> ::= <arith exp>

Check that expression is integer or scalar.

<u>Production 244</u> <sub exp> ::= <# expression>

If <# expression> = #, generate an indirect stack entry.

<u>Production 245</u> <# expression> ::= #

Set FIXL to indicate only a #.

<u>Production 246, 247</u> <# expression> ::= <# expression> + <term>
| <# expression> - <term>

If <# expression> is just a sharp, set FIXL to indicate + or -; otherwise, call ADD_AND_SUBTRACT to add together the current non-# part of <# expression> and the <term>.

<u>Production 248</u> <=1> ::= =

Save arrayness of left side.

<u>Production 249</u> <$> ::= $

If this is a subscript on a function invocation issue:

function level	1	XXST	N	0
sym pointer for function name	0	SYT	0	1

If this is not already a subscript then SAVE_ARRAYNESS.

Increment SUBSCRIPT_LEVEL by 0 for <empty> subscript or 1 for $. Build empty indirect stack entry for the subscript.

ADD_AND_SUBTRACT	-- 851000
ARITH_LITERAL	-- 843600
LIT_RESULT_TYPE	-- 849500
MATCH_ARITH	-- 847500
MATCH_SIMPLES	-- 834100
MATRIX_COMPARE	-- 819200
VECTOR_COMPARE	-- 818500

ARITH_LITERAL sets up its two arguments for a MONITOR call and returns true if they are both literals.

LIT_RESULT_TYPE returns INT_TYPE if both of its arguments are integers and the result of the operation is integerizeable; otherwise it returns SCALAR_TYPE.

MATCH_ARITH checks that addition and subtraction are defined between its two arguments. If they are integer/scalar MATCH_SIMPLES generates any necessary integer to scalar conversion. If they are matrices or vectors, MATRIX_COMPARE and VECTOR_COMPARE check that the sizes match.

ADD_AND_SUBTRACT performs an addition (arg=0) or subtraction (arg=1) on the elements pointed to by SP and MP. If both operands are literals, the computation is done by a MONITOR(9) call. If they are not both literals, then HALMAT code is generated to do the arithmetic creating a VAC. In either case the result goes into MP and the indirect stack is popped down to there.

ASSOCIATE	-- 1095700

Check that any overpunches are consistent with the final type after subscripting. Insert proper type for the output writer.

If this is a NAME or % macro argument, then SAVE_ARRAYNESS. If all the copies were subscripted out, pop off the value just saved and, if requested, fix up the HALMAT pointed to by the argument (TAG).

Set brace and bracket flags for the output writer.

4.4.5 <expression> and <relational exp>

This section deals with productions:

4-32, 82-120, 121-135, 177, 178,
181-192, 206-208, 250-272.

Notice that productions 18-20 are grouped with production 28 immediately before 27, rather than in the obvious numerical order.

```
 4    <ARITH EXP> ::= <TERM>
 5                  | + <TERM>
 6                  | - <TERM>
 7                  | <ARITH EXP> + <TERM>
 8                  | <ARITH EXP> - <TERM>

 9    <TERM> ::= <PRODUCT>
10             | <PRODUCT> / <TERM>

11    <PRODUCT> ::= <FACTOR>
12                | <FACTOR> * <PRODUCT>
13                | <FACTOR> . <PRODUCT>
14                | <FACTOR> <PRODUCT>

15    <FACTOR> ::= <PRIMARY>
16               | <PRIMARY> <**> <FACTOR>

17    <**> ::= **

18    <PRE PRIMARY> ::= ( <ARITH EXP> )
19                    | <NUMBER>
20                    | <COMPOUND NUMBER>

21    <ARITH FUNC HEAD> ::= <ARITH FUNC>
22                        | <ARITH CONV> <SUBSCRIPT>

23    <ARITH CONV> ::= INTEGER
24                   | SCALAR
25                   | VECTOR
26                   | MATRIX

27    <PRIMARY> ::= <ARITH VAR>

28    <PRE PRIMARY> ::= <ARITH FUNC HEAD> ( <CALL LIST> )

29    <PRIMARY> ::= <MODIFIED ARITH FUNC>
30                | <ARITH INLINE DEF> <BLOCK BODY> <CLOSING> ;
31                | <PRE PRIMARY>
32                | <PRE PRIMARY> <QUALIFIER>
```

```
 82  <BIT PRIM> ::= <BIT VAR>
 83                | <LABEL VAR>
 84                | <EVENT VAR>
 85                | <BIT CONST>
 86                | ( <BIT EXP> )
 87                | <MODIFIED BIT FUNC>
 88                | <BIT INLINE DEF> <BLOCK BODY> <CLOSING> ;
 89                | <SUBBIT HEAD> <EXPRESSION> )

 90                | <BIT FUNC HEAD> ( <CALL LIST> )

 91  <BIT FUNC HEAD> ::= <BIT FUNC>
 92                     | BIT <SUB OR QUALIFIER>

 93  <BIT CAT> ::= <BIT PRIM>
 94              | <BIT CAT> <CAT> <BIT PRIM>
 95              | <NOT> <BIT PRIM>
 96              | <BIT CAT> <CAT> <NOT> <BIT PRIM>

 97  <BIT FACTOR> ::= <BIT CAT>
 98                 | <BIT FACTOR> <AND> <BIT CAT>

 99  <BIT EXP> ::= <BIT FACTOR>
100              | <BIT EXP> <OR> <BIT FACTOR>

101  <RELATIONAL OP> ::= =
102                     | <NOT> =
103                     | <
104                     | >
105                     | < =
106                     | > =
107                     | <NOT> <
108                     | <NOT> >

109  <COMPARISON> ::= <ARITH EXP> <RELATIONAL OP> <ARITH EXP>
110                 | <CHAR EXP> <RELATIONAL OP> <CHAR EXP>
111                 | <BIT CAT> <RELATIONAL OP> <BIT CAT>
112                 | <STRUCTURE EXP> <RELATIONAL OP> <STRUCTURE EXP>
113                 | <NAME EXP> <RELATIONAL OP> <NAME EXP>

114  <RELATIONAL FACTOR> ::= <REL PRIM>
115                        | <RELATIONAL FACTOR> <AND> <REL PRIM>

116  <RELATIONAL EXP> ::= <RELATIONAL FACTOR>
117                     | <RELATIONAL EXP> <OR> <RELATIONAL FACTOR>

118  <REL PRIM> ::= ( <RELATIONAL EXP> )
119               | <NOT> ( <RELATIONAL EXP> )
120               | <COMPARISON>
```

```
121   <CHAR PRIM> ::= <CHAR VAR>
122                 | <CHAR CONST>
123                 | <MODIFIED CHAR FUNC>
124                 | <CHAR INLINE DEF> <BLOCK BODY> <CLOSING> ;
125                 | <CHAR FUNC HEAD> ( <CALL LIST> )
126                 | ( <CHAR EXP> )

127   <CHAR FUNC HEAD> ::= <CHAR FUNC>
128                      | CHARACTER <SUB OR QUALIFIER>

129   <SUB OR QUALIFIER> ::= <SUBSCRIPT>
130                        | <BIT QUALIFIER>

131   <CHAR EXP> ::= <CHAR PRIM>
132                | <CHAR EXP> <CAT> <CHAR PRIM>
133                | <CHAR EXP> <CAT> <ARITH EXP>
134                | <ARITH EXP> <CAT> <ARITH EXP>
135                | <ARITH EXP> <CAT> <CHAR PRIM>



177   <CALL LIST> ::= <LIST EXP>
178                 | <CALL LIST> , <LIST EXP>



181   <EXPRESSION> ::= <ARITH EXP>
182                  | <BIT EXP>
183                  | <CHAR EXP>
184                  | <STRUCTURE EXP>
185                  | <NAME EXP>

186   <STRUCTURE EXP> ::= <STRUCTURE VAR>
187                     | <MODIFIED STRUCT FUNC>
188                     | <STRUC INLINE DEF> <BLOCK BODY> <CLOSING> ;
189                     | <STRUCT FUNC HEAD> ( <CALL LIST> )

190   <STRUCT FUNC HEAD> ::= <STRUCT FUNC>

191   <LIST EXP> ::= <EXPRESSION>
192                | <ARITH EXP> # <EXPRESSION>



206   <NAME EXP> ::= <NAME KEY> ( <NAME VAR> )
207                | NULL
208                | <NAME KEY> ( NULL )
```

```
250   <AND> ::= &
251             | AND

252   <OR> ::= |
253          | OR

254   <NOT> ::= ¬
255             | NOT

256   <CAT> ::= ||

257             | CAT

258   <QUALIFIER> ::= <$> ( @ <PREC SPEC> )

259   <BIT QUALIFIER> ::= <$> ( @ <RADIX> )

260   <RADIX> ::= HEX
261             | OCT
262             | BIN
263             | DEC

264   <BIT CONST HEAD> ::= <RADIX>
265                      | <RADIX> ( <NUMBER> )

266   <BIT CONST> ::= <BIT CONST HEAD> <CHAR STRING>
267                 | TRUE
268                 | FALSE
269                 | ON
270                 | OFF

271   <CHAR CONST> ::= <CHAR STRING>
272                  | CHAR ( <NUMBER> ) <CHAR STRING>
```

Production 4, 5 <arith exp> ::= <term>
| + <term>

Nothing.

Production 6 <arith exp> ::= -<term>

If the <term> is a constant, negate it at compile time; otherwise, generate HALMAT to do the negation.

Production 7, 8 <arith exp> ::= <arith exp> + <term>
| <arith exp> - <term>

Generate HALMAT (or perform compile time add or subtract) via ADD_AND_SUBTRACT.

Production 9 <term> ::= <product>

Nothing.

Production 10 <term> ::= <product>/<term>

If the arguments are constant, do the division at compile time.

Force divisor to be scalar. If numerator is integer, force it to scalar. Issue HALMAT to perform the division. Pop the indirect stack.

Production 11 <product> ::= <factor>

Because multiplication is associative, the compiler can perform multiplies in the order it wants to. The best order is much faster than the worst. By making productions 12, 13, and 14 right recursive, the compiler forces all multiplies to stack up. This production is reached at the point where all the multiplies must be issued and it issues them, thus leaving nothing for productions 12, 13, and 14.

Count up the number of dot products, cross products, matrices, vectors, and scalars involved in the whole product. If there are no multiplications to be done, do nothing.

Move through the stack generating multiplies for all the scalars via MULTIPLY_SYNTHESIZE.

If there are no vectors in the product then move through the stack generating multiplies for all the matrices. Multiply the final matrix product by the final product of scalars -- all done.

If there are vectors then we want to do the matrix*vector calculations first. Scan from left to right finding strings of the form:

$$\text{matrix}_1 * \text{matrix}_2 * \ldots * \text{vector}_1$$

and generating HALMAT to compute:

$$\text{vector } 2 = \text{matrix}_1 * (\text{matrix}_2 * \ldots * (\text{matrix}_n * \text{vector}))$$

followed by HALMAT to compute:

$$\text{vector } 3 = ((\text{vector}_2 * \text{matrix}_{n+1}) + \ldots * \text{matrix}_{n+m}).$$

If there are no vector -- vector multiplications, multiply the final product of vectors by the final product of scalars. Now copy all the information about the result into the indirect stack entry of the leftmost factor in the product. The only product that can be left is a single outer product so generate the HALMAT if necessary -- all done.

Move through the stack generating HALMAT to do all the cross products. If there are no dot products then join the preceding code for "all vector products done".

Move through the stack generating HALMAT to do all the dot products. Join preceding code to multiply in the final product of scalars.

Productions 12, 13, 14 <product> ::= <factor> * <product>
| <factor> • <product>
| <factor> <product>

See production 11.

Production 15 <factor> ::= <primary>

Just syntax.

Production 16   <factor> ::= <primary> ** <factor>

Generate a cross reference for primary and decrement EXPONENT_LEVEL.

For matrices, check that the exponent is not a "T" and that it is an integer constant. Then generate a HALMAT MEXP. If the exponent is a "T", generate a HALMAT MTRA.

Vectors cannot have exponents.

For integers and scalars, try doing it at compile time. If that fails then generate an IPEX for an integer to a positive integer constant power. If that fails, force <primary> to a scalar and generate an SEXP or SIEX.

Generate a cross reference for <factor>.

Production 17   <**> ::= **

Bump EXPONENT_LEVEL.

Production 21   <arith func level> ::= <arith func>

START_NORMAL_FCN.

Production 22   <arith func head> ::= <arith conv> <subscript>

Set global flags to point to <subscript> entries on indirect stack. If the subscript is null, then fill in default sizes; otherwise, use ARITH_SHAPER_SUB to compute the correct size and check the subscript for validity. Build a function stack entry. Issue HALMAT to start the shaping function reference.

```
Production 23, 24, 25, 26   arith conv   ::=  INTEGER
                                            |  SCALAR
                                            |  VECTOR
                                            |  MATRIX
```

Set up FIXL for production 22.

Production 18 <pre primary> ::= (<arith exp>)

Copy the VAR and indirect stack entries from <arith exp> to <pre primary>.

Production 19, 20 <pre primary> ::= <number>
| <compound number>

Build an indirect stack entry.

Production 28 <pre primary> ::= <arith func head> (<call list>)

END_ANY_FCN.

Production 27 <primary> ::= <arith var>

Nothing.

Production 29 <primary> ::= <modified arith func>

SETUP_NO_ARG_FCN.

Production 30 <primary> ::= <arith inline def> <block body> <closing>;

Set up and then join production 289 to handle the closing of the inline block.

Production 31 <primary> ::= <pre primary>

Just drop FIXF.

Production 32 <primary> ::= <pre primary><qualifier>

Generate code to do the precision conversion and then pop indirect stack down to the <primary>. Drop FIXF.

Production 82 <bit prim> ::= <bit var>

Generate a cross reference and set that this is not an event.

Production 83 <bit prim> ::= <label var>

Generate a cross reference. Set PSEUDO_TYPE and PSEUDO_LENGTH to bit string length 1.

Production 84 <bit prim> ::= <event var>

Same as 83.

Production 85 <bit prim> ::= <bit constant>

Same as 82 without the cross reference.

Production 86 <bit prim> ::= (<bit exp>)

Copy the indirect stack entry from <bit exp> to <bit prim>.

Production 87 <bit prim> ::= <modified bit func>

SETUP_NO_ARG_FCN. Join 82.

Production 88 <bit prim> ::= <bit inline def> <block body> <closing>;

Same as production 30.

Production 89 <bit prim> ::= <subbit head> <expression>)

END_SUBBIT_FCN. Set that was not an event.

Production 90 <bit prim> ::= <bit func head> (<call list>)

END_ANY_FCN. Set that was not an event.

Production 91   <bit func head>   ::= <bit func>

START_NORMAL_FUNCTION. If user defined function, ASSOCIATE.

Production 92   <bit func head> ::= BIT <sub or qualifier>

Set for pretty output. Copy the indirect stack entry from <sub or qualifier> to <bit func head>. Set that the type is bit. Build a function stack entry.

Production 93   <bit cat>   ::= <bit prim>

Just syntax.

Production 94   <bit cat> ::= <bit cat> <cat> <bit prim>

Set that it is not an event. Generate HALMAT to do the catenation.

Production 95   <bit cat>   ::= <not> <bit prim>

If <bit prim> is a literal, do the NOT at compile time; otherwise, generate HALMAT to do it. Drop the "2" bit in INX. Copy the indirect stack entry from <bit prim> to <bit cat>.

Production 96   <bit cat> ::= <bit cat> <cat> <not> <bit prim>

Generate HALMAT to do the NOT and then join production 94.

Production 97   <bit factor>   ::= <bit cat>

Just syntax.

Production 98   <bit factor> ::= <bit factor> AND <bit cat>

If both operands are literals, do the AND at compile time; otherwise, generate HALMAT to do it. Notice that BIT_LITERAL also puts the values of the literals in their FIXV entries.

Production 99   <bit exp>   ::= <bit factor>

Just syntax.

Production 100 <bit exp> ::= <bit exp> OR <bit factor>

Join production 98.

Production 101, 102, 103, 104, 105, 106, 107, 108

```
<relational op>  ::=  =
                   |  <not> =
                   |  <
                   |  >
                   |  <=
                   |  >=
                   |   <not> <
                   |   <not> >
```

Set up REL_OP for <comparison> productions.

Production 109 <comparison> ::= <arith exp> <relational op> <arith exp>

Match the types of the operands if possible. Issue a HALMAT comparison for the appropriate arithmetic type.

Production 110, 111

```
<comparison>  ::= <char exp> <relational op> <char exp>
                | <bit cat> <relational op> <bit cat>
```

Emit appropriate HALMAT comparison operation.

Production 112

<comparison> ::= <structure exp> <relational op> <structure exp>

STRUCTURE_COMPARE(a_1, a_2, eclass, num) does a structure walk of templates a_1 and a_2. If they are not equivalent, it issues the error message in class eclass and number num.

Emit a structure comparison HALMAT operation.

Production 113

<comparison> ::= <name exp> <relational op> <name exp>

NAME_COMPARE(a_1, a_2, eclass, num, fs) compares the names described by stack entries a_1 and a_2. If they are not NAMEs of comparable things, issue the error message in class eclass and number num. If fs then their arrayness stack entries must match; otherwise, the arrayness stack entry of a_1 must match CURRENT_ARRAYNESS. Also check that the data is not locked.

COPINESS(l,r) compares the copiness of its operands.

- identical copiness → return 0
- copies(r) = 0 → make copies(r) = copies(l) and return 2.
- copies(l) = 0 → return 4
- none of the above → return 3

NAME_ARRAYNESS(SP) sets up CURRENT_ARRAYNESS to describe the item the NAME is pointing at.

Finally, emit a name comparison HALMAT instruction.

Production 114 <relational factor> ::= <rel prim>

Just syntax.

Production 115

<relational factor> ::= <relational factor> <and> <rel prim>

Generate a HALMAT CAND instruction.

Production 116 <relational exp> ::= <relational factor>

Just syntax.

Production 117

<relational exp> ::= <relational exp> <or> <relational factor>

Issue HALMAT COR instruction.

Production 118 <rel prim> ::=(<relational exp>)

Copy indirect stack pointer to <rel prim>.

Production 119 <rel prim> ::= <not>(<relational exp>)

Issue HALMAT CNOT instruction and then copy indirect stack pointer to <rel prim>.

Production 120 <rel prim> ::= <comparison>

Relational operators other than = and ¬= are defined only for unarrayed integers, scalars, and character strings.

EMIT_ARRAYNESS.

Production 121 <char prim> ::= <char var>

Generate a cross reference.

Production 122 <char prim> ::= <char const>

Just syntax.

Production 123 <char prim> ::= <modified char func>

SETUP_NO_ARG_FCN.

Production 124

<char prim> ::= <char inline def> <block body> <closing>;

Join production 30.

Production 125 <char prim> ::= <char func head> (<call list>)

END_ANY_FCN.

Production 126 <char prim> ::= (<char exp>)

Copy indirect stack pointer to <char prim>.

Production 127 <char func head> :: <char func>

START_NORMAL_FCN.

If it is a user defined function, ASSOCIATE.

Production 128 <char func head> ::= CHARACTER <sub or qualifier>

Set up for pretty output. Reserve space on function stack.

Production 129 <sub or qualifier> ::= <subscript>

Check that subscript is not a precision modifier. There must be 0 or 1 component subscripts and nothing else.

Production 130 <sub or qualifier> ::= <bit qualifer>

Drop INX.

Production 131 <char exp> ::= <char prim>

Just syntax.

Production 132 <char exp> ::= <char exp> <cat> <char prim>

If both operands are literals, do the catenation at compile time; otherwise, issue a HALMAT CCAT instruction.

Production 133 <char exp> ::= <char exp> <cat> <arith exp>

Call ARITH_TO_CHAR to check type of <arith exp> and issue HALMAT STOC or ITOC instruction.

Join production 132.

Production 134, 135 <char exp> ::= <arith exp> <cat> <arith exp>
| <arith exp> <cat> <char exp>

See production 133.

Production 177, 178 <call list> ::= <list exp>
| <call list>, <list exp>

Call SETUP_CALL_ARG to check that the function nesting is not too deep and that the argument is legal for a function if this is a function.

For user defined procedures and functions

Cannot make these calls from inline functions.

Issue an XXAR instruction for the argument.

Arguments for procedures can be NAMEs -- drop the NAME_PSEUDO and clean up.

Use GET_FCN_PARM to get the symbol table entry describing the formal parameter.

Build a pseudo indirect stack entry at level 0 of the stack. Build a CURRENT_ARRAYNESS entry. Check that the formal and actual parameters agree.

For normal built in functions

Just count the argument.

For arithmetic shapers

Check that the argument's type is legal. Issue an SFAR instruction for the argument. Count the argument.

For string shapers

Just count the argument.

For list functions

Issue an SFAR instruction for the argument on and count it.

```
Production 181, 182, 183, 184, 185   <expression> ::= <arith exp>
                                                    | <bit exp>
                                                    | <char exp>
                                                    | <structure exp>
                                                    | <name exp>
```

Production 186 <structure exp> ::= <structure var>

Generate a cross reference.

Production 187 <structure exp> ::= <modified struct func>

SETUP_NO_ARG_FUNC.

Production 188

<structure exp> ::= <struc inline def> <block body> <closing>

Join production 30.

Production 189 <structure exp> ::= <struct func head> (<call list>)

END_ANY_FCN.

Production 190 <struct func head> ::= <struct func>

START_NORMAL_FCN.

If it is a user defined function, ASSOCIATE.

Production 191 <list exp> ::= <expression>

Drop INX for non-built-in functions.

Production 192 <list exp> ::= <arith exp> # <expression>

The function must be an arithmetic shaping function. Copy indirect stack entry from <expression> to <list exp>.

Production 206 <name exp> ::= <name key> (<name var>)

CHECK_NAMING and drop DELAY_CONTEXT_CHECK.

Production 207 <name exp> ::= NULL

Build an indirect stack entry describing the null pointer.

Production 208 <name exp> ::= <name key> (NULL)

Drop NAMING and DELAY_CONTEXT_CHECK, then join production 207.

Productions 250-257

```
<and>  ::=  &
         |  AND
 <or>  ::=  |
         |  OR
<not>  ::=  ¬
         |  NOT
<cat>  ::=  ||
         |  CAT
```

Just syntax.

Production 258 <qualifier>::= <$> (@ <prec spec>)

Set PSEUDO_FORM to 1 for SINGLE and 2 for DOUBLE.

Decrement subscript level.

Production 259 <bit qualifier> ::= <$> (@ <radix>)

If <radix> was DEC, set TEMP3 to 2.

Set up PSEUDO_FORM.

Decrement SUBSCRIPT_LEVEL.

Productions 260-263

```
<radix>  ::=  HEX
           |  OCT
           |  BIN
           |  DEC
```

Set TEMP3.

Production 264 <bit const head> ::= <radix>

Set that there was only a radix.

Production 265 <bit const head> ::= <radix> (<number>)

Point FIXL at value of number.

Production 266 <bit const> ::= <bit const head> <char string>

Convert the character string to a number in the base defined in <radix> (i.e. TEMP3). Check that all the digits and the total size of the number are legal. For non-decimal radix, repetition factors are legal and are implemented by shifting and ORing. Finally, build an indirect stack entry for the constant.

```
Production 267 - 270  <bit const>  ::=  TRUE
                                       | FALSE
                                       | ON
                                       | OFF
```

Build an indirect stack entry for the constant with the proper value.

```
Production 271  <char const>  ::= <char string>
```

Build an indirect stack entry.

```
Production 272  <char const>  ::=  CHAR(<number>) <char string>
```

Build the character string by multiple concatenations, then join 271.

START_NORMAL_FCN -- 896300
PUSH_FCN_STACK -- 841500

Build an indirect stack entry.

For built-in functions

Generate a cross-reference entry. Fill in the type, form and symbol table pointer in the indirect stack entry. Build a function stack entry via PUSH_FCN_STACK. If necessary, SAVE_ARRAYNESS and issue HALMAT for beginning of list function.

Return false.

For user defined functions

Fill in indirect stack entry. Build a function stack entry. SAVE_ARRAYNESS. Issue HALMAT to start function reference. Guarantee that update blocks do not call non-imbedded functions. Setup FIXL and FIXV properly for structure valued functions. Generate a cross reference.

Return true.

SETUP_NO_ARG_FCN	-- 891000
SET_BI_XREF	-- 551300
UPDATE_BLOCK_CHECK	-- 842800
STRUCTURE_FCN	-- 890200

For built-in functions

Build a cross reference entry via SET_BI_XREF.

If this is an initial/constant context and the function has special processing in that case, do the special processing; otherwise, generate HALMAT for a built-in function call.

For user defined functions

Check that the function is not access protected. Use UPDATE_BLOCK_CHECK to check that update blocks do not call non-imbedded functions. Use STRUCTURE_FCN to convert FIXL and FIXV of the function to the structure form if the value of the function is a structure. Generate HALMAT to do the function call:

function nest	1	FCAL	0	0
sym pointer for fnc	0	0	0	1
function nest	0	XXND	0	0

Generate a cross reference.

For all functions

After generating the call, if there was a precision modifier specified in the argument to SETUP_NO_ARG_FCN, generate the HALMAT to do the conversion.

END_ANY_FCN -- 964900

For procedure and user defined functions

Generate a HALMAT PCAL or FCAL. Then end it off with an XXND.

For normal built-in functions

Check that the proper number of arguments were encountered and that the types match. Then branch depending on the type of the first argument.

- BIT — Set length of result string.
- CHARACTER — Generate HALMAT to convert operands to proper types if necessary.
- MATRIX — Check and set up dimensions of argument and result.
- VECTOR — Set up dimension of result.
- SCALAR — Attempt to perform compile time evaluation via BI_COMPILE_TIME. Generate integer to scalar conversions if necessary.
- INTEGER — Same as scalar case except scalar to integer conversion are performed.
- INTEGER or SCALAR — Make type of arguments match via MATCH_SIMPLES. Then set type of returned value to type of arguments if it was originally IORS.

After handling the individual cases, generate the HALMAT call of the function.

```
                                   1
            built-in function #    2     BFNC    0    0
                                   3

                                 type   form
(1,2,or     pointer to arg        of     of      0    1
3 args)                          arg    arg
```

For arithmetic shaping functions

For integers and scalars -- restore arrayness of argument. If argument is simple, generate HALMAT shaping function call targeting to scalar or integer result; otherwise, generate an MSHP HALMAT instruction, taking the arguments from LOC_P (ARG# + 1), ...

For vector and matrix shaping functions issue an MSHP HALMAT instruction.

For string shaping functions

Check that the call is legal. Generate a HALMAT shaping function call targeted to either bit or character string. Issue one or two HALMAT operands for each subscript (AT and TO require two operands). Go back and fill in the proper number of arguments in the operator.

For list functions

Check that the type of the argument is okay and that there is only one argument (an array). Generate a HALMAT LFNC to call the function followed by an SFND to end the function invocation. Finally, RESET_ARRAYNESS.

4.4.6 <statement>

This section deals with productions: 33-81, 136-176, 179, 180, 273-288, 429-449.

```
33  <OTHER STATEMENT>  ::= <ON PHRASE> <STATEMENT>
34                        | <IF STATEMENT>
35                        | <LABEL DEFINITION > <OTHER STATEMENT >

36  <STATEMENT>  ::=< BASIC STATEMENT >
37                 | < OTHER STATEMENT >

38  < ANY STATEMENT > ::=< STATEMENT >
39                      | < BLOCK DEFINITION >

40  < BASIC STATEMENT > ::=< LABEL DEFINITION ><BASIC STATEMENT >
41                        | < ASSIGNMENT > ;
42                        | EXIT ;
43                        | EXIT <LABEL > ;
44                        | REPEAT ;
45                        | REPEAT <LABEL > ;
46                        | GO TO < LABEL> ;
47                        | ;
48                        | <CALL KEY > ;
49                        | <CALL KEY > (  < CALL LIST > )  ;
50                        | <CALL KEY >  ASSIGN   (< CALL ASSIGN LIST > ) ;
51                        | <CALL KEY > (<CALL LIST>) <ASSIGN > (<CALL ASSIGN LIST > ) ;
52                        | RETURN ;
53                        | RETURN < EXPRESSION >  ;
54                        | <DO GROUP HEAD> <ENDING > ;
55                        | <READ KEY>  ;
56                        | <READ PHRASE>  ;
57                        | <WRITE KEY>  ;
58                        | <WRITE PHRASE>  ;
59                        | <FILE EXP> = <EXPRESSION>  ;
60                        | <VARIABLE> = <FILE EXP>  ;
61                        | <WAIT KEY> FOR·DEPENDENT ;
62                        | <WAIT KEY><ARITH EXP>  ;
63                        | <WAIT KEY> UNTIL <ARITH EXP>  ;
64                        | <WAIT KEY> FOR <BIT EXP>  ;
65                        | <TERMINATOR>  ;
66                        | <TERMINATOR> <TERMINATE LIST> ;
67                        | UPDATE PRIORITY TO <ARITH EXP>  ;
68                        | UPDATE PRIORITY <LABEL VAR> TO <ARITH EXP>  ;
69                        | <SCHEDULE PHRASE>  ;
70                        | <SCHEDULE PHRASE><SCHEDULE CONTROL>  ;
71                        | <SIGNAL CLAUSE>  ;
72                        | SEND ERROR <SUBSCRIPT>  ;
73                        | <ON CLAUSE>  ;
74                        | <ON CLAUSE> AND <SIGNAL CLAUSE>  ;
75                        | OFF ERROR  <SUBSCRIPT>  ;
76                        | <% MACRO NAME>  ;
77                        | <% MACRO HEAD><% MACRO ARG> ) ;
```

```
 78  <% MACRO HEAD> ::= <% MACRO NAME> (
 79                   | <% MACRO HEAD> <% MACRO ARG> ,

 80  <% MACRO ARG>   ::= <NAME VAR >
                      | <CONSTANT>


136  <ASSIGNMENT> ::= <VARIABLE> <=1> <EXPRESSION>
137                 | <VARIABLE> ,<ASSIGNMENT>

138  <IF STATEMENT> ::= <IF CLAUSE> <STATEMENT>
139                   | <TRUE PART> <STATEMENT>

140  < TRUE PART> ::= <IF CLAUSE> <BASIC STATEMENT> ELSE

141  <IF CLAUSE> ::= <IF> <RELATIONAL EXP> THEN
142                | <IF> <BIT EXP> THEN

143  <IF> ::=  IF

144  <DO GROUP HEAD> ::=  DO;
145                     | DO <FOR LIST>  ;
146                     | DO <FOR LIST> <WHILE CLAUSE>  ;
147                     | DO <WHILE CLAUSE>  ;
148                     | DO CASE <ARITH EXP>  ;
149                     | <CASE ELSE> <STATEMENT>
150                     | <DO GROUP HEAD> <ANY STATEMENT>
151                     | <DO GROUP HEAD> <TEMPORARY STMT>

152  <CASE ELSE>  ::=  DO CASE <ARITH EXP>  ;  ELSE

153  <WHILE KEY>  ::=  WHILE
154                 |  UNTIL

155  <WHILE CLAUSE> ::= <WHILE KEY> <BIT EXP>
156                   | <WHILE KEY> <RELATIONAL EXP>

157  <FOR LIST> ::= <FOR KEY> <ARITH EXP> <ITERATION CONTROL>
158               | <FOR KEY> <ITERATION BODY>

159  <ITERATION BODY> ::= <ARITH EXP>
160                       <ITERATION BODY> , <ARITH EXP>

161  <ITERATION CONTROL> ::=  TO <ARITH EXP>
162                        |  TO <ARITH EXP> BY <ARITH EXP>

163  <FOR KEY>   ::=  FOR <ARITH VAR> =
164                |  FOR TEMPORARY <IDENTIFIER> =

165  <ENDING> ::=  END
166             |  END <LABEL>
167             |  <LABEL DEFINITION> <ENDING>

168  <ON PHRASE> ::=  ON ERROR <SUBSCRIPT>
```

```
169  <ON CLAUSE>  ::=  ON ERROR <SUBSCRIPT> SYSTEM
170                  |  ON ERROR <SUBSCRIPT> IGNORE

171  <SIGNAL CLAUSE> ::=  SET <EVENT VAR>
172                    |  RESET <EVENT VAR>
173                    |  SIGNAL <EVENT VAR>

174  <FILE EXP> ::= <FILE HEAD> , <ARITH EXP> )

175  <FILE HEAD> ::=  FILE ( <NUMBER>

176  <CALL KEY> ::=  CALL <LABEL VAR>

179  <CALL ASSIGN LIST> ::= <VARIABLE>
                         | <CALL ASSIGN LIST> , <VARIABLE>


273  <IO CONTROL> ::=  SKIP ( <ARITH EXP> )
274                 |  TAB ( <ARITH EXP> )
275                 |  COLUMN ( <ARITH EXP> )
276                 |  LINE ( <ARITH EXP> )
277                 |  PAGE ( <ARITH EXP> )

278  <READ PHRASE> ::= <READ KEY> <READ ARG>
279                  | <READ PHRASE> , <READ ARG>

280  <WRITE PHRASE> ::= <WRITE KEY> <WRITE ARG>
281                   | <WRITE PHRASE> , <WRITE ARG>

282  <READ ARG> ::= <VARIABLE>
283              | <IO CONTROL>

284  <WRITE ARG> ::= <EXPRESSION>
285               | <IO CONTROL>

286  <READ KEY> ::=  READ ( <NUMBER> )
287             |  READALL ( <NUMBER> )

288  <WRITE KEY> ::=  WRITE ( <NUMBER> )
```

```
429  <TERMINATOR>  ::=  TERMINATE
430                   |  CANCEL

431  <TERMINATE LIST> ::= <LABEL VAR>
432                     | <TERMINATE LIST> , <LABEL VAR>

433  <WAIT KEY> ::=  WAIT

434  <SCHEDULE HEAD> ::=  SCHEDULE <LABEL VAR>
435                     | <SCHEDULE HEAD> AT <ARITH EXP>
436                     | <SCHEDULE HEAD> IN <ARITH EXP>
437                     | <SCHEDULE HEAD> ON <BIT EXP>

438  <SCHEDULE PHRASE> ::= <SCHEDULE HEAD>
439                      | <SCHEDULE HEAD> PRIORITY ( <ARITH EXP> )
440                      | <SCHEDULE PHRASE> DEPENDENT

441  <SCHEDULE CONTROL> ::= <STOPPING>
442                       | <TIMING>
443                       | <TIMING> <STOPPING>

444  <TIMING> ::= <REPEAT> EVERY <ARITH EXP>
445             | <REPEAT> AFTER <ARITH EXP>
446             | <REPEAT>

447  <REPEAT> ::=  , REPEAT

448  <STOPPING> ::= <WHILE KEY> <ARITH EXP>
449               | <WHILE KEY> <BIT EXP>
```

Production 33 <other statement> ::= <on phrase><statement>

Define an internal label to jump to after executing the ON statement. This is necessary because the <statement> code is generated in line and must be jumped over. Check that there have been no branches to <statement> via UNBRANCHABLE. Set that no labels have been processed yet.

Production 34 <other statement> ::= <if statement>

Set that no labels yet processed.

Production 35

<other statement> ::= <label definition> <other statement>

Link in the label in the SYT_PTR chain for this statement. SET_LABEL_TYPE.

Production 36 <statement> ::= <basic statement>

There should be no transposes hanging. Print the statement. EMIT_SMRK.

Production 37 <statement> ::= <other statement>

Just syntax.

```
Production 38, 39  <any statement> ::= <statement>
                                     | <block definition>
```

Reset PTR.

Production 40

<basic statement> ::= <label definition> <basic statement>

See production 35.

Production 41 <basic statement> ::= <assignment>

Pop all old entries off indirect stack. If the assignment involved a NAME operation call NAME_ARRAYNESS. Fill the number of left sides into the HALMAT assignment operator. EMIT_ARRAYNESS. Set that no labels have been processed yet.

```
Production 42, 43   <basic statement> ::=  EXIT;
                                         |  EXIT <label> ;
```

Search through enclosing DO nests until LABEL_MATCH detects a DO with a label matching <label> (a null <label> matches everything). Issue a HALMAT BRA to the statement immediately after the end of the appropriate DO group. Set that no labels have been processed yet.

```
Production 44, 45   <basic statement> ::=  REPEAT;
                                         |  REPEAT <label>;
```

The same as 42, 43, except the BRA targets to the test on the loop instead of the outside of the loop.

```
Production 46   <basic statement> ::=  GO TO <label>;
```

Check that the <label> is a legal target from the current DO nest position. Generate a HALMAT BRA to <label>. Set that no labels have been processed yet.

```
Production 47   <basic statement> ::= ;
```

Set that no labels have been processed yet.

Production 48-51

```
<basic statement> ::= <call key>;
                    | <call key> (<call list>);
                    | <call key> <assign> (<call assign list>);
                    | <call key>(<call list>)<assign>(<call assign list>);
```

END_ANY_FCN.

```
Production 52   <basic statement> ::=  RETURN;
```

Check that the current block is compatible with a RETURN containing no <expression>. Generate a HALMAT RTRN. Set that no labels have been processed yet.

```
Production 53   <basic statement> ::=  RETURN <expression>;
```

Drop any residual effects from a NAME pseudo via KILL_NAME and generate an error message if there were any. Same for arrayness. Check that this is a function block.

Check that the <expression> is compatible with the type of the function and generate any necessary conversions. Generate a HALMAT RTRN instruction with <expression> as operand.

Set that no labels have been processed.

Production 54 <basic statement> ::= <do group head> <ending>;

For a DO CASE, fill in the tag on the last CLBL operation to indicate it is the last.

Issue the appropriate ending HALMAT (i.e. ESMP, EFOR, ECAS, ETST).

Pass over the list of labels: for each label defined in the group being closed, remove the label from the list and make its DO level an impossible value so that no other references can target it.

DISCONNECT all of the group's temporaries from the hash table.

Decrement DO_LEVEL and set that no labels have been processed yet.

```
Production 55-58   <basic statement>::= <read key>;
                                      | <read phrase>;
                                      | <write key>;
                                      | <write phrase>;
```

Issue a HALMAT I/O instruction.

Issue a HALMAT XXND to terminate the I/O (which looks like a subroutine call) reference.

Set that no labels have been processed yet.

Production 59 <basic statement> ::= <file exp> = <expression>;

Issue a FILE instruction and fill in the specific information about <expression> in argument 2.

EMIT_ARRAYNESS.

Set that no labels have been processed yet.

Production 60 <basic statement> ::= <variable> = <file exp>;

Issue a FILE instruction and fill in the specific information about <variable> into argument 2.

Check <variable> for legality.

Set that no labels have been processed yet.

Production 61 <basic statement> ::= <wait key> FOR DEPENDENT;

Issue a WAIT instruction.

Check that the context is valid for a real time statement. Set that no labels have been processed yet.

Production 62-64 <basic statement> ::= <wait key> <arith exp>;
| <wait key> UNTIL <arith exp>;
| <wait key> FOR <bit exp>;

Check that the <arith exp> or <bit exp> is valid. Issue a WAIT instruction. Join production 61.

Production 65 <basic statement> ::= <terminator>;

Issue the HALMAT instruction built by the <terminator> productions and join production 61.

Production 66

<basic statement> ::= <terminator> <terminate list>;

Issue the HALMAT instruction built by the <terminator> productions -- EXT_P is the length of <terminate list>. Issue one operand for each program/task on the list. Join production 61.

Production 67, 68

<basic statement> ::= UPDATE PRIORITY TO <arith exp>;
| UPDATE PRIORITY <label var> TO <arith exp>;

Check that the <label var> is a program or task and that the <arith exp> is an unarrayed integer or scalar. Issue a HALMAT PRIO instruction. Join production 61.

Production 69, 70

<basic statement> ::= <schedule phrase>;
| <schedule phrase> <schedule control>;

Issue a HALMAT SCHD instruction. Issue an operand for each of the optional clauses that were specified. Join production 61.

Production 71 <basic statement> ::= <signal clause>;

Issue a SGNL instruction. Set that no labels have been processed yet.

Production 72 <basic statement> ::= SEND ERROR <subscript>;

ERROR_SUB checks that the <subscript>is a legal error specification for:

```
SEND ERROR -- arg = 2
ON ERROR   -- arg = 1
OFF ERROR  -- arg = 0
```

sets up FIXV (<subscript>) for use as an operand in the HALMAT instruction and adds the error specification to the EXT_ARRAY list if it is a new one. The internal routine ERROR_SS_FIX examines the individual components of the subscript and returns their values.

Emit an ERSE instruction. Make an entry for the block summary. Join production 61.

Production 73, 74

```
<basic statement> ::= <on clause>;
                    | <on clause> AND <signal clause>;
```

Issue ERON instruction (see ERROR_SUB) and go set that no labels have been processed yet.

Production 75 <basic statement> ::= OFF ERROR <subscript>;

Use ERROR_SUB to check the <subscript> for legality and to set up FIXV. Issue an ERON instruction. Go set that no labels have been processed yet.

Production 76 <basic statement> ::= <% macro name>;

Issue PMHD and PMIN instructions. Check that the % macro does not expect arguments. Set that no labels have been processed yet.

Production 77

<basic statement> ::= <% macro head> <% macro arg>) ;

Check that the correct number of arguments have been seen. Issue a PMAR for the last argument. Restore normal checking of variables. Issue a PMIN instruction to close the % macro invocation. Restore lock group checking. Set that no labels have been processed.

Production 78 <% macro head> ::= <% macro name> (

Issue a PMHD instruction. DELAY_CONTEXT_CHECK. If this is %COPY, inhibit lock group checking.

Production 79

<% macro head> ::= <% macro head> <% macro arg>,

Issue a PMAR instruction for the argument.

Production 80 <% macro arg> ::= <name var>

Check that the <name var> meets the specification for the macro's arguments as listed in PCARGTYPE and PCARGBITS.

Production 81 <% macro arg> ::= <constant>

Similar to production 80 but simpler.

Production 136 <assignment> ::= <variable> <=1> <expression>

Initialize count of operands of HALMAT assignment operator to 2. Issue a NASN or XASN instruction to perform the assignment and check that the left and right sides are compatible for an assignment. Copy the description of <expression> into <assignment>.

Production 137 <assignment> ::= <variable> , <assignment>

Issue another operand for the assignment operator issued in production 136. Add 1 to the count of operands. Check that the left and right sides are compatible for assignment and copy description of <expression> to <assignment>.

Production 138, 139 <if statement> ::= <if clause> <statement>
| <true part> <statement>

Do not allow branching to <statement>. Issue an LBL instruction to define the flow number generated to allow branching around the <statement>.

Production 140 <true part> ::= <if clause> <basic statement> ELSE

Do not allow branching to the <basic statement>. Drop any implicit transposes. List everything up to but not including ELSE. EMIT_SMRK. SRN_UPDATE. List the ELSE. Issue a BRA instruction so that the <basic statement> code does not fall into the ELSE statement code. Issue an LBL instruction to define the flow number for the false branch on the IF, and save the flow number in FIXV for production 139.

Production 141 <if clause> ::= <if> <relational exp> THEN

Issue a branch to the false part and save the flow number for definition by production 140 or 138. List the statement. EMIT_SMRK.

Production 142 <if clause> ::= <if> <bit exp> THEN

Issue a BTRU to transform the <bit exp> to a condition. Check that the <bit exp> is one bit long. EMIT_ARRAYNESS. Join production 141.

Production 143 <if> ::= IF

Issue an IFHD instruction to start things rolling.

Production 144 <do group head> ::= DO;

Issue a DSMP and EMIT_PUSH_DO.

Check that there are no implicit transposes hanging. List the statement. If there was a TEMPORARY, issue a TDCL to declare it. EMIT_SMRK.

Production 145 <do group head> ::= DO <for list>;

Fill into the DFOR a tag indicating whether it is a discrete DO, an implicit 1 increment DO, or an explicit increment DO.

Join 144.

Production 146 <do group head> ::= DO <for list> <while clause>;

Fix the DFOR as in 145, including also a high order 1 bit if there is an UNTIL clause. Issue a CFOR to end the conditional.

Join 144.

Production 147

Issue a CTST to close the DTST.

Join 146.

Production 148, 152 <do group head> ::= DO CASE <arith exp>;

Check that <arith exp> is an unarrayed integer or scalar.

Emit a DCAS (n.b. FIXL indicates whether or not there is an ELSE). EMIT_PUSH_DO. Emit the second operand describing the <arith exp>.

Check that there are no hanging transposes.

If there is an ELSE, print the statement without the ELSE. EMIT_SMRK, SRN_UPDATE. Print the ELSE.

If there is no ELSE, print the statement, EMIT_SRMK. Join production 149.

Production 149 <do group head> ::= <case else> <statement>

Check that there are no branches to <statement>.

Initialize that no cases have yet been processed in this nested case. Join production 150.

Production 150

<do group head> ::= <do group head> <any statement>

If the DO group is a DO CASE and <any statement> is not a real <block definition> then:

- Set up for pretty output of case number.
- Issue a HALMAT CLBL instruction which points to the end of the DO CASE and defines the location of the case.
- Point FIXV at the last CLBL operator

Production 151 <do group head> ::= <do group head> <temporary stmt>

Check that the <temporary stmt> is at the beginning of the DO group and that the group is not a DO CASE.

Output the statement. EMIT_SMRK.

Production 152 <case else> ::= DO CASE <arith exp>; ELSE

See production 148.

Production 153, 154 <while key> ::= WHILE
| UNTIL

If this is not a DO FOR then issue a HALMAT DTST and EMIT_PUSH_DO.

Set FIXL:

0 for WHILE
1 for UNTIL

Production 155 <while clause> ::= <while key> <bit exp>

Check that <bit exp> is an unarrayed boolean. Emit a HALMAT BTRU to transform it to a relation.

Copy WHILE/UNTIL indicator to INX and copy indirect stack pointer.

Production 156 <while clause> ::= <while key> <relational exp>

Join production 155.

Production 157

<for list> ::= <for key> <arith exp> <iteration control>

Check that <arith exp> is an unarrayed integer or scalar. Issue a DFOR. EMIT_PUSH_DO. Emit two or one operands depending on whether or not there is a BY clause. Point FIXV at the DFOR. Set PTR to the number of operands.

Production 158 <for list> ::= <for key> <iteration body>

Fill in tag field in last AFOR to indicate that it is the end of the list.

Set PTR to indicate a discrete DO.

Production 159 <iteration body> ::= <arith exp>

This is the beginning of the list of values so issue the HALMAT DFOR operator here.

Call EMIT_PUSH_DO to build a DO stack entry, reserve enough flow numbers for the entire DO group processing and emit the first operand of the DFOR which is the flow number of the instruction immediately following the end of the DO group.

Issue a HALMAT operand for the variable in the <for key>.

Set FIXV of <for key> to point to the DFOR.

Fall into production 160 to finish processing <arith exp>.

Production 160 <iteration body> ::= <iteration body>, <arith exp>

Check that the <arith exp> is an unarrayed integer or scalar. Issue a HALMAT AFOR instruction for the <arith exp>. Reserve a flow number just in case.

Set FIXV of <iteration> body to point to the last AFOR issued.

Production 161, 162

```
    <iteration control> ::=  TO <arith exp>
                          |  TO <arith exp> BY <arith exp>
```

Check that <arith exp> is an unarrayed integer or scalar. Set TEMP2 to 2 if BY is present; otherwise to 1.

Production 163 <for key> ::= FOR <arith var> =

Check legality of assignment.

Check that <arith var> is an unarrayed integer or scalar via UNARRAYED_SIMPLE. Drop <arith var>'s FIXL entry.

Production 164 <for key> ::= FOR TEMPORARY <identifier> =

Build an indirect stack entry to describe <identifier>.

Production 165 <ending> ::= END

Just syntax.

Production 166 <ending> ::= END <label>

Check that the <label> matches the <label definition> on the innermost DO.

Production 167 <ending> ::= <label definition> <ending>

SET_LABEL_TYPE.

Production 168 <on phrase> ::= ON ERROR <subscript>

Check the <subscript> for validity and set up FIXV. Issue an ERON instruction and save the "branch around" flow number in FIXL. List the statement. EMIT_SMRK.

```
Production 169, 170 <on clause> ::=  ON ERROR <subscript> SYSTEM
                                  |  ON ERROR <subscript> IGNORE
```

Save action in FIXL. Check the <subscript> and set up FIXV.

```
Production 171-173   <signal clause> ::=  SET <event var>
                                        | RESET <event var>
                                        | SIGNAL <event var>
```

Check that this is not an inline and that the event is latched (except for SIGNLA). Check that there is not any arrayness. Save the action in INX.

Production 174 <file exp> ::= <file head> , <arith exp>

Check that the device number is small enough. Check that <arith exp> is an unarrayed scalar or integer. RESET_ARRAYNESS.

Production 175 <file head> ::= FILE (<number>

Save device number = <number>+2.

SAVE_ARRAYNESS.

Production 176 <call key> ::= CALL <label var>

Trace back through the IND_CALL_LAB chain to locate the symbol table entry for the procedure and check that it is a procedure and not access protected.

Initialize argument count to 0.

Production 179, 180

```
<call assign list> ::= <variable>
                     | <call assign list> , <variable>
```

Count the argument. Issue an XXAR instruction to specify the argument. Drop any arrayness. Check that the argument is legal for an assign parameter.

```
Production 273 - 277   <io control> ::=  SKIP (<arith exp>)
                                       | TAB (<arith exp>)
                                       | COLUMN (<arith exp>)
                                       | LINE (<arith exp>)
                                       | PAGE (<arith exp>)
```

Save the control function in TEMP. Check that the <arith exp> is an unarrayed integer or scalar.

Production 278, 279

```
<read phrase> ::= <read key> <read arg>
                | <read phrase> , <read arg>
```

```
TEMP =   0 - <expression> or <variable>
         1 -  TAB
         2 -  COLUMN
         3 -  SKIP
         4 -  LINE
         5 -  PAGE
```

If this is a READ, check that the argument is legal. Otherwise, call READ_ALL_TYPE to check whether the argument contains any non-character string variables.

Production 280, 281

```
<write phrase> ::= <write key> <write arg>
                 | <write phrase>, <write arg>
```

Just syntax.

Production 282-285

```
<read arg> ::= <variable>
             | <io control>
<write arg> ::= <expression>
              | <io control>
```

Check that the item is legal for I/O and that this is not an inline function. Issue an XXAR instruction for this operand of the I/O subroutine call. If it is a structure, there cannot be any NAMEs in the structure. EMIT_ARRAYNESS.

Productions 286-288

```
 <read key> ::=  READ (<number>)
              |  READALL (<number>)
<write key> ::=  WRITE (<number>)
```

```
TEMP =  0 - READ
        1 - READALL
        2 - WRITE
```

Issue an XXST instruction to start the I/O reference. Build an indirect stack entry for <read key> or <write key> describing the device. Check that the device is legal for the I/O requested. Save TEMP in INX.

```
Production 429, 430  <terminator> ::=  TERMINATE
                                     |  CANCEL
```

Incorporate type of terminator in FIXL, FIXV.

Production 431 <terminate list> ::= <label var>

Set up to count the number of <label var>s in EXT_P. Join production 432.

Production 432 <terminate list> ::= <terminate list>, <label var>

Count the <label var>. Build a cross reference. Check that the <label var> is either a program or a task via PROCESS_CHECK.

Production 433 <wait key> ::= WAIT

Initialize REFER_LOC.

Production 434 <schedule head> ::= SCHEDULE <label var>

Check that <label var> is a program or task. Initialize REFER_LOC.

Production 435-437

```
<schedule head> ::= <schedule head> AT <arith exp>
                  | <schedule head> IN <arith exp>
                  | <schedule head> ON <bit exp>
```

Check that the <arith exp> or <bit exp> is legal. Check that only one of the three forms was specified. Set INX(<label var>) to indicate which of the three forms.

Production 438 <schedule phrase> ::= <schedule head>

There must be a priority specified.

Production 439, 440

```
<schedule phrase> ::= <schedule head> PRIORITY  (<arith exp>)
                    | <schedule phrase> DEPENDENT
```

Set bits in INX(<label var>).

```
Productions 441-443  <schedule control> ::= <stopping>
                                          | <timing>
                                          | <timing> <stopping>
```

Syntax.

```
Productions 444-446  <timing> ::= <repeat> EVERY <arith exp>
                                | <repeat> AFTER <arith exp>
                                | <repeat>
```

<arith exp> must be an unarrayed integer or scalar. Set the appropriate bit in INX.

```
Production 447  <repeat> ::= , REPEAT
```

Syntax.

```
Production 448  <stopping> ::= <while key> <arith key>
```

Check that this is UNTIL situation and <arith exp> is an unarrayed integer or scalar. Set bit in INX.

```
Production 449  <stopping> ::= <while key> <bit exp>
```

Check that the <bit exp> is legal via CHECK_EVENT_EXP. Set bit in INX.

4.4.7 <compilation>

This section deals with productions 1-3, 289-292, and 426-428.

```
1     <COMPILATION>  ::= <COMPILE LIST> _|_

2     <COMPILE LIST> ::= <BLOCK DEFINITION>
3                      | <COMPILE LIST> <BLOCK DEFINITION>


289   <BLOCK DEFINITION> ::= <BLOCK STMT> <BLOCK BODY> <CLOSING> ;

290   <BLOCK BODY>::=
291                 |<DECLARE GROUP>
292                 |<BLOCK BODY> <ANY STATEMENT>


426   <CLOSING> ::=  CLOSE
427               |  CLOSE <LABEL>
428               |  <LABEL DEFINITION> <CLOSING>
```

Production 1 <compilation> ::= <compile list> _|_

Check that the parse stack is empty and that this is a compilation unit. Issue an XREC instruction and flush the HALMAT buffer. Flush out the LITFILE. Set COMPILING.

Production 2, 3

```
<compile list> ::= <block definition>
                 | <compile list> <block definition>
```

Just syntax.

Production 289

```
<block definition> ::= <block stmt> <block body> <closing> ;
```

TEMP = ICLS for inline,
CLOSE for normal.

TEMP2 = INLINE_LEVEL for inline function,
0 otherwise.

Issue the ICLS or CLOSE instruction.

Make a pass over all the symbol table entries for this scope, DISCONNECTing them along the way.

- functions should have been defined
- statement labels should have been defined
- in the outermost scope, procedures and tasks should have been defined
- in embedded scopes, block summary information should be supplied for undefined procedures and tasks
- if a procedure call referencing an IND_CALL_LAB can definitely be associated with a procedure definition, add the cross reference data from the IND_CALL_LAB to the definition entry using TIE_XREF

If the <closing> specified a name, check that it matches the name of this scope.

If it is an inline function, save the inline counters and restore the regular ones.

If it is not an inline BLOCK_SUMMARY prints the block summary information for the scope being closed.

Count the unique errors handled by the block, encode the information in SYT_ARRAY and discard the now useless EXT_ARRAY entries.

Productions 290, 291 <block body> ::=
| <declare group>

Issue an EDCL indicating whether or not there was a <declare group>.

For functions and procedures, check that all parameters have been declared.

Set that no statements have been processed yet.

Production 292 <block body> ::= <block body> <any statement>

Set that a statement has been found.

Production 426-428 <closing> ::= CLOSE
| CLOSE <label>
| <label definition> <closing>

If there is a <label definition> SET_LABEL_TYPE. If there is a <label> save it in VAR(<closing>) to check it in production 289.

4.4.8 HALMAT and Initialization Routines

HALMAT_POP -- 804000
HALMAT_FIX_POPTAG -- 808000
HALMAT_FIX_PIP# -- 807200

CALL HALMAT_POP(OP, n, C, tag) creates

CURRENT_ATOM =

tag	n	OP	C	0
8	8	12	3	1

This is inserted in the HALMAT block and LAST_POP# points to it.

HALMAT_FIX_POPTAG resets tag field.

HALMAT_FIX_PIP# resets field n.

HALMAT_PIP -- 805000
HALMAT_FIX_PIPTAGS -- 808800

CALL HALMAT_PIP(A, B, C, D) creates

CURRENT_ATOM =

A	C	B	D	1
16	8	4	3	1

and enters it into the current HALMAT block.

HALMAT_FIX_PIPTAGS resets field C with argument 1, and field D with argument 2.

HALMAT_TUPLE -- 805900

CALL HALMAT_TUPLE (op, b, oprnd1, oprnd2, tag, rnd1t1, rnd1t2, rnd2t1, rnd2t2)

tag	0 1 2	op	b	0	
sym pointer 1	rnd1t1	form 1	rnd1t2	1	← if oprnd1≠0
sym pointer 2	rnd2t1	form 2	rnd2t2	1	← if oprnd2≠1

HALMAT	--	801100
HALMAT_BACKUP	--	803400
HALMAT_BLAB	--	790000
HALMAT_RELOCATE	--	794100

HALMAT calls HALMAT_OUT to output the current block if necessary and then puts CURRENT_ATOM into the block. HALMAT_BLAB prints a HALMAT instruction. HALMAT_RELOCATE moves down some HALMAT code when the previous code has been forced out leaving an empty space. HALMAT_BACKUP resets the pointer, thereby erasing some HALMAT.

INITIALIZATION -- 1055200

Pick up the options specified in the JCL invocation of the compiler. Print the heading using the TITLE if supplied; otherwise, the default. Print the parameter field from the JCL. Print the type 1 and type 2 options and store their values in more accessible places.

Allocate space for the based variables other than the symbol table used in Phase I via STORAGE_MGT.

Define all the pointers into the DW area.

Allocate space for the common and then non-common symbol table arrays.

Define the card type characters using the defaults and the CARDTYPE parameter.

Read a card, determine the style of input and if the first card could not follow a comment, skip cards until a reasonable one is found.

Initialize the scanner with calls to STREAM and SCAN.

Initialize the parser to the initial state and build the VOCAB_INDEX array for it.

4.5 Global Names of Phase I

4.5.1 Variables

#PRODUCE_NAME	See Parser.
ACCESS_FLAG	See symbol table -- SYT_FLAGS.
ACCESS_FOUND	See STREAM.
ADD_AND_SUBTRACT	Procedure.
ADDR_FIXED_LIMIT	See SCAN.
ADDR_FIXER	See SCAN.
ADDR_PRESENT	On if ADDRS option requested in JCL.
ADDR_ROUNDER	See DW.
ADDR_VALUE	See SCAN.
ALDENSE_FLAGS	See symbol table -- SYT_FLAGS.
ALIGNED_FLAG	
ALMOST_DISASTER	Label.
ANY_TYPE	See symbol table -- SYT_TYPE.
APPLY1	See Parser.
APPLY2	See Parser.
ARITH_FUNC_TOKEN	See TOKEN.
ARITH_LITERAL	Procedure.
ARITH_SHAPER_SUB	Procedure.
ARITH_TO_CHAR	Procedure.
ARITH_TOKEN	See TOKEN.
ARRAY_DIM_LIM	The maximum size of an array dimension.
ARRAY_FLAG	See symbol table -- SYT_FLAGS.
ARRAY_SUB_COUNT	Section 4.4.
ARRAYNESS_FLAG	Current expression is arrayed.

ARRAYNESS_NEST	Not used.
ARRAYNESS_STACK	Section 4.4.
AS_PTR	Section 4.4.
ASSIGN_ARG_LIST	Section 4.4.
ASSIGN_CONTEXT	See CONTEXT in SCAN.
ASSIGN_PARM	See symbol table -- SYT_FLAGS.
ASSIGN_TYPE	Section 4.4.
ASSOCIATE	Procedure.
AST_STACKER	Procedure.
ATOM#_FAULT	Section 4.4.
ATOMS	Section 4.4.
ATTACH_SUB_ARRAY	Procedure.
ATTACH_SUB_COMPONENT	
ATTACH_SUB_STRUCTURE	
ATTACH_SUBSCRIPT	
ATTR_BEGIN_FLAG	See GRAMMAR_FLAGS.
ATTR_FOUND	Section 4.4.
ATTR_INDENT	The amount to indent after an attribute.
ATTR_LOC	Section 4.4
ATTR_MASK	
ATTRIBUTES	
AUTO_FLAG	See symbol table -- SYT_FLAGS.
AUTSTAT_FLAGS	See symbol table -- SYT_FLAGS.
BASE_PARM_LEVEL	See STREAM.
BCD	See SCAN.

BCD_PTR	See GRAMMAR_FLAGS.
BEGINP	Temporary.
BI_ARG_TYPE	Section 4.4.
BI_FLAGS	Section 4.4.
BI_FUNC_FLAG	Section 4.4.
BI_INDEX	See SCAN.
BI_INFO	See SYNTHESIZE.
BI_NAME	See SCAN.
BI_XREF	Section 4.4.
BIT_FUNC_TOKEN	See TOKEN.
BIT_LENGTH	Section 4.4.
BIT_LENGTH_LIM	Section 4.4.
BIT_LITERAL	Procedure.
BIT_TOKEN	See TOKEN.
BIT_TYPE	See symbol table -- SYT_TYPE.
BLANK	Procedure.
BLANK_COUNT	See STREAM.
BLOCK_MODE	See SYNTHESIZE.
BLOCK_SUMMARY	Procedure.
BLOCK_SUMMARY_ISSUED	Not used.
BLOCK_SYTREF	Section 4.4
BORC_TYPE	See symbol table -- SYT_TYPE.
BUILDING_TEMPLATE	Section 4.4

C	Temporary.
CALL_SCAN	Procedure.
CALLED_LABEL	Not used.
CARD_COUNT	See STREAM.
CARD_TYPE	See STREAM.
CASE_LEVEL	Section 4.4.
CASE_STACK	Section 4.4.
CHAR_FUNC_TOKEN	See TOKEN.
CHAR_INDEX	Procedure.
CHAR_LENGTH	Section 4.4.
CHAR_LENGTH_LIM	Section 4.4.
CHAR_LITERAL	Procedure.
CHAR_OP	See O-W and SCAN.
CHAR_TOKEN	See TOKEN.
CHAR_TYPE	See symbol table -- SYT_TYPE.
CHARACTER_STRING	See TOKEN.
CHARDATE	Procedure.
CHARTIME	Procedure.
CHARTYPE	See STREAM.
CHECK_ARRAYNESS	Procedure.
CHECK_ASSIGN_CONTEXT	
CHECK_CONFLICTS	
CHECK_CONSISTENCY	
CHECK_EVENT_CONFLICTS	
CHECK_EVENT_EXP	
CHECK_IMPLICIT_T	

CHECK_NAMING	Procedure.
CHECK_STRUC_CONFLICTS	
CHECK_SUBSCRIPT	
CHECK_TOKEN	
CLASS	Section 4.4.
CLASS_A	Error codes -- see User's Manual.
CLASS_AA	
CLASS_AV	
⋮	
CLASS_XM	
CLASS_XU	
CLASS_XV	
CLOCK	0 - beginning of time. 1 - time at end of set up. 2 - time at end of processing. 3 - time at end of clean up.
CLOSE_BCD	Section 4.4.
CMPL_MODE	Compiling a COMPOOL.
COMMA	See TOKEN.
COMMENT_COUNT	See O-W.
COMMENTING	See STREAM.
COMMON_SYTSIZES(i)	The number of bytes for an entry in the i^th^ common symbol table array.
COMPARE	Procedure.
COMPILATION_LOOP	Procedure.
COMPILING	Switch on while computation is continuing normally.
COMPOOL_LABEL	See symbol table -- SYT_TYPE.
COMPRESS_OUTER_REF	Procedure.
CONCATENATE	See TOKEN.
CONSTANT_FLAG	See symbol table -- SYT_FLAGS.
CONTEXT	See SCAN.

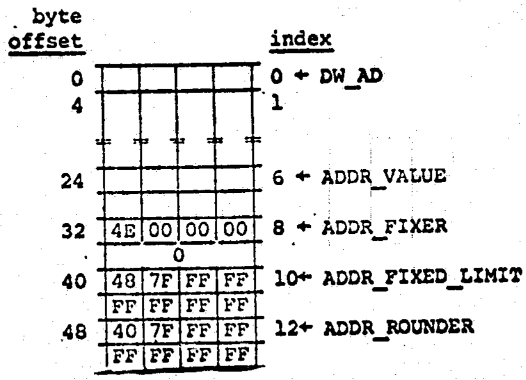
CONTROL	There are a collection of diagnostic control toggles that can be set by ¢toggle on DEBUG directives (Section 2.2.7), CONTROL(0) is status of ¢0, ..., CONTROL("F") is status of ¢F.
COPINESS	Procedure.
CPD_NUMBER	See TOKEN.
CROSS	Signal for a cross product.
CROSS_COUNT	The number of cross products in a product.
CROSS_TOKEN	See TOKEN.
CUR_IC_BLK	Section 4.4.
CURLBLK	See literal table.
CURRENT_ARRAYNESS	Section 4.4.
CURRENT_ATOM	Section 4.4.
CURRENT_CARD	See STREAM.
CURRENT_SCOPE	Name of the block actually being read by STREAM.
DECLARE_CONTEXT	See SCAN -- CONTEXT.
DECLARE_TOKEN	See TOKEN.
DECOMPRESS	Procedure.
DEF_BIT_LENGTH	See SCAN.
DEF_CHAR_LENGTH	
DEF_MAT_LENGTH	
DEF_VEC_LENGTH	
DEFAULT_ATTR	
DEFAULT_TYPE	
DEFINED_LABEL	See symbol table -- SYT_FLAGS.

DELAY_CONTEXT_CHECK	Section 4.4.
DESNE_FLAG	See symbol table -- SYT_FLAGS.
DESCORE	Procedure.
DISASTER	Procedure.
DISCONNECT	Procedure.
DO_CHAIN	Section 4.4.
DO_INIT	
DO_INX	
DO_LEVEL	
DO_LOC	
DO_PARSE	
DO_TOKEN	See TOKEN.
DOLLAR	See TOKEN.
DONT_SET_WAIT	See SCAN -- PRINTING_ENABLED.
DOT	Signal for a dot product.
DOT_COUNT	Count of dot products in a product.
DOT_TOKEN	See TOKEN.
DOUBLE	See O-W.
DOUBLE_FLAG	See symbol table -- SYT_FLAGS.
DUMMY_FLAG	See symbol table -- SYT_FLAGS.
DUMP_MACRO_LIST	See O-W.
DUMPIT	Procedure.
DUPL_FLAG	See symbol table -- SYT_FLAGS

DW

An area set aside for communication with the MONITOR.

Map of DW:



DW_AD	The address of DW(0).
EMIT_ARRAYNESS	Procedure.
EMIT_EXTERNAL	
EMIT_PUSH_DO	
EMIT_SMRK	
EMIT_SUBSCRIPT	
END_ANY_FCN	
END_GROUP	See STREAM.
END_OF_INPUT	See STREAM.
END_SUBBIT_FCN	Procedure.
ENDITNOW	Procedure.
ENDSCOPE_FLAG	See symbol table -- SYT_FLAGS.
ENTER	Procedure.
ENTER_DIMS	Procedure.
ENTER_XREF	Procedure.
EOFILE	See TOKEN.
EQUATE_CONTEXT	See CONTEXT.
EQUATE_IMPLIED	EQUATE names are kept in the symbol table with a @ prepended to them. EQUATE_IMPLIED is on until this transformation is made.
EQUATE_LABEL	See symbol table -- SYT_TYPE.

EQUATE_TOKEN	See TOKEN.
ERROR	Procedure.
ERROR_CLASSES	A character string used to produce the two letter error class code.
ERROR_COUNT	Number of errors accumulated during compilation.
ERROR_SUB	Procedure.
ERROR_SUMMARY	Procedure.
ESCAPE	Non-HAL escape character.
EVENT_TOKEN	See TOKEN.
EVENT_TYPE	See symbol table -- SYT_TYPE.
EVIL_FLAG	See symbol table -- SYT_FLAGS.
EXCLUSIVE_FLAG	See symbol table -- SYT_FLAGS.
EXP_OVERFLOW	See SCAN.
EXP_TYPE	See SCAN.
EXPONENT	See TOKEN.
EXPONENT_LEVEL	Incremented by one for every **, decremented at the end of the exponent.
EXPONENTIATE	See TOKEN.
EXPRESSION_CONTEXT	See SCAN -- CONTEXT.
EXT_ARRAY	See symbol table.
EXT_ARRAY_PTR	See symbol table - EXT_ARRAY.
EXT_P	Section 4.4.
EXTERNAL	Section 4.4.
EXTERNAL_FLAG	See symbol table -- SYT_FLAGS.
EXTERNALIZE	Section 4.4.
FACTOR	See TOKEN.
FACTOR_FOUND	Section 4.4.

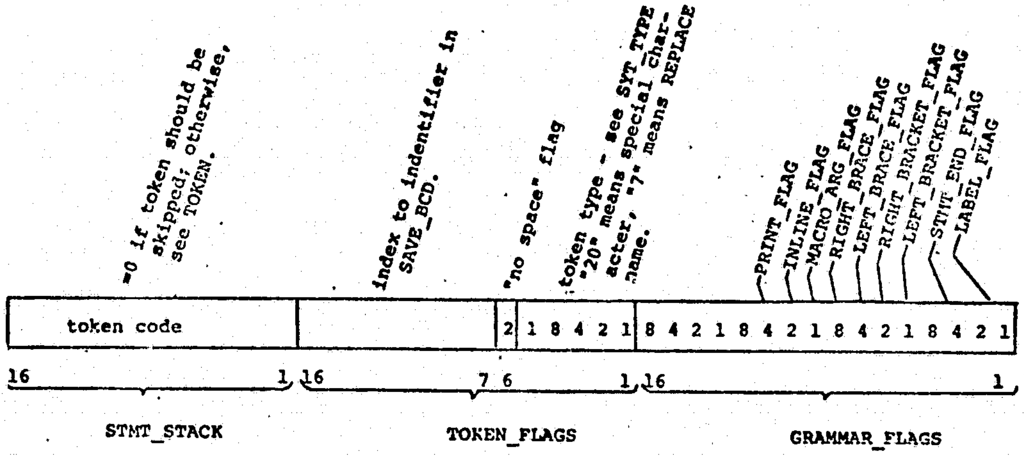
FACTORED_ATTR_MASK	FACTORED_XXX is copied to and from XXX by a loop copying between the "array" TYPE and the "array" FACTORED_TYPE.
FACTORED_ATTRIBUTES	
FACTORED_BIT_LENGTH	
FACTORED_CHAR_LENGTH	
FACTORED_CLASS	
FACTORED_IC_FND	
FACTORED_IC_PTR	
FACTORED_LOCK#	
FACTORED_MAT_LENGTH	
FACTORED_N_DIM	
FACTORED_NONHAL	
FACTORED_S_ARRAY	
FACTORED_STRUC_DIM	
FACTORED_STRUC_PTR	
FACTORED TYPE	
FACTORED_VEC_LENGTH	
FACTORING	Section 4.4
FCN_ARG	
FCN_LOC	
FCN_LV	
FCN_MODE	
FIRST_FREE	See MACRO_TEXT in SCAN.
FRIST_STMT	Section 4.4.

FIRST_TIME	See STREAM.
FIRST_TIME_PARM	See STREAM.
FIX_DIM	Section 4.4.
FIXF	Section 4.4.
FIXING	See SCAN.
FIXL	Section 4.4.
FIXV	Section 4.4.
FL_NO	Section 4.4.
FL_NO_MAX	Section 4.4.
FLOATING	Procedure.
FOUND_CENT	See SCAN.
FUNC_CLASS	See symbol table -- SYT_FLAGS.
FUNC_FLAG	See GRAMMAR_FLAGS.
FUNC_MODE	Section 4.4.
GET_ARRAYNESS	Procedure.
GET_FCN_PARM	Procedure.
GET_ICQ	Procedure.
GET_LITERAL	Procedure.

GRAMMAR_FLAGS

The statement stack is used to store up a source statement before printing. The stack is built of three parallel arrays as indicated in the diagram. STMT_PTR points to the top-most entry in the stack. Notice that the actual character strings are stored in SAVE_BCD. TOKEN_FLAGS simply contains an index into SAVE_BCD. BCD_PTR points to the last entry in SAVE_BCD. In the general case, some of the material in the stack has been printed and LAST_WRITE points to the first unprinted item.

A Statement Stack Item:



In order to associate items in the parser's stack with their entries in the statement stack, the parser maintains STACK_PTR entires. STACK_PTR (parser stack pointer) points to the element's entry in the statement stack.

GRAMMAR_FLAGS values.

0042	ATTR_BEGIN_FLAG	
0428	FUNC_FLAG	Token is a function call.
0577	INLINE_FLAG	Token is an inline function.
0671	LABEL_FLAG	Token is a label.
0687	LEFT_BRACE_FLAG	Preceed token by '{' on output.
0688	LEFT_BRACKET_FLAG	Preceed token by '[' on output.
0786	MACRO_ARG_FLAG	Token is an argument to a macro.
0976	PRINT_FLAG	Token should be printed.
0978	PRINT_FLAG_OFF	¬PRINT_FLAG -- Used to turn off PRINT_FLAG.
1047	RIGHT_BRACE_FLAG	Append "}" after token on output.
1048	RIGHT_BRACKET_FLAG	Append "]" after token on output.
1160	STMT_END_FLAG	Final token in statement.

GRAMMAR_FLAGS_UNFLO	Not used.
GROUP_NEEDED	See STREAM.
HALMAT	Procedure.
HALMAT_BACKUP	Procedure
HALMAT_BLAB	Procedure.
HALMAT_BLOCK	Section 4.4.
HALMAT_CRAP	The HALMAT file is bad.
HALMAT_FILE	Section 4.4.
HALMAT_FIX_PIP#	Procedure.
HALMAT_FIX_PIPTAGS	Procedure.
HALMAT_FIX_POPTAG	Procedure.
HALMAT_INIT_CONST	Procedure.
HALMAT_OK	The HALMAT file is good.
HALMAT_OUT	Procedure.
HALMAT_PIP	Procedure.
HALMAT_POP	Procedure.
HALMAT_RELOCATE	PROCEDURE
HALMAT_RELOCATE_FLAG	The HALMAT is not positioned at the bottom of the buffer and should be moved down.
HALMAT_TUPLE	Procedure.
HALMAT_XNOP	Procedure.
HASH	Procedure.
HEX	Procedure.
HOW_TO_INIT_ARGS	Procedure.
I	Temporary.
I_FORMAT	Procedure.

IC_FILE	Section 4.4.
IC_FND	
IC_FORM	
IC_FOUND	
IC_LEN	
IC_LIM	
IC_LINE	
IC_LOC	
IC_MAX	
IC_ORG	
IC_PTR	
IC_PTR1	
IC_PTR2	
IC_TYPE	
IC_VAL	
ICQ	
ICQ_ARRAY#	Procedure.
ICQ_ARRAYNESS_OUTPUT	
ICQ_CHECK_TYPE	
ICQ_OUTPUT	
ICQ_TERM#	
ID_LOC	Section 4.4.
ID_TOKEN	See TOKEN.
IDENT_COUNT	See SCAN.
IDENTIFY	Procedure.

ILL_ATTR	Section 4.4.
ILL_CLASS_ATTR	
ILL_EQUATE_ATTR	
ILL_INIT_ATTR	
ILL_LATCHED_ATTR	
ILL_MINOR_STRUC	
ILL_NAME_ATTR	
ILL_TEMPL_ATTR	
ILL_TEMPORARY_ATTR	
ILL_TERM_ATTR	
IMP_DECL	See symbol table -- SYT_FLAGS.
IMPL_T_FLAG	See symbol table -- SYT_FLAGS.
IMPLICIT_T	See SCAN.
IMPLIED_TYPE	See SCAN.
IMPLIED_UPDATE_LABEL	Section 4.4.
INACTIVE_FLAG	See symbol table -- SYT_FLAGS.
INCLUDE_CHAR	See O-W.
INCLUDE_COMPRESSED	See STREAM.
INCLUDE_END	
INCLUDE_LIST	
INCLUDE_LIST2	
INCLUDE_MSG	
INCLUDE_OFFSET	
INCLUDE_OPENED	
INCLUDING	

IND_CALL_LAB	See symbol table -- SYT_TYPE.
IND_ERR_#	Temporary.
IND_LINK	Section 4.4.
IND_STMT_LAB	See symbol table -- SYT_TYPE.
INDENT_INCR	Section 4.4.
INDENT_LEVEL	See O-W.
INDEX1	See Parser.
INDEX2	See Parser.
INFORMATION	See O-W.
INIT_CONST	See symbol table -- SYT_FLAGS.
INIT_EMISSION	Section 4.4.
INIT_FLAG	See symbol table -- SYT_FLAGS.
INITCONST_OFF	See symbol table -- SYT_FLAGS.
INITIAL_INCLUDE_RECORD	See STREAM.
INITIALIZATION	Procedure.
INLINE_FLAG	See GRAMMAR_FLAGS.
INLINE_INDENT	See O-W.
INLINE_INDENT_RESET	See O-W.
INLINE_LABEL	Section 4.4.
INLINE_LEVEL	Section 4.4.
INLINE_MODE	See BLOCK_MODE.
INLINE_NAME	Section 4.4.
INLINE_STMT_RESET	Used in inline processing to allow temporary resetting of STMT_NUM.
INP_OR_CONST	See symbol table -- SYT_FLAGS.

INPUT_DEV	See STREAM.
INPUT_PARM	See symbol table -- SYT_FLAGS.
INPUT_REC	See STREAM.
INT_TYPE	See symbol table -- SYT_TYPE.
INTERPRET_ACCESS_FILE	Procedure.
INX	Section 4.4.
IODEV	See SYNTHESIZE.
IORS	Procedure.
IORS_TYPE	See symbol table -- SYT_TYPE.
J	Temporary.
K	Temporary.
KILL_NAME	Procedure.
KIN	See SCAN.
L	Temporary.
LAB_TOKEN	See TOKEN.
LABEL_CLASS	Class for label symbols.
LABEL_COUNT	Number of labels on current statement.
LABEL_DEFINITION	See TOKEN.
LABEL_FLAG	See GRAMMAR_FLAGS.
LABEL_IMPLIED	See SCAN -- CONTEXT.
LABEL_MATCH	Procedure.
LAST	See O-W.
LAST_POP#	Section 4.4.
LAST_SPACE	See O-W.
LAST_WRITE	See GRAMMAR_FLAGS.
LATCHED_FLAG	See symbol table -- SYT_FLAGS.
LEFT_BRACE_FLAG	See GRAMMAR_FLAGS.
LEFT_BRACKET_FLAG	See GRAMMAR_FLAGS.

LEFT_PAD	Procedure.
LEFT_PAREN	See TOKEN.
LETTER_OR_DIGIT	See STREAM.
LEVEL	See TOKEN.
LINE_LIM	See O-W.
LINE_MAX	See O-W.
LISTING2	ON for second listing.
LISTING2_COUNT	The number of lines already used on the current LISTING2 page.
LIT_CHAR	See literal table.
LIT_CHAR_AD	See literal table.
LIT_CHAR_FREE	See literal table.
LIT_CHAR_SIZE	See literal table.
LIT_DUMP	Procedure.
LIT_PTR	
LIT_RESULT_TYPE	Procedure.
LIT_TOP	See literal table.
LITLIM	
LITMAX	
LITORG	
LIT1	
LIT2	
LIT3	
LOC_P	Section 4.4.
LOCK_FLAG	See symbol table -- SYT_FLAGS.
LOCK_LIM	The maximum LOCK#.
LOCK#	Section 4.4.

LOOK	See Parser.
LOOK_STACK	See Parser.
LOOKUP_ONLY	See SCAN.
LOOK1	See Parser.
LOOK2	See Parser.
LRECL	See STREAM.
M_BLANK_COUNT	See SCAN.
M_CENT	See STREAM.
M_P	See SCAN.
M_PRINT	See SCAN.
M_TOKENS	See SCAN.
MAC_NUM	See O-W.
MACRO_ADDR	A word containing a dummy character string descriptor of the REPLACE text area.
MACRO_ARG_COUNT	See SCAN.
MACRO_ARG_FLAG	See GRAMMAR_FLAGS.
MACRO_CALL_PARM_TABLE	See SCAN.
MACRO_EXPAN_LEVEL	See STREAM.
MACRO_EXPAN_STACK	See SCAN.
MACRO_FOUND	See STREAM.
MACRO_INDEX	See O-W. .
MACRO_NAME	See SCAN.
MACRO_POINT	See SCAN.
MACRO_TEXT	See SCAN.
MACRO_TEXT_DUMP	Procedure.

MACRO_TEXT_LIM	Number of characters of storage allocated for REPLACE <text>.
MAIN_SCOPE	The SYT_SCOPE value of the compilation unit.
MAJ_STRUC	See symbol table -- SYT_TYPE.
MAKE_FIXED_LIT	Procedure.
MAT_DIM_LIM	Largest legal matrix dimension.
MAT_LENGTH	Section 4.4.
MAT_TYPE	See symbol table -- SYT_TYPE.
MATCH_ARITH	Procedure.
MATCH_ARRAYNESS	Procedure.
MATCH_SIMPLES	Procedure.
MATRIX_COMPARE	Procedure.
MATRIX_COUNT	The number of matrices in a product.
MATRIX_PASSED	The number of matrices to multiply by a vector.
MATRIXP	Pointer to the stack entry for the current product of matrices.
MAX	Procedure.
MAX_PTR_TOP	Section 4.4.
MAX_SCOPE#	Section 4.4.
MAX_SEVERITY	Maximum error severity encountered.
MAXNEST	Section 4.4.
MAXSP	The maximum stack size achieved.
MAX	Procedure.
MIN	Procedure.
I	Temporaries.
J	Temporaries.
MISC_NAME_FLAG	See symbol table -- SYT_FLAGS.

MP	See Parser.
MPP1	See Parser.
MULTIPLY_SYNTHESIZE	Procedure.
N_DIM	Section 4.4.
NAME_ARRAYNESS	Procedure.
NAME_BIT	Item is NAME (something).
NAME_COMPARE	Procedure.
NAME_FLAG	See symbol table -- SYT_FLAGS.
NAME_HASH	See STREAM.
NAME_IMPLIED	Processing a declaration for a NAME variable.
NAME_PSEUDOS	Processing a NAME variable.
NAMING	Have seen a NAME pseudo-function and have not yet encountered the closing paren.
NDECSY	See symbol table.
NEST	Section 4.4.
NEW_LEVEL	See STREAM.
NEW_MEL	See SCAN.
NEXT	See STREAM.
NEXT_ATOM#	Section 4.4.
NEXT_CHAR	See STREAM.
NEXT_RECORD	Procedure.
NEXT_SUB	Section 4.4.
NEXTIME_LOC	50.

NO_ARG_ARITH_FUNC	See TOKEN.
NO_ARG_BIT_FUNC	
NO_ARG_CHAR_FUNC	
NO_ARG_STRUCT_FUNC	
NO_LOOK_AHEAD_DONE	See Parser.
NONBLANK_FOUND	See STREAM.
NONCOMMON_SYTSIZES(i)	The number of bytes in an entry in the i^{th} non-common symbol table array.
NONHAL	Section 4.4.
NONHAL_FLAG	See symbol table -- SYT_FLAGS.
NOT_ASSIGNED_FLAG	A local variable of SYT_DUMP.
NT_PLUS_1	Not used.
NUM_ELEMENTS	Section 4.4.
NUM_FL_NO	Section 4.4.
NUM_OF_PARM	See STREAM.
NUM_STACKS	Section 4.4.
NUMBER	See TOKEN.
OLD_LEVEL	See STREAM.
OLD_MEL	See SCAN.
OLD_MP	See SCAN.
OLD_PEL	See SCAN.
OLD_TOPS	See SCAN.
ON_ERROR_PTR	See symbol table -- EXT_ARRAY.
ONE_BYTE	Temporary.

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

OPTIONS_CODE	See COMM(7).
ORDER_OK	Procedure.
OUT_PREV_ERROR	See O-W.
OUTER_REF	See OUTER_REF in SCAN.
OUTER_REF_INDEX	See OUTER_REF in SCAN.
OUTER_REF_PTR	See OUTER_REF in SCAN.
OUTPUT_GROUP	Procedure.
OUTPUT_WRITER	Procedure.
OUTPUT_WRITER_DISASTER	Label in main program. OUTPUT_WRITER jumps here when all is lost.
OVER_PUNCH	See STREAM.
OVER_PUNCH_TYPE	See O-W and SCAN.
P_CENT	See STREAM.
PAD	Procedure.
PAD1	See O-W.
PAD2	See O-W.
PAGE	See O-W.
PAGE_THROWN	See O-W.
PARM_CONTEXT	See CONTEXT in SCAN.
PARM_COUNT	See SCAN.
PARM_EXPAN_LEVEL	See STREAM.
PARM_FLAGS	See symbol table -- SYT_FLAGS.
PARM_REPLACE_PTR	See STREAM.
PARM_STACK_PTR	See STREAM.

PARMS_PRESENT	Section 4.4.
PARMS_WATCH	Section 4.4.
PARSE_STACK	See Parser.
PARTIAL_PARSE	ON if PARSE request in JCL option.
PASS	See SCAN.
PC_LIMIT	See SCAN.
PCARG#	Section 4.4.
PCARGBITS	Section 4.4.
PCARGOFF	Section 4.4.
PCARGTYPE	Section 4.4.
PCNAME	See SCAN.
PERCENT_MACRO	See TOKEN.
PERIOD	A ".".
PHASE1_FREESIZE	Storage avove this point is for Phase 1 only and can be returned at the end.
PHASE2_STUFF	Not used.
PLUS	See O-W.
PP	Temporary.
PPTEMP	Temporary.
PREP_LITERAL	Procedure.
PREVIOUS_ERROR	See O-W.
PRINT_DATE_AND_TIME	Procedure.
PRINT_FLAG	See GRAMMAR_FLAGS.
PRINT_FLAG_OFF	See GRAMMAR_FLAGS.

PRINT_SUMMARY	Procedure.
PRINT_TIME	Procedure.
PRINTING_ENABLED	See SCAN.
PRINT2	Procedure.
PROC_LABEL	See symbol table -- SYT_TYPE.
PROC_MODE	Section 4.4.
PROCESS_CHECK	Procedure.
PROCMARK	Section 4.4.
PROCMARK_STACK	Section 4.4.
PROG_LABEL	See symbol table -- SYT_TYPE.
PROG_MODE	Section 4.4.
PROGRAM_ID	See STREAM.
PROGRAM_LAYOUT	Section 4.4.
PROGRAM_LAYOUT_INDEX	
PSEUDO_FORM	
PSEUDO_LENGTH	
PSEUDO_TYPE	
PTR	
PTR_TOP	
PUSH_FCN_STACK	Procedure.
PUSH_INDIRECT	Procedure.
QUALIFICATION	See SCAN.
READ_ACCESS_FLAG	See symbol table -- SYT_FLAGS.
READ_ALL_TYPE	Procedure.

READ_TYPE	See Parser.
READ1	See Parser.
READ2	See Parser.
RECOVER	Procedure.
RECOVERING	See O-W.
REDUCE_SUBSCRIPT	Procedure.
REDUCTIONS	See Parser.
REENTRANT_FLAG	See symbol table -- SYT_FLAGS.
REF_ID_LOC	Section 4.4.
REFER_LOC	Section 4.4.
REGULAR_PROCMARK	A pointer to the first symbol table entry for the current procedure.
REL_OP	Section 4.4.
REMOTE_FLAG	See symbol table -- SYT_FLAG.
REPL_ARG_CLASS	See symbol table -- SYT_CLASS.
REPL_CLASS	See symbol table -- SYT_CLASS.
REPL_CONTEXT	See SCAN -- CONTEXT.
REPLACE_PARM_CONTEXT	See SCAN -- CONTEXT.
REPLACE_TEXT	See TOKEN.
REPLACE_TOKEN	See TOKEN.
RESERVED_LIMIT	See SCAN.
RESERVED_WORD	See SCAN.
RESET_ARRAYNESS	Procedure.
RESTORE	See SCAN.

RIGHT_BRACE_FLAG	See GRAMMAR_FLAG.
RIGHT_BRACKET_FLAG	See GRAMMAR_FLAG.
RIGID_FLAG	See symbol table -- SYT_FLAGS.
RT_PAREN	See TOKEN.
S	Temporary character string.
S_ARRAY	Section 4.4.
SAVE_ARRAYNESS	Procedure.
SAVE_ARRAYNESS_FLAG	ARRAYNESS_FLAG saved here while processing subscripts.
SAVE_BCD	See GRAMMAR_FLAGS.
SAVE_BLANK_COUNT	See SCAN.
SAVE_CARD	See STREAM.
SAVE_COMMENT	See O-W.
SAVE_DUMP	Procedure.
SAVE_ERROR_MESSAGE	See O-W.
SAVE_GROUP	See STREAM.
SAVE_INDENT_LEVEL	Section 4.4.
SAVE_INPUT	Procedure.
SAVE_LINE_#	Array of line numbers on which error occurred.
SAVE_LITERAL	Procedure.
SAVE_NEXT_CHAR	See STREAM.
SAVE_OVER_PUNCH	See STREAM.
SAVE_PE	See SCAN.
SAVE_SCOPE	See O-W.
SAVE_SEVERITY	See O-W.
SAVE_	See O-W.
SAVE_TOKEN	Procedure.

SCALAR_COUNT	Number of scalars invovled in a product.
SCALAR_TYPE	See symbol table -- SYT_TYPE.
SCALARP	Stack pointer for product of scalars.
SCAN	Procedure.
SCAN_COUNT	See SCAN.
SCOPE#	Section 4.4.
SCOPE#_STACK	Section 4.4.
SD_FLAGS	See symbol table -- SYT_FLAGS.
SDL_OPTION	See O-W.
SEMI_COLON	See TOKEN.
SET_BI_XREF	Procedure.
SET_CONTEXT	See SCAN -- CONTEXT.
SET_LABEL_TYPE	Procedure.
SET_OUTER_REF	
SET_SYT_ENTRIES	
SET_XREF	
SET_XREF_RORS	
SETUP_CALL_ARG	
SETUP_NO_ARG_FCN	
SETUP_VAC	
SIGNAL_STMT	On if TABLES option was requested.
SIMULATING	See symbol table -- SYT_FLAGS.
SINGLE_FLAG	
SLIP_SUBSCRIPT	Procedure.

SM_FLAGS	See symbol table -- SYT_FLAGS.
SMRK_LOC	Not used.
SOME_BCD	See SCAN.
SP	See Parser.
SPACE_FLAGS	See O-W.
SQUEEZING	See O-W.
SREF_OPTION	On if SREF selected on JCL.
SRN	See O-W.
SRN_COUNT	See O-W.
SRN_COUNT_MARK	Section 4.4.
SRN_FLAG	On if something hanging for an SRN_UPDATE.
SRN_MARK	Section 4.4.
SRN_PRESENT	ON if SRN option requested on JCL.
SRN_UPDATE	Procedure.
STAB	A buffer used for accumulating information to be written on the statement file.
STAB_BLK	The number of STAB blocks written.
STAB_CLOSE	Procedure.
STAB_ENTER	Procedure.
STAB_HDR	Procedure.
STAB_INX	Pointer to the next available word in the STAB buffer.
STAB_LAB	Procedure.
STAB_MARK	Section 4.4.
STAB_SKIP	The number of extra words in a STAB entry required by subsequent phases.
STAB_STACK	Section 4.4.
STAB_STACKER	Procedure.
STAB_STACKTOP	Section 4.4.

STAB_VAR	Procedure.
STACK_DUMP	Procedure.
STACK_DUMP_PTR	See O-W.
STACK_DUMPED	See O-W.
STACK_PTR	See GRAMMAR_FLAGS.
STACKING_COUNT	Section 4.4.
STARRED_DIMS	Section 4.4.
STARS	"*****"
START_NORMAL_FCN	Procedure.
START_POINT	See MACRO_TEXT in SCAN.
STATE	See Parser.
STATE_NAME	See Parser.
STATE_STACK	See Parser.
STATEMENT_SEVERITY	Maximum error severity in current statement.
STATIC_FLAG	See symbol table -- SYT_FLAGS.
STMT_END_FLAG	See GRAMMAR_FLAGS.
STMT_LABEL	See symbol table -- SYT_TYPE.
STMT_NUM	Statement number.
STMT_PTR	See GRAMMAR_FLAGS.
STMT_STACK	See GRAMMAR_FLAGS.
STMT_TYPE	The type of the statement -- used for writing on statement file.
STORAGE_FLAGS	Not used.
STREAM	Procedure.
STRING	Procedure.
STRING_GT	Procedure.
STRING_OVERFLOW	See SCAN.

STRUC_DIM	Section 4.4.
STRUC_PTR	Section 4.4.
STRUC_SIZE	The size of the structure whose template is being declared.
STRUC_TOKEN	See TOKEN.
STRUCT_FUNC_TOKEN	See TOKEN.
STRUC_TEMPLATE	See TOKEN.
STRUCTURE_COMPARE	Procedure.
STRUCTURE_FCN	Procedure.
STRUCTURE_SUB_COUNT	Section 4.4.
STRUCTURE_WORD	See TOKEN.
SUB_COUNT	Section 4.4.
SUB_SEEN	Section 4.4.
SUBHEADING	A constant character string.
SUBSCRIPT_LEVEL	See Parser.
SUPPRESS_THIS_TOKEN_ONLY	See PRINTING_ENABLED in SCAN.
SYNTHESIZE	Procedure.
SYSIN_COMPRESSED	On if input is in compressed format.
SYT_ADDR	See symbol table.
SYT_ARRAY	See symbol table.
SYT_CLASS	See symbol table.
SYT_DUMP	Procedure.
SYT_FLAGS	See symbol table.
SYT_HASHLINK	See symbol table.
SYT_HASHSTART	See symbol table.
SYT_INDEX	See SCAN.
SYT_LINK1	See symbol table.

SYT_LINK2	See symbol table.
SYT_LOCK#	
SYT_NAME	
SYT_NEST	
SYT_PTR	
SYT_SCOPE	
SYT_SORT	
SYT_TYPE	
SYT_XREF	
SYTSIZE	
T_INDEX	See MACRO_TEXT in SCAN.
TASK_LABEL	See symbol table -- SYT_TYPE.
TASK_MODE	Section 4.4.
TEMP	Temporary.
TEMP_INDEX	Local variable of PARM_FOUND.
TEMP_STRING	See SCAN.
TEMP_SYN	Temporary.
TEMPL_NAME	See symbol table -- SYT_TYPE.
TEMPLATE_CLASS	See symbol table -- SYT_CLASS.
TEMPLATE_IMPLIED	See SCAN -- CONTEXT.
TEMPORARY	See TOKEN.
TEMPORARY_FLAG	See symbol table -- SYT_FLAG.
TEMPORARY_IMPLIED	See SCAN.
TEMP1	Temporary.
TEMP2	Temporary.
TEMP3	Temporary.
TERMP	Temporary.

TEXT_LIMIT	See STREAM.
THE_BEGINNING	Procedure.
TIE_XREF	Procedure.
TOGGLE	Literally COMM(6).
TOKEN	Type of current token. Value of -1 indicates REPLACE name; otherwise:
ARITH_FUNC_TOKEN	Functioning returning an arithmetic value.
ARITH_TOKEN	Arithmetic value such as matrix, vector, scalar, integer.
BIT_FUNC_TOKEN	Functioning returning a bit string value.
BIT_TOKEN	Bit string value.
CHAR_FUNC_TOKEN	Function returning a character string value.
CHAR_TOKEN	Character string value.
CHARACTER_STRING	Character literal.
COMMA	",".
CONCATENATE	Concatenation operator "||".
CPD_NUMBER	Invalid numeric token.
CROSS_TOKEN	Cross product operator (*).
DECLARE_TOKEN	Keyword DECLARE.
DO_TOKEN	Keyword DO.
DOLLAR	"$".
DOT_TOKEN	Dot product operator (.).
EOFILE	End of file marker (X"FE").
EVENT_TOKEN	Keyword EVENT.
EXPONENT	
EXPONENTIATE	"**".
FACTOR	

ID_TOKEN	Identifier (parameter and replace macro names) - also used for undefined names in error.
LAB_TOKEN	Label value.
LABEL_DEFINITION	
LEFT_PAREN	"(".
LEVEL	Structure declaration level number.
NO_ARG_ARITH_FUNC	Function with no arguments following.
NO_ARG_BIT_FUNC	Function with no arguments following.
NO_ARG_CHAR_FUNC	Function with no arguments following.
NO_ARG_STRUCT_FUNC	Function with no arguments following.
NUMBER	Numeric literal.
PERCENT_MACRO	%macro name.
REPLACE_TEXT	The <text> part of a REPLACE statement.
REPLACE_TOKEN	Keyword REPLACE.
RT_PAREN	")".
SEMI_COLON	";"
STRUC_TOKEN	Structure value.
STRUCT_FUNC_TOKEN	Function returning a structure value.
STRUCT_TEMPLATE	Not used.
STRUCTURE_WORD	Not used.
TEMPORARY	Keyword TEMPORARY.
TOKEN_FLAGS	See GRAMMAR_FLAGS.
TOKEN_FLAGS_UNFLO	
TOKEN_WAS_COMMA	
TOO_MANY_ERRORS	ON if error stack overflowed.
TOO_MANY_LINES	See STREAM.
TOP_OF_PARM_STACK	See SCAN.

TPL_FLAG	On if XO option request in JCL.
TPL_FUNC_CLASS	See symbol table -- SYT_CLASS.
TPL_LAB_CLASS	See symbol table -- SYT_CLASS.
TPL_LRECL	Line length for template = LRECL+1.
TPL_NAME	The name of the current template being processed.
TPL_VERSION	The template version number.
TRANS_IN	See SCAN.
TRANS_OUT	See O-W.
TX	See SCAN.
TYPE	Section 4.4.
UNARRAYED_INTEGER	Procedure.
UNARRAYED_SCALAR	Procedure.
UNARRAYED_SIMPLE	Procedure.
UNBRANCHABLE	Procedure
UNSPEC	Procedure.
UNSPEC_LABEL	See symbol table -- SYT_TYPE.
UPDATE_BLOCK_CHECK	Procedure.
UPDATE_BLOCK_LEVEL	Section 4.4.
UPDATE_MODE	Section 4.4.
V_INDEX	See procedure SCAN -- identifier.
VAL_P	Section 4.4.
VALID_00_CHAR	See SCAN.
VALID_00_OP	See SCAN.
VALUE	See SCAN.
VAR	Section 4.4.
VAR_ARRAYNESS	Section 4.4.
VAR_CLASS	See symbol table -- SYT_CLASS.

VAR_LENGTH	See symbol table -- identical to SYT_DIMS.
VBAR	See O-W.
VEC_LENGTH	Section 4.4.
VEC_LENGTH_LIM	Section 4.4.
VEC_TYPE	See symbol table -- SYT_TYPE.
VECTOR_COMPARE	Procedure.
VECTOR_COUNT	The number of vectors involved in a product.
VECTORP	Stack pointer for current product of vectors.
VERSION	Version of the compiler.
VERSION_LEVEL	Fractional version of the compiler.
VOCAB	See SCAN.
VOCAB_INDEX	See procedure SCAN -- identifiers.
WAIT	See SCAN.
WAS_HERE	See O-W.
XADLP	HALMAT Codes.
HAFOR	HALMAT Codes.
XAST	Form.
XASZ	Form.
XBAND	HALMAT Codes.
HBCAT	
HBEQU	
XBFNC	
XBINT	
XBNOT	
XBOR	
XBRA	
XBTOB	

XBTOC	HALMAT Codes.
XBTOI	
XBTOQ	
XBTOS	
XBTRU	
XCANC	
XCAND	
XCCAT	
XCDEF	
XCEQU	
XCFOR	
XCLBL	
XCLOS	
XCNOT	
XCO_D	Code Optimizer Bits.
XCO_N	Code Optimizer Bits.
XCOR	HALMAT Codes.
XCSZ	Form.
XCTST	HALMAT Codes.
XDCAS	
XDFOR	
XDLPE	
XDSMP	
XDSMP	
XDSUB	
XDTST	

XECAS	HALMAT Codes.
XEDCL	
XEFOR	
XEINT	
XELRI	
XERON	
XERSE	
XESMP	
XETRI	
XETST	
XFXTN	
HFASN	
XFBRA	
XFCAL	
XFDEF	
XFILE	
XICLS	
XIDEF	
XIDLP	
XIEQU	
XIFDH	
XIMD	Form.

XIMRK	HALMAT Code.
XINL	Form.
XITOS	HALMAT Code.
XLBL	HALMAT Codes.
XLFNC	HALMAT Codes.
XLIT	Form.
XMADD	HALMAT Codes.
XMDEF	
XMEQU	
XMINV	
XMMPR	
XMNEG	
XMSDV	
XMSHP	
XMSPR	
XMSUB	
XMTOM	
XMTRA	
XMVPR	
XNASN	
XNEQU	
XNINT	
XNOP	
XOFF	Form.

XPCAL	HALMAT Codes.
XPDEF	
XPMAR	
XPMHD	
XPMIN	
XPRIO	
XREAD	
XREF	See symbol table -- SYT_XREF.
XREF_ASSIGN	
XREF_FULL	
XREF_INDEX	
XREF_LIM	
XREF_MASK	
XREF_REF	
XREF_SUBSCR	
XRTRN	HALMAT Codes.
HSADD	
HSCHD	
XSEQU	
XSEXP	
XSFAR	
XSFNO	
XSFST	
XSGNL	
XSIEX	
XSLRI	

XSMRK	HALMAT Codes.
XSPEX	
XSSPR	
XSSUB	
XSTOI	
XSTRI	
XSYT	Form.
XTASN	HALMAT Codes.
XTDCL	
HTDEF	
XTEQU	
XTERM	
XTINT	
XTSUB	
XUDEF	
XVAC	Form.
XVCRS	HALMAT Codes.
XVDOT	
XVEQU	
XVMPR	
XVSPR	
XVVPR	
XWAIT	
XXASN	
XXPT	Form.
XXREC	HALMAT Codes.
XXXAR	HALMAT Codes.

XXXND	HALMAT Codes.
XXXST	HALMAT Codes.
X1	n blanks.
X2	
X32	
X4	
X70	
X8	

4.5.2 Index to Procedure Descriptions

427200	DECOMPRESS	X
270400	DESCORE	X
1093300	DISCONNECT	X
1083200	DUMPIT	X
863300	EMIT_ARRAYNESS	X
764600	EMIT_EXTERNAL	4.1.2
812100	EMIT_PUSH_DO	Sec. 4.4.6 Production 159
809600	EMIT_SMRK	X
926300	EMIT_SUBSCRIPT	Sec. 4.4.4 Production 219
964900	END_ANY_FCN	Sec. 4.4.5
993300	END_SUBBIT_FCN	Similar to END_ANY_FCN
556200	ENTER	4.2.1
1043400	ENTER_DIMS	See SET_SYT_ENTRIES
549400	ENTER_XREF	4.2.1
281800	ERROR	X
827200	ERROR_SUB	Sec. 4.4.6 Production 72
1554300	ERROR_SUMMARY	X
287000	FLOATING	X
871100	GET_ARRAYNESS	Sec. 4.4.4 Production 219
901900	GET_FCN_PARM	Sec. 4.4.5 Production 178
997400	GET_ICQ	See data description IC_LINE
175900	GET_LITERAL	4.2.1
801100	HALMAT	4.4.8
803400	HALMAT_BACKUP	4.4.8
790000	HALMAT_BLAB	4.4.8
807200	HALMAT_FIX_PIP#	4.4.8
808800	HALMAT_FIX_PIPTAGS	4.4.8
80800	HALMAT_FIX_POPTAG	4.4.8
1015200	HALMAT_INIT_CONST	4.4.3
798700	HALMAT_OUT	Sec. 4.4.7 Production 1
80500	HALMAT_PIP	4.4.8

80400	HALMAT_POP	4.4.8
794100	HALMAT_RELOCATE	4.4.8
805900	HALMAT_TUPLE	4.4.8
802800	HALMAT_XNOP	X.
288200	HASH	X
272400	HEX	X
1013200	HOW_TO_INIT_ARGS	4.4.3
273900	I_FORMAT	X
1000800	ICQ_ARRAY#	X
1002000	ICQ_ARRAYNESS_OUTPUT	4.4.3
1003900	ICQ_CHECK_TYPE	4.4.3
1007200	ICQ_OUTPUT	4.4.3
999100	ICQ_TERM#	X
557900	IDENTIFY	4.2.1
1055200	INITIALIZATION	4.4.8
316800	INTERPRET_ACROSS_FILE	4.2.2
861700	IORS	Sec. 4.4.4 Production 229
878100	KILL_NAME	Sec. 4.4.6 Production 53
815100	LABEL_MATCH	Sec. 4.4.6 Production 42
269600	LEFT_PAD	X
663800	LIT_DUMP	X
849500	LIT_RESULT_TYPE	4.4.4
662300	MACRO_TEXT_DUMP	X
284400	MAKE_FIXED_LIT	X
847500	MATCH_ARITH	4.4.4
887800	MATCH_ARRAYNESS	Sec. 4.4.4 Production 219
834100	MATCH_SIMPLES	4.4.4
819200	MATRIX_COMPARE	4.4
262300	MAX	X
261700	MIN	X
853500	MULTIPLY_SYNTHESIZE	Sec. 4.4.5 Production 11
881300	NAME_ARRAYNESS	Sec. 4.4.5 Production 113
873400	NAME_COMPARE	Sec. 4.4.5 Production 113
434500	NEXT_RECORD	4.2.2
436200	ORDER_OK	4.2.2
278900	OUTPUT_GROUP	4.2.2
291000	OUTPUT_WRITER	4.3.3

268700	PAD	X
574100	PREP_LITERAL	4.2.1
781700	PRINT_DATE_AND_TIME	X
1555900	PRINT_SUMMARY	Close files
781000	PRINT_TIME	X
277200	PRINT2	X
824100	PROCESS_CHECK	Sec. 4.4.6 Production 432
841500	PUSH_FCN_STACK	4.4.5
814100	PUSH_INDIRECT	X
835400	READ_ALL_TYPE	Sec. 4.4.6 Production 278
1534500	RECOVER	4.1.2
932600	REDUCE_SUBSCRIPT	Sec. 4.4.4 Production 219
866500	RESET_ARRAYNESS	
865000	SAVE_ARRAYNESS	See data définition of VAR_ARRAYNESS Sec. 4.4
280600	SAVE_DUMP	4.1.2
274900	SAVE_INPUT	4.2.2
569800	SAVE_LITERAL	4.2.1
399700	SAVE_TOKEN	4.2.1
577700	SCAN	4.2.1
551300	SET_BI_XREF	4.4.5
1089200	SET_LABEL_TYPE	4.4.2 (Production 307)
547800	SET_OUTER_REF	4.2.1
1047500	SET_SYT_ENTRIES	4.4.3
552300	SET_XREF	4.2.1
554400	SET_XREF_RORS	X
904100	SETUP_CALL_ARG	Sec. 4.4.5 Production 178
891000	SETUP_NO_ARG_FCN	4.4.5
817500	SETUP_VAC	X
941900	SLIP_SUBSCRIPT	Sec. 4.4.4 Production 219
777100	SRN_UPDATE	X
786900	STAB_CLOSE	X
782400	STAB_ENTER	X
787600	STAB_HDR	X
786200	STAB_LAB	4.4.2 (Production 304)
783700	STAB_STACKER	X

784700	STAB_VAR	X
1087300	STACK_DUMP	4.1.2
896300	START_NORMAL_FCN	4.4.5
440700	STREAM	4.2.2
264700	STRING	X
617400	STRING_GT	X
837500	STRUCTURE_COMPARE	4.4.5 Production 112
890200	STRUCTURE_FCN	4.4.5
1101600	SYNTHESIZE	4.4
618600	SYT_DUMP	Print a formatted dump of the symbol table with all cross references
660300	TIE_XREF	Sec. 4.4.7 Production 289
820600	UNARRAYED_INTEGER	X
822600	UNARRAYED_SCALAR	X
821800	UNARRAYED_SIMPLE	Sec. 4.4.6 Production 163
816300	UNBRANCHABLE	Sec. 4.4.6 Production 33
265400	UNSPEC	X
842800	UPDATE_BLOCK_CHECK	4.4.5
818500	VECTOR_COMPARE	4.4

5.0 PHASE II

5.1 Data Structures

5.1.1 Block Definition Table

A group of arrays of length PROC #. These arrays contain information about all CSECTs in a HAL/S compilation unit. There are CSECTs for programs, compools, tasks, procedures, functions, update blocks, and external templates, as well as compiler created CSECTs. Each CSECT is given a number which also serves as its ESDID. For symbol table entries, this number corresponds to the entry's SYT_SCOPE.

Not all the arrays are relevant to each type of CSECT. The possible information associated with each CSECT consists of:

CALL#	Value	Block Type
	0	Procedure, function, task
	1	Program
	2	Compool
	4	Exclusive or update blocks

ERRALL

The number of error groups for which ON ERROR statements appear for all members.

ERRALLGRP

1 if ON ERROR control for all errors is on at some point during the block, 0 otherwise.

ERRPTR

A pointer to the first ERR_STACK entry associated with the block. ERRPTR(0) is the total number of errors in the error stack.

ERRSEG

1) The displacement of the beginning of the error vector within the block's run time stack frame (i.e. the maximum temporary storage excluding the error vector).

2) During object code generation, this array is used to store the beginning address of the last HAL/S source statement processed within a block.

INDEXNEST

The ESDID number of the block enclosing a given block. INDEXNEST (0) is the currently active csect (either code or data); that is, it is the current scope of the location counter. Most of the time the rest of the arrays are accessed using INDEXNEST as the subscript.

LASTBASE

The last base register used for addresssing data declared in a block.

LASTLABEL

Pointer to the statement number of the last label set within a block. This is the beginning of a linked list of all the labels set within a block and connected by LOCATION_LINK.

LOCCTR

The location counter of each CSECT.

MAXERR

The number of errors for which ON ERROR statements exist in a block.

MAXTEMP

The maximum temporary storage required by a CSECT in the runtime stack.

NARGS

The number of arguments of a procedure, function.

ORIGIN

A value used to provide an origin for addresses within a CSECT.

PROC_LEVEL

A pointer to a block's symbol table entry.

PROC_LINK (scope #)

Pointer to the symbol table entry for the last name declared in scope #. This variable is used to set up a list of all variables within the block (see SYT_LEVEL).

PTRARG

Is 1 if register 2 (FC only) has been reserved for something in the calling sequence.

REMOTE_LEVEL

The ESDID of a CSECT used for storing REMOTE variables declared in the CSECT, if the CSECT is an EXTERNAL template.

RIGID_BLOCK

Literally INDEXNEST. Is TRUE if EXTERNAL template or COMPOOL compilation unit is RIGID.

STACKSPACE

During object code generation, this is the ending address of the last HAL/S source statement passed within the block.

WORKSEG

The displacement of the beginning of the area used for storing intermediate results (i.e. the amount of temporary storage required for the block's register save area, error vector, parameters, temporary variables, and AUTOMATIC variables).

While processing a block, additional information for ON ERROR statements is kept in two additional arrays of dimension ARG_STACK#.

ERR_DISP

ERR_DISP(I) is the displacement (relative to beginning of error vector) of the error described in ERR_STACK(I).

ERR_STACK(I)

Is an entry of the form:

errror number	error group
9	6

Notice that ERRALL is the number of distinct error number = * entries already appearing in ERRSTACK for the current block.

5.1.2 CALL STACK

A group of arrays of length CALL_LEVEL# containing information necessary for setting up calls to procedures, functions, I/O routines, and shaping functions.

For every nest level of invocation at any time during compilation, the arrays specify the following information.

ARG_COUNTER

Initially the number of arguments to a procedure, function, or I/O routine. Decremented after each HALMAT XXAR instruction.

ARG_POINTER

1) Initially points to the symbol table entry of the first argument of a procedure, function, or I/O routine. Incremented after each HALMAT XXAR statement.

2) For integer and scalar shaping functions, it is a pointer to the first free entry in SF_RANGE.

CALL_CONTEXT

The context of the call:

1	I/O routine
2	Shaping function, non-HAL function or procedure, other
4	Function or procedure

SAVE_ARG_STACK_PTR

The value of ARG_STACK_PTR at the beginning of the invocation.

SAVE_CALL_LEVEL

The value of CALL_LEVEL at the beginning of the invocation.

5.1.3 INDIRECT STACK

The code generation phase of a compiler requires a place to keep descriptors for the items it is manipulating. One candidate is the symbol table. This choice has the disadvantage of being very space inefficient. Specifically,

- it requires a symbol table entry for every temporary, even though temporaries are of interest for a very short time

- it requires the addition of many more fields to every entry in the symbol table even though these fields hold information of a transient nature (e.g. the register containing the variable).

Because of these considerations, a far better choice is to set up an auxiliary, transient, expanded symbol table. There is one descriptor in this table for each item currently of interest. Since the number of items is small, the amount of information per item can be large.

Many compilers use a stack mechanism for allocating space for these descriptors (thus our name "INDIRECT STACK"). As the code generation process becomes more sophisticated, a stack mechanism becomes less and less appropriate. Thus, our "stack" is actually an array with a free list (STACK_PTR). Pointers to this array are kept in immediately active locations (e.g. the operands of the current instruction) and in the HALMAT where they overwrite the instruction used to generate them.

The indirect stack is a group of parallel arrays of length STACK_SIZE.

BACKUP_REG

This is the same as the base register associated with an entry except in certain cases where it is used to save the base register. This is done because when a register is checkpointed, a pointer to its contents in temporary storage is kept, but the number of the register which held the contents is forgotten. BACKUP_REG can be used to retain this number. This is necessary in code generation for DO FOR loops where a checkpointed loop index must be reloaded into the name register it originally occupied.

BASE

The base register associated with the entry. If BASE < 0, it is a virtual register which must be assigned to a hardware register and loaded before use.

COLUMN

The significant of COLUMN depends on the entry's TYPE.

1) MATRIX: The number of columns.

2) VECTOR: The number of components.

3) BIT: A pointer to an indirect stack entry representing the position of the first bit of a bit string in a location in core. This is necessary because of dense storage and subscripting.

4) CHARACTER: A pointer to an Indirect Stack entry representing the position of the first character in a string referenced by a subscript.

CONST

1) A constant term that must be added to the value of an entry. This is used to keep track of constant terms in mixed mode expressions, and allows stack entries for constants to be dropped while avoiding incorporating the constant into the expression until necessary, thus permitting further constant folding.

2) For type RELATIONAL entries, a Phase 2 generated label for the location immediately after the test.

3) Used to chain together entries of the same SELECTYPE in multiple assignment statements.

COPT

Non-zero for a common sub-expression.

COPY

The number of dimensions of arrayness of an entry, or the number of copies of a structure. Notice that this may differ from DOCOPY because an arrayed expression can have simple variables in it (DOCOPY > COPY) or an ASSIGN parameter can be an array (COPY > DOCOPY).

DEL

1) WORK: If the entry represents the contents of a register saved in temporary storage, DEL is the number of entries using the register.

2) STRUCTURE: A pointer to the symbol table entry for its template.

3) CHAR: A pointer to an indirect stack entry for the position of the last character in a character string subscript reference.

4) MATRIX: An indexing value used to locate the non-adjacent entries in a matrix partition. The matrix elements are stored as a linear array, row by row. In a partition, certain elements are picked out of each row. Adding DEL to the last element picked out in a row will give the location of the first element to be picked out in the next row.

5) VECTOR: An indexing value to locate the non-adjacent entries in a column VECTOR.

DISP

A displacement used together with BASE for addressing an indirect stack entry.

FORM

1) The form of the entry:

0	
1	SYM
2	AIDX - 1-dimensional subscript index
3	VAC
5	LIT
6	IMD
7	CSYM
10	OFFSET
18	LBL
19	FLNO
20	STNO
22	EXTSYM
30	AIDX2 - 2-dimensional subscript index
31	WORK - stored VAC

This value helps determine the significance of the other fields.

2) In some special cases immediately before calling code emitting routines, such as EMITOP, FORM is set to an intermediate output code qualifier. This is done in SAVE_LITERAL and ARITH_BY_MODE.

INX

The index register associated with an indirect stack entry. If the register has been checkpointed, it is a negative pointer to an indirect stack entry pointing to the contents of the register in temporary storage.

INX_CON

1) A constant indexing term associated with the entry.

2) For formal parameters, this is the amount of storage necessary for passing * arrayness and character size information.

3) For EXTSYM's that are tasks, programs, or compools, it is the offset in PCEBASE of addressing information.

INX_MUL

When dealing with multi-dimensioned arrays, an attempt is made to forstall generating the code to do the multiply so that a comparison with existing registers can be made. INX_MUL is the accumulated constant multiplier.

INX_SHIFT

When describing a variable used as a subscript, there are two interesting values:

1) The value of the variable.

2) The appropriate offset.

Value 2 takes into account the width of the data item and is a multiple of value 1. Since this multiple is always a power of two, the multiplication can be done by shifting. INX_SHIFT is the required shift.

LOC

The significance of LOC depends on the indirect stack entry's form:

1) WORK: Pointer to a temporary storage entry.

2) SYM or LBL: Pointer to the symbol table entry represented by the indirect stack entry except for structure nodes where it is a pointer to the symbol table entry for the structure.

3) LIT: Pointer to associated Literal Table entry or -1 if literal is not in table.

4) FLNO: Phase 2 generated label number.

5) 0: The actual value of the entry.

6) AIDX: Pointer to the indirect stack entry set up as an index variable for a do loop to process a subscript.

7) AIDX2: Pointer to the indirect stack entry set up as an index variable for a do loop to process the second subscript in a two-dimensional reference.

8) EXTSYM: For EXTERNAL templates and procedures, the CSECT number. For tasks and programs, the PCEBASE.

LOC2

1) SYM: A pointer to the corresponding symbol table entry.

2) AIDX: Pointer to the indirect stack entry set up as an index variable for the do loop to process the second subscript in a two-dimensional reference.

REG

If TYPE(entry) = RELATIONAL then REG(entry) = condition code; otherwise, REG(entry) is the register containing the value of the entry.

ROW

Meaning depends on the entry's type:

1) Matrix: Size of the rows.

2) Bit or Character: The length of the string.

3) Integer or Scalar: 1.

4) Structure: EXTENT of its symbol table entry.

5) Vector: 1.

SIZE

LITERALLY 'ROW'.

STACK_MAX

The maximum size of the indirect stack.

STACK_PTR

Used for chaining together free indirect stack entries. Initially, each entry points to the next entry on the stack. As entries are allocated, their STACK_PTR becomes -1. STACK_PTR(0) points to the first free entry, and for free entries, STACK_PTR points to the next free entry.

STRUCT

0 - just an array
1 - structure with copies

If 1, the value is set to 0 after indexing is set up for the structure.

STRUCT_CON

A constant term used for addressing the terminal nodes of a structure. This term is later incorporated into INX_CON.

STRUCT_INX

A value used for determining how to compute index values for subscripted arrayed variables.

STRUCT_INX	Description
2	Array reference unconnected with a subscripted structure.
4	Array reference for a node of an arrayed structure with one copy after subscripting.
5	Array reference for a node of an arrayed structure or a subscripted structure where the subscript picks out several copies.

TYPE

The operand type of an indirect stack entry. If bit 3 is one, then double precision is specified; if zero, single precision. The numerical values for TYPE can be found in the table "operand types and properites" in the HALMAT section.

VAL

The meaning of VAL depends on the indirect stack entry form.

1) LIT: The literal's value. For character literals this is the LIT_CHAR pointer copied out of the LIT2 table.

2) STNO, LBL, FLNO: The statement number.

3) OFFSET: The value of the offset.

4) VAC: If the entry is used for emitting shaping function repeats, this is a register used in the process.

5) 0: The statement number of a label used for generating a "failure" conditional branch.

6) WORK: If the entry represents temporary storage for an integer or scalar shaping function, this is a pointer to the first entry in SF_RANGE containing arrayness information.

XVAL

The meaning of XVAL depends on the indirect stack entry's FORM:

1) LIT: For double precision literals VAL and XVAL together contain the literal's value.

2) SYM: a) If the entry is used for referencing arrayed structures, XVAL is AREASAVE.

b) If the entry represents a subscript in a two-dimensional reference, XVAL is a constant multiplier used for creating the indexes.

3) VAC: If the entry is used for emitting shaping function repeats, this is the index register used.

4) 0: The statement number of a label used for generating a successful conditional branch.

5) AIDX2: A constant multiplier used for generating two-dimensional subscript references. This usage is set in several places but used only in SEARCH_INDEX2.

5.1.4 REGISTER TABLE

One of the critical elements in optimization is eliminating redundant operations (i.e. loading a variable which is already in a register). The greater the optimization, the more record keeping is necessary. The HAL compilers go to great lengths including recognition of the fact that a multiply subscripted variable is already in a register. The appropriate information is kept in the register table.

A group of arrays of length REG_NUM, which describe the contents of the registers. These arrays are:

INDEXING: BIT(8)

This value indicates whether a register may be used as an index register or not.

R_BASE: FIXED

The contents of the register if it is used as a base register.

R_CON: FIXED

1) If the register contains a literal, this is the literal's value. For double precision literals R_CON and R_XCON together hold the literal's value.

2) Any constant terms that are to be added to the contents of a register are added to R_CON.

3) For registers of form SYM2, R_CON is the indexing constant associated with the first subscript.

R_CONTENTS: BIT(8)

The form of the register's contents (LIT, SYM, VAC, AIDX, XPT, POINTER, SYM2).

R_INX: BIT(16)

The index register associated with the contents of a register, or a negative pointer to an indirect stack entry representing the register if the register has been checkpointed.

R_INX_CON: FIXED

A constant indexing term associated with the register contents.

R_INX_SHIFT: BIT(8)

If R_CONTENTS is AIDX, this is the number of bits the contents must be shifted before indexing. The register contents corresponds to the number of data items to be indexed. Shifting the contents is equivalent to multiplying by the byte width of the operand type in the register to obtain the number of halfwords to be indexed.

R_MULT: BIT(16)

A constant multiplier used for two-dimensional arrayness references.

R_SECTION: BIT(8)

For a virtual base register, this is the number of the csect containing the variable(s) which required this base.

R_TYPE: BIT(8)

The operand type of the register's contents.

R_VAR: BIT(16)

Form of Contents	Significance of R_VAR
SYM	Pointer to the symbol table entry.
AIDX	Pointer to the indirect stack entry for the array index variable.
AIDX2/SYM2	Pointer to the indirect stack entry set up as an index variable for the do loop to process the second subscript in a two-dimensional reference.
XPT	Associated virtual base register number.
POINTER	Pointer to the symbol table entry of pointer type parameter or NAME variable.

R_VAR2: BIT(16)

If R_CONTENTS is AIDX2 or SYM2, R_VAR2 is a pointer to the indirect stack entry set up as an index variable for the do loop to process the second subscript in a two-dimensional reference.

R_XCON: FIXED

1) Used together with R_CON to hold the value of a double precision literal.

2) If R_CONTENTS is SYM2, R_XCON is the indexing constant associated with the second subscript in a two-dimensional reference.

USAGE: BIT(16)

Reflects the claims on a register. The number of claims on a register is the greatest integer of USAGE/2. An even value indicates that the contents of the register is unknown; an odd value indicates that it is known. A value of 1 means that the contents is known but not currently needed.

USAGE_LINE: BIT(16)

A pointer to the HALMAT operator being decoded when the register was last allocated. This value is used to decide which register to store when no free register is available.

5.1.5 Storage Descriptor Stack

A set of arrays of size LASTEMP. These arrays contain information about all the entries in temporary storage.

The arrays are:

ARRAYPOINT

Pointer chaining together all the temporary storage entries for a given do block, or 0 for the last temporary storage entry for that block. DOTEMP of each do level points to the beginning of the chain.

LOWER

Initial BIGNUMBER. The address of the beginning of a temporary storage entry; the lower bound of an entry in storage.

POINT

Pointer to the temporary storage entry that occupies the space following a given entry. POINT(0) always points to the first entry in this linked list. POINT of the last entry points to zero.

SAVEPOINT

An array of pointers to temporary storage entries that are no longer necessary.

SAVEPTR

A pointer to the last entry in SAVEPOINT.

UPPER

Points to the upper bound of an entry in storage. If less than or equal to 0, the temporary storage is not in use.

WORK_CTR

A pointer to the HALMAT operator word at the time storage is required.

WORK_USAGE

The number of indirect stack entries using the value in temporary storage when the value is the contents of a register. This number is necessary for determining which temporary space can be dropped.

5.1.6 DO Loop Descriptor Declarations

A group of arrays of length DOSIZE is used for storing information necessary for generating code for DO loops. The stack contains entries for each nested DO loop that is being processed.

DOBASE

Is an array of size 1 used for generating code for DO FOR loops. DOBASE is the base register used for addressing the index variable of the DO FOR loop. DOBASE(1) is a negative pointer to the indirect stack entry representing DOBASE if the index variable is a CSYM; otherwise, it is DOBASE.

DOCASECTR

Is the number of cases associated with a DO CASE statement.

DOFORCLBL

Is the LABEL ARRAY (entry = flow number) for the label pointing to the value of a discrete DO FOR loop entry.

DOINX

Is the index register used for addressing the index variable in a DO FOR loop. If the index variable is a CSYM, DOINX(1) is a negative pointer to the indirect stack entry set up for storing the contents of DOINX; otherwise it is DOINX.

DOLEVEL

DOLEVEL is a pointer to the stack entries for the do loop for which code is currently being generated. The zeroeth array entries are used as well as the entries with index DOLEVEL to describe the current DOLEVEL.

The array entries associated with each DO LOOP are:

DOFORFINAL

A pointer to a temporary storage location containing the final value of an iterative DO FOR loop.

DOFORINCR

A pointer to a temporary storage location containing the increment for iterative DO FOR loops.

DOFOROP

A pointer to the indirect stack entry for the index variable in a DO FOR loop.

DOFORREG

The register containing the value of the index variable for a DO FOR loop.

DOLBL

Pointer to the label array entry for a label marking the code following a DO loop. The label array entries following DOLBL are also used for DO loop code generation.

DOTEMP

Pointer to a chain of temporary storage entries for temporary variables in the DO loop. (See ARRAYPOINT.)

DOTYPE

```
The type of DO FOR loop:  0  if discrete loop
                          1  implicit increment of 1
                          2  explicit increment
```

DOUNTIL

A temporary storage location used to generate code so that a DO FOR loop is executed at least once before a DO UNTIL condition is tested. If no UNTIL clause, DOUNTIL = 0.

5.1.7 ARRAY-DO-LOOP Declarations

Two stacks are used to create the do loops implied by HAL/S arrayed statements. Arrayness is specified by a HALMAT ADLP or IDLP operator; some of the information associated with each stack entry is applicable to only one of these operators.

I. ARRAY REFERENCE STACK

A group of arrays of length DONEST used to keep track of information about array references at specific call levels. The stack entries are pointed to by CALL_LEVEL.

DOCOPY

The number of dimensions of arrayness of the context (cf. COPY).

DOCTR

Pointer to HALMAT ADLP operator.

DOFORM

The form of the reference:

Value	Description
0	All cases except those below.
1	Static Initialization.
2	Simple array parameter reference not followed by an expression and not part of an I/O routine. This is an interesting case because the parameter can simply be passed by reference with no iterative processing involved.

DOPTR

Pointer to the first entry in the Array DO LOOP Stack associated with the reference.

DOPTR#

A pointer to the array-do-loop stack entry associated with a subscript referenced by a HALMAT TSUB or DSUB operator.

DOTOT

Pointer to the last entry in the Array-Do-Loop Stack associated with the reference. (Equivalent to DOPTR(CALL_LEVEL) + DOCOPY(CALL_LEVEL.)

SDOLEVEL

The CALL_LEVEL at the beginning of the HALMAT ADLP operator processing.

SDOPTR

Pointer to the first entry in Array DO LOOP Stack associated with the reference.

SDOTEMP

Pointer to the first entry in a chain of temporary storage entries used in setting up the array do loops for a reference. The other entries in the chain are linked by ARRAYPOINT.

II. ARRAY DO LOOP STACK

A group of arrays of length DOLOOPS containing information about the do loops that are necessary for processing each dimension of arrayness. The entries in the Array DO LOOP Stack are pointed to by ADOPTR.

ADOPTR

A pointer to table entries for the most current DO LOOP that is being set up for array processing.

DOBLK

HALMAT block containing IDLP operator.

DOINDEX

For IDLP references, the actual value of an index variable which is compared with DORANGE. Otherwise, it is the pointer to an indirect stack entry for a register set up to be used as an index variable for the loop to process the dimension of arrayness.

DOLABEL

For IDLP references, it is a pointer to the current HALMAT operand. Otherwise, a statement number pointing to the code within the do loop.

DORANGE

For IDLP references, the array dimension minus one. Otherwise, a pointer to an indirect stack entry representing the size of the dimension.

DOSTEP

The increment used in the do loop. (Not applicable to IDLP.)

STACK#

A pointer into the SUBLIMIT array. It is 0 for an ordinary array reference. For a subscripted variable it is the array dimension + 1. In this way, if the subscripts are arrayed, STACK# points to the first SUBLIMIT entry containing information about the subscript's arrayness.

SUBLIMIT

An array used to contain information about the arrayness and size of a variable being subscripted and of the subscript. If the variable has n dimensions, the 0 - n-1st entries are the size of the 1st to nth dimensions and the nth entry is AREASAVE (= size of individual element). The n+1st to n+mth entries are the size of the m dimension of the subscript, if it is arrayed, and the n+m+1st entry is the subscript's AREASAVE.

SUBRANGE

1) Used as an array of temporary variables to set up SUBLIMIT.

2) The ith entry is used for the range of the ith subscript in a subscript reference.

5.1.8 HALMAT and Associated Material

This section describes the variables used in reading, decoding, and interpreting the HALMAT created by Phase I. HALMAT is described in the "HAL/S-360 Compiler System Specification", Appendix A.

Decoding HALMAT Instructions

General Declarations:

CODEFILE: The file created by phase 1 and massaged by phase 1.5 which contains the HALMAT instructions. The file is broken into blocks. All the HALMAT for a single HAL/S statement must fit in one block. Although the current block may be examined several times, previous blocks are never reread.

CURCBLK: The next block of CODEFILE to be referenced.

OPR: An array used for storing the HALMAT block currently being referenced.

CTR: A pointer to the HALMAT operator in OPR being decoded.

READ_CTR: Pointer to a HALMAT READ or RDAL instruction.

SMRK_CTR: Pointer to the next HALMAT SMRK instruction.

RESET: Pointers to HALMAT operators.

PP: The number of HALMAT operators converted by Phase 2.

Operator Word:

Phase 2	TAG	NUMOP	CLASS	OPCODE		0
HAL/S-360 Compiler Spec.	T	N	OP		P	0
	8	8	4 + 8		3	1

The P field has no Phase 1.5 name, but two of its values have Phase 1.5 mnemonics.

P Value	Phase1.5	HAL/S-360 Compiler Spec.
1	XN	N
2	XD	D

The P field on exit from phase 1.5 is used to convey code optimization information

P Value	Phase 2	Meaning
4	-	CSE (at least 2 references)

CLASS:

The class of the current HALMAT operator.

0 formatting, program organization, execution control, linkage, system control, subscripting

1 bit operations

2 character operations

3 matrix arithmetic

4 vector arithmetic

5 scalar arithmetic

6 integer arithmetic

7 conditional arithmetic

8 initialization

If the CLASS≠0, the eight bit OPCODE is broken down further into a three bit SUBCODE and a five bit OPCODE.

SUBCODE	OPCODE
3	5

SUBCODE: A value generated by Phase 2 used to classify opcodes within the same class.

Operand Word:

Phase 2	OP1	TAG3	TAG1	TAG2	1
HAL/S-360 Compiler Spec.	D	T1	Q	T2	1
	16	8	4	3	1

TAG2 extracted by X_BITS
TAG1 extracted by TAG_BITS
TAG3 extracted by TYPE_BITS

TAGS: Used to extract information from the general purpose tag field of a HALMAT SCHD operator. For this operator the tag field specifies the presence of options in the schedule statement.

Operand Qualifiers:

Value	Phase 2	HAL/S-360 Compiler Spec.
0		
1	SYM	SYT or SYL
2	INL	GLI or INL
3	VAC	VAC
4	XPT	XPT
5	LIT	LIT
6	IMD	IMD
7	no equivalent	AST
8	CSIZ	CSZ
9	ASIZ	ASZ
10	OFFSET	OFF

OPERATOR PROPERTIES

A collection of arrays of OPSIZE

	Value	Operator	Unary	Commutative	Condition	Reverse	Additive	Destructive	Arith (
	0	Load	0	0	7	0	0	0	"18"
	1	Store	0	0	0	0	0	0	"10"
	2	AND	0	1	0	0	0	1	"14"
	3	Or	0	1	0	0	0	1	"16"
	4	Not/EXOR	1	0	0	0	0	1	"17"
	5	Not Equal	0	0	4	≠	0	0	"19"
	6	Equal	0	0	3	=	0	0	"19"
	7	Not Greater Than	0	0	1	¬<	0	0	"19"
	8	Greater than	0	0	6	<	0	0	"19"
	9	Not less than	0	0	2	¬>	0	0	"19"
A	10	Less than	0	0	5	>	0	0	"19"
B	11	SUM	0	1			1	1	"1A"
C	12	MINUS	0	0			1	1	"1B"
D	13	Multiplication	0	1			0	1	"1C"
E	14	Division	0	0			0	1	"1D"
F	15	Exponentiation	0	0			0	1	"1C"
1	16	PREFIXMINUS	1	0			0	1	"13"
11	17	Integer Exponent	0	0			0	1	"1C"
12	18	Positive Integer Exponent	0	0			0	1	"1C"
13	19	ABS	1	0			0	1	"13"
14	20	Test	1	0			0	0	"18"
15	21	Exclusive Or	0	1			0	1	
16	22	Midval	0	0			0	1	

OPERAND TYPES AND PROPERTIES

A collection of arrays of size TYP_SIZE

Phase 2 Names	Description	Value	PACKTYPE	SELECTYPE	CHARTYPE	DATATYPE	CVTTYPE	BIGHTS	OPMODE	RCLASS	SHIFT
		0	1	5	5	0	0	2	3	2	1
BITS/BOOLEAN	Halfword bit	1	1	0	4	1	1	1	1	3	0
CHAR	Character	2	2	4	4	2	0	1	0	3	0
MATRIX	Single precision matrix	3	0	5	5	3	0	2	3	1	1
VECTOR	Single precision vector	4	0	5	5	4	0	2	3	1	1
SCALAR	Single precision scalar	5	3	2	2	5	0	2	3	1	1
INTEGER	Single precision integer	6	3	0	0	6	1	1	1	3	0
POINTER		7	3	0	0	7	1	2	1	3	0
		8	0	5	5	0	0	4	4	0	2
FULLBIT	Fullword bit	9	1	4	4	1	1	2	2	3	1
		10	1	4	4	1	1	1	0	3	0
	Double precision matrix	11	0	5	5	3	0	4	4	0	2
	Double precision vector	12	0	5	5	4	0	4	4	0	2
	Double precision scalar	13	3	3	3	5	0	4	4	0	2
DINTEGER	Double precision integer	14	3	1	1	6	1	2	2	3	1
EXTRABIT	Double word item used in SUBBIT context	15	1	5	4	1	1	2	2	3	1
STRUCTURE	Structure	16	4	0	0	0	0	4	5	0	0
EVENT	Event	17	1	0	4	1	1	1	1	3	0
CHARSUBBIT	Character subbit	18	1	5	4	1	1	1	0	3	

HALMAT Opcodes

XADD	"OB"	integer and scalar addition
XBNEQ	"7250"	operator for Bit String Comparison for Inequality
XCFOR	"0120"	operator for a DO FOR Condition Delimiter
XCSIO	"07"	used for generating calls to character I/O routines
XCSLD	"09"	used for generating calls to a character library routine
XCSST	"0A"	used for generating calls to a character library routine
XCTST	"0160"	operator for a DO WHILE/UNTIL delimiter
XDIV	"0E"	scalar division
XDLPE	"0180"	operator for an end arrayness specifier
XEXP	"0F"	scalar exponentiation
XEXTN	"0010"	operators which lists the multiple symbol table references required for referencing a structure variable
XFBRA	"00A0"	operator for branch on false
XFILE	"0220"	operator for file I/O
XICLS	"0520"	operators used to close an inline function block
XIDEF	"0510"	operator for the opening of an inline function block
XILT	"7CA0"	operator for integer less than comparison
XIMRK	"0030"	operator for an inline function statement marker
XMASN	"15"	used for generating calls to vector-matrix assignment routines
XMDET	"11"	used for matrix determinant routines
XMEXP	"19"	used for generating calls to matrix exponentiation routines
XMIDN	"13"	used for generating calls to identify matrix routines

XMINV	"0A"	used for generating calls to matrix inverse routines
XMTRA	"09"	for matrix transposes, also used for generating calls to the routine that performs this operation
XMVPR	"0C"	for vector-matrix products, also used for generatings calls to the appropriate library routine
XNOT	"04"	for logical not, also used as an index into the operator table
XOR	"03"	for logical or, also used as an index into the operator table to obtain information about the operator
XPASN	"18"	used for generating calls to matrix assignment routines
XPEX	"12"	for integer exponentiation
XRDAL	"0200"	operator for Readall I/O
XREAD	"01F0"	operator for Read I/O
XSASN	"14"	used for generating calls to vector-matrix assignment routines
XSFAR	"0470"	operator for shaping function arguments' reference
XSFND	"0460"	operator marking the end of a shaping function reference definition
XSFST	"0450"	operator marking the start of a shaping function reference definition
XSMRK	"0040"	operator for a Statement Marker
XVMIO	"16"	used for generating calls to vector-matrix I/O routines
XWRIT	"0210"	operator for write I/O
XXASN	"01"	for assignment
XXREC	"0020"	operating indicating the end of a HALMAT record
XXXAR	"0270"	operator marking an argument reference
XXXND	"0260"	operator marking the end of a reference definition
XXXST	"0250"	operator marking the start of a reference definition

Other Associated Variables

INITBLK (*nest level*)

The HALMAT block being referenced at each *nest level* of an initialization repetition specification. Used to backspace the HALMAT in order to perform a repetition.

INITCTR (*nest level*)

Pointer to the beginning of a repeated block of HALMAT initialization. The HALMAT is backspaced to this point once for each repetition (i.e. INITREPT (*next level*) times).

LEFTOP

Pointer to indirect stack entry for operand of HALMAT instruction.

LHSPTR

1) An index variable used to address the HALMAT operand words for the receivers in multiple assignment statements.

2) Used to reference the HALMAT operand words for time and event expressions.

NEWPREC

Precision of result specified by HALMAT instruction.

0: arbitrary
1: single
2: double

OPTYPE

Type of result of current instruction.

RESULT

Pointer to indirect stack entry representing the result of a HALMAT instruction, (e.g. result of a function call).

RIGHTOP

A pointer to the indirect stack entry for an operand of a HALMAT instruction.

SUBOP

The HALMAT operand word that is currently being decoded in a TSUB or DSUB instruction.

XD

Initial 2. A HALMAT operator word pseudo-optimizer tag field mnemonic for an arrayness specification.

XN

Initial 1. A HALMAT operator word pseudo-optimizer tag field mnemonic for an arrayness upshift stopper.

XPT

Initial 4. The HALMAT operand qualifier for an extended pointer. This pointer is used for referencing structure variables; the operand field is a pointer to the HALMAT EXTN operator listing the multiple symbol table references required to specify the variable.

5.1.9 Arguments of Procedures and Functions

ARG_NAME

1 if argument is a name parameter,
0 otherwise.

ARG_STACK

Pointer to the indirect stack entry which corresponds to each argument in the argument stack.

ARG_STACK#

Size of argument stack.

ARG_ STACK_PTR

Pointer to next free entry in argument stack.

ARG_TYPE

For list and shaping functions specifies repeat factor (cf. HAL/S-360 Compiler Spec., page A-60); otherwise, true if assign parameter.

ARGNO

The actual number of an argument (1,2,...) rather than its index in the argument stack.

ARGPOINT

A pointer to the symbol table entry for an argument.

ARGTYPE

The type of the entry pointed at by ARGPOINT.

FIXARG1

Initial 5, FC only. Register for use as an index register, for passing integer and bit parameters, and for returning integer and bit function values.

FIXARG2

Initial 6, FC only. Register for use as an index register and for passing integer and bit parameters.

FIXARG3

Initial 7, FC only. Register for use as an index register and for passing integer and bit parameters.

5.1.10 Runtime Stack Frame and Local Block Data Area

A runtime stack mechanism is used by the compiler to provide subroutine linkage area, temporary work areas, error vectors, and local storage for reentrant code blocks. The precise format of the runtime stack frame can be found in the compiler system specification.

Registers used for addressing the stack frame and associated data:

Register			
FC	360	Phase 2 Name	Contents
0	13	TEMPBASE	Points to the runtime stack frame of block in execution
1	10	PROGBASE	Points to the program level data base
2		PTRARG	Work addressing register used to pass address parameters and dereference name variables
3		PROCBASE	Used to address data local to the block in execution
4	14	LINKREG	Contains the return address for intrinsic or leaf procedure linkages

NARGINDEX is tne scope number of the current block and its index in the Block Definition Table.

TEMPBASE is a register which points to the beginning of the current runtime stack frame. Certain offsets from the beginning of the frame have been given mnemonic names to reflect their contents:

Phase 2 Name	Contents
REGISTER_SAVE_AREA	The caller's register save area is stored beginning at this offset.
STACK_LINK	The contents is the pointer to the preceding stack frame. This is the previous value of TEMPBASE.
NEW_LOCAL_BASE	The contents is the pointer to the current block's Local Data Area. This is equivalent to the current value contained in PROCBASE.
NEW_GLOBAL_BASE	The contents is the pointer to the current block's Program Data Area. This is equivalent to the current value contained in PROGBASE.
NEW_STACK_LOC	The next value of the stack pointer. This value is set when a procedure is entered, except when no new procedure is to be called. (Used only when SCAL linkage is not used.)
STACK_FREEPOINT	The first location following the register save area. The contents of the caller's floating point register are saved starting at this offset.

Phase 2 Name	Contents
ERRSEG(NARGINDEX)	The displacement in the frame where the error vector starts.
WORKSEG(NARGINDEX)	The displacement in the frame where the work area starts.
MAXTEMP(NARGINDEX)	The maximum space occupied by a run-time stack frame. The displacement of the end of the frame.

The code for setting up a new run-time stack frame when a block is entered is generated by BLOCK_OPEN.

The ERROR VECTOR in a runtime stack frame contains an entry of 2 halfwords for each ON ERROR statement in the block. The information contained in the error vector is contained in the Error Stack and augmented by the Block Definition Table entry for the runtime stack frame.

The Block Definition Table provides the following information:

ERRPTR: A pointer to the first error stack entry associated with the block.

ERRSEG: The displacement of the beginning of the error vector within the runtime stack frame.

ERRALLGRP: 1 if there is an ON ERROR*:* statement in the block, 0 otherwise.

ERRALL: The number of error groups for which an ON ERROR$_{group:*}$ statement appears in the block.

MAXERR: The number of errors for which ON ERROR statements exist in the block.

The information in the Block Definition Table is used primarily for determining the displacement of each error within the vector. This is done in the procedure SET_ERRLOC. The errors are arranged so that entries for single errors in a group are at the beginning of the vectors. These are followed by entries for error groups with all errors on. The last entry indicates the action to be taken if all errors are on at some point during the block's execution.

The Error Vector Entries have the format shown below:

<table>
<tr><td>A$_{(4)}$</td><td>Error Number$_{(6)}$</td><td>Error Group$_{(6)}$</td></tr>
<tr><td></td><td colspan="2">Address (16 bits)</td></tr>
</table>

The displacement of an Error Vector is in the Error Stack ERR_DISP field.

Error Group
Error Number } This information is in the Error Stack's ERR_STACK field.

A:	0000	GO TO Address
	XX01	SYSTEM
	XX11	IGNORE
	00XX	No event action
	01XX	SET
	10XX	RESET
	11XX	SIGNAL

} Determined in GENERATE from the HALMAT ERON instruction.

Address: The address of an Event Variable or GO TO.

The Local Block Data Area

A Block Data Area may exist for any program, procedure, function, task, or update block. The Block Data Areas are created by Phase 2 of the compiler and are part of DATABASE, the program level data CSECT. Storage is allocated for the Block Data Area by INITIALIZE, and the address of the area is stored in the block's SYT_ADDR entry. Register 3, PROCBASE, is loaded with the address of the Block Data Area for the block being entered by the compiler code emitted by BLOCK_ENTRY. PROCBASE points to the Local Block Data Area for the block in execution. The previous values of PROCBASE are saved in the runtime stack frames.

The Block Data Area consists of two or five consecutive halfword locations. The values stored in the first two locations are determined by the procedure BLOCK_OPEN, the remaining ones by BLOCK_CLOSE. The format of a Block Data Area is shown below, followed by an explanation of each field and the Phase 2 variables containing the information.

<table>
<tr><th></th><th>Fields</th><th colspan="3"></th><th></th></tr>
<tr><td>PROCBASE →</td><td>1</td><td colspan="3">Block ID</td><td></td></tr>
<tr><td></td><td>2</td><td>XU</td><td>ONERRS</td><td>ERRDISP</td><td></td></tr>
<tr><td></td><td>3</td><td>TYP</td><td>UNUSED</td><td>RESERVE SVC#</td><td rowspan="3">} Only required if XU = 1.</td></tr>
<tr><td></td><td>4</td><td colspan="2">UNUSED</td><td>RELEASE SVC#</td></tr>
<tr><td></td><td>5</td><td colspan="3">LOCK ID</td></tr>
</table>

Assume that BLOCK is a pointer to the block's symbol table entry, and SCOPE is the block's SYT_SCOPE.

	Field	Phase 2 Reference	Definition
1.	Block ID		A 16 bit field uniquely identifying the block.
		CMPUNIT_ID	(9 bits). The first 9 bits are a user supplied compilation unit number.
		SYT_SCOPE(BLOCK)	(7 bits). The last 7 bits are the compiler generated number identifying each block. This is to provide a pointer to the information about the block in the Block Definition Table.
2.	XU	CALL#(SCOPE)=4	(1 bit) EXCLUSIVE_UPDATE flag. Set to 1 if the block is either an EXCLUSIVE or UPDATE block. This is indicated by a CALL#(SCOPE) of 4.
	ONERRS	MAXERR(SCOPE)	(6 bits). The number of discrete errors for which an ON ERROR statement exists in the block.
	ERRDISP	ERRSEG(SCOPE)	(9 bits). The displacement in halfwords from the beginning of the block's runtime stack frame to the error vector.
3.	TYP	Determined from SYT_CONST(BLOCK)	(1 bit). Set to zero for EXCLUSIVE functions or procedures. For update blocks, set to one if shared data variables are read only, and set to zero if shared data variables are to be written.

Field	Phase 2 Reference	Definition
RESERVE SVC#		(8 bits). SVC number for the reserve SVC: 15 for a code block 16 for a data area
4. RELEASE SVC#		(8 bits). SVC number for release SVC: 17 for a code block 18 for a data area
5. LOCK ID		(15 bits). Indicates which code blocks or data areas are being used.
	The contents of an offset in EXCLBASE	For an EXCLUSIVE block, it is the address of its EXCLUSIVE data CSECT.
	SYT_CONST(BLOCK) &"7FFF"	For a data area, it is a bit string specifying which lock groups are involved.

5.1.11 Vector-Matrix Optimization

The temporary storing of the result of a HALMAT vector-matrix operation immediately before an assignment can be eliminated if certain conditions hold. A detailed description of these conditions may be found in the HAL/S-FC Compiler Specification, Section 3.1.5.5.

The variables associated with vector-matrix optimization can be grouped in the following way:

I. Global Flags:

NO_VM_OPT
A compiler option specifying that vector-matrix optimization is not required. In this case, some unnecessary temporary stores for the results of vector-matrix operations will be generated.

ALL_FAILS
True if optimization probably not possible.

OK_TO_ASSIGN
True if optimization probably possible.

II. Variables Associated with the HALMAT Operation:

STMT_PREC
True if either operand is double precision.

CLASS1_OP
True if the operation is in Class 1. This class only includes raising a matrix to the 0th power.

CLASS3_OP
A flag indicating that the operation is in class 3. Class 3 operations include matrix-scalar and vector-scalar multiplication and division, vector-matrix addition and subtraction, vector and matrix negation, and the built-in function UNIT.

SRCE
A pointer to the value being assigned in a vector or matrix assignment statement.

ASNCTR
A pointer to the HALMAT assignment operation following the V-M operation to be optimized.

Variables associated with either operand while its properties are being determined:

OPER_SYMPTR — A pointer to the symbol table entry of an operand.

OPER_PARM_FLAG — A flag used to indicate whether an operand of a vector or matrix instruction is a parameter.

START_PART — The offset used to find the beginning of a matrix partition for an operand in a matrix instruction that is being considered for vector-matrix optimization.

SRCEPART_SIZE — The extent of the partition of an operand in a matrix instruction being considered for vector-matrix optimization.

NAME_OP_FLAG — A flag used to indicate whether the last HALMAT operator is a name variable. (Name variables that are operands stored in the temporary work area cause this flag to be false.)

VAC_FLAG — A flag used to indicate whether the last operand decoded is in the temporary work area.

SUBSTRUCT_FLAG — A flag used to indicate whether the last operand examined is a terminal of a subscripted structure.

The properties associated with the operands are:

LEFT_NSEC or RIGHT_NSEC

A flag used to indicate that the left-hand (or right-hand) operand of a vector-matrix operation is in temporary storage.

LNON_IDENT or RNON_IDENT

A flag used to indicate that the left (or right) operand of a vector-matrix instruction and the receiver of an immediately following assignment statement with one receiver are not identical.

LEFT_DISJOINT or RIGHT_DISJOINT

A flag used to indicate that the left (or right) operand of a vector-matrix operation and a suitable receiver of an assignment statement are disjoint.

III. Variables Associated with the Receiver:

RECVR

A pointer to the indirect stack entry for the receiver in an assignment statement with a single receiver.

RECVR_SYMPTR

The pointer to the symbol table entry for the receiver for an assignment statement being considered for vector-matrix optimization.

RECVR_NEST_LEVEL

The nest level of the receiver in an assignment statement being considered for vector-matrix optimization.

RTYPE

A flag used to indicate the precision of the receiver in an assignment statement with single receivers.

START_OFF

The offset used to find the beginning of a matrix partition for the receiver of a matrix assignment statement being considered for vector-matrix optimization.

PART_SIZE

The extent of the indexing term associated with a partitioned matrix receiver in an assignment statement used for vector-matrix optimization.

Intermediate flags associated with the Receiver:

REMOTE_RECVR

A flag used to indicate that the receiver in an assignment statement with a single receiver has the REMOTE attribute.

Flags associated with the Receiver:

INX_OK

A flag used to indicate that a receiver in an assignment statement with a single receiver does not have variable subscripting.

NONPART (¬VAC_FLAG & ¬SUBSTRUCT_FLAG)

A flag used to indicate whether the receiver in an assignment statement with a single receiver is non-partitioned.

ASSIGN_PARM_FLAG

A flag used to indicate whether the receiver in an assignment statement with a single receiver is an assign parameter.

RECVR_OK(¬NAME_OP_FLAG & ¬REMOTE_RECVR & ¬SUBSTRUCT_FLAG)

A flag used to indicate that the receiver in an assignment statement with a single receiver is not a REMOTE or NAME variable, and is not a terminal of a subscripted structure.

5.1.12 Other Useful Compendia

Register Names Used by Phase 2

Register	Phase 2 Reference Number	Names
R0	0	TEMPBASE
R1	1	PROGBASE, SYSARG0
R2	2	PTRARG, SYSARG1
R3	3	PROCBASE, SYSARG2
R4	4	LINKREG
R5	5	FIXARG1
R6	6	FIXARG2
R7	7	FIXARG3
F0	8	FR0
F1	9	FR1, REMOTE_BASE
F2	10	FR2
F3	11	
F4	12	FR4
F5	13	
F6	14	FR6
F7	15	FR7

Operand Qualifiers Declared in Phase 2

Operand qualifiers are used in Phase 2 by HALMAT operand words, Indirect Stack Entries, the Register Table, and the Intermediate Code File to classify the operands and give significance to the other operand fields. The operand qualifiers used by each table do not form groups with mutually exclusive names or values. The table below lists the possible qualifiers values, their Phase 2 names if they exist, and which tables use them.

		USERS			
Value	Phase 2 Mnemonic	HALMAT Operands	Indirect Stack Entries	Register Table	Intermediate Output Code
0			✓		✓
1	SYM	SYT/SYL	✓	✓	✓
2	INL	GLI/INL			
	AIDX		✓	✓	
3	VAC	VAC	✓	✓	
4	XPT	XPT		✓	X
5	LIT	LIT	✓	✓	
6	IMD	IMD	✓		
7	CSYM	X(AST)	✓		✓
	POINTER			✓	
8	CHARLIT				✓
	CSIZ	CSZ			
9	ASIZ	ASZ			X
10	EIXLIT				✓
	OFFSET	OFF	✓		
11-13					✓
14	NOT USED				
15	CLBL				✓
16	ADCON				✓
17	LOCREL				✓
18	LBL		✓		✓
19	FLNO		✓		✓
20	STNO		✓		✓
21	SYSINT				✓
22	EXTSYM		✓		✓
23	SHCOUNT				✓
24-28	NOT USED				
29	SYM2			✓	
30	AIDX2		✓		
31	WORK		✓		

✓ Qualifier value is used.   X Qualifier is used, but has no Phase 2 mnemonic.

(For HALMAT operands, ✓ and X have been replaced or supplemented by the mnemonic used in the HAL/S-360 Compiler Spec., Appendix A.)

Intermediate Output Code Opcodes

32	RRTYPE	45	PDELTA
33	RXTYPE	46	C_STRING
34	SSTYPE	47	CODE_END
35	DELTA	48	PLBL
36	ULBL	49	DATA_LIST
37	ILBL	50	SRSTYPE
38	CSECT	51	CNOP
39	DATABLK	52	NO
40	DADDR	53	NADDR
41	PADDR	54	PROLOG
42	LADDR	55	ZADDR
43	RLD	56	SMADDR
44	STMTNO		

Opcode Construction

Declarations Involved:

ARITH_OP, OPMODE, MODE_MOD, RR, RX, RI.

ARITH_OP: An array giving the basic RR opcode for each of the OPSIZE operators in the operator table.

OPMODE: An array giving the operation mode for each operand type:

OPMODE	Operand Type
0	Character
1	Halfword bit, single precision integer
2	Fullword bit, double precision integer
3	Single precision scalar
4	Double precision scalar
5	Structure

There are four instruction types:

```
     RR initial(0)
     RX initial(5)
     RI initial(10)
     RS initial(15)
```

which can take on the above modes. The instruction type and the mode are added together to get an index into the MODE_MOD array. This array provides a value used for modifying the basic opcode.

The complete sequence of operations for generating an opcode is:

```
  ARITH_OP(operator+MODE_MOD(OPMODE(operand type)+instruction type)
```

This results in a two digit hex opcode whose first digit indicates the instruction mode whose second digit indicates the operator. (This method cannot be used for all opcode generation.)

	MODE_MOD	1^{st}Hex Digit	Description
0	0		
1	0	1	RR
2	0	1	RR
3	"20"	3	Single precision RR
4	"10"	2	Double precision RR
5	1		Illegal on FC
6	"30"	4	Halfword RX
7	"40"	5	Fullword RX
8	"60"	7	Single precision RX
9	"50"	6	Double precision RX
10	0		
11	"90"	A	Halfword RI
12	0		
13	0		
14	0		
15	0		
16	0		
17	"BO"	C	Fullword RX to storage

5.1.13 Alphabetical Listing of Global Phase 2 Data

A	Initial("5A"). Opcode used for code generation.
ABS	label
ADCON	Initial(16). An intermediate code qualifier which indicates an address constant to be used as a displacement in RX instructions.
ADD	Initial("0B"). An operator code for addition used as an index into the tables of properties of operators.
ADDITIVE	See HALMAT Operator properties.
ADDR_FIXED_LIMIT	Common value passed from Phase 1.
ADDR_FIXER	Common value passed from Phase 1.
ADDR_ROUNDER	Common value passed from Phase 1.
ADDRESS_STRUCTURE	label
ADDRS_ISSUED	A flag indicating whether the source statement number of the current statement has been output to the intermediate output code file.
ADJUST	label
ADOPTR	See Array Do Loop declarations.
AH	Initial("4A"). Opcode used in code generation.
AHI	Initial("AA"). Opcode used in code generation.
AIDX	Initial (2). Array index. One of the possible forms of an indirect stack entry.
AIDX2	Initial (30). An indirect stack entry form for a two-dimensional array index.
ALCOP	literally RESULT
ALL_FAILS	See Vector-Matrix.
ALLOCATE_TEMPORARY	label

ALWAYS

Initial (7). Used in generating branch instructions to represent a test condition of 7 (always branch).

AM

Addressing mode field of an RS format instruction.

AND

See HALMAT operator properties.

ANY_LABEL

Initial ("40"). One of the entry types in the symbol table used to distinguish label entries (type ⩾ "40") from other entries.

AP101INST

Array of size OPMAX. Array of AP-101 opcodes indexed by the corresponding 360 opcode.

AR

Initial ("1A"). Opcode used in code generation.

AREA

Product of AREASAVE and the size of each dimension arrayness of an operand.

AREASAVE

A number used as a basis for computing the area a terminal operand occupies since the product of AREASAVE and the bytes the operand's optype occupies gives the value determined by the operand's packtype.

PACKTYPE	AREASAVE
0 Matrix/Vector	ROW x COLUMN or Number of components
1 Bit	1
2	1/2 (length+2) + (length+2) & 1
3 Integer/Scalar	1
4 Structure	

ARG_ASSEMBLE

label

ARG_COUNTER

Array of size CALL_LEVEL#. See Call Stack.

ARG_NAME

See Arguments.

ARG_POINTER

Array of size CALL_LEVEL#. See Call Stack.

ARG_STACK	Array of size ARG_STACK#. See Arguments.
ARG_STACK_PTR	See Arguments.
ARG_TYPE	See Arguments.
ARG#	Dummy variable used as an index in do-loops that access all arguments of a procedure, function I/O routine.
ARGFIX	See Arguments.
ARGNO	See Arguments.
ARGPOINT	See Arguments.
ARGTYPE	See Arguments
ARITH_BY_MODE	label
ARITH_OP	See HALMAT operator properties.
ARRAY_FLAG	A flag indicating that a conditional operation is occurring during array processing. This means that code for closing the loops set up for array processing must precede any code for conditional branches.
ARRAY_INDEX_MOD	label
ARRAYNESS	Number of array dimensions of an operand.
ARRAYPOINT	Array of size LASTEMP. See Storage Descriptor Stack.
ARRAY2_INDEX_MOD	label
ARRCONST	The product of AREASAVE and an offset computed from the array dimensions of an operand. The offset is computed as follows: (N_i is the i^{th} array dimension).

# Dimensions	Offset
0	0
1	-1
2	$(-1\ N_2) - 1$
3	$((-1\ N_2) - 1\ N_3) - 1$

ASIZ — See HALMAT decoding.

ASNCTR — See Vector-Matrix.

ASSEMBLER_CODE — 1 if assembler code listing of program that is being compiled has been requested, 0 otherwise.

ASSIGN_FLAG — See Symbol Table SYT_FLAGS.

ASSIGN_HEAD — Array of size 5. Used for scalar and integer assignment. The entries are indexed by selectype, and each entry points to the first entry in a chain of operands of the same selectype that are to be assigned with the same value.

ASSIGN_OR_NAME — See Symbol Table SYT_FLAGS.

ASSIGN_PARM_FLAG — See Vector-Matrix.

ASSIGN_START — Array of size 4; initial (0,6,12,18,24). Used for integer and scalar assignment to determinate the order in which conversion should be done. The entries are indexed by the SELECTYPE of the right side of the assignment. They provide an index into ASSIGN_TYPES.

ASSIGN_TYPES — Array of size 23; initialized. Used for integer and scalar assignment. ASSIGN_START provides an index into this array which is then used to determine in what order any conversions necessary to carry out assignment should be made.

AUTO_FLAG — See Symbol Table SYT_FLAGS.

AVAILABLE_FROM_STORAGE — label

B — Current base register. (Used in generating object code.)

BACKUP_REG — Array of size STACK_SIZE . See Indirect Stack.

BAL — Initial ("45"). Opcode used for code generation.

BALR — Initial ("05"). Opcode used for code generation.

BASE
Array of size STACK_SIZE.
See Indirect Stack

BC
Initial ("47"). Opcode used for code generation.

BCF
Initial ("87"). Opcode used for code generation.

BCR
Initial ("07"). Opcode used for code generation.

BCRE
Initial ("0F"). Opcode used for code generation.

BCT
Initial ("46"). Opcode used for code generation.

BCTR
Initial ("06"). Opcode used for code generation.

BD_BASE_REGS
The location in DATABASE of the values of the virtual base registers.

BEGIN_SF_TABLE
label

BIFCLASS
Array of size BIFNUM; initialized. An array giving the class of each built-in function. The classes are:

0 Arithmetic Functions
1 Algebraic Functions
2 Vector-Matrix Function
3 Character Functions
4 Supervisor Built-in Functions

BIFNAMES
Character array; initialized. An array used for generating the names of library or external routines that perform built-in functions. The names in this array are prefixed to specify precision, argument type.

BIFOPCODE Array of size BIFNUM initialized. This array gives the index of the built-in function name in BIFNAMES.

BIFREG Array of size 3; initial (8,10,5,6). Used to determine what registers to use for arguments of arithmetic built-in function. Registers 8 and 10 are used for scalar operands; registers 5 and 6 are used otherwise.

BIFTYPE Array of size BIFNUM; initialized. This array gives the type of each built-in function.

BIGHTS Array of size TYP_SIZE; initialized. The number of halfwords occupied by an item of each data type.

BIT_MASK label

BIT_SHIFT label

BIT_STORE label

BIT_SUBSCRIPT label

BITESIZE Initial (16). 16 bits. Used to compute storage.

BITS Initial (1). The halfword bit operand type.

BLANK Initial (' ').

BLOCK_CLASS Array of size (11); initialized. Array with an entry for each symbol table class with value 1 if class is a label name, and 0 if it is a data name.

BLOCK_CLOSE label

BLOCK_OPEN label

BOOLEAN	Initial (1). The halfword bit operand type.
BOUNDARY_ALIGN	label
BYTES_REMAINING	The number of character positions left in the current card.
CALL_CONTEXT	Array of size CALL_LEVEL#. See Call Stack.
CALL_LEVEL	The current nest level; 0 for procedure calls and $\geq$ 1 for nested function invocations.
CALL#	Array of size PROC#. See Block Definition Table.
CARDIMAGE	See COLUMN.
CASE2SET	Array of size VMOPSIZE used by VMCALL to determine which operand's dimensions contain all necessary information for the subroutine call.
CCREG	A number describing the side effects of an instruction on the condition code: CCREG<0 indicates a logical condition code. CCREG=0 indicates the condition code is no longer valid. CCREG>0 indicates the register affecting the condition code.
CH	Initial ("49"). Opcode used for code generation.
CHAR	Initial (2). The character operand type.
CHAR_CALL	label
CHAR_CONVERT	label
CHAR_INDEX	label
CHAR_SUBSCRIPT	label
CHARACTER_TERMINAL	label
CHARLIT	Initial (18). An intermediate output code qualifer referring to the character literal pool.

CHARSTRING	Used to build up part of a line of assembler code for output as the assembler listing.
CHARSUBBIT	Initial (18). The operand type for character strings referenced as bit strings.
CHARTYPE	See HALMAT Operand types.
CHECK_ADDR_NEST	label
CHECK_AGGREGATE_ASSIGN	label
CHECK_ASSIGN	label
CHECK_ASSIGN_PARM	label
CHECK_CSYM_INX	label
CHECK_LOCAL_SYM	label
CHECK_LOCK#	label
CHECK_NAME_ARG	label
CHECK_REMOTE	label
CHECK_SI	label
CHECK_SRCE	label
CHECK_SRS	label
CHECK_STRUCTURE_PARM	label
CHECK_VAC	label
CHECK_VM_ARG_DIMS	label
CHECKPOINT_REG	label
CHECKSIZE	label

CHI	Initial "A9". Opcode used for code generation.
CLASS	See HALMAT decoding.
CLASS_B	Initial (110). Errors resulting in compiler termination.
CLASS_BS	Initial (9). Error resulting in compiler termination due to stack size limitations.
CLASS_BX	Initial (n). Compiler Error.
CLASS_D	Initial (18). Declaration errors.
CLASS_DI	Initial (23). Declaration error: initialization.
CLASS_DQ	Initial (112). Declaration error: structure template tree organization.
CLASS_DU	Initial (100). Declaration error: undeclared data.
CLASS_E	Initial (29). Expression errors.
CLASS_EA	Initial (30). Expression error: arrayness.
CLASS_F	Initial (115). Formal parameters and arguments error.
CLASS_FD	Initial (37). Formal parameter and arguments error due to dimension agreement.
CLASS_FN	Initial (38). Formal parameter and argument error: number of arguments.
CLASS_FT	Initial (40). Formal parameter and argument error: type agreegment.
CLASS_PE	Initial (95). Program control and internal consistance error: external templates.

CLASS_PF	Initial (58). Program control and internal consisteance error: function return expressions.
CLASS_QD	Initial (69). Shaping function dimension information error.
CLASS_RT	Initial (97). Real time statement error timing expression.
CLASS_SR	Initial (76). Subscript usage error: range of subscript values.
CLASS1_OP	A flag indicating that a vector-matrix operation is a Class 1 operation. This class only includes raising a matrix to the 0th power.
CLASS3_OP	A flag indicating that a vector-matrix operation is a Class 3 operation. Class 3 operations include matrix-scalar and vector-scalar multiplication and division, vector-matrix addition and subtraction, vector and matrix negation, adn the built-in function UNIT.
CLBL	Initial (15). An intermediate code qualifier indicating the address of the beginning of a data area containing the address of the beginning of the code for each case in a DO CASE statement.
CLEAR_CALL_REGS	label
CLEAR_NAME_SAFE	label
CLEAR_R	label
CLEAR_REGS	label
CLEAR_SCOPED_REGS	label
CLEAR_STMT_REGS	label

CLOCK

Array of size 2.

CLOCK(0): time at beginning of phase 2.
CLOCK(1): time at end of phase 2 set up.
CLOCK(2): time at end of phase 2 generation.

CMPUNIT_ID

A user supplied number used to identify a compilation unit.

CNOP

Initial (51). An intermediate code qualifier indicating how to align data areas to proper boundaries.

CODE

Array of size CODE_SIZE. Array containing the block of intermediate code which is currently being referenced or modified.

CODE_BASE

The lowest line from the intermediate code file in CODE.

CODE_BLK

The block of the intermediate output code file which is currently in CODE.

CODE_END

Initial (47). An intermediate code qualifier indicating the end of a compilation unit.

CODE_LIM

The highest line from the intermediate code file in CODE.

CODE_LINE

The line of intermediate code that is currently being referenced, added, or modified. CODE_LINE is an absolute value relative to all the lines of code generated, it is not a pointer into CODE.

CODE_LISTING_REQUESTED

A compiler option: 1 if code listing is requested, 0 otherwise.

CODE_MAX

the maximum number of lines of code in the intermediate output file.

CODEFILE

See HALMAT decoding.

COLON

Initial (:).

COLUMN(OBJECT GENERATOR)

An array used to set up card images to be output. CARDIMAGE(I) are the four bytes of COLUMN starting at COLUMN(4*(I-1)). DUMMY_CHAR is built to be a descriptor pointing to column so that COLUMN can be output as a normal character string.

COLUMN(otherwise) See Indirect Stack.

COMMA Initial (,).

COMMON_SYTSIZES Array of size #COM_SYTSIZES used by storage_mgt for dynamic allocation.

COMMUTATIVE See HALMAT operator properties.

COMMUTFM label

COMPACT_CODE A compiler option.

COMPARE Initial "05". An Operator Code for comparison used as an index into the table of properties of operators.

COMPARE_STRUCTURE label

COMPILER A character string indicating the compiler type.

COMPOOL_LABEL See Symbol Table SYT_TYPE.

CONDITION See HALMAT Operator Properties.

CONST See Indirect Stack.

CONSTANT_CTR Pointer to the constant table entry for the last literal put into the constant area.

CONSTANT_FLAG See Symbol Table SYT_FLAGS.

CONSTANT_HEAD For each opmode, a pointer to the beginning of the last of literal pool entries for that opmode.

CONSTANT_PTR Array of size CONST_LIM.
In GENERATE, a pointer to the next constant of the same opmode in the constant area. GENERATE_CONSTANTS overwrites the pointer with the actual address of the constant.

CONSTANTS — Array of size CONST_LIM. The value of the literals in the constant area. For double precision literals, the i^{th} and $i+1^{st}$ entries together contain the value.

COPT — See Indirect Stack.

COPY — See Indirect Stack.

COSTBASE

COUNT#GETL

CS — label

CSE_FLAG — A flag indicating whether or not a HALMAT instruction is a common subexpression.

CSECT — Initial (38). An intermediate code opcode which switches processing from one control section to another or switch origins within control sections.

CSIZ — See HALMAT decoding.

CSTRING — Initial (46). An intermediate code opcode indicating character data.

CSYM — Initial (7). An indirect stack entry form indicating a symbolic reference which is referenced by its own base and displacement rather than letting these values be computed dynamically during object code generation.

CTON — label

CTR — See HALMAT decoding.

CTRSET — Array of size VMOPSIZE used by VMCALL to break the possible opcodes into four classes for further processing.

CURCBLK — See HALMAT decoding.

CURLBLK — The literal file block that is currently being referenced.

CURRENT_ESDID — The CSECT for which object code is currently being generated.

CVFL — Initial ("3F"). Opcode used for code generation.

CVFX — Initial ("1F"). Opcode used for code generation.

CVTTYPE — No code is required to convert between types if their CVTTYPEs are the same. See HALMAT "operand types and properties".

D — The displacement used in base-displacement addressing during object code generation.

DADDR — Initial (40). Data address; an intermediate code opcode indicating an address constant which refers to a specified absolute position within a CSECT.

DATA_LIST — Initial (49). An intermediate code opcode indicating local code list control.

DATA_WIDTH — The data width of a vector or matrix element in halfwords; 2 for single precision operands, and 4 for double precision operands.

DATABASE — Array of size 1. DATABASE(0) is the ESDID number of the CSECT which contains static data without the REMOTE attribute; DATABASE(1) initially indicates the existence (1) of remote data. If there is remote data, DATABASE(1) will be set to the ESDID of the CSECT for remote data by SETUP_REMOTE_DATA.

DATABLK — Initial (39). An intermediate code opcode used to indicate the definition of one or more full words of data.

DATALIMIT — The last CSECT number assigned for REMOTE data for EXTERNAL templates.

DATATYPE

Extracts essential information about a type (e.g. double and single precision types have same DATATYPE, EVENT and BOOLEAN have same DATATYPE. See HALMAT operand types.

DECK_REQUESTED

A compiler option: 1 if deck requested,
0 otherwise.

DECLMODE

A flag which is set at the beginning of a block, and reset to zero at the end of the declarations for the block. This is to ensure that any code generated during variable initialization is not intermixed with the data.

DECODEPIP

label

DECODEPOP

label

DEFINE_LABEL

label

DEFINED_BLOCK

See Symbol Table SYT_FLAGS.

DEFINED_LABEL

See Symbol Table SYT_FLAGS.

DEL

Array of size STACK_SIZE.
See Indirect Stack.

DELTA

Initial (35). An intermediate code opcode indicating a value used to modify the address of the following instruction.

DENSE_FLAG

See Symbol Table SYT_FLAGS.

DENSEADDR

The address in the data CSECT of a data item requiring dense initialization.

DENSESHIFT

The number of bit positions an initial value for a bit data item with dense initialization must be shifted so that it is at the proper bit position in its location in core.

DENSETYPE

The data type of a data item requiring dense initialization.

DENSEVAL

The initial values of the data items requiring dense initialization that are to be stored in the same word after the initial values have been shifted appropriately so that they are at the proper positions.

DESC	Literally 'STRING'; magic XPL conversion function for descriptors.
DESCENDENT	label
DESTRUCTIVE	See HALMAT Operator properties.
DIAGNOSTICS	1 if diagnostics are required, 0 otherwise.
DIMFIX	label
DINTEGER	Initial (14). Double precision integer operand type.
DISP	Array of size STACK_SIZE. See Indirect Stack.
DO_ASSIGNMENT	label
DOBASE	See Do Loop Descriptor Declarations.
DOBLK	Array of size DOLOOPS. See array Do Loop Declarations.
DOCASECTR	See Do Loop Descriptor Declarations.
DOCLOSE	label
DOCOPY	Array of size DONEST. See Array Do Loop Declarations.
DOCTR	Array of size DONEST. See Array Do Loop Declarations.
DOFORCLBL	See Do Loop Descriptor Declarations.
DOFORFINAL	Array of size DOSIZE. See Do Loop Descriptor Declarations.
DOFORINCR	Array of size DOSIZE. See Do Loop Descriptor Declarations.

DOFORM	Array of size DONEST. See Array Do Loop Declarations.
DOFOROP	Array of size DOSIZE. See Do Loop Descriptor Declarations.
DOFORREG	Array of size DOSIZE. See Do Loop Descriptor Declarations.
DOFORSETUP	label
DOINDEX	Array of size DOLOOPS. Set Array Do Loop Declarations.
DOINX	See Do Loop Descriptor Declarations.
DOLABEL	Array of size DOLOOPS. See Array Do Loop Declarations.
DOLBL	Array of size DOSIZE. See Do Loop Descriptor Declarations.
DELEVEL	The number of nested do loops at any point during code generation. See Do Loop Descriptor Declarations.
DOMOVE	label
DOOPEN	label
DOPTR	Array of size DONEST. See Array Do Loop Declarations.
DOPTR#	See Array Do Loop Declarations.
DORANGE	Array of size DOLOOPS. See Array Do Loop Declaration.
DOSTEP	Array of size DOLOOPS. See Array Do Loop Declarations.
DOTEMP	Array of size DOSIZE. See Do Loop Descriptor Declarations.
DOTOT	Array of size DONEST. See Array Do Loop Declarations.

DOTOT#	
DOTYPE	Array of size DOSIZE. See Do Loop Descriptor Declarations.
DOUBLE_FLAG	See Symbol Table SYT_FLAGS
DOUBLEFLAG	
DOUNTIL	Array of size DOSIZE . See Do Loop Descriptor Declarations.
DROP_INX	label
DROP_VAR	label
DROPFREESPACE	label
DROPLIST	label
DROPOUT	label
DROPSAVE	label
DROPTEMP	label
DSCALAR	Initial (13). The double precision scalar operand type.
DUMMY	A dummy character string with several uses.
DUMMY_CHAR	See COLUMN.
DW	DOUBLEWORD aligned work area. Used for Inline Scalar Arithmetic.
EMIT_ADDRESS	label
EMIT_ARRAY_DO	label
EMIT_BY_MODE	label
EMIT_ENTRY	label

EMIT_EVENT_EXPRESSION	label
EMIT_RETURN	label
EMIT_WHILE_TEST	label
EMIT_Z)CON	label
EMITADDR	label
EMITBFW	label
EMITC	label
EMITDELTA	label
EMITDENSE	label
EMITEVENTADDR	label
EMITLFW	label
EMITOP	label
EMITP	label
EMITPCFADDR	label
EMITPDELTA	label
EMITPFW	label
EMITRR	label
EMITRZ	label
EMITSI	label
EMITSIOP	label
EMITSP	label
EMITSTRING	label
EMITW	label
EMITXOP	label

END_SF_REPEAT	label
ENDSCOPE_FLAG	See Symbol Table SYT_FLAGS.
ENTER_CALL	label
ENTER_CHAR_LIT	label
ENTER_ESD	label
ENTRYPOINT	See SYT_LINK1 in, symbol table.
EQ	Initial (4). Condition code used as a test for equality when generating conditional branch instructions.
ERR_DISP	See Block Definition Table.
ERR_STACK	See Block Definition Table.
ERRALL	Array of size PROC# . See Block Definition Table.
ERRALLGRP	Array of size PROC# . See Block Definition Table.
ERRCALL	label
ERROR_POINT	Initial (1). Never referenced.
ERROR#	The number of errors detected in Phase 2 of compilation.
ERRORS	label
ERRPTR	Array of size PROC#. See Block Definition Table.
ERRSEG	Array of size PROC#. See Block Definition Table. In OBJECT_ GENERATOR ERRSEG(ESD) = first location for that ESD.
ESD_LINK	Array of size ESD_LIMIT. Pointers chaining together ESD table entries whose names HASH to the same number.

ESD_MAX Initial (1). The maximum number of entries in the ESD table.

ESD_NAME Array of size ESD_CHAR_LIMIT. Packed tables of ESD names. The ESD number can be decoded to give the array entry and index in that entry where a name begins.

ESD_NAME_LENGTH Array of size ESD_LIMIT. The length of each ESD name in the ESD table.

ESD_START Array of size HASHSIZE. Each entry is a pointer to the beginning of ESD names that hash to the same index.

ESD_TABLE Character Procedure.

ESD_TYPE Array of size ESD_LIMIT. The type of each entry in the ESD table, the types used by phase 2 are:
0 - csect
1 - entry
2 - external

EV_EXP Array of size EV_EXPTR_MAX. Event Expression Stack: value of each entry is:

0 for an operand
1 for OR operator
2 for NOT operator
3 for AND operator

EV_EXPTR Pointer to last entry in Event Expression Stack (EV_EXP).

EV_OP Array of size EV_PTR_MAX. Stack of pointers to indirect stack entries for operands of an event expression.

EV_PTR Pointer to the last entry in Event Operand Stack (EV_OP).

EVALUATE label

EVENT Initial (17). The event operand type.

EVENT_OPERATOR
label

EVIL_FLAGS
See Symbol Table SYT_FLAGS

EXAMINING
Initial (1). Initially 1, but set to 0 if an error of severity 1 is found before the program has reached the error unit.

EXCLBASE
The CSECT used for storing exclusive data.

EXCLUSIVE_FLAG
See Symbol Table SYT_FLAGS.

EXCLUSIVE#
The number of exclusive procedures and functions. By bumping the number by 1 each time a new exclusive procedure is found, unique numbers are generated for SYT_LINK1.

EXOR
Initial ("04"). An operator code for not used as an index into the operator table. The not operation is performed by finding the exclusive or of the operand and a string of 1's the length of the operand.

EXPONENTIAL
label

EXPRESSION
label

EXT_ARRAY
Array of size EXT_SIZE.
Passed from Phase 1. See Symbol Table.

EXTENT
Common Based array. See Symbol Table.

EXTERNAL_FLAG
See Symbol Table SYT_FLAGS.

EXTOP
A pointer to an indirect stack entry with one of the following uses:

1) to represent an unknown array size,
2) for additional information for TO and AT partitions in subscripting,
3) to represent amount of input or output data in file I/O,
4) a pointer to temporary storage needed for real time operators, and
5) a pointer to temporary storage for matrix inversion.

EXTRA_LISTING — A compiler option.

EXTRABIT — See HALMAT operand types. A bit operand type used when performing a SUBBIT operation on a double word item.

EXTSYM — Initial (22). External symbol: 1) one of the possible forms of an Indirect Stack entry, 2) used as flag to the code emitting routines to ensure RLDs are generated if an external symbol is referenced, 3) used as an intermediate code qualifier to indicate an external symbol.

EZ — Initial (4). Condition code. A test for zero, used in generating branch instructions.

F — The I field of an RS format instruction with the indexed addressing mode.

FILECONTROL — Names of FILE I/O library routines.

FINDAC — label

FIRST_INST — Set to 1 at the beginning of every statement, and then back to zero after the first instruction of the statement has been generated.

FIRSTLABEL — A statement number generated by Phase 2 to use as a label for the destination of branch instructions.

FIRSTREMOTE — A pointer to the symbol table entry for the first REMOTE variable declared. The remote variables are chained together by SYT_LINK2.

FIRSTSTMT# — The statement number generated in Phase 1 for the first HAL/S source statement not contained in an EXTERNAL TEMPLATE block.

FIX_INTLBL — label

FIX_LABEL — label

FIX_STRUCT_INX — label

FIX_TERM_INX	label
FIXARG1	See Arguments.
FIXARG2	See Arguments.
FIXARG3	See Arguments.
FIXLIT	Initial (10). An intermediate code qualifier referring to the fullword literal pool.
FIXONE	Never referenced.
FL_NO_MAX	Value passed from Phase 1.
FLNO	Initial (19). Internal flow of control label. One of the forms of an indirect stack entry and one of the qualifiers used in the intermediate output code.
FORCE_ACCUMULATOR	label
FORCE_ADDRESS	label
FORCE_ARRAY_SIZE	label
FORCE_BY_MODE	label
FORM	Array of size STACK_SIZE. See Indirect Stack.
FORM_CHARNAME	Character Procedure.
FORM_VMNAME	label
FORMAT	Character Procedure.
FORMAT_OPERANDS	label
FREE_ARRAYNESS	label
FREE_TEMPORARY	label
FR0	Initial (8). Floating pointer register 0. Used to pass scalar paraemters, 1 to return scalar function results, and as a floating point accumulator.

FR1	Initial (9). Floating pointer register 1. Used to pass scalar parameters, to return scalar function results, and as a floating point accumulator.
FR2	Initial (10). Floating point register 2. Used to pass scalar paraemters and as a floating point accumulator.
FR4	Initial (12). Floating point register 4. Used to pass scalar parameters and as a floating point accumulator.
FR6	Initial (14). Floating point register 6. Used as a floating point accumulator.
FR7	Initial (15). Floating point register 7. Used as a floating point accumulator.
FSIMBASE	
FULLBIT	Initial (9). The fullword bit operand type.
FULLTEMP	Maximum temporary storage stack size.
FUNC_CLASS	See Symbol Table SYT_CLASS.
FUNC_LEVEL	The nest level of a function or of an inline function invocation.
GEN_ARRAY_TEMP	label
GEN_STORE	label
GENCALL	label
GENERATE	label
GENERATE_CONSTANTS	label
GENERATING	Initial (1). A flag used to indicate that code generation is occurring.
GENEVENTADDR	label

GENLIBCALL	label
GENSI	
GENSVC	
GENSVCADDR	
GET_ARRAYSIZE	
GET_ASIZ	
GET_CHAR_OPERANDS	
GET_CODE	
GET_CSIZ	
GET_EVENT_OPERANDS	
GET_FUNC_RESULT	
GET_INIT_LIT	
GET_INST_R_X	
GET_INTEGER_LITERAL	
GET_LIT_ONE	
GET_LITERAL	
GET_OPERAND	
GET_OPERANDS	
GET_R	
GET_STACK_ENTRY	
GET_STRUCTOP	
GET_SUBSCRIPT	
GET_VAC	
GET_VM_TEMP	
GETARRAY#	
GETARRAYDIM	

GETFREESPACE	label
GETINTLBL	label
GETINVTEMP	label
GETLABEL	label
GETSTATNO	label
GETSTMTLBL	label
GETSTRUCT#	label
GO	Initial (5). Condition code. Used as a test for greater than or equal to when generating branch instructions.
GT	Initial (1). Condition code. Used as a test for greater than when generating branch instructions.
GUARANTEE_ADDRESSABLE	label
HADDR	Initial (53). An intermediate code qualifier which indicates a halfword address constant.
HALFMAX	Initial ("7FFF"). Literals whose absolute value are greater than this are double precision.
HALFWORDSIZE	Initial (16). The number of bits in a halfword.
HALMAT_REQUESTED	A compiler option. 1 if a HALMAT listing is requested, 0 otherwise.
HASH	label
HEX	Character procedure.
HEX_LOCCTR	Character procedure.
HEXCODES	Initial ('0123456789ABCDEF'). A string used to convert internal binary to external hex notation.

IA Indirect Address field of RS format. AP-101 instrction with indexed addressing mode. This field specifies indirect addressing when one.

IAL Initial ("4F"). Opcode used for code generation.

IDENT_DISJOING_CHECK label

IGNORE_FLAG See Symbol Table SYT_FLAGS.

ILBL Initial (37). Internal label. An intermediate code opcode indicating a flow of control label.

IMD Initial (6). A HALMAT operand qualifier and indirect stack entry form specifying an actual numerical value.

INCORPORATE label

IND_CALL_LAB See Symbol Table SYT_TYPE.

IND_PTR Initial ("3F").

IND_STMT_LAB Initial ("41"). Indirect statement label.

INDEX
1) An indirect stack entry used as an index variable for setting up shaping function repeats.
2) Pointer to a symbol table entry for a block name.
3) Number of arguments in a percent macro.

INDEXING Array of size REG_NUM. Initial (0,1,1,1,$\overline{1}$,1,1,1). See Register Table.

INDEXNEST Array of size PROC#. See Block Definition Table.

INDIRECT label

INDIRECTION Array of size (3). Initial (' ', '@', '#', '@#'). Indirection characters used in generating instruction mnemonics for assembler code listing.

INFO

A dummy character string used while generating a line of assembler code for output.

INITADDR

The address relative to INITBASE of the data structure to be initialized if it requires static initialization; 0 if the data item requires automatic initialization. Notice that an offset from INITADDR must be added to get the individual item to be initialized.

INITAGAIN

Initially 0. The number of consecutive data items of the same type starting at a given offset that are to be initialized from the same intial list. The initial values of these items are stored in consecutive entries in the literal table. INITAGAIN is decremented as each value is assigned to a data item.

INITAUTO

1 if variable requiring initialization is automatic, 0 if it is static.

INITBASE

DATABASE(0) if variable requiring initialization does not have the remote attribute, DATABASE(1) if it does.

INITBLK

See HALMAT.

INITCTR

See HALMAT.

INITDECR

The offset of the parent node of a structure terminal item that is being initialized. The offset is in terms of the number of preceding elements in the structure. INITDECR is necessary since INITINCR gives an offset for the terminal node relative to the structure's beginning, but structure addressing is relative to the parent node, INITINCR - INITDECR is the offset relative to the parent node.

INITDENSE

1 if variable requires dense initialization, 0 otherwise.

INITIALIZE

label

INITINCR

When handling a list of initial values, INITINCR counts the number of items in the "natural sequence" (ref. language spec. 5.5). This value can be used either directly or indirectly (using STRUCTURE_WALK) to compute the address of the item to be initialized.

INITINX

The index register associated with the variable being initialized.

INITLITMOD

(Ref. 360 Compiler Spec. A.1.9.3). When "repeat" is non-zero, the initial values are in consecutive locations in the LITeral table. INITLITMOD is used to index the base address given in the HALMAT operand word so that consecutive literals can be extracted without requiring a separate initialization instruction for each element.

INITMOD

The storage space occupied by a structure; used as an offset when computing the address of a data item in a structure with several copies that is being initialized.

INITMULT

If INITTYPE is structure then 1. otherwise, the data width of the operand type in half words.

INITOP

A pointer to the symbol table entry of the data item being initialized.

INITREL

Array of size INITMAX . This array saves the value of INITINCR at the beginning of each nest level of initializtion repetition specification.

INITREPT

Array of size INITMAX . The number of repetitions of the initial list for a given initialization nest level that must still be made.

INITRESET

Saves the value of INITINCR due to initializing lists with a repeat factor, so that INITINCR can be used to index element by element initializations of members of the initial list.

INITSTART The address in INITBASE of a structure that requires static initialization.

INITSTEP Array of size INITMAX.
The number of values on the repetition list for a given initialization nest level.

INITSTRUCT 1 if the item being initialized is a structure, 0 if it is not.

INITTYPE The type of the data item being initialized.

INITWALK A counter used while walking through a structure to find the terminal element that is being initialized. The purpose of the walk is to find the offset of the terminal element's parent node.

INL See HALMAT decoding.

INLINE_RESULT Pointer to an indirect stack entry representing an inline function result.

INSMOD Instruction modifier.

INST The opcode of an instruction. Used to index the AP-101 instruction array to get the corresponding AP-101 opcode.

INSTRUCTION Character procedure.

INTEGER Initial (16). The single precision integer operand type.

INTEGER_DIVIDE label

INTEGER_MULTIPLY label

INTEGERIZABLE label

INTRINSIC label

INTSCA Initial (3). The PACKTYPE of the integer and scalar operand types.

INX Array of size STACK_SIZE. See Indirect Stack.

INX_CON Array of size STACK_SIZE. See Indirect Stack.

INX_MUL Array of size STACK_SIZE. See Indirect Stack.

INX_OK See Vector-Matrix.

INX_SHIFT Array of size STACK_SIZE. See Indirect Stack.

INXMOD Used for array and structure subscripting. A pointer to the indirect stack entry set up for the index variable for the do loop generated to process a subscript.

IOCONTROL Array of size (5), initial (' ', 'TAB', 'COLUMN', 'SKIP', 'LINE', 'PAGE'). Used to generate a library call to the routine whose index in the array corresponds to ARG_TYPE of the arguments of an I/O Reference.

IODEV Array of size (9), COMMON.

IOINIT
label

IOMODE
The type of I/O in an I/O routine invocation: 0 for read, 1 for write.

ITYPES
Array of size (4), initial ('B', 'H', 'I', 'E', 'D'). Used for generating calls to the library routine corresponding to the OPMODE of the arguments, by concatenating the letter whose index corresponds to the opmode with the library routine name.

IX
1) The index field of RS format AP-101 instructions with indirect addressing mode.
2) The second register operand of RR format AP-101 instructions.

IX1
1) Pointer used for searching temporary storage stack.
2) Dummy variable used while generating program names.
3) Dummy variable used while allocating structure templates.
4) Do loop index.

IX2
1) Pointer used for searching temporary storage stack.
2) Dummy variable used while generating program names.
3) Dummy variable used while allocating structure templates.

KIN
Pointer to symbol table entries for structure nodes used when walking through a structure.

KNOWN_SYM
label

L
Initial ("58"). Opcode used in code generation.

LA
Initial ("61"). Opcode used in code generation.

LABEL_ARRAY BASED. The statement number generated by Phase 2 that is associated with each internal flow number.

LABEL_CLASS See Symbol Table SYT_CLASS.

LABELSIZE Number of internal flow labels.

LADDR Initial (42). The intermediate code opcode for an address constant which points to a literal pool entry.

LASTBASE Array of size PROC#. See Block Definition Table.

LASTLABEL Array of size PROC#. See Block Definition Table.

LASTREMOTE Pointer to the symbol table entry for the last REMOTE variable declared.

LASTRESULT A pointer to the indirect stack entry for a library routine or built-in function reuslt.

LASTSTMT# The last statement number generated in Phase 1 for a HAL/S source program.

LATCH_FLAG See Symbol Table SYT_FLAGS.

LBL Initial (18). An indirect stack entry form and intermediate code qualifier for a user defined label.

LCR Initial (13). An opcode used for code generation.

LEFT_DISJOINT See Vector-Matrix.

LEFT_NSEC See Vector-Matrix.

LEFTBRACKET Initial ('(').

LEFTOP See HALMAT.

LFLI Initial ("03"). Opcode used for code generation.

LFXI Initial ("02"). Opcode used for code generation.

LH Initial ("48"). An opcode used for code generation.

LHI Initial ("A8"). An opcode used for code generation.

LHS The opcode field of an intermediate code output word.

LHSPTR See HALMAT

LIBNAME Character procedure.

LINE# A statement number for a line of HAL/S source code generated in Phase 1.

LINKREG See Runtime Stack Frame.

LIT Initial (5). A HALMAT operand qualifier and indirect stack entry form for a literal. (See Literal Table.)

LIT_CHAR COMMON BASED. See Literal Table.

LIT_CHAR_ADDR Beginning of free area in the storage for character string literals.

LIT_CHAR_LEFT Area left in the storage for character string literals.

LITERAL label

LITLIM The limit of the page of the literal file that is currently being read.

LITORG The beginning of the page of the literal file that is currently being read.

LITTYTPE The type of literals used with a HALMAT instruction.

LITTYPSET Array of size (12), initial (6).

LIT1 COMMON BASE. See Literal Table.

LIT2 COMMON BASE. See Literal Table.

MESSAGE	A variable used for building a line of assembler code for an assembler listing.
MH	Initial ("4C"). An opcode used in code generation.
MHI	Initial ("AC"). An opcode used in code generation.
MIH	Initial ("4E"). An opcode used in code generation.
MIN	label
MINUS	See HALMAT Operator properties.
MIX_ASSEMBLE	label
MOD_GET_OPERAND	label
MODE_MOD	Array. A number added to the basic opcode for one of the operator codes (given by ARITH_OPS) to generate the appropriate variable of an instruction.
MOVE_STRUTURE	label
MOVEREG	label
MR	Initial ("1C"). Opcode used for code generation.
MSTH	Initial ("BA"). An opcode used in code generation.
NAME_FLAG	See Symbol Table SYT_FLAG.
NAME_OP_FLAG	See Vector-Matrix.
NAME_SUB	1 if a subscript is enclosed in a NAME pseudo-function; 0 otherwise.
NAMELOAD	Initial ("48"). Opcode used for code generation.

NAMESIZE	Initial (1). The number of half words required for storing a NAME variable.
NAMESTORE	Initial ("40"). Opcode used for code generation.
NARGINDEX	The ESDID (scope) number of the block for which code is being generated.
NARGS	Array of size PROC#. See Block Definition Table.
NDECSY	Number of declared symbols.
NEGLIT	1 if a literal in the literal table is negative, 0 otherwise.
NEGMAX	Initial ("80000000"). Maximum negative value.
NEQ	Initial (3). Condition code. Used as a test for not equal to in branch instruction generation.
NESTFUNC	The nest level of a function or of an in-line function invocation.
NEW_GLOBAL_BASE	See Runtime Stack Frame.
NEW_HALMAT_BLOCK	label
NEW_LOCAL_BASE	See Runtime Stack Frame.
NEW_REG	label
NEW_STACK_LOC	See Runtime Stack Frame.
NEW_USAGE	label
NEWPREC	See HALMAT.
NEXT_REC	label
NEXTCODE	label
NEXTDECLREG	
NEXTPOPCODE	label

NHI Initiail ("A4"). Opcode used for code generation.

NIST Initial ("B4").

NO_VM_OPT See Vector-Matrix.

NONCOMMON_SYTSIZES Array of size NC_SYTSIZES#. Used by storage_mgt for dynamic allocation.

NONHAL_FLAG See Symbol Table SYT_FLAGS.

NONHAL_PROC_FUNC_CALL label

NONHAL_PROC_FUNC_SETUP label

NONPART See Vector-Matrix.

NOP Initial (52). An intermediate code opcode for No Operation Used to eliminate a previously generated RX or SI instruction.

NOT_MODIFIER Array of size (64). If NOT_MODIFIER (type of intermediate code) then increment the location counter; otherwise, do not. Used when outputting intermediate code lines which will not use words on the target machine. Values assigned in INITIALISE.

NSEC_CHECK label

NTOC label

NULL_ADDR Initial (0). An address or value of zero.

NUMOP See HALMAT decoding.

OBJECT_CONDENSER label

OBJECT_GENERATOR label

OFF_INX label

OFF_TARGET label

OFFSET Initial (10). A HALMAT operand qualifier and indirect stack entry form for an offset value.

OK_TO_ASSIGN See Vector-Matrix.

OPCC Array of size OPMAX, initialized. An array indexing the offset of each AP-101 instruction on the condition code:

OPCC	SIGNIFICANCE
0	Condition Code Unaffected
1	Register affected by condition code
2	Condition code no longer valid
3	Logical condition code

OPCODE See HALMAT decoding

OPCOUNT Array of size OPMAX. An array which records the number of times each of the opcodes used in code generation occurs.

OPER Array of size OPMAX, initialized. An array which gives the index in the appropriate OPNAMES entry for the mnemonic corresponding to each opcode used in code generation.

OPER_PARM_FLAG See Vector-Matrix.

OPER_SYMPTR See Vector-Matrix.

OPERATOR Array of size OPMAX, initialized. Array used to test whether a generated opcode actually exists. The array entry corresponding to the opcode is 1 if it exists, and 0 if it does not.

OPMODE Things have the same OPMODE if operations between them require no conversions (e.g. MATRIX, VECTOR and SCALAR have same OPMODE). See HALMAT Operand types.

OPMODE See HALMAT Operand types.

OPNAMES Array of size (3), initialized. This array consists of three character strings containing the mnemonics associated with the opcodes use for code generation. SHR(OPCODE,6) gives the string the mnemonic is in, and OPER(OPCODE) gives the index in the string where the mnemonic occurs.

OPR COMMON BASED. See HALMAT decoding.

OPSTAT label

OPTIMIZE label

OPTION_BITS COMMON bits indicating which of the user defined compiler options have been specified.

OPTYPE See HALMAT.

OP1
1) See HALMAT.
2) A pointer to a symbol table entry, used in INITIALIZE.

OP2
1) Dummy variable used in INITIALIZE and GENERATE.
2) Pointer to the symbol table entry for a template associated with a structure, and then to other symbol table entries associated with the template when walking through the template.

PACKFORM

Array of size (31). Used for choosing the form of the intermediate code. The values are:

0 for all operand field qualifiers except as noted
1 for CSYM WORK
2 for LIT, VAC

PACKFUNC_CLASS

Array of size (11), initial (0,0,0,1, 0,0,0,0,0,1,0,0).

PACKTYPE

Array of size TYP_SIZE, initialized. Value associated with each operand type to determine storage requirements.

Value	Description	Name
0	Vector,Matrix	VECMAT
1	Bit	BITS
2	Character	CHAR
3	Integer, Scalar	INTSCA
4	Structure	

PAD

Character procedure.

PADDR

Initial ('4'). An intermediate code opcode indicating an address constant which points to a literal pool entry.

PARM_FLAGS

See Symbol Table SYT_FLAGS.

PART_SIZE

See Vector-Matrix.

PCEBASE

A CSECT used for Process Director Entries. This CSECT provides information about task addresses to the operating system.

PDELTA	Initial (45). An intermediate code opcode indicating that the next instruction musc be modified by the maximum temporary storage size of the CSECT specified by the intermediate code instruction.
PLBL	Initial (48). An intermediate code opcode indicating a Phase 2 generated label.
PLUS	Initial ('+').
PM_FLAGS	See Symbol Table SYT_FLAGS.
PMINDEX	The index number of a %MACRO.
POINT	Array of size LASTEMP. See Storage Descriptor Stack.
POINTER	Initial (7). The pointer operand type.
POINTER_FLAG	See Symbol Table SYT_FLAGS.
POINTER_OR_NAME	See Symbol Table SYT_FLAGS.
POSITION_HALMAT	label
POSMAX	Initial ("7FFFFFFF").
POWER_OF_TWO	label
PP	See HALMAT decoding.
PREFIXMINUS	See HALMAT operator properties.
PRINT_DATE_AND_TIME	label
PRINT_TIME	label
PRINTSUMMARY	label
PROC_FUNC_CALL	label
PROC_FUNC_SETUP	label

C5

PROC_LABEL	Initial ("4+"). See Symbol Table SYT_TYPE.
PROC_LEVEL	Array of size PROC#. See Block Definition Table.
PROC_LINK	Array of size PROC#. See Block Definition Table.
PROCBASE	See Runtime Stack Frame.
PROCLIMIT	The last CSECT number assigned to a program, procedure, function, task, or Compool by Phase 1.
PROCPOINT	The CSECT number = scope number of a procedure, program, function, task or COMPOOL whose symbol table entry is being processed by INITIALIZE.
PROC#	Literally '255'. The maximum number of csects that are processable.
PROG_LABEL	See Symbol Table SYT_TYPE.
PROGBASE	See Runtime Stack Frame.
PROGCODE	The number of halfwords of program generated by Phase 2.
PROGDATA	PROGDATA(0) is the number of halfwords of local data generated by Phase 2. PROGDATA(1) is the equivalent for REMOTE data.
PROGNAME	Character procedure.
PROGPOINT	The CSECT number = scope number of the outer block of a compilation unit.
PTRARG1	See Runtime Stack Frame.
PUSH_ADDLEVEL	label
PUSH_ARRAYNESS	label
QUOTE	Initial ('''').
R	The register field of an AP-101 instruction.
R_BASE	See Register Table.

R_CON

R_CONTENTS

R_INX

R_INX_CON

R_INX_SHIFT

R_MULT

R_SECTION

R_TYPE

R_VAR

R_VAR2

R_XCON

See Register Table.

R_CLASS

Array of size TYP_SIZE.
Array giving the type of register used by each operand type for finding an appropriate register for an operand.

RCLASS	Register Type
0	Double Floating Accumulator
1	Floating Accumulator
2	Double Accumulator
3	Fixed Accumulator
4	Index Register
5	Odd

RCLASS_START	See Registers.
READCTR	See HALMAT decoding
REAL_LABEL	label
RECVR	See Vector-Matrix.
RECVR_NEST_LEVEL	See Vector-Matrix.
RECVR_OK	See Vector-Matrix.
RECVR_SYMPTR	See Vector-Matrix.
REENTRANT_FLAG	See Symbol Table SYT_FLAGS.
REF_STRUCTURE	label
REC	See Indirect Stack.
REG_NUM	The maximum number of base registers (real & virtual).
REGISTER_SAVE_AREA	See Runtime Stack Frame.
REGISTER_STATUS	label
REGISTERS	This array is used in conjunction with RCLASS_START to obtain a list of all registers in any class. RCLASS_START gives the entry in REGISTERS where the list of registers of a certain class starts (e.g. if begin=RCLASS_START(DOUBLE_AC) and end=RCLASS_START(DOUBLE_AC+1)-1, then REGISTERS(begin), REGISTERS(begin+1),... REGISTERS(end) are all the double accumulators.
RELATIONAL	Initial (21). An indirect stack entry type used for generation conditional branches.
RELEASETEMP	label
REMOTE_ADDRS	A flag indicating wheter any of the operands of a HALMAT instruction is remote data.

REMOTE_BASE	Initial (9). Register 9, used for addressing remote data.
REMOTE_FLAG	See Symbol Table SYT_FLAGS.
REMOTE_LEVEL	Array of size PROC#. See Block Definition Table.
REMOTE_RECVR	See Vector-Matrix.
RESET	See HALMAT decoding.
RESTART	A location in MAIN_PROGRAM where GENERATE is called.
RESULT	See HALMAT.
RESUME_LOCCTR	label
RETURN_STACK_ENTRIES	label
RETURN_STACK_ENTRY	label
REVERSE	See HALMAT operator properties. Used to change operator when commuting an operation.
RHS	The operand field of an intermediate code output word.
RI	Initial (10). A value used to generate the opcode for various RI instructions. Used with MODE_MOD, OPMODE, and ARITH_OP.
RIGHT_DISJOINT	See Vector-Matrix.
RIGHT_NSEC	See Vector-Matrix.
RIGHTBRACKET	Initial (')').
RIGHTOP	See HALMAT.
RIGID_FLAG	See Symbol Table SYT_FLAGS.
RLD	Initial (43). An intermediate output code opcode used to specify an ESDID as the reference entry for an RLD specification.
RM	Initial ("7"). Register 7.
RNON_IDENT	See Vector-Matrix.
ROW	Array of size STACK_SIZE. See Indirect Stack.

RR Initial (0). A value used to generate the opcode for various RR instructions. Used with MODE_MOD, OPMODE, and ARITH_OP.

RRTYPE Initial (32). An intermediate code opcode indicating an RR format instruction.

RTYPE See Vector-Matrix.

RX Initial (5). A value used to generate the opcode for various RX instructions. Used with MODE_MOD, OPMODE, and ARITH_OP.

RXTYPE Initial (33). An intermediate code opcode indicating an RX format instruction.

R0 Initial (0). Register 0: the stack register points to register save area. Formal parameters, temporaries, and AUTOMATIC variables in REENTRANT procedures are based off of it.

R1 Initial (1). Register 1. Used to address all variables and values within a compilation unit.

SAFE_INX label

SAVE_ARG_STACK_PTR Array of size CALL_LEVEL #. See Call Stack.

SAVE_CALL_LEVEL Array of size CALL_LEVEL #. See Call Stack.

SAVE_FLOATING_REGS label

SAVE_LITERAL label

SAVE_REGS label

SAVEPOINT Array of size LASTEMP . See Storage Descriptor Stack.

SAVEPTR See Storage Descriptor Stack.

SB Initial ("B6"). Opcode used for code generation.

SCALAR Initital (5). The single precision scalar operand type.

SDL — A compiler option informing the compiler whether it is operating within the SDL.

SDOLEVEL — Array of size DONEST. See Array Do Loop Declarations.

SDOPTR — Array of size DONEST. See Array Do Loop Declarations.

SDOTEMP — Array of size DONEST. See Array Do Loop Declarations.

SDR — Initial ("28"). An opcode used for code generation.

SEARCH_INDEX2 — label

SEARCH_REGS — label

SECONDLABEL — A statement label generated by Phase 2 to use as the destination of a branch instruction.

SELECTYPE — Array of size TYP_SIZE. initialized. A value associated with each operand type used for generating appropriate library calls and for determining the sequence of conversions in assignment statements.

SELF_ALIGNING — A compiler option.

SELFNAMELOC	A pointer to the symbol table entry for the outer block of the compilation unit, (set by CHECK_COMPILABLE).
SER	Initial ("3B"). An opcode used for code generation.
SET_AREA	label
SET_ARRAY_SIZE	
SET_AUTO_IMPLIED	
SET_AUTO_INIT	
SET_BINDEX	
SET_CHAR_DESC	
SET_CHAR_INX	
SET_CINDEX	
SET_ERRLOC	
SET_EVENT_OPERAND	
SET_INIT_SYM	
SET_IO_LIST	
SET_LABEL	
SET_LOCCTR	
SET_OPERAND	
SET_RESULT_REG	
SETUP_ADCON	
SETUP_BOOLEAN	
SETUP_CANC_OR_TERM	
SETUP_EVENT	

SETUP_INX
SETUP_NONHAL_ARG
SETUP_PRIORITY
SETUP_RELATIONAL
SETUP_STACK
SETUP_STRUCTURE
SETUP_TIME_OR_EVENT
} label

SF_DISP   The byte width of the operand type.

SF_RANGE   Array of size CALL_LEVEL#. The range of each dimension of arrayness of a shaping function.

SF_RANGE_PTR   A pointer to the first free entry in SF_RANGE.

SGNLNAME   Array of size (2), initial (12,13, 14). An array giving the SVC number corresponding to each kind of event signalling.

Value	Description	SVC #
0	SIGNAL	12
1	SET	13
2	RESET	14

SHAPING_CALL   label

SHAPING_FUNCTIONS   label

SHCOUNT   Initial (23). An intermediate code qualifier idnicating a shift count.

SHIFT   See HALMAT operand types.

SHOULD_COMMUTE   label

SHW   An opcode used for code generation.

SIMPLE_ARRAY_PARAMETER   label

SIZEFIX   label

SIZE3 Array of size VMOPSIZE literally '25', initialized. Array specifying whether each vector-matrix operation has special routines for 3x3 matrices and vectors with 3 components.

SLDL Initial ("8D"). Opcode used for code generation.

SLL Initial ("89"). Opcode used for code generation.

SM_FLAGS Initial ("00C2008C"). Used for matching structure terminal SYT_FLAGS.

SMADDR Initial (56). An intermediate code opcode indicating a HAL/S source line member.

SMRK_CTR See HALMAT decoding.

SORD Array of size (1), initial (' ', 'D'). Prefixes for built-in function names indicating whether to use the function that gives a single or double precision result.

SPM Initial ("04"). Opcode used for code generation.

SR Initial ("1B"). Opcode used for code generation.

SRA Initial ("8A"). Opcode used for code generation.

SRCE See Vector-Matrix.

SRCEPART_SIZE See Vector-Matrix.

SRCERR The location in MAIN_PROGRAM where control goes after an error in HAL/S source has been found.

SRDA Initial ("8E"). Opcode used for code generation.

SRL Initial ("88"). Opcode used for code generation.

SRSTYPE	Initial (50). An intermediate output code opcode indicating an SRS format instruction.
SSTYPE	Initial (34). An intermediate output code opcode indicating an SS format instruction.
ST	Initial (50). Opcode used for code generation.
STACK_EVENT	label
STACK_FREEPOINT	See Runtime Stack Frame.
STACK_LINK	See Runtime Stack Frame.
STACK_MAX	See Indirect Stack.
STACK_PTR	Array of size STACK_SIZF. See Indirect Stack.
STACK#	See Array Do Loop Declarations.
STACKPOINT	The first of a sequence of ESDID numbers assigned to the unresolved external control sections for the stack for each program or task.
STACKSPACE	Array of size PROC#. See Block Definition Table. In OBJECT_GENERATOR STACKSPACE(end) = last location used for that end.
STACKSPACE	Array of size PROC#. See Block Definition Table.
START_OFF	See Vector-Matrix.
START_PART	See Vector-Matrix.
STATIC_BLOCK	label
STATNO	The number of statement labels generated by Phase 2.
STEP_LINE#	label
STH	Initial ("40"). Opcode used for code generation.
STM	Initial ("90"). Opcode used for code generation.

STMT_LABEL See Symbol Table SYT_TYPE.

STMT_NUM

STMT_PREC ≠0 if dealing with double precision matrix result. See Vector-Matrix.

STMTNO Initial (44). An intermediate output code opcode marking HAL source statement boundaries.

STNO Initial (20). An indirect stack entry form and intermediate code qualifier indicating a Phase 2 generated label.

STOPPERFLAG A flag used to prevent the emitting of code for branching around ELSE clauses for IF statements whose THEN clause ends in an unconditional branch.(For example, GO TOs, RETURN.)

STORE Initial ("01"). An operator code for storing used as an index into the table containing information about the different operators.

STRACE Never referenced.

STRI_ACTIVE 1 if initialization of data items is occurring, 0 otherwise.

STRUCT Array of size STACK_SIZE. See Indirect Stack.

STRUCT_CON Array of size STACK_SIZE. See Indirect Stack.

STRUCT_INX Array of size STACK_SIZE. See Indirect Stack.

STRUCT_LINK A pointer to a structure template's symbol table entry used for chaining through a linked list of structure templates.

STRUCT_MOD Array of size (1). A modifier used for computing the address of a terminal element of a structure. STRUCT_MOD gives the offset of a mode from the beginning of a structure copy plus the displacement of the structure copy the address is in from the beginning of the structure. The array has two entries so that it can keep information about two structures at once.

STRUCT_REF	Array of size (1). A pointer to the symbol table entry for a structure's template used when walking through structures. The array has two entries so that it can keep pointers to two structures for processing structure conditions.
STRUCT_START	A pointer to the symbol table entry for the first structure template in a linked list of structure templates. The SYT_LEVEL entry of each template points to the next member of the list.
STRUCT_TEMPL	Array of size (1). A pointer to the symbol table entry for a structur- template which is currently being referenced. The array has two entries so that it can have pointers to two structures for structure conditionals.
STRUCTFIX	label
STRUCTURE	Initial (16). The structure operand type.
STRUCTURE_ADVANCE	label
STRUCTURE_COMPARE	label
STRUCTURE_DECODE	label
STRUCTURE_WALK	label
SUB#	Subscript number.
SUBCODE	See HALMAT decode.
SUBLIMIT	See Array Do Loop declarations.
SUBMONITOR	The location in MAIN_PROGRAM where control goes if compilation is abandoned or at the end of compilation.
SUBOP	See HALMAT.
SUBRANGE	See Array Do Loop declarations.
SUBSCRIPT_MULT	label
SUBSCRIPT_RANGE_CHECK	label
SUBSCRIPT2_MULT	label

SUBSTRUCT_FLAG See Vector-Matrix.

SUCCESSOR label

SUM See HALMAT operator properties.

SVC Initial ("9A"). Opcode used for code generation.

SYM Initial (1). A HALMAT operand qualifier, indirect stack entry form, and intermediate code qualifier indicating a symbol table entry.

SYMBREAK The ESD number of the last function or procedure that has a symbol table entry. The last number assigned by Phase 1.

SYMFORM Array of size (31). 1 for SYM, CSYM, IMD, and INL; 0 for other intermediate code qualifiers. Initialized by INITIALISE.

SYM2 Initial (29). An operand qualifier used for indicating that a register is bieng used for two-dimensional subscripting.

SYSARG0 Initial (1). Register 1. This name refers to its use for vector-matrix routine input and output.

SYSARG1 Initial (2). Register 2. This name refers to its use for vector and matrix routine input and output.

SYSARG2 Initial (3). Register 3. This name refers to its use for vector and matrix routine input and output.

SYSINT Initial (21). An intermediate code qualifier indicating a System Intrinsic Library member.

SYT_ADDR Common Based. See Symbol Table.

SYT_ARRAY Common Based. See Symbol Table.

SYT_BASE	Based. See Symbol Table.
SYT_CLASS	Common Based. See Symbol Table.
SYT_CONST	Based. See Symbol Table.
SYT_COPIES	label
SYT_DIMS	Common Based. See Symbol Table.
SYT_DISP	Based. See Symbol Table.
SYT_FLAGS	Common Based. See Symbol Table.
SYT_LEVEL	Based. See Symbol Table.
SYT_LINK1	Common Based. See Symbol Table.
SYT_LINK2	Common Based. See Symbol Table.
SYT_LOCK#	Common Based. See Symbol Table.
SYT_NAME	Common Based. See Symbol Table.
SYT_NEST	Common Based. See Symbol Table.
SYT_PARM	Based. See Symbol Table.
SYT_PTR	Common Based. See Symbol Table.
SYT_SCOPE	Common Based. See Symbol Table.
SYT_SIZE	The size of the symbol table. See Symbol Table.
SYT_SORT	Based. See Symbol Table.
SYT_TYPE	Common Based. See Symbol Table.
SYT_XREF	Common Based. See Symbol Table.
TABLE_ADDR	Common. Never referenced.
TAG	See HALMAT decoding.
TAG_BITS	label

TAGS
See HALMAT decoding.

TAG1
See HALMAT decoding.

TAG2
See HALMAT decoding.

TAG3
See HALMAT decoding.

TARGET_R
Initial (-1). If TARGET_R is positive, routines that search for a free register will checkpoint it and return it.

TARGET_REGISTER
Initial (-1). If TARGET_REGISTER ≥ 0, routines that force values into registers will force them into this register.

TASK_LABEL
See Symbol Table SYT_TYPE.

TASK#
The number of tasks in the compilation unit.

TASKPOINT
A pointer to the symbol table entry for the first task in a linked list of all the tasks in a program. The tasks are linked through SYT_LINK1.

TB
Initial ("B1"). Opcode used for code generation.

TD
Initial ("9B"). Opcode used for code generation.

TEMP
A temporary variable with a variety of uses used in object code generation.

TEMPBASE
See Runtime Stack Frame.

TEMPL_NAME
See Symbol Table SYT_TYPE.

TEMPORARY_FLAG
See Symbol Table SYT_FLAG.

TEMPSPACE
1) Used to compute the EXTENT of a symbol table entry for variables that are not formal parameters.
2) The number of elements in a matrix, vector, or array.

TERMFLAG Used to distinguish matrix subscripting from subscripting of other data types. Value is 0 for non-matrix data types. For matrices value is 1 while subscripting the rows, and then set to 0 for subscripting the columns.

TERMINATE label

TEST See HALMAT Operator properties.

TH Initial ("91"). An opcode used for code generation.

TMP A temporary variable with a variety of localized uses.

TO_BE_INCORPORATED Initial (1). A flag indicating the presence of integer constants that are to be incorporated into terms.

TO_BE_MODIFIED Initial (1). A flag indicating whether the contents of a register will be modified or not.

TOGGLE Common.

TRACING A flag indicating if the TRACE compiler option is in effect.

TRUE_INX label

TS Initial ("93"). Opcode used for code generation.

TYPE See Indirect Stack.

TYPE_BITS label

TYPES Array of size (8), initial ('H','I','E', 'D','B','B','K','O','X'). Entries from this array are chosen according to the SELECTYPE of an operand and used to generate calls to appropriate library routines by prefixing or suffixing the letter to the name of the routine.

ULBL	Initial (36). An intermediate code opcode for a user defined label.
UNARY	See HALMAT Operator properties.
UNARYOP	label
UNIMPLEMENTED	Location to which control is transferred if an unimplemented feature is encountered.
UNRECOGNIZABLE	label
UPDATE_ASSIGN_CHECK	label
UPDATE_CHECK	label
UPDATE_INX_USAGE	label
UPDATING	If greater than 0, this is the block number of an update block for which code is being generated.
UPPER	Array of size LASTEMP. See Storage Descriptor Stack.
USAGE	Array of size REG_NUM. See Register Table.
USAGE_LINE	Array of size REG_NUM. See Register Table.
VAC	Initial (3). 1) A HALMAT operand qualifier for a virtual accumulator, a block pointer to the results of a previous HALMAT instruction. 2) An indirect stack entry form for a register being used as a temporary variable.
VAC_COPIES	label

VAC_FLAG See Vector-Matrix.

VAL Array of size STACK_SIZE literally '100'. See Indirect Stack.

VALMOD Used to modify the offset calculated for TO or AT partiion subscripts to take into account the indexing method used.

VALMUL The size of a subscript used in an array, component, or structure subscripting operation.

VALS Based.

VAR_CLASS See Symbol Table SYT_CLASS.

VECMAT Initial (0). The PACKTYPE of vector and matrix operands.

VECMAT_ASSIGN label

VECMAT_CONVERT label

VECTOR Initial (4). The single precision vector operand type.

VERIFY_INX_USAGE label

VERSION Initial (8). The compiler version number.

VERSION_LEVEL Initial (5). The compiler version level.

VMCALL label

VMREMOTEOP Array of size VMOPSIZE. initialized. An array used to generate the opcode used for calling the versions of vector-matrix routines for remote data.

WAITNAME Array of size (3), initial (9,6,7,8). This array gives the SVC number associated with each kind of WAIT.

HALMAT WAIT operator tag	Kind of WAIT	SVC #
0	WAIT FOR DEPENDENT	9
1	WAIT (timing expression)	6
2	WAIT UNTIL (timing expressions)	7
3	WAIT FOR (event expresion)	8

WORDSIZE

Initial (32). The number of bits in a word.

WORK

Initial (31). An indirect stack entry form for a location in the temporary storage area of a CSECT.

WORK_CTR

Array of size LASTEMP. See Storage Descriptor Stack.

WORK_USAGE

Array of size LASTEMP. See Storage Descriptor Stack.

WORKSEG

Array of size PROC#. See Block Definition Table.

WORK1

A temporary variable with a variety of uses including:
1) Setting up labels for DO CASE statements.
2) Pointer to symbol table entries for struture terminals.

WORK2

A temporary variable with a variety of uses including:
1) Setting up labels for DO CASE statements.
2) Pointer to symbol table entries for structure terminals.

WORK3

Records value of FREELIMIT after dynamic allocation of COMMON tables to be passed to Phase 3.

X_BITS

label

XADD
XBNEQ
XCFOR
XCSIO
XCSLD
XCSST
XCTST

See HALMAT opcodes.

XD XDIV XDLPE XEXP XEXTN XFBRA XFILE XICLS XIDEF XILT XIMRK	See HALMAT opcodes
XIST	Initial ("B7"). Opcode used for code generation.
XITAB	Array of size (32), initialized. Character strings used for masking bit operands according to size.
XMASN XMDET XMEXP XMIDN XMINV XMTRA XMVPR XN	See HALMAT opcodes

XNOT XOR XPASN XPEX	See HALMAT Opcodes.
XPROGLINK	A pointer to the beginning of a chian of external non-HAL procedures or function. XPROGLINK points to the symbol table entry of the first such procedure or function. SYT_LINK1 is used by each member of the chain to pointto the next member.
XPT	See HALMAT.
XR	Initial ("17"). An opcode use for code generation.
XRDAL	See HALMAT opcodes.
XREAD	See HALMAT opcodes.
XREF	See Symbol Table SYT_XREF.
XSASN XSFAR XSFNO XSFST XSMRK	See HALMAT opcodes
XVAL	Array of size STACK_SIZE. See Indirect Stack.

XVMIO	See HALMAT opcodes.
XWRIT	
XXASN	
XXREC	
XXXAR	
XXXNO	
XXST	
X2	Initial (' '). A string of two blanks.
X3	Initial (' '). A string of three blanks.
X4	Initial (' '). A string of four blanks.
X72	Initialized. A string of seventy-two blanks.
Z_LINKAGE	A compiler option indicating that external linkage conventions are to be used.
ZADDR	Initial (55). An intermediate code opcode indicating a Z-type address constant.
ZB	Initial ("BE"). Opcode used for code generation.
ZH	Initial ("99"). Opcode used for code generation.

5.2 Procedure Descriptions

Name		
ABS	✓	
ADDRESS_STRUCTURE	✓	
ADDRESSABLE	✓	
ADJUST	X	
ALLOCATE_TEMPLATE	✓	
ALLOCATE_TEMPORARY	✓	
ARG_ASSEMBLE	✓	
ARITH_BY_MODE	✓	
ARRAY_INDEX_MOD	✓	
ARRAY2_INDEX_MOD	-	Similar to ARRAY_INDEX_MOD for two dimensional arrays.
ASSIGN_CLEAR	X	
AVAILABLE_FROM_STORAGE	X	
BEGIN_SF_REPEAT	X	
BESTAC	X	
BIT_MASK	✓	
BIT_SHIFT	✓	
BIT_STORE	✓	
BIT_SUBSCRIPT	X	
BLOCK_CLOSE	✓	
BLOCK_OPEN	✓	
BOUNDARY_ALIGN	X	
CHAR_CALL	✓	
CHAR_CONVERT	X	
CHAR_INDEX	✓	
CHAR_SUBSCRIPT	X	
CHARACTER_TERMINAL	X	
CHECK_ADDR_NEST	✓	
CHECK_AGGREGATE_ASSIGN	X	
CHECK_AND_DROP_VAC	X	
CHECK_ASSIGN	-	See GENERATE MTRA
CHECK_ASSIGN_PARM	X	
CHECK_COMPILABLE	X	

CHECK_CSYM_INX	✓	
CHECK_LINKREG	X	
CHECK_LOCAL_SYM	X	
CHECK_LOCK#	X	
CHECK_NAME_PARM	X	
CHECK_REMOTE	✓	
CHECK_SI	X	
CHECK_SRCE	-	See GENERATE MTRA
CHECK_SRS	X	
CHECK_STRUCTURE_PARM	X	
CHECK_VAC	✓	
CHECK_VM_ARG_DIMS	X	
CHECKPOINT_REG	✓	
CHECKSIZE	✓	
CLEAR_CALL_REGS	X	
CLEAR_NAME_SAFE	X	
CLEAR_R	✓	
CLEAR_REGS	X	
CLEAR_SCOPED_REGS	X	
CLEAR_STMT_REGS	X	
COMMUTEM	X	
COMPARE_STRUCTURE	X	
CONSTERM	X	
COPY_STACK_ENTRY	X	
CS	X	
CTON	X	
DECODEPIP	✓	
DECODEPOP	✓	
DEFINE_LABEL	✓	
DESC	✓	
DESCENDENT	-	See STRUCTURE_WALK
DIMFIX	✓	
DO_ASSIGNMENT	✓	
DO_EXPRESSION	X	

DOCLOSE	✓	
DOFORSETUP	X	
DOMOVE	X	
DOOPEN	✓	
DROP_INX	✓	
DROP_PARM_STACK	X	
DROP_REG	X	
DROP_VAC	✓	
DROPFREESPACE	✓	
DROPLIST	✓	
DROPOUT	✓	
DROPSAVE	✓	
DROPTEMP	✓	
EMIT_ADDRS	-	See OBJECT_GENERATOR (56)
EMIT_ARRAY_DO	✓	
EMIT_BY_MODE	X	
EMIT_CALL	X	
EMIT_CARD	✓	
EMIT_ENTRY	X	
EMIT_ESD_CARDS	✓	
EMIT_SYM_CARDS	✓	
EMIT_EVENT_EXPRESSION	✓	
EMIT_RETURN	X	
EMIT_WHILE_TEST	✓	
EMIT_Z_CON	X	
EMITADDR	X	
EMITBFW	X	
EMITC	X	
EMITDELTA	X	
EMITDENSE	X	
EMITEVENTADDR	X	
EMITLFW	X	
EMITOP	X	
EMITP	X	

EMITPCEADDR	X
EMITPDELTA	X
EMITPFW	X
EMITRR	X
EMITRX	X
EMITSI	X
EMITSIOP	X
EMITSP	X
EMITSTRING	✓
EMITW	X
EMITXOP	X
END_SF_REPEAT	X
ENTER	X
ENTER_CALL	X
ENTER_CHAR_LIT	✓
ENTER_ESD	X
ERRCALL	X
ERRORS	X
ESD_TABLE	X
EVALUATE	X
EVENT_OPERATOR	X
EXPONENTIAL	✓
EXPRESSION	X
FETCH_VAC	X
FINDAC	X
FIX_INTLBL	✓
FIX_LABEL	✓
FIX_STRUCT_INX	✓
FIX_TERM_INX	X
FORCE_ACCUMULATOR	✓
FORCE_ADDR_LIT	X
FORCE_ADDRESS	✓
FORCE_ARRAY_SIZE	X
FORCE_BY_MODE	✓

FORCE_NUM	X
FORM_BD	✓
FORM_CHARNAME	X
FORM_VMNAME	X
FORMAT	✓
FORMAT_OPERANDS	X
FREE_ARRAYNESS	✓
FREE_TEMPORARY	X
GEN_ARRAY_TEMP	✓
GEN_STORE	✓
GENCALL	X
GENERATE	✓
GENERATE_CONSTANTS	✓
GENEVENTADDR	X
GENLIBCALL	X
GENSI	X
GENSVC	X
GENSVCADDR	X
GET_ARRAYSIZE	X
GET_ASIZ	✓
GET_CHAR_OPERANDS	X
GET_CODE	X
GET_CSIZ	X
GET_EVENT_OPERANDS	X
GET_FUNC_RESULT	✓
GET_INIT_LIT	X
GET_INST_R_X	X
GET_INTEGER_LITERAL	✓
GET_LIT_ONE	X
GET_LITERAL	✓
GET_OPERAND	✓
GET_OPERANDS	X
GET_R	✓
GET_STACK_ENTRY	✓
GET_STRUCTOP	X

GET_SUBSCRIPT	X	
GET_VAC	✓	
GET_VM_TEMP	X	
GETARRAY#	✓	
GETARRAYDIM	✓	
GETFREESPACE	✓	
GETINTLBL	✓	
GETINVTEMP	X	
GETLABEL	X	
GETSTATNO	✓	
GETSTMTLBL	✓	
GETSTRUCT#	X	
GUARANTEE_ADDRESSABLE	✓	
HASH	X	
HEX	✓	
HEX_LOCCTR	✓	
IDENTI_DISJOINT_CHECK	-	See Section on V-M Optimization and FC Spec. 3.1.5.5.
INCORPORATE	✓	
INCR_USAGE	X	
INDIRECT	X	
INITIALISE	✓	
INSTRUCTION	X	
INTEGER_DIVIDE	X	
INTEGER_MULTIPLY	✓	
INTEGER_VALUE	X	
INTEGERIZABLE	✓	
INTRINSIC	X	
IOINIT	X	
KNOWN_SYM	X	
LIB_LOOK	X	
LIBNAME	X	
LITERAL	✓	
LOAD_NUM	✓	

LOAD_TEMP	X	
LUMP_ARRAYSIZE	X	
LUMP_TERMINALSIZE	X	
MAIN_PROGRAM	X	
MAJOR_STRUCTURE	✓	
MAKE_INST	X	
MARKER	X	
MASK_BIT_LIST	X	
MAX	✓	
MIN	✓	
MIX_ASSEMBLE	-	Similar to ARG ASSEMBLE with second operand scalar
MOD_GET_OPERAND	-	A restricted version of GET_OPERAND
MOVE_STRUCTURE	X	
MOVEREG	✓	
NEED_STACK	X	
NEW_HALMAT_BLOCK	✓	
NEW_REG	✓	
NEW_USAGE	✓	
NEXT_REC	X	
NEXT_STACK	X	
NEXTCODE	✓	
NEXTPOPCODE	X	
NONHAL_PROC_FUNC_CALL	X	
NONHAL_PROC_FUNC_SETUP	X	
NSEC_CHECK	X	
NTOC	X	
OBJECT_CONDENSER	✓	
OBJECT_GENERATOR	✓	
OFF_INX	✓	
OFF_TARGET	X	
OPDECODE	X	
OPSTAT	X	
OPTIMISE	✓	
PAD	✓	
PARAMETER_ALLOCATE	✓	

PARMTEMP	X	
POSITION_HALMAT	✓	
POWER_OF_TWO	✓	
PRINT_DATE_AND_TIME	X	
PRINT_TIME	X	
PRINTSUMMARY	X	
PROC_FUNC_CALL	X	
PROC_FUNC_SETUP	✓	
PROCENTRY	✓	
PROGNAME	X	See Section 6.2 of User's Manual
PUSH_ADOLEVEL	X	
PUSH_ARRAYNESS	✓	
REAL_LABEL	X	
REF_STRUCTURE	X	
REGISTER_STATUS	✓	
RELEASETEMP	✓	
RESUME_LOCCTR	✓	
RETURN_EXP_OR_FH	X	
RETURN_STACK_ENTRIES	X	
RETURN_STACK_ENTRY	✓	
SAFE_INX	X	
SAVE_FLOATING_REGS	✓	
SAVE_LITERAL	✓	
SAVE_REGS	✓	
SEARCH_INDEX2	X	
SEARCH_REGS	✓	
SET_AREA	✓	
SET_ARRAY_SIZE	✓	
SET_AUTO_IMPLIED	-	See BLOCK_OPEN
SET_AUTO_INIT	X	
SET_BINDEX	X	
SET_BIT_TYPE	X	
SET_CHAR_DESC	X	
SET_CHAR_INX	X	
SET_CINDEX	X	

SET_ERRLOC	✓
SET_EVENT_OPERAND	X
SET_INIT_SYM	X
SET_IO_LIST	X
SET_LABEL	✓
SET_LOCCTR	✓
SET_NEST_AND_LOCKS	X
SET_OPERAND	X
SET_PROCESS_SIZE	X
SET_RESULT_REG	X
SETUP_ADCON	✓
SETUP_BOOLEAN	✓
SETUP_CANC_OR_TERM	X
SETUP_DATA	X
SETUP_EVENT	X
SETUP_INX	X
SETUP_NONHAL_ARG	X
SETUP_PRIORITY	✓
SETUP_RELATIONAL	X
SETUP_REMOTE_DATA	X
SETUP_STACK	✓
SETUP_STACKS	X
SETUP_STRUCTURE	X
SETUP_TIME_OR_EVENT	X
SETUP_TOTAL_SIZE	X
SETUP_VAC	X
SETUP_XPROG	X
SHAPING_CALL	X
SHAPING_FUNCTIONS	X
SHORTCUT_BIT_LIT	X
SHOULD_COMMUTE	X
SIMPLE_ARRAY_PARAMETER	X
SIZEFIX	✓
SKIP	X
SKIP_ADDR	X

SKIP_NOP	X	
STACK_EVENT	X	
STACK_PARM	X	
STACK_REG_PARM	X	
STACK_TARGET	X	
STATIC_BLOCK	X	
STEP_LINE#	✓	
STORAGE_ASSIGNMENT	✓	
STORAGE_MGT	X	
STRUCTFIX	✓	
STRUCTURE_ADVANCE	-	See STRUCTURE_WALK
STRUCTURE_COMPARE	X	
STRUCTURE_DECODE	✓	
STRUCTURE_WALK	✓	
SUBSCRIPT_MULT	✓	
SUBSCRIPT_RANGE_CHECK	✓	
SUBSCRIPT2_MULT	✓	
SUCCESSOR	-	See STRUCTURE_WALK
SYT_COPIES	✓	
TAG_BITS	-	See HALMAT decoding Section 3.3.8
TERMINATE	✓	
TRUE_INX	X	
TYPE_BITS	-	See HALMAT decoding Section 3.3.8
UNARYOP	X	
UNRECOGNIZABLE	✓	
UNSPEC	X	
UPDATE_ASSIGN_CHECK	X	
UPDATE_CHECK	✓	
UPDATE_INX_USAGE	✓	
VAC_COPIES	✓	
VARIABLES	✓	
VECMAT_ASSIGN	X	
VECMAT_CONVERT	X	
VERIFY_INX_USAGE	✓	
VMCALL	✓	
X_BITS	-	See HALMAT decoding Section 3.3.8

Purpose:

Absolute value function.

Parameters Passed:

VALUE: A value.

Local Variables:

None.

Value Returned:

The absolute value of VALUE.

Procedure

Purpose:

To establish addressing for a structure terminal. If BACKUP_REG ≥ 0 then REG is a base register; otherwise, BACKUP_REG points to a checkpointed base register. To take care of large displacements, everything is incorporated into the base if necessary.

Parameters:

PTR: stack entry for structure

OP: symbol table entry for terminal

REF: 0 or 1 for first or second structure in comparison

TBASE: if $\neq 0$, then desired base register.

Procedure

Purpose:

Given a symbol table entry and the run-time location for it, assign it a specific base register and a specific displacement taking into account all addressing modes of the hardware. Notice that if the location cannot be reached from an existing base register, one must be created. Since the hardware has a limited number of registers, virtual registers are created (SYT_BASE<0) and subsequently code will be generated to load virtual base registers into hardware registers.

Procedure

Purpose:

To lay out storage for a structure template. When INITIALISE has completed processing a minor node of a structure template, the symbol table pointers for all of the nodes are at the end of SYT_SORT. A minor node must have contiguous storage to allow passing such structures as procedure parameters; thus, layout is done for each minor node, rather than once for the entire structure. Storage is layed out using the same algorithms as for regular storage allocation (i.e. packing, minimizing offsets, minimizing boundary alignments). Addresses relative to the minor node point are filled into SYT_ADDR. These will be amended to be relative to the major structure node when INITIALISE completes the structure. Notice that storage is only layed out here, storage will be allocated if a variable of this type is declared.

Parameters:

PTR: Symbol table entry for minor node point.

ALLOCATE_TEMPORARY

Purpose:

Set up storage for temporaries of DO group.

Local Variables:

TEMP - pointer to indirect stack.

TYP - type of variable.

Parameters Passed:

ptr - symbol table pointer to first temporary.

Communicates via:

Symbol table.

Description:

If it has already been done, return; otherwise, follow list in SYT_LINK1, allocate space; copy information from indirect stack to symbol table; set up implied initialization; return stack entry.

Procedure

Purpose:

To set up arguments for vector-matrix operations. This includes GET_OPERAND, conversion if the precisions do not agree, conversion of either operand if it is remote or is a partition.

Procedure

Purpose:

To emit code for RX and RR arithmetic by mode.

Parameters Passed:

OP:	The operator code.
OP1:	A pointer to the Indirect Stack entry for the first operand.
OP2:	A pointer to the Indirect Stack entry for the second operator.
OPTYPE:	The operand type.
BIAS:	The bias for the instruction: RR or RX.

Local Variables:

INST:	The opcode for the instruction.

Communicates via:

Calling the code emitting routines.

Description:

The register type of the first operand's register, R_TYPE(REG(OP1)), is set to OPTYPE. If the operand type is double precision scalar and one of several certain operators is being used, the operator type can be considered to be single precision scalar. If the second Indirect Stack entry's form is VAC, it is a register temporary. This means that an RR type instruction can be used, BIAS=RR; otherwise, an RX instruction must be used.

The instruction's opcode INST is computed using the following equation:

ARITH_OP(OP) + MODE_MOD(OPMODE(OPTYPE) + BIAS)

where ARITH_OP primarily provies the second hex digit of the opcode, and MODE_MOD modifies the fist hex digit according to the instruction mode.

For an RR instruction, EMITRR is called to emit the code. If the operator is binary, the usage of the second operand's register is decremented since its contents have one less claim on them.

For an RX instruction, if the form of the second operand is a literal and the operand mode is halfword integer, a check is made to see if there is an RI form of the instruction. The instruction has an RI form if the AP101INST entry for INST+"60" is non-zero. Halfword integer opcodes have a first digit of 3; adding "60", gives a first digit of 9, which characterizes RI instruction. If the instruction has an RI form, the FORM and LOC fields of OP2's Stack entry are changed to a form appropriate for generating intermediate code. If the second operand is a literal and no RI instruction form exists. SAVE_LITERAL is called to save the literal in the appropriate literal pool.

For all RX instructions, GUARANTEE_ADDRESSABLE is called to amke sure that OP2 can be addressed using the instruction, INST. EMITOP is called to emit the instruction. DROP_INX(OP2) is called to drop OP2's index register. DROPSAVE(OP2) is called to indicate, that if OP2's form is WORK, the temporary storage used by it has one less claim.

References:

The Operand and Operator Tables, Opcode Construction.

Function

Purpose:

To generate code to load or modify current index by array loop index. If OP (see below) is 0 then just generate code to load the index. Notice that if SHIFTCT ≠ 0, an attempt is made to find the index in a register both with the given value and with 0 before the load code is emitted. If OP ≠ 0, generate code to add increment to index. Notice that an attempt is first made to find the increment in a register and use RR code; if impossible, then do AH.

Returns:

Stack pointer for index.

Parameters:

OP: Stack pointer for index or 0

INDEX: Initial value or increment

SHIFTCT: Required shift to convert array subscript to index (depends on width of data)

BIT_MASK

Procedure

Purpose:

To mask bit operands according to size.

Parameters Passed:

OPCODE: The operator used.

OP: A pointer to the Indirect Stack entry for the bit operand.

SIZE: The bit length of the operand.

SHCOUNT: A pointer to the Indirect Stack entry indicating the bit position within a location the bit operand starts at.

Local Variables:

MASK: The mask used.

PTR: A pointer to the Indirect Stack entry for the mask.

RM: A pointer to an Indirect Stack entry of form VAC used for shifting the mask if SHCOUNT is not a literal.

Communicates via:

Calling routines to emit code.

Description:

If there is shift and the FORM of the shift's stack entry is LITERAL, then the amount of shift is known. MASK is then XITAB(SIZE), a string of 1's of length SIZE, shifted by VAL(SHIFT), the shift. Since the shift is incorporated into the mask, SHIFT can be set to zero. Otherwise, MASK, the mask, is XITAB(SIZE). GET_INTEGER_LITERAL is called to set up a stack entry for the mask, and to get a pointer to it, PTR. The type of this entry will be fullword or halfword integer according to whether OP is full or halfword.

If SHIFT is still non-zero, it represents the result of bit subscripting and has form VAC or WORK. GET_VAC is called to get a pointer to a VAC Indirect Stack entry, RM. The register for this VAC entry is loaded with the mask by calling LOAD_NUM, and BIT_SHIFT is called to shift the mask by the amount represented by SHIFT. CHECK_VAC is called in case OP was checkpointed by getting a register for the mask. Then ARITH_BY_MODE is called to perform the masking. DROP_VAC is called to drop the entry for the mask which is no longer needed.

If the shift is zero, ARITH_BY_MODE is still called to do the masking, but the pointer to the Indirect Stack entry for the mask is used as a parameter instead of the pointer to the VAC for the mask used in the previous case.

The stack entry for the mask is returned.

Procedure

Purpose:

To shift bit operands according to stack shift description.

Parameters Passed:

OPCODE: The opcode for the shift type.

R: A pointer to an Indirect Stack entry indicating the shift of form:

LITERAL : if no subscripting has taken place, the entry's VAL is the shift.

VAC/WORK: if bit subscripting has taken place, the entry's REG contains the shift.

FLAG: A flag indicating that if the shift is in a register, the register's usage should not be decremented after the shift instruction.

Local Variables:

None.

Communicates via:

Emitting code.

Description:

BIT_SHIFT generates shift instructions according to the form of OP since the shift information is stored in different fields of the stack entry according to the form of the operand. Also, if the operand is not a literal, CHECK_VAC must be called before emitting code in case the VAC has been checkpointed. After generating the code, if the flag is not true, the usage of the register containing the shift must be decremented.

BIT_STORE

Procedure

Purpose:

To generate code to store a bit variable. If the store is into a character SUBBIT or double word scalar SUBBIT, out of line code is generated using GENLIBCALL('DSST') or CHAR_CALL(XCSST). In all other cases, in-line code is generated which may include:

FORCE_ACCUMULATOR (value to be stored)

GUARANTEE_ADDRESSABLE (place to store)

loading of contents of place to store, and shift, masking, and ORing operations.

Parameters:

ROP: indirect stack entry for value to store

OP: indirect stack entry for place to store into

CONFLICT: true if ROP will be used again (CSE or multiple assignment)

Local Variables:

BOP: temporary

IMPMASK: true if contents of register containing ROP is scrambled.

SHORTLIT: true if ROP is literal consisting of all zeros or all ones of the length of OP.

Procedure

Purpose:

To clean up at the end of a block. If this is a function and close is reachable, insert run time error message. Generate SVC if not just a procedure/function. Restore previous location counter and set that register contents are unknown.

BLOCK_OPEN

Procedure

Purpose:

To initialize at HALMAT block open. Emit identifier for scope number in compilation unit. Emit MAXERR and ERRSEG. Emit Z-cons for all remotes. For each variable in the block

if NAME, initialize to null,

if BIT, set to zeroes,

if character string, emit maximum size,

if structure, walk structure performing above operations on the nodes.

Emit standard header code.

Handle parameters in Registers.

For each temporary, generate automatic initialization code via SET_AUTO_IMPLIED.

Procedure

Purpose:

To generate calls to character manipulation library routines. The routine generates load of all necessary registers with some help from SET_CHAR_DESC if there is components subscripting. GENLIBCALL then actually issues the call.

Parameters:

OPCODE: the operation to be performed

OP0: if ≠0 then result goes to address of OP0

OP1: operand

OP2: optional second operand

OP3: optional third operand (bit string for SUBBIT)

CHAR_INDEX

Function

Purpose:

To initialize at HALMAT block open. Emit identifier for scope number in compilation unit. Emit MAXERR and ERRSEG. Emit Z-cons for all remotes. For each variable in the block

if NAME, initialize to null,

if character string, emit maximum size,

if structure, walk structure performing above operations on the nodes.

Emit standard header code.

Handle parameters.

For each temporary, generate automatic initialization code via SET_AUTO_IMPLIED.

Function

Purpose:

To find an occurrence of one character string in another.

Parameters Passed:

STRING1: The character string being searched.

STRING2: The character string being searched for.

Local Variables:

L1: Length of STRING1.

L2: Length of STRING2.

I: Temporary Do Loop variable.

Value Returned:

The index of the beginning of STRING2 in STRING1 or -1 if it is not there.

Procedure

Purpose:

To generate code to perform a stack walk and set up pointer addresses for addressing of scoped variables allocated on the stack.

Parameters Passed:

R: The register used in addressing; a negative value means no register specified.

OP: A pointer to the Indirect Stack entry whose address is being determined.

Local Variables:

ALOC: The Symbol Table entry associated with the Indirect Stack entry.

SCOPE: SYT_SCOPE (ALOC), the CSECT the variable is defined in.

IX: An index register used for addressing.

Communicates via:

Indirect Stack.

References:

The Block Definition Table, The Local Block Data area, addressing the Runtime Stack Frame, Section 3.1.1.3, Scoped Formal Parameter Addressing Forms, HAL/S-FC Compiler Spec.

Description:

If the Stack entry is a pointer to a task, program, or compool, SETUP_ADCON is called to set up addressing and the procedure returns. If the scope of the entry is INDEXNEST, the CSECT for which code is being generated, the procedure returns.

If no register number has been specified, GET_R is called to get a register to use, R. FINDAC is called to find an index register, IX. Then the appropriate code emitters are called to generate a loop, which goes back through the runtime stack frames until it finds a frame whose nest level equals that of the parameter. The code generated is:

```
LHI IX, <Block ID>       Block ID is SHL(COMPUNIT_ID,7)+SCOPE
LR   R, TEMPBASE         Load R with the address of the run-
                           time stack frame
L    R,STACK_LINK(R)     Load R with the address of the
                           preceding frame
CH@ IX,NEW_LOCAL_BASE(R)
                         Compare the variable's scope with
                           the scope number of the frame.
                           NEW_LOCAL_BASE(R) is the address
                           of the Local Block Data Area.
BNE -3
```

The USAGE of R is 2 since there is one claim on the register. The USAGE of IX is set to 0 to show it is no longer being used. OP's stack entry is changed to have form CSYM. This indicates that it has its own base and displacement for addressing. The following fields of the entry are modified:

```
FORM(OP) = CSYM

BASE(OP),BACKUP_REG(OP)=R    the register now contains a
                             base address for OP

DISP(OP) = SYT_DISP(ALOC)
```

CHECK_CSYM_INX

Procedure

Purpose:

To combine the contents of an Indirect Stack entry's index register with the contents of a register containing a value used for subscript or array subscripting.

Parameters Passed:

OP: An Indirect Stack entry.

R: A register containing a term that is used for array and subscript indexing for OP.

Local Variables:

None.

Communications via:

Indirect Stack.

Description:

If the register has more than one user, the contents of OP's index register cannot be combined with it. If OP has a shift associated with its operand type and the self-aligning option is in effect, the contents of INX(OP) and R cannot be directly combined.

If it is possible to combine the register contents and OP's index register has been checkpointed, the contents of OP's index register are added to R. DROP_INX is called to drop OP's index register since it is no longer needed. R is marked unrecognizable since it has been modified.

Function

Purpose:

To check if an Indirect Stack entry refers to remote data.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

None.

Value Returned:

True if entry refers to remote data, false otherwise.

Description:

The entry's form is checked to see if it has a corresponding Symbol Table entry using the SYMFORM array. If it does, the entry's flags are checked for the REMOTE attribute.

CHECK_VAC

Procedure

Purpose:

To check an Indirect Stack entry for a supposed VAC, to see if it has been checkpointed.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

R: An optional parameter to specify a register for the VAC.

Local Variables:

None.

Communicates via:

Indirect Stack.

References:

The procedures CHECKPOINT_REG, GET_VAC.

Description:

If the form of the stack entry is WORK, then the VAC has been checkpointed. If R is not specified, FINDAC is called to find a new indexing register, REG(OP), for the VAC. EMIT_BY_MODE is called to load the register with the contents of the VAC. The usage of REG(OP) is set to 2 to show there is a claim on the register; the DEL Add of OP is decremented by 2 to show there is one less claim on the WORD entry's Storage Descriptor Stack entry. DROPSAVE is called to see if the Storage Descriptor Stack entry is still necessary. The form of OP is changed back to VAC.

Procedure

Purpose:

To save the contents of a register in a temporary location, and to modify Indirect Stack Entries refering to it.

Parameters Passed:

R: The register to be saved.

Local Variables:

RTYPE: The operand type contained in the register.

PTR: A pointer to an Indirect Stack Entry set up to point to the Storage Desdriptor Stack entry for the register contents.

I: A do loop temporary.

Communicates via:

Changes the Indirect Stack.

Message Condition:

DIAGNOSTICS.

Description:

The procedure checks USAGE(R) to see if it is worth saving the contents of the register. If it is, it calls GETFREESPACE to get storage for the register in the Runtime Stack. PTR is set to the Indirect Stack Entry returned by GETFREESPACE. A line of code to store the contents of the register in Temporary Storage is provided by calling EMIT_BY_MODE.

The WORK_USAGE of the Storage Descriptor Stack entry describing the temporary storage is set to zero. CHECKPOINT_REG is going to check all allocated Indirect Stack entries. For those whose STACK_PTR is negative, (if they use the register) the entries will be modified to reflect the storing of the register, and WORK_USAGE(LOC(PTR) and DEL(PTR) are used to keep track of the use of the stored entry. There are three ways that an Indirect Stack entry, I, may use the register.

1. FORM(I) = VAC and REG(I) = R

   In this case, the form of the entry is changed to WORK, and the remaining fields of I are modified to agree with PTR's fields. WORK_USAGE(LOC(PTR)) is incremented.

2. INX(I) = R

   In this case INX(I) is set to -PTR to indicate the register's contents are stored. If this is the first use of the register as an index, indicated by DEL(PTR) = 0, WORK_USAGE(LOC(PTR)) is incremented to show another usage for the Temporary Storage. DEL(PTR) is incremented by two to show another use of the register, it corresponds to USAGE(R).

3. FORM(I) = CSYM and BACKUP_REG(I) = R

   In this case, BASE(I) and BACKUP_REG(I) are set to -PTR to indicate the register's contents are stored. Since the CSYM is the only user of the register, DEL(PTR) is set to two. WORK_USAGE(LOC(PTR)) is incremented to show another usage of Temporary Storage.

After modifying the relevant Indirect Stack entries, DEL(PTR) is checked. If DEL(PTR)=0, the stack entry, PTR, is not being used, and is returned. Finally, the register is cleared, and if the contents of the register were DSCALAR, R+1 is cleared as well.

Procedure

Errors Detected:

BS 105: Data storage capacity exceeded (Severity 1).

BS 120: Data storage capacity exceeded (Severity 2).

Purpose:

To check for too much storage allocation in a runtime stack frame.

Parameters Passed:

NUMBER: The size of the storage allocated.

SEVERITY: A number used to determine which error to report.

Local Variables:

None.

Communicates via:

Calling the appropriate error routine if necessary.

Description:

If NUMBER > 200,000, the maximum bytes of storage, the error is reported to ERRORS.

CLEAR_R

Procedure

Purpose:

To clear the Register Table entries associated with a given register.

Parameters Passed:

R: A register number.

Local Variables:

None.

Communicates via:

The Register Table.

Description:

This procedure sets all Register Table fields with index R to zero.

Procedure

Purpose:

To clear the Register Table entries for all registers.

Parameters Passed:

None.

Local Variables:

I: Do Loop temporary.

Communicates via:

Register Table.

Description:

CLEAR_REGS calls CLEAR_REG for each register to clear its Register Table entries.

Function

Purpose:

To determine the core requirements of a character string.

Parameters Passed:

LEN: The size of the string.

Local Variables:

None.

Value Returned:

SHR(LEN,1)+LEN & 1).

Description:

If LEN is even, the value returned is 1/2 LEN.

If LEN is odd, the value returned is 1/2(LEN+1).

DECODEPIP

Procedure

Purpose:

To decode a HALMAT operand.

Parameters Passed:

OP: The number of the operand word in the HALMAT instruction which is to be decoded.

N: The entry in the TAG2 and TAG3 arrays that is to be used.

Communicates via:

Global variables for the HALMAT operand word fields.

References:

Appendix A1, HAL/S-360 Compiler Spec.

Description:

DECODEPIP takes the OPth operand word following the current HALMAT operator (pointed to by CTR) and decodes it as follows:

OP1	TAG3	TAG1	TAG2	1
16	8	4	3	1

where:

OP1: operand field

TAG1: qualifier field

TAG2,TAG3: tag fields

TAG2 and TAG3 are arrayed variables so that information about several HALMAT operand words may be retained. DECODEPIP uses the array entry specified by N.

If the HALMAT compiler option is in effect, DECODEPIP outputs the operand word in the following format:

```
OP1(TAG1) TAG3, TAG2: BLOCK#,(CTR + OP)
```

DECODEPOP

Procedure

Purpose:

To decode a HALMAT operator word.

Parameters Passed:

CTR: A pointer to the HALMAT operator to be decoded.

Communicates via:

Global variables for the HALMAT operator word fields.

References:

Appendix A.1, HAL/S-360 Compiler Spec.

Description:

DECODEPOP takes the HALMAT operator pointed to by CTR and decodes it as follows:

TAG	NUMOP	CLASS	SUBCODE/ OPCODE	COPT	0
8	7	4	8	3	1

CLASS: The operator class
NUMOP: Number of operands
TAG: Tag field
COPT: pseudo-optimizer tag field

```
IF CLASS=0,  SUBCODE=0
             OPCODE is all 8 bits of its field.

otherwise,   SUBCODE:  first 3 bits
              OPCODE:  last 5 bits
```

SUBCODE and OPCODE are used for classifying the operators.

If the HALMAT compiler option is in effect, DECODEPOP outputs the operator word in the following format:

```
SUBCODE/OPCODE(NUMOP) TAG, COPT: BLOCK#, CTR
```

Procedure

Purpose:

To define the value of a generated statement label.

Parameters Passed:

PTR: A pointer to an Indirect Stack entry of form LBL, FLNO.

FLAG: Indicates user defined statement labels unreachable by GO TO statements, and not marking update blocks, or otherwise unreachable label.

Local Variables:

CODE: The intermediate code opcode for the label.

Communicates via:

Calling SET_LABEL.

Message Conditions:

ASSEMBLER_CODE.

References:

Appendix C, Section on Label Definition in HAL/S-360 Compiler Spec. SYT_DIMS field of the Symbol Table.

Description:

If the stack entry represents a user defined statement label, its type is checked by examining SYT_DIMS. FLAG is set if the label is unreachable by GOTO, and does not define an Update Block to indicate to SET_LABEL that the registers do not have to be cleared. SET_LABEL is called with the following parameters:

VAL(PTR) The phase 2 generated statement number associated with the label.

FLAG described above.

1 indicating that the label is not a phase 2 generated label.

CODE is the intermediate code opcode for the label. It will be ULBL if FORM(PTR) is LBL, and 'LBL if FORM(PTR) is FLNO. EMITC emits the output code indicating the definition of the label. The stack entry is returned since it is no longer necessary.

DESC

Function

Purpose:

To create a descriptor out of a pointer. The argument passed to DESC is in the XPL descriptor format; however, in the calling routine it is not of type CHARACTER. DESC is of type CHARACTER so DESC (ptr) returns exactly what it was passed but the XPL compiler now understands that it is a string.

Parameters Passed:

D: A character string descriptor which is not of type CHARACTER.

Local Variables:

None.

Value Returned:

The same character string descriptor.

Procedure

Purpose:

To determine the size of Indirect Stack entries and whether they are arrayed.

Parameters Passed:

PTR: A pointer to an Indirect Stack entry.

OP1: A pointer to the Symbol Table entry associated with PTR.

Local Variables:

None.

Communicates via:

The global variables AREASAVE, ARRAYNESS and the COPY Indirect Stack field.

Description:

This routine sets ARRAYNESS to the result of GETARRAY#(OP1), a procedure which returns information about the number of dimensions of a Symbol Table entry, and calls SET_AREA(PTR) to determine the size of the entry.

In addition, it sets COPY(PTR) to the number of array dimensions of a stack entry. For most Indirect Stack Entries, this is ARRAYNESS. For terminal nodes of arrayed structures that also possess arrayness, ARRAYNESS only indicates the arrayness of the terminal node. COPY(PTR) must be set to ARRAYNESS+1 to reflect the extra dimension of arrayness induced by the structure itself.

Procedure

Purpose:

Generate code to store HALMAT operand 1 into operands 2 through NUMOP. The left hand sides are first sorted by type and then assignment code is generated for each type in turn. The order in which types are chosen is determined by ASSIGN_TYPES.

In the special case that there is only one left hand side, that the left hand side is a halfword, and the right hand side is a literal, an attempt is made to optimize by generating special purpose code.

Local variables:

ASSIGNC: number of types of left sides

ASSIGNS: number of assignments processed

ASSIGNT: temporary

PROTECT_RIGHTOP: true unless this is the last assignment to be generated. This is used to prevent routines from destroying the value before all assignments have been made.

Procedure

Purpose:

To close outstanding array do loops.

Parameters Passed:

None.

Local Variables:

PTR: Pointer to Indirect Stack entries.

LITOP: Never referenced.

Communicates via:

Array Do Loop Stack, code emitting, ADOPTR.

References:

The HALMAT ADLP, ALPE, IDLP operators, the Array DO Loop and Array Reference Stack, the procedures CHECKPOINT_REG, DOOPEN, GENERATE(ADLP, IDLP, DLPE cases), SEction 3.1.7 HAL/S FC Compiler Spec.

Description:

If there are any outstanding array do loops, DOCOPY(CALL_LEVEL) > 0, DOCLOSE closes them according to DOFORM(CALL_LEVEL).

I. DOFORM(CALL_LEVEL)=0: Was set up for HALMAT ADLP parameters, except for simply array parameters.

Each do loop outstanding at the call level is closed in the following manner. PTR is set to DOINDEX(ADOPTR). ADOPTR is the index of the Array Do Loop Stack entries for the loop to be closed; DOINDEX(ADOPTR) is the pointer to the Indirect Stack entry for the register, TMP, used as the Do loop index. TMP is BACKUP_REG(PTR) rather than REG(PTR) because if the register had been checkpointed, the value of REG would be set to -1 but BACKUP_REG remains unchanged. If the stack entry's form is WORK, the register has been checkpointed. The procedure CHECKPOINT_REG(TMP) is called to clear the register and code is emitted to load it with its former value. DROPTEMP(LOC(TMP)) is called to drop the Temporary storage used for the register's contents, if necessary.

Code is emitted to add DOSTEP(ADOPTR), the increment, to the index register. The zeroeth Indirect Stack entry is given FORM=AIDX and LOC=PTR so that NEW_USAGE(0) can be called to mark all users of the index register unrecognizable. If DOSTEP=1, the special case BIX instruction is generated, which combines the increment and test functions.

PTR is now set to the stack entry for the final value, DORANGE(ADOPTR). Code is emitted for comparing the contents of TMP or PTR according to whether the final value is or known array size or an unknown array size. If the size is unknwon CHECK_ADDRS_NEST must be called to check the scoping of the variable before emitting the code. TMP is no longer needed so its usage is set to zero. The stack entry for the final value is no longer needed so it is returned. EMITBFW is called to emit a conditional branch to the label set up in DOOPEN marking the beginning of the code within the loop.

ADOPTR is decremented and the next array do loop is closed. This process continues until ADOPTR equals SDOPTR(CALL_LEVEL), the value of ADOPTR at the beginning of the reference. The number of Do Loops closed may be greater than the number opened at a call level if arrayness is pushed from an outer to inner level.

II. DOFORM(CALL_LEVEL)=1: IDLP processing - Static Initialization.

In the static initialization case, no code is actually generated for array do loops, rather DOCLOSE runs through all the possible values of the array indices. For each set of values, the DOBLK and DOCTR values are used to position the HALMAT to the IDLP operator. NEXTCODE is called to decode the following HALMAT instruction and control goes to RESTART, the part of the main program that calls GENERATE. The HALMAT following the IDLP operator is decoded with the new index values; when the DLPE operator is reached. DOCLOSE is called again. This continues until all the array indices have been gone through.

III. DOFORM(CALL_LEVEL)=2: Simply array parameters.

No do loops are necessary so DOCLOSE does nothing. In the case of simple structure array parameters with arrayness, STRUCTFIX calls DOOPEN and DOOPEN changes DOFORM to 0.

After all the do loops are closed, DOCOPY is set to zero to reflect the end of the array reference.

Procedure

Errors Detected:

BS 119: Exceeded arrayness stack size.

Purpose:

To set up a do loop to process a dimension of arrayness.

Parameters Passed:

START: The starting value of the do loop index.

STEP: The step by which the index is incremented.

STOP: The final value of the Do Loop index if array size is known, a negative pointer to a Symbol Table entry if the array size is unknown.

Local Variables:

PTR: A pointer to an Indirect Stack entry used for a register as an index variable for the loop that is set up.

Communicates via:

The Array Do Loop Stack.

References:

The HALMAT ADLP operator, the Array Do Loop Stack, the procedure DOCLOSE, Section 3.1.7 HAL/S-FC Compiler Spec.

Description:

ADOPTR, the pointer to the last allocated Array Do Loop stack entry is incremented. Several fields associated with the new entry are assigned. DOSTEP is set to step. GETSTATNO is called to get a statement number to assign to DOLABEL. This statement number will be used to label the beginning of the code within the Loop. GET_VAC(-1) is called to get a register that can be used as an index, DOINDEX and PTR are assigned to this value. BACKUP_REG(PTR) is set to REG(PTR) to ensure that the value of the number of the register used as an index is saved if the register is checkpointed, so that the correct code may be generated.

DORANGE(ADOPTR) is set to a pointer for an Indirect Stack entry for the final loop value determined by SET_ARRAY_SIZE if the array size is unknown, and GET_INTEGER_LITERAL if it is known.

The procedure DOCLOSE takes care of the remaining code generation for the loop including incrementing the index and checking to see if it has attained its final value.

Before calling GET_VAC, RESUME_LOCCTR(NARGINDEX) is called. This is to ensure that the proper location counter is in use; this call is needed because if initialization is in progress, the location counter will be using the data CSECT, DATABASE. Once an index register, TMP, has been obtained, LOAD_NUM is called to load it with a starting value. SET_LABEL(DOLABEL(ADOPTR),1) is called to set the label marking the beginning of the code within the loop. The flag of 1 indicates that the registers do not have to be cleared since there are no external GO TOs to the label.

TMP's Register Table entry is updated as follows:

R_CONTENTS = AIDX	an array index
USAGE = 3	usage is known
R_VAR = PTR	the stack entry describing it
R_TYPE = INTEGER	

DROP_INX

Procedure

Purpose:

To drop the index register used by an Indirect Stack entry.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

None.

Communicates via:

Indirect Stack.

Description:

OFF_INX(INX(OP)) is called to decrement the usage of INX(OP). INX(OP) is set to zero to show there is no index register.

DROP_VAC

Procedure

Purpose:

To drop an Indirect Stack entry set up as a register temporary.

Parameters Passed:

PTR: A pointer to the Indirect Stack entry.

Local Variables:

None.

Communicates via:

Indirect Stack, Register Table.

Description:

If the form of the entry is VAC, the usage of its register is decremented, and RETURN_STACK_ENTRY is called to return the entry.

Procedure

Purpose:

To drop temporary storage space saved in the SAVEPOINT array.

Parameters Passed:

None.

Local Variables:

I: A Do Loop temporary.

Communicates via:

SAVEPTR, calling DROPTEMP to modify the Storage Descriptor Stack.

References:

The procedure DROPSAVE.

Description:

DROPFREESPACE calls DROPTEMP to drop all undropped Storage Descriptor Stack entries saved in the SAVEPOINT array. If SAVEPOINT is zero, the entry has been dropped by DROPOUT. SAVEPTR is set to zero to indicate there are no saved entries to be dropped.

DROPLIST

Procedure

Purpose:

To drop temporary space saved due to arrayness.

Parameters Passed:

LEVEL: The level of array reference.

Local Variables:

PTR: A pointer used for chaining through the ARRAYPOINT entries pointed to by SDOTEMP(LEVEL).

Communicates via:

Calling DROPTEMP to change the Storage Descriptor Stack.

References:

The procedures DROPSAVE, FREE_TEMPORARY, The Storage Descriptor and Array Reference Stacks.

Description:

SDOTEMP(LEVEL) points to the beginning of a linked list of Storage Descriptor Stack entries used for processing array references that are no longer needed. ARRAYPOINT of each list member points to the next member; ARRAYPOINT of the last entry is zero. DROPLIST goes down the linked list calling DROPTEMP for each list member. It leaves SDOTEMP(LEVEL) equal to zero indicating that there are no unneeded temporary storage entries left at that call level.

DROPOUT

Procedure

Purpose:

To force the immediate release of a dropped temporary storage entry.

Parameters Passed:

ENTRY: A pointer to an Indirect Stack Entry.

Local Variables:

I: A do loop temporary.

Communicates via:

Calling DROPTEMP to change the Storage Descriptor Stack.

References:

SAVEPOINT, the procedure DROPSAVE.

Description:

ENTRY is checked to see that its form is WORK; if it is not, it does not represent a Storage Descriptor Stack Entry. If its form is WORK, then ENTRY is set to LOC(ENTRY), the pointer to the Storage Descriptor Stack entry. SAVEPOINT, the array of entries to be dropped is searched to see whether it has been dropped already. If it has not been dropped, DROPTEMP is called to drop the entry. The SAVEPOINT entry that pointed to ENTRY is set to zero to show that SAVEPOINT entry has been dropped when DROPFREESPACE is called.

Procedure

Purpose:

To determine if a Storage Descriptor Stack entry is no longer needed, and if so, to save details of the entry.

Parameters Passed:

ENTRY: A pointer to an Indirect Stack entry.

Local Variables:

I, J: Temporary variables.

Communicates via:

The arrays SAVEPOINT, ARRAYPOINT, SDOTEMP.

Description:

The Indirect Stack entry's form is checked since only WORK entries represent Storage Descriptor Stack entries. If the entry's form is WORK, a pointer to the Storage Descriptor Stack entry is obtained from the Indirect Stack entry's LOC field. The usage of the Storage Descriptor stack entry, WORK_USAGE is decremented. If the entry is no longer needed, WORK_USAGE=0, details identifying the entry are saved in one of two places:

1) A linked list pointed to by SDOTEMP of any currently nested call level.

   If an array reference is being processed at any of the current levels of nesting and it is a simple arrayed parameter reference, or the reference occurred after the storage was allocated, the SDOTEMP linked list is used. This ensures the storage will not be freed until the reference is completed. The linked list pointed to by SDOTEMP and linked by the member's ARRAYPOINT fields is searched for the entry since FREE_TEMPORARY may have added it to the list. If the entry is not on the list, it is added to the beginning of the list, and SDOTEMP will point to it.

2) The SAVEPOINT array

The SAVEPOINT array is searched for the entry. If the entry is not there, SAVEPTR is incremented to the first free SAVEPOINT entry. The SAVEPOINT entry will contain a pointer to the Storage Descriptor Stack entry. The SAVEPOINT entries are allocated consecutively and dropped after each HAL/S source statement.

This approach enables the compiler to indicate that a temporary will not be needed after the current operation and to actually deallocate the space after code has been generated to perform the operation.

Procedure

Purpose:

To release a Storage Descriptor Stack entry.

Parameters Passed:

ENTRY: A pointer to a Storage Descriptor Stack entry.

Local Variables:

None.

Communicates via:

Storage Descriptor Stack.

References:

Storage Descriptor Stack, the procedure GETFREESPACE.

Description:

The procedure searches the linked list of allocated Storage Descriptor Stack entries formed by the entry's POINT field until it finds the entry whose POINT field is ENTRY. To do this, the procedure uses two temporary variables, IX1 and IX2, providing pointers to a member of the list and to the member it is linked to. Two pointers are necessary since a link may be removed from the middle of the chain. Chaining continues until the second pointer points to ENTRY. Then UPPER(ENTRY) is set to -1 to show the entry has been deallocated. The POINT field of the first pointer is set to POINT(ENTRY) so that ENTRY is removed from the linked list without breaking the chain.

Procedure

Purpose:

To prepare for setting up array do loops from HALMAT, and to call DOOPEN to set them up.

Parameters Passed:

LEVEL: The Array Reference Level.

Local Variables:

SAVCTR: A variable used to temporarily save the pointer to the current HALMAT operator.

Communicates via:

Array Reference Stacks, Calling DOOPEN to set up array do loops.

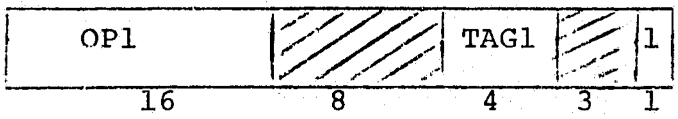
References:

Array Do Loop Declarations, the HALMAT ADLP operator.

Description:

SAVCTR saves the value of CTR, the current HALMAT operator, so that CTR can take on the value of DOCTR(LEVEL), the pointer to the HALMAT ADLP operator for the array reference. Saving CTR is unnecessary when the procedure is called from GENERATE, but is necessary when it is called from STRUCTFIX xince their CTR is not the same as DOCTR(LEVEL).

After calling SAVE_REGS to save the necessary registers, a do loop is opened for each dimension of arrayness. There are DOCOPY(CALL_LEVEL) dimensions. Before opening the do loop, all registers in use much be checkpointed and the HALMAT operand word for the array dimension decoded by calling CHECKPOINT_REG and DECODEPIP. A HALMAT ADLP operand word has the form shown below:



TAG1 is IMD when array size is known: OP1 gives value.

ASIZ when array size is unknown: OP1 gives a symbol table reference.

EMIT_ARRAY_DO calls DOOPEN with parameters indicating an index starting at 1, with a step of 1. The third parameter indicates the end condition which is OP1 if TAG1=IMD, and -OP1 if TAG1=ASIZ.

EMIT_ARRAY_DO sets DOFORM(LEVEL) to zero to indicate that array do loops have been set up. It restores the value of CTR before returning.

Procedure

Purpose:

To actually emit a card for the linkage editor.

Notice that CARDIMAGE and COLUMN are really the same array. On initial entry, a descriptor is built in DUMMY_CHAR so that COLUMN can be manipulated as a character string. All other times, the current contents of COLUMN are output unless either no data is on the card or the type of card has not been set (i.e. CARDIMAGE=0). After outputting the card, the array is overwritten with blanks, the identification field (CARDIMAGE(19)) is set to "I**2", the card count is bumped and inserted after the I**2.

EMIT_ESD_CARDS

Purpose:

To produce SYM and ESD cards. The SYM cards are produced using EMIT_SYM_CARDS. The ESD cards are then emitted three ESDs to a card (DO I = 1 TO (ESD_MAX+2)/3) in CARDIMAGE columns 5, 9, and 13 (DO J = 5 TO 13 BY 4). Since the actual character string (not a pointer) must be inserted, INLINE code is used to copy the string.

Reference:

AP-101 Support Software/SDL ICD Chapter 2.

Procedure

Purpose:

Build the SVC argument list describing an event expression. All necessary information has already been inserted in EV_EXP (by STACK_EVENT) and in EV_OP (by SET_EVENT_OPERAND).

Procedure

Purpose:

To emit SYM cards.

Example:

Assume:

A. Compilation Unit Name is COMP_UNIT, a COMSUB
B. Version number is 20
C. Stack size is 100
D. References are made to COMSUBS EXT1 of Version 10, and EXT2 of Version 100
E. Local variables are A and B

The FC compiler will produce SYM cards for:

	NAME	TYPE	ADDRESS	COMMENT
1.	#CCOMPUN	CSECT	0	Defines CSECT
2.	STACK	DSECT	0	
3.	STACKEND	VAR	100	Address of 100 is stack size
4.	HALS/FC	DSECT	20	Invalid label, HALS/FC or HALS/360 used to indicate beginning of version data. Address of HALS/FC is the version of COMP_UNIT
5.	EXT1	DSECT	10	Version of EXT1
6.	EXT2	DSECT	100	Version of EXT2
7.	HALS/END	DSECT		

	NAME	TYPE	ADDRESS	COMMENT
8.	#DCOMPUN	CSECT	2010	
9.		SPOFF		turn off storage protect
10.	A	VAR	2010	
11.	B	VAR	2012	

If the compilation unit contains nested scopes, a triplet of cards (similar to cards 1-3) will be produced after card 7 for each such CSECT. If the compilation unit is a COMPOOL, a copy of card 1 is inserted immediately before card 8. Notice that all non-stack allocated data is described in one long list after card 9.

The 360 compiler does not produce cards 2 and 3. The name on card 4 is changed to HALS/360. Cards 8-11 are not produced.

The information between HALS/FC (or HALS/360) DSECTS and the HALS/END can be generated only by the compiler; therefore, no template checking of assembly routines can be accomplished.

Local Procedures:

EMIT_SYM_CARD	outputs the current card and initializes the for next one
EMIT_SYM	inserts symbolic information into the current SYM card
EMIT_SYM_DATA	inserts numeric data into the current SYM card

Local Variables:

I: ESD counter

J: current column on card

B: procedure number counter

P: pointer to symbol table entry for variable

T: type of variable.

Reference:

HAL/S-SDL ICD, Chapter 2.

Procedure

Purpose:

To emit the necessary branch instruction or modify the branch address of an existing instruction so as to perform a WHILE/UNTIL test.

Parameters Passed:

OP: An indirect stack entry for the "condition" to be tested.

LBL: Branch address

Communicates via:

Generating code and LOCATION array.

Description:

If this is UNTIL, true and false conditions are inverted. If OP is a RELATIONAL then generate branch instruction using condition in REG(OP). If OP is not a relational then the necessary branch instructions have already been generated in evaluating an expression FIX_INTLBL is used to set the true branch to jump to LBL and SET_LABEL is used to define the label point for the false branch to be the current location yielding the effect of falling through.

In all cases, return the stack entry for OP.

Procedure

Purpose:

To emit a string into the intermediate code file. The routine is complicated because a normal string assigment only copies a pointer, not the actual string; therefore, INLINE code must be used to actually copy the string. In the FC compiler, a translation is made from EBCDIC to DEU code.

Parameters:

STRING: The literal to be emitted.

ILEN: The maximum length that the string can attain. This is necessary when EMITSTRING is used to perform static initialization.

ENTER_CHAR_LIT

Function

Purpose:

To enter a literal string into LIT_CHAR. This routine is necessary because a normal XPL character string assignment moves descriptors, not strings. LIT_CHAR must actually contain the string. If not, when string storage is compactified, strings pointed to from LIT pages not in core would be garbage collected.

Parameters:

STR: a character string to be moved to LIT_CHAR.

Returns: a pointer into LIT_CHAR.

Procedure

Purpose:

To generate code for A^B. If B is a positive integer constant and DATATYPE(A) is scalar then special purpose code is emitted to do successive multiplies; otherwise, the operands are forced into accumulators and a library call is generated.

Parameters:

OPCODE: opcode part of HALMAT instruction.

Local Variables:

R: register containing $A^?$ where ? is a power of 2

WRK: register containing partial result

I: what remains of B after some multiplications have been generated

EXP_RCLASS: A mapping from TYPE to the register type required for exponentiating TYPE.

FIX_INTLBL

Procedure

Purpose:

To generate effect of identifying an internal flow label with a phase 2 statement number.

Parameters Passed:

LBL: internal flow number

STATNO: phase 2 statement number

Communicates via:

LABEL_ARRAY, LOCATION, and code generation.

Description:

If LBL is already defined, define STATNO to be the current location and generate an unconditional jump to LBL; otherwise, define LBL to have the same LOCATION as STATNO.

Procedure

Purpose:

To redefine the destination of a statement number.

Parameters Passed:

LAB1: The statement number whose location is to be redefined.

LAB2: The new destination of the statement.

Local Variables:

None.

Message Condition:

ASSEMBLER_CODE.

Description:

LOCATION(LAB1) is set to -LAB2 to indicate its destination is the same as LAB2. A positive LOCATION value is the actual destination of the label, a negative value indicates the index in LOCATION to try to find the destination.

FIX_STRUCT_INX

Procedure

Purpose:

To combine the contents of a register used for computing an array or subscript indexing term for an Indirect Stack entry with the entry's index register, and aligning absolute index values if self alignment is present.

Parameters Passed:

IX: A register used to compute an array of subscript indexing term.

OP: A pointer to the Indirect Stack entry that the indexing term will be used to address.

Local Variables:

SHFT: The shift associated with OP's operand type.

R: An index register.

TEMP: A temporary pointer to an Indirect Stack entry.

Communications via:

Indirect Stack, Emitting code.

Description:

If IX is not zero, CHECK_CSYM_INDEX is called to see whether OP's index should be combined with the contents of IX. If it is combined, INX(OP) will be set to zero by CHECK_CSYM_INX, and FIX_STRUCT_INX will set it to IX and return.

If INX(OP) is not zero, OP has an index register. If the register has been checkpointed, FINDAC is called to find an index register that can be loaded with the stored index value. If the SELF_ALIGNING option is in effect VERIFY_INX_USAGE is called to protect any other users of the index register before it is modified. Then the absolute contents of the index register are re-aligned by shifting them right. This is necessary because at this point the index register is used only for addressing structures and since structure nodes do not all have the same halfword width so the index is absolute and must be shifted to take into account the automatic alignment.

VERIFY_INX_USAGE is called to protect any users of OP's index register in case the procedure was not called previously since the self-aligning option was not in effect. The contents of IX are added to the contents of INX(OP).

Function

Purpose:

To generate code to force a value into an accumulator. If the value is not a VAC, an attempt is made to find it in a register both shifted and not shifted. If necessary, an existing register copy is copied. If all else fails, code to load the register is issued.

Returns:

Register containing the value.

Parameters:

OP: indirect stack entry for value to be loaded

OPTYPE: desired type in register

ACCLASS: type of register desired

SHIFTCT: shift to be applied to value (useful if value will be used as an index)

Procedure

Purpose:

To generate code to force an address pointer of the right type into a register, including storing the current contents if necessary.

Parameters:

TR: the register number, if TR < 0 then routine will GET_R

OP: indirect stack entry for item whose address is interesting

FLAG: {1 reserve register (i.e. USAGE=2)
 0 otherwise

FOR_NAME: pointer should be of type suitable for a name assignment

BY_NAME: pointer should be of type suitable for ASSIGN parameter. This may be a pointer to a pointer.

FORCE_BY_MODE

Procedure

Purpose:

To generate code to force an element into an accumulator (FORCE_ACCUMULATOR) and do all necessary type conversions.

Parameters:

OP: indirect stack entry for item desired

MODE: type item should be forced to

RTYPE: type of accumulator desired, if 0 then FORCE_ACCUMULATOR will make an automatic choice

6

FORM_BD

Procedure

Purpose:

To form a base (B) displacement (D) pair for an address. Most cases are simple, the bulk of the code simply formats listing. For relocateable entries, an attempt is made to base them off PROGBASE instead. Relocateable entries with negative displacementsbecome positive displacements with a flag in RLD_REF. Branch displacements are in turn handled by an internal routine FORM_BADDR.

In FORM_BADDR

SRSTYPE will only occur for a specific pair of branch forward branch backward instructions. Otherwise, negative displacements are handled by the bit immediately before the displacement. If the displacement is too large, switch to extended addressing. Notice that extended addresses must be relocated.

Parameters:

I: The LHS-RHS subscript for addressing the argument.

FORMAT

Function

Purpose:

To format fixed numbers to strings of a specific length.

Parameters Passed:

IVAL: A fixed number.

N: The minimum length of the resulting string.

Local Variables:

STRING: A temporary character string.

Value Returned:

A character string of the number padded with blanks on the left, if necessary.

Procedure

Purpose:

To generate implicit subscripting for arrays and structures with copies which do not have explicit subscripts. There is something to be done only if the context has arrayness (DOCOPY > 0) and the variable has arrayness (COPY > 0).

If this is not static initialization then try to optimize by searching register tables for a register already containing the index required. If optimization cannot be done because there are too many dimensions or the size of the variable is too hard to handle, then generate code. For static initialization, the addressing computation is done right here.

Function

Purpose:

To generate a temporary copy of an array or multiple copy structure. The size of the required AREA is computed and allocated using GETFREESPACE. Then the array is copied to the temporary.

Parameters:

OP: indirect stack entry for array

LTYPE: type of entry in array

CONTEXT: if > 0 then take LTYPE from OP

Returns:

Stack entry for copy.

GEN_STORE

Procedure

Purpose:

To generate code to store a value. This includes a FORCE_ACCUMULATOR to load the value if necessary, GUARANTEE_ADDRESSABLE followed by EMIT_BY_MODE (store), updating the register stack if this is a KNOWN_SYM.

Parameters:

ROP: stack entry for value to store

OP: stack entry for place to store into

FLAG: if false, decrement usage of REG(OP)

BY_NAME: if true, value is a pointer which should not be dereferenced

Local Variables:

R: the register containing the value

Procedure

Reference:

HALMAT is defined in Appendix A of the HAL/S-360 Compiler System Specification.

Purpose:

Translate HALMAT to intermediate code.

GENERATE makes a pass over the current HALMAT block. It processes one source statement at a time, calling OPTIMISE to set up the next statement. Notice that since OPTIMISE prescans all the HALMAT for an entire source statement, GENERATE has advance warning about interesting subjects. The procedure is a do while "there's some HALMAT left in the block", which is immediately broken into several disjoint subparts by a do case on the CLASS of the HALMAT operator. After processing each HALMAT instruction, DROPFREESPACE returns no longer needed runtime temporaries and NEXTCODE advances to the next HALMAT instruction.

CLASS=0

NOP (A-6)

Do nothing.

EXTN (A-87)

Do nothing now.

XREC (A-6)

Return to main program indicating end of HALMAT if appropriate.

IMRK & SMRK (A-6, 7)

Clean up after statement and prepare next statement.

IFHD (A-49)

Mark beginning of IF statement.

LBL (A-49)

Define the label unless it is an unused exit label of an IF statement.

BRA (A-50)

If branch not redundant, emit unconditional branch.

FBRA (A-50)

Emit code or fill in addresses in existing instructions to perform branch on false.

DCAS (A-51)

Initialize for DO CASE and generate standard code to perform case selection.

ECAS (A-52)

Set up table of indirect jumps to actually get to individual cases, define a label for the location after all cases.

CLBL (A-51)

Generate jump to the location after the DO CASE statement; if this is not the last case, define the flow number of this one and link it into the list built in LABEL_ARRAY.

DTST (A-52)

If this is DO UNTIL, generate jump around test; define beginning of loop.

ETST (A-53)

Generate jump back to the beginning of the loop; define a label for the location after the loop; free temporary storage.

DFOR (A-54,55)

Set DOTYPE equal to tag field (n.b. the description of the tag field is given in the compiler spec. is wrong. The flag for WHILE/UNTIL is actually only for UNTIL). DOFOROP becomes index variable; if this is DO FOR UNTIL, get a location for a boolean and generate code to initialize it to zero; allocate space for temporaries if this is DO FOR TEMPORARY;

Iterative Case

Set up final value; set up increment; generate code to convert initial value to type of index variable; set up addressing information for location of do index and register for do initial value, set up indirect stack entry for do index; if this is DO UNTIL generate code to test boolean flag for first-time through.

Discrete Case

This is simple because code for inserting new values is generated by AFOR instructions; get space for value and do check points if necessary.

EFOR (A-57)

Define label which is end of loop; in normal-situation DOTYPE ≠ "FF".

discrete case

Generate subroutine return style code.

iterative case

Generate code to increment do index, compare with final value, and branch back.

Define label for code after loop; drop temporary storage and descriptions for loop parameters.

In abnormal situation, DOTYPE = "FF";

define label for location after loop
lower do level
issue error message

CFOR (A-56)

At this point loop header code and code to evaluate condition have been issued. Emit code to perform WHILE/UNTIL test. Define label of beginning of actual code to allow skipping around UNTIL code on first iteration.

DSMP (A-57)

Bump DO LEVEL.

ESMP (A-57)

If anybody needed the address of the end of the loop, define it; free temporaries used in loop.

AFOR (A-56)

Increment flow number counter DOFORCLBL (n.b. these flow numbers are never used); generate code to put next value of index in proper place; If this is not last value (i.e. TAG=0) then:

Generate subroutine call style code
Define DOFORCLBL flow label as current location

If this is the last value (i.e. TAG=1):

Generate code to load the address of the instruction after the loop into LINKREG so that loop will exit instead of looping.

The code will fall into the loop so no branch is necessary.

Define label for beginning of loop code. Notice that WHILE/UNTIL code is part of loop.

Generate code to save linkage register so that code can get next index value.

Set up descriptor of register containing DO index.

Generate code to store DO index.

Generate code to skip UNTIL.

Check on first index value if appropriate.

CTST (A-53)

Generate code to perform WHILE/UNTIL test and label for skipping UNTIL test.

ADLP (A-85)

Initialize tables for constructing do loop for arrayed expression and generate initial code in complicated cases.

DLPE (A-86)

Generate end of loop code and clean up.

DSUB (A-89, 95)

Generate all necessary code to evaluate subscript expression and put value of expression in an index register.

NAME_SUB = 1 if in name pseudo

LITTYPE = real tag

TAG = real tag

TMP = 1 if in name pseudo and assignment context

ALCOP = stack entry for item to be subscripted

TERMFLAG = boolean 1 - matrix subscript
0 - no

SUB# = # of the current subscript. Notice that many kinds of subscripts have more than one operand associated with them (see SUBOP).

LEFTOP = Stack entry for accumulated subscript.

SUBOP = The number of the current operand. Notice that sometimes several operands make up one subscript. This value is changed by subroutines called by GENERATE.

RIGHTOP = stack entry for subscript.

EXTOP = stack entry for second part of subscript in TO and AT subscripts.

Notice that the code at DO_DSUB is referenced also from many class 1 operations.

IDLP (A-86)

Set up array do loop parameters to describe the arrayness as copied from the IDLP operands.

TSUB (A-88, 94)

Similar to a very stripped down DSUB. There is only one level of subscripting and that applies across the entire structure.

PCAL (A-61)

Check that we are not in a nested function call (n.b. TAG will be 0); make stack entry for procedure name;

If normal HAL procedure

Generate set-up code using PROC_FUNC_SETUP

Generate subroutine branch code using PROC_FUNC_CALL.

Otherwise,

DO equivalent for non-HAL

This will actuall generate a "NOT IMPLEMENTED" error on the FC compiler.

FCAL (A-61)

Check that we are nested to the proper depth in function calls; make stack entry for function name;

If normal HAL function:

Generate set-up code using PROC_FUNC_SETUP

Set up indirect stack entry and needed run-time temporaries using GET_FUNC_RESULT

Generate subroutine branch code using PROC_FUNC_CALL

If result will be returned in a register, set up register stack description of it.

If non-HAL function, similar to PCAL
(not implemented on FC)

Replace the HALMAT FCAL instruction with indirect stack pointer for RESULT.

FILE (A-63)

Generate code to do library call. Notice that the argument is passed according to non-HAL conventions.

XXST (A-58)

Save existing status so it can be restored; Set CALL_LEVEL to that indicated in instruction. Note that since this could be a call moved out of a nest, TAG may not be CALL_LEVEL+1.

If this is I/O then temporarily set HALMAT instruction counter to point to the I/O instruction involved and call IOINIT for appropriate initialization for specific type of I/O. READCTR was set in OPTIMISE.

If this is not I/O then set up stack entry for routine; set that this is HAL-type; check that routine is already defined; extract information from symbol table and block definition table and insert it in the call stack.

Return stack entry. Copy array-do-loop entry from enclosing level to this level using PUSH_ARRAYNESS.

XXND (A-59)

Restore CALL_LEVEL and ARG_STACK_PTR to their values in outer level.

XXAR (A-58, 94)

Check that nest level is consistent. Check that argument stack has not overflowed. If normal procedure/function call get argument type from symbol table; otherwise, from instruction.

ARG_NAME indicates if argument is of name type. TMP indicates assign if argument is of name type.

Get indirect stack entry for argument.

If this is I/O then process using SET_IO_LIST.

If not I/O, then update information if assign argument, and update ARG_STACK, ARG_STACK_PTR, ARG_COUNTER, and ARG_POINTER.

Also, if not I/O and this is the call level of a previous ADLP operator, then generate necessary temporaries depending on form.

TDEF, MDEF, FDEF, PDEF, UDEF, CDEF (A-8,9)

Call BLOCK_OPEN to set up block definitions.

CLOS (A-9)

Check that close is at correct level and call BLOCK_CLOSE.

EDCL (A-9)

RESUME LOCCTR to Code Csect and set that declarations are finished.

RTRN (A-10)

For functions, generate code to return result. Generate jump to return.

WAIT (A-77)

Allocate run time space and generate code to perform WAIT SVC.

SGNL (A-77)

Allocate run time space and generate code to perform SIGNAL, SET, or RESET SVC.

CANC or TERM (A-78)

CALL SETUP_CANC_OR_TERM.

PRIO (A-79)

Allocate run time space and generate code to perform an UPDATE PRIORITY SVC.

SCHD (A-79)

Allocate run time space and generate code to perform a SCHEDULE SVC.

ERON (A-76)

Generate code to manipulate runtime error stack for ON ERROR and OFF ERROR statements.

ERSE (A-76)

Generate SVC instruction to perform SEND ERROR.

MSHP (A-73)

Allocate runtime temporary and then generate code to perform shaping using SHAPING_FUNCTIONS.

VSHP (A-73)

Identical to MSHP with ROW=1.

SSHP (A-71)

Essentially the same as MSHP.

ISHP (A-72)

Identical to ISHP with OPTYPE=INTEGER.

SFST (A-59)

Set up call stack for shaping function call.

SFND (A-60)

Pop up call stack.

SFAR

Stack argument for later processing by SHAPING_FUNCTIONS.

BFNC (A-64)

Generate code to call (or perform in-line) built-in function.

LFNC (A-75)

Get runtime temporary and generate code to perform library call for list-type built-in functions.

TNEQ,TEQU

Set up stack entries for the structures to be compared. Set up branch points one way or the other depending on whether this is TNEQ or TEQU. Generate code to compare the entirety of the two structures. Notice that if the structures contain character strings, the filler between current length and max length does not have to match, consequently, structures containing character strings must be compared node by node.

TASN

Generate code to copy the strutcure.

IDEF (A-10)

Generate code to save registers, call BLOCK_OPEN and set aside space to receive inline function result.

ICLS (A-10)

Call BLOCK_CLOSE to finish off inline function.

NNEQ (A-92)

Generate code to compare the two NAME operands and jump accordingly.

NEQV (A-92)

Identical to NNEQ.

NASN (A-91)

Generate code to put into arguments 2, 3, ..., a pointer to argument 1.

PMHD (A-96)

Initialize for %MACRO.

PMAR (A-96)

Put %MACRO argument into ARG_STACK.

PMIN (A-96)

Generate in-line code to perform a %MACRO.

CLASS 1 OPCODES

SUBCODE = 0

TAG≠0 implies event operation

GET_EVENT_OPERANDS
EVENT_OPERATOR
EMIT_EVENT_EXPRESSION when expression is complete.

This HALMAT is generated only for real time statements.

Notice that code is not built to evaluate the expression. Rather, the events and operators are put together into an agrument for an SVC call. The supervisor will actually evaluate the expression.

TAG=0

BASN (A-29)

DO_ASSIGNMENT.

BAND (A-31)

EVALUATE(BAND).

BOR (A-31)

EVALUATE(BOR).

BNOT

If next operation is "convert to relation" then just set some flags; otherwise, EVALUATE(BNOT).

BCAT (A-30)

Emit code to shift and OR operands.

Do not forget the code at the very end of the case statement for class 1. This code processes all the subscripts operands, regardless of opcode.

SUBCODE=1

BTOB (A-36)

Just process subscripts.

BTOQ (A-38)

Just process subscripts.

SUBCODE=2

CTOB (A-36)

Generate code to transform from character to bit string and then process subscripts.

SUBCODE=5

STOB (A-35)

Generate code to force into accumulator as integer and then process subscripts.

STOQ (A-37)

If operand is single precision, just process subscript normally; otherwise, generate appropriate code for all possible component subscripting of bit string.

SUBCODE=6

ITOB (A-35)

Just handle subscripts.

ITOQ (A-37)

Just handle subscripts.

CLASS 2 OPCODES

SUBCODE=0

CASN (A-29)

Create temporary copy if necessary. Call CHAR_CALL for each left hand side.

CCAT (A-30)

Get temporary space for result. Call CHAR_CALL.

SUBCODE=1

BTOC (A-34)

NTOC (operand) and then handle subscript.

SUBCODE=2

CTOC (A-34)

Just handle subscript.

SUBCODE=5

STOC (A-33)

See BTOC.

SUBCODE=6

ITOC (A-33)

See BTOC.

CLASS 3 OPCODES

SUBCODE=0

MASN (A-21)

VECMAT_ASSIGN (each left side, right side).

SUBCODE=1

MTRA (A-23)

The code from MAT_TEMP up to MAT_CALL is entered from several places to check for Vector-Matrix Optimization possibilities. This code then enters MAT_CALL which uses VMCALL to generate the appropriate library call.

CHECK_ASSIGN (which calls CHECK_SRCE) checks the overall conditions specified in the HAL/S-FC Compiler System Specification, Section 3.1.5.5 and sets OK_TO_ASSIGN true if the current operation is a good candidate. The code at MAT_TEMP checks the four alternate conditions specified and branches to:

- STACK_ENTRY_ASN if the optimization is to be performed. Here, the HALMAT location counter is advanced to the assignment operator and RESULT is set to the result operand of the assignment.

- TEMP_ASN if optimization is impossible. RESULT is set to a temporary.

For MTRA instruction

```
    ARG_ASSEMBLE
    goto MAT_TEMP
```

SUBCODE=2

MNEG (A-22)

```
OPCODE≠XXASN, ARG_ASSEMBLE
              goto MAT_TEMP
```

MTOM (A-20)

OPCODE = XXASN

VECMAT_CONVERT (operand) if necessary.

SUBCODE=3

MADD (A-21); MSUB, MMPR (A-22)

```
ARG_ASSEMBLE
goto MAT_TEMP
```

SUBCODE=4

VVPR (A-24)

```
ARG_ASSEMBLE
goto MAT_TEMP
```

SUBCODE=5

MSPR, MSDV (A-24)

```
MIX_ASSEMBLE
goto MAT_TEMP
```

SUBCODE=6

MEXP (A-23)

If exponent <-1, generate inverse and fall into code for positive exponent.

If exponent >1, set OPCODE to exponentiate.

Allocate temporary space for computing inverse or successive multiplications.

goto MAT_TEMP.

CLASS 4 OPERATIONS

SUBCODE=0

VASN (A-25)

See Class 3.

SUBCODE=2

VNEG (A-27), VTOV (A-25)

See Class 3.

SUBCODE=3

VMPR (A-26), MVPR (A-27)

```
ARG_ASSEMBLE
goto MAT_TEMP
```

SUBCODE=4

VADD, VSUB (A-26); VCRS (A-27)

```
ARG_ASSEMBLE
goto MAT_TEMP
```

SUBCODE=5

VSPR, VSDV (A-28)

```
MIX_ASSEMBLE
goto MAT_TEMP
```

CLASS 5 OPERATIONS

SUBCODE=0

SASN (A-13)

DO_ASSIGNMENT

SUBCODE=1

BTOS (A-12)

FORCE_BY_MODE (operand)

SUBCODE=2

CTOS (A-12)

CTON (operand)

SUBCODE=3

SIEX (A-15), SPEX (A-16)

EXPONENTIAL (opcode)

SUBCODE=4

VDOT (A-16)

```
ARG_ASSEMBLE
VMCALL (vdot, ...
```

SUBCODE=5

STOS (A-13); SADD, SSUB, SSDV (A-14); SSPR, SEXP (A-15)

For STOS -- GET_OPERANDS (operand, DSCAL) and FORCE_ACCUMULATOR (operand, DSCALAR) if necessary.

```
For non-exponentials -- EVALUATE (opcode)
            for SEXP -- EXPONENTIAL (opcode)
```

SUBCODE=6

ITOS (A-13)

FORCE_BY_MODE or LITERAL.

CLASS 6 OPERATIONS

IASN (A-18)

DO_ASSIGNMENT

BTOI (A-17)

If operand is not literal FORCE_ACCUMULATOR.

CTOI (A-17)

CTON (operand)

STOI (A-17)

FORCE_BY_MODE or LITERAL.

SUBCODE=6

ITOI (A-17)

FORCE_ACCUMULATOR if different type and not literal.

IADD (A-18)

If operands are not CSE's, try folding constant parts. If not completely successful, call EXPRESSION for what is left over.

ISUB (A-19)

See IADD.

IIPR (A-19)

INTEGER_MULTIPLY (opcode)

IIDV (non-existent)

Generate code for an integer divide if one is added to the language.

INEG (A-20)

EVALUATE (opcode).

IPEX (A-19)

EXPONENTIAL (opcode).

CLASS 7 OPERATIONS

SUBCODE=1

BTRU (A-45)

Generate code to transform bit string to a relation. If string is a literal, just change some pointers. If string is in storage, attempt to use test storage instructions. If all else fails, generate code to load and test.

BEQU, BNEQ (A-45)

See subcode 5.

SUBCODE=2

CEQU, CNEQ (A-46)

```
CHAR_CALL
goto SETAG_CONDITIONAL
```

SUBCODE=3

MEQU, MNEQ (A-47)

```
ARG_ASSEMBLE
VMCALL
goto SETAG_CONDITIONAL
```

The code at SETAG_CONDITIONAL calls SETUP_RELATIONAL if simple case, SETUP_BOOLEAN if not simple and intermediate boolean is required.

VEQU, VNEQ (A-48)

See subcode = 3.

SUBCODE=5

SEQU, SNEQ, SGT (A-41); SNGT, SLT, SNLT (A-42)

Generate code to perform comparison with special case code for the situation where one of the operands is literal 0.

SUBCODE=6

IEQU, INEQ, IGT (A-43), INGT, ILT, INLT (A-44)

Attempt folding constants and then go to subcode 5.

SUBCODE=7

CNOT (A-40)

Invert the labels for the true and false conditions on the conditional operand.

CAND (A-39)

Make failure labels identical. Make success of first test fall through to second test.

COR (A-40)

Make failure label of first test fall through to second test. Make success labels identical.

CLASS 8 OPERATIONS

Notice that class 8 is a loop which counts down INITAGAIN (the repeat factor when an OFF qualifier appears) and does not move the HALMAT. The loop structure is the reason that INITLITMOD and INITINCR are necessary. The simpler approach of having a separate HALMAT for every initialization was rejected because it would generate arbitrarily large HALMAT sequences for one HAL source statement thereby violating the requirement of each HAL statement being completely enclosed in one HALMAT block.

SUBCODE=0

STRI (A-81)

Set up addressing information using SET_INIT_SYM. Initialize for generating initialization. If item is STRUCTURE, set up pointers, etc. for STRUCTURE_WALK, STRUCUTRE_ADVANCE, ...

SLRI (A-82)

Initialize repetition count and length of initial value list.

ELRI (A-82)

If not all repetitions are done, update counters and reposition HALMAT file at beginning of repeated initial value list.

ETRI (A-81)

Clean up after handling initialization.

SUBCODE=1

BINT (A-83)

See IINT - subcode 3.

SUBCODE=2

CINT (A-83)

If variable is automatic set up addressing information and generate code to store value; otherwise, insert value in proper place.

SUBCODE=3 or 4

VINT, MINT (A-83)

If variable is automatic, generate VMCALL to assiqn the value: otherwise. insert the value in the entire matrix using a do looo.

SUBCODE=1 or 5 or 6

BINT, SINT, IINT (A-83)

Very similar to CINT.

SUBCODE=7

NINT (A-92)

Same story.

TINT (A-83)

Set up addressing using STRUCTURE_WALK (see STRI), check for type compatibility, then simulate simple initialization by setting OPCODE, SUBCODE, and going to beginning of initialization again.

EINT

Set up address of operand to be used as external entry point.

Procedure

Purpose:

To emit all necessary constants into the database. First the values of the virtual base registers are emitted, then the lists of constants are traversed, the constant is emitted and the CONSTANT_PTR is overwritten with the address of the constant.

Function

Purpose:

To set up indirect stack for ASZ style HALMAT subscript operand. This includes reading next operand when necessary and generating arithmetic code to load array size into a register to evaluate subscript expression at runtime.

Returns:

Indirect stack pointer.

Parameters:

MARK: tag field of ASZ operand.

Local variables:

PTR: pointer to be returned

OP: pointer to indirect stack entry for optional expression operand

Function

Purpose:

Built an indirect stack entry for a function result.

Parameters Passed:

OP: an indirect stack entry for the function.

Returns:

The stack entry built.

Communicates via:

Symbol table and indirect stack.

Description:

Build a stack entry with the information about the function result obtained from entry for function. If necessary, allocate a runtime temporary for the result. If the function has a register, give it to the result.

Function

Purpose:

To create an Indirect Stack entry for an integer literal.

Parameters Passed:

VALUE: The value of the literal.

Local Variables:

PTR: Pointer to the Indirect Stack entry for the literal.

Value Returned:

PTR: Pointer to the Indirect Stack entry for the literal.

Description:

The procedure calls GET_STACK_ENTRY to get an Indirect Stack entry and then sets up the relevant fields associated with the entry as follows:

FORM:	LIT
TYPE:	INTEGER or DINTEGER
VAL:	The value passed to the procedure
LOC:	-1 to show the literal is not in the Literal Table

It returns the pointer to the entry it set up.

GET_LITERAL

Function

Purpose:

To locate the actual literal in LITERAL file (cf. 3.1.1). It returns the offset into the LIT array of the literal. Notice that this may require reading in the correct page of the table.

Parameters:

PTR: absolute (unpaged) pointer into the literal table

FLAG: if true, then when changing pages, write out current page and if PTR points to a page not yet generated, increment counter rather than reading in page.

Comments:

Object_generator routine(s) call this routine GET_RLD, and re-uses the literal file for accumulating RLD information.

Function

Purpose:

To set up an Indirect Stack entry for a HALMAT operator word.

Parameters Passed:

OP: The operand word number.

FLAG: 3 for a SIZE shaping function argument,
1 for a variable that is to be subscripted.

BY_NAME: The operand is part of a NAME pseudo-function.

N: The entry in the TAG2 and TAG3 arrays that should be used.

Local Variables:

SAVCTR: A temporary variable used to save the current value of CTR.

PTR: A pointer to the operand's Indirect Stack entry.

Value Returned:

A Pointer to the operand's Indirect Stack entry.

Description:

DECODEPIP is called to decode the HALMAT operand word. An Indirect Stack entry is set up according to the operand's Qualifier; TAG1.

1) Symbol Table Variable (TAG1=1)

GET_STACK_ENTRY is called to get an Indirect Stack entry. The entry's FORM is SYM to show it is a Symbol Table entry, and LOC and LOC2 point to the Symbol Table entry. UPDATE_CHECK is called to update the CSECT's lock group references. The stack entry's TYPE is determined from the Symbol Table entry. SIZEFIX is called to set the stack entry's size parameters. DIMFIX and SYT_COPIES are called to set up arrayness information. If the operand is not being subscripted, FREE_ARRAYNESS is called to set up indexing of the variable if it is an unsubscripted array reference.

2) Virtual Accumulator (TAG1=3)

A virtual accumulator is a pointer to the result of a previous HALMAT instruction. The OPR entry for the previous instruction was set to the stack entry containing the result of the entry. The VAC's OP1 field is a pointer to the OPR entry, and OPR(OP1) is the pointer to the stack entry. VAC_COPIES is called to set up arrayness information about the VAC in the SUBLIMIT stack. VAC_COPIES calls FREE_ARRAYNESS to set up indexing for the VAC if it is an unsubscripted array reference. VAC_COPIES parallels the function of SYT_COPIES.

3) Pointer (TAG1=4)

The EXTN opcode and subsequent operands are traversed to establish an indirect stack entry describing a reference to a structure node or structure terminal. STRUCTFIX is called to set up the major structure, and STRUCTURE_DECODE is called for each EXTN node to establish addressing and perform any implicit NAME de-referencing. Then control is passed to the symbol table variable process to complete the task.

4) Literal (TAG1=5)

For a literal, an Indirect Stack entry of the form LITERAL is set up whose LOC field points to the literal's Literal Table entry. The procedure LITERAL is called to put information about the literal in the appropriate fields of the Indirect Stack entry.

5) Immediate (TAG1=6)

For an immediate value, the OP1 field of the operand word is the value. GET_INTEGER_LITERAL is called to set up a stack entry for it.

6) Offset (TAG1=10)

A stack entry with FORM of OFFSET and whose VAL field is the offset is set up.

GET_OPERAND does not set up stack entries for other qualifier values.

GET_R

Function

Purpose:

To get an addressing register.

Parameters Passed:

None.

Local Variables:

R: The register chosen.

TR: Never references.

Value Returned:

R: The chosen register.

Description:

If TARGET_R is greater than or equal to zero, then it is the register chosen, Otherwise, register 2 is chosen. The register that has been chosen is checkpointed by calling CHECKPOINT_REG. Then the appropriate Register Table fields are assigned.

Function

Errors Detected:

Indirect Stack Overflow.

Purpose:

Gets a free Indirect Stack Entry.

Parameters Passed:

None.

Value Returned:

Pointer to the Indirect Stack Frame.

Local Variables:

PTR: A pointer to the first free Indirect Stack entry.

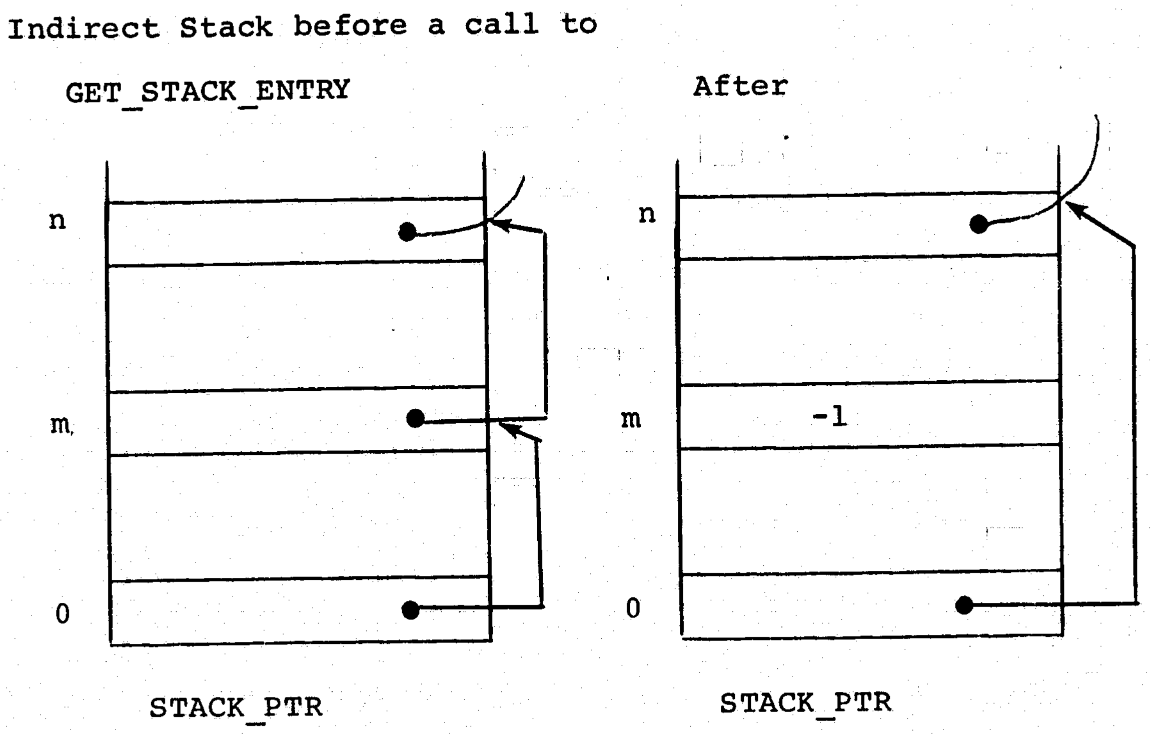
Description:

PTR takes on the value of STACK_PTR, the pointer to the first free stack entry. STACK_PTR takes on the value of STACK_PTR(PTR). The stack is checked for overflow: STACK_PTR(PTR)=0. If there is none, STACK_PTR(PTR) is set to -1 to show the entry has been allocated. All the fields associated with the stack entry are initialized:

REG, BACKUP_REG, STACK_PTR = -1.

INX_MUL = 1

INX, BASE, INX_SHIFT, COLUMN, DEL, CONST, INX_CON, STRUCT_CON, COPY, STRUCT, STRUCT_INX = 0.



Indirect Stack before a call to GET_STACK_ENTRY

After

If compiler diagnostics have been requested, then a message is pointed naming the allocated stack entry.

References:

SETUP_STACK, RETURN_STACK_ENTRY together with GET_STACK_ENTRY provide a complete picture of allocation and deallocation of Indirect Stack Frames.

GET_VAC

Function

Purpose:

To set up an Indirect Stack entry for a register temporary.

Parameters Passed:

R: The register number, or a negative value if no particular register specified.

TYP: The type of the register contents. The default of 0 is taken to indicate type INTEGER.

Local Variables:

PTR: A pointer to an Indirect Stack entry.

Value Returned:

PTR: A pointer to an Indirect Stack entry.

Description:

If R is negative, FINDAC is called to find an index register to use as a temporary. GET_STACK_ENTRY is called to get a new Indirect Stack Entry to represent the temporary. PTR points to it. The relevant fields are set: FORM to VAC, REG to R, and TYPE to TYP. The Register Table field R_TYPE for R is set to TYP. The pointer to the entry is returned.

GETARRAYDIM

Function

Purpose:

To pick up an array dimension from the Symbol Table.

Parameters Passed:

IX: The array dimension.

OP1: Pointer to the array's Symbol Table entry.

Local Variables:

None.

Value Returned:

The IXth array dimension of OP1.

References:

See SYT_ARRAY field of the Symbol Table.

Description:

This function returns the number of copies of a structure determined by SYT_ARRAY(OP1) or the IXth dimension of an array determined by EXT_ARRAY(SYT_ARRAY(OP)+IX).

GETARRAY#

Function

Purpose:

To determine the number of array dimensions of a Symbol Table entry.

Parameters Passed:

OP: Pointer to a Symbol Table entry.

Local Variables:

None.

Value Returned:

Arrayness information.

References:

The SYT_ARRAY field of a Symbol Table Entry.

Description:

GETARRAY# returns 0 if the Symbol Table entry is unarrayed, or if it has * size arrayness indicated by SYT_ARRAY(OP) < 0. Otherwise, it returns the number of array dimensions. This information is found in EXT_ARRAY(SYT_ARRAY(OP)).

Function

Errors Detected:

BS112: Storage Descriptor Stack overflow.

BS113: Exceeded temporary storage.

Purpose:

To find temporary storage in the runtime stack frame of the block for which code generation is occuring, and to set up the Storage Descriptor and Indirect Stack entries to represent it.

Parameters Passed:

OPTYPE: The operand type to be stored.

TEMPSPACE: The amount of temporary storage needed in terms of the product of any dimensions of arrayness and the halfwords occupied by a structure, the length of a character string, or the number of data items in the other data types.

Local Variables:

TYPESIZE: The number of halfwords occupied by one data item.

SIZE: The number of halfwords of storage necessary.

TEMP: A temporary value used while searching for sufficient storage.

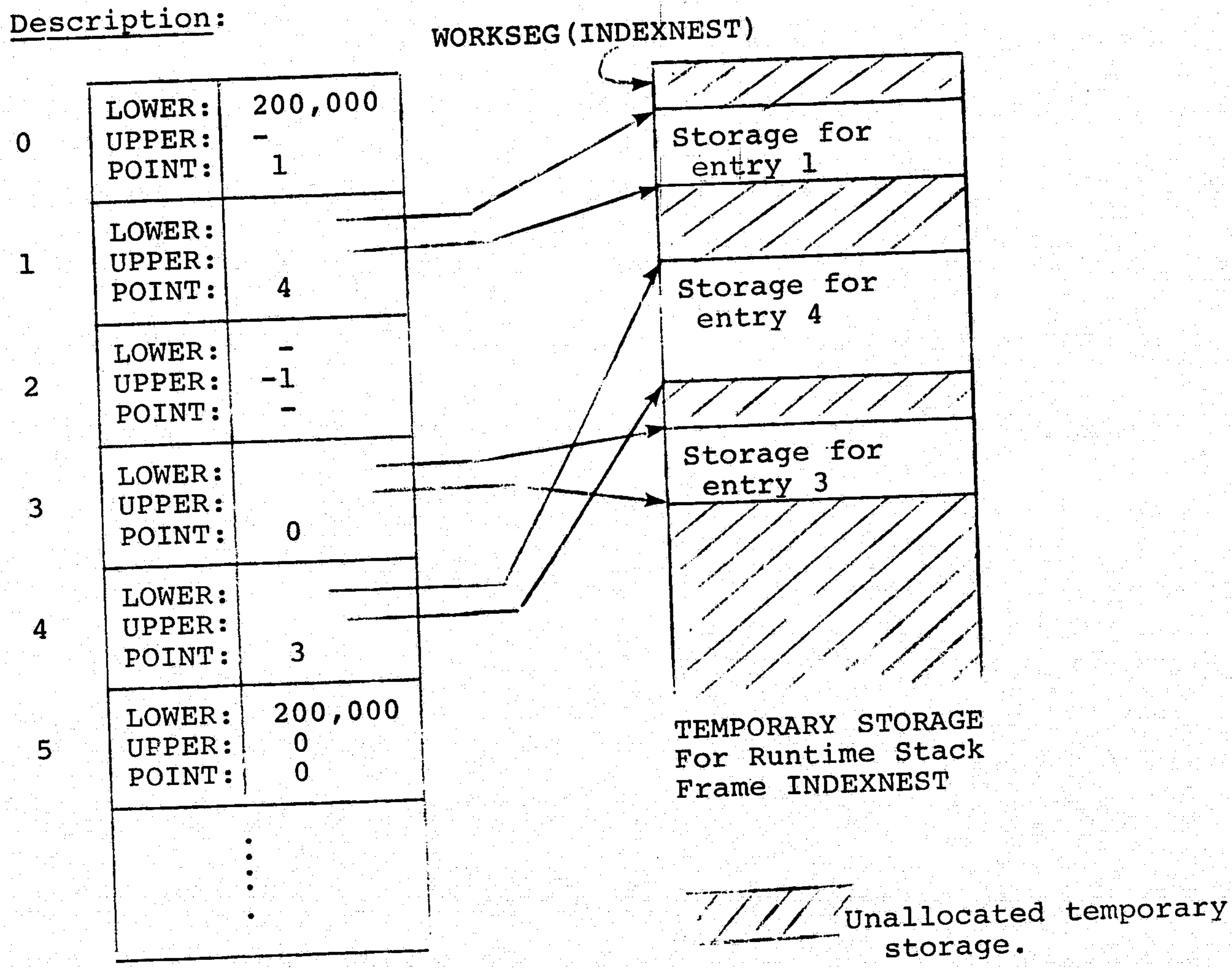
Value Returned:

A pointer to the Indirect Stack entry representing the storage.

Communicates via:

Creates a new Storage Descriptor Stack and a new Indirect Stack entry.

Description:



	Field	Value
0	LOWER: UPPER: POINT:	200,000 - 1
1	LOWER: UPPER: POINT:	 4
2	LOWER: UPPER: POINT:	- -1 -
3	LOWER: UPPER: POINT:	 0
4	LOWER: UPPER: POINT:	 3
5	LOWER: UPPER: POINT:	200,000 0 0
	⋮	

TEMPORARY STORAGE
For Runtime Stack
Frame INDEXNEST

STORAGE DESCRIPTOR
STACK

Above is a diagram of a possible configuration of the Storage Descriptor Stack and Temporary Storage at some time during code generation. Only the fields of the Storage Descriptor entries relating to storage allocation have been shown; ARRAYPOINT, WORK_OR, WORK_USAGE have been omitted.

GETFREESPACE searches the Storage Descriptor Stack for the first entry whose UPPER field is not greater than zero since this indicates the entry is not being used. If UPPER is zero, then the entry has never been allocated previously, and FULLTEMP, the maximum Storage Descriptor Stack size, is incremented. An UPPER of -1 indicates that the entry was previously allocated but is no longer needed.

The procedure computes SIZE, the number of halfwords of storage necessary. The allocated temporary storage is searched to see if there is room for the new entry between two existing entries. The space between entries is due to alignment requirements and storage entries that have been released. Searching for space involves using the linked list formed by the POINT fields of the entries. POINT(0) points to the first allocated storage in the work area of the runtime stack frame. POINT of each subsequent Storage Descriptor Stack entry points to the entry occupying the next allocated storage. The last member of the list points to 0. LOWER of each entry points to the beginning of the area in Temporary Storage occupied by the entry, UPPER points to the end.

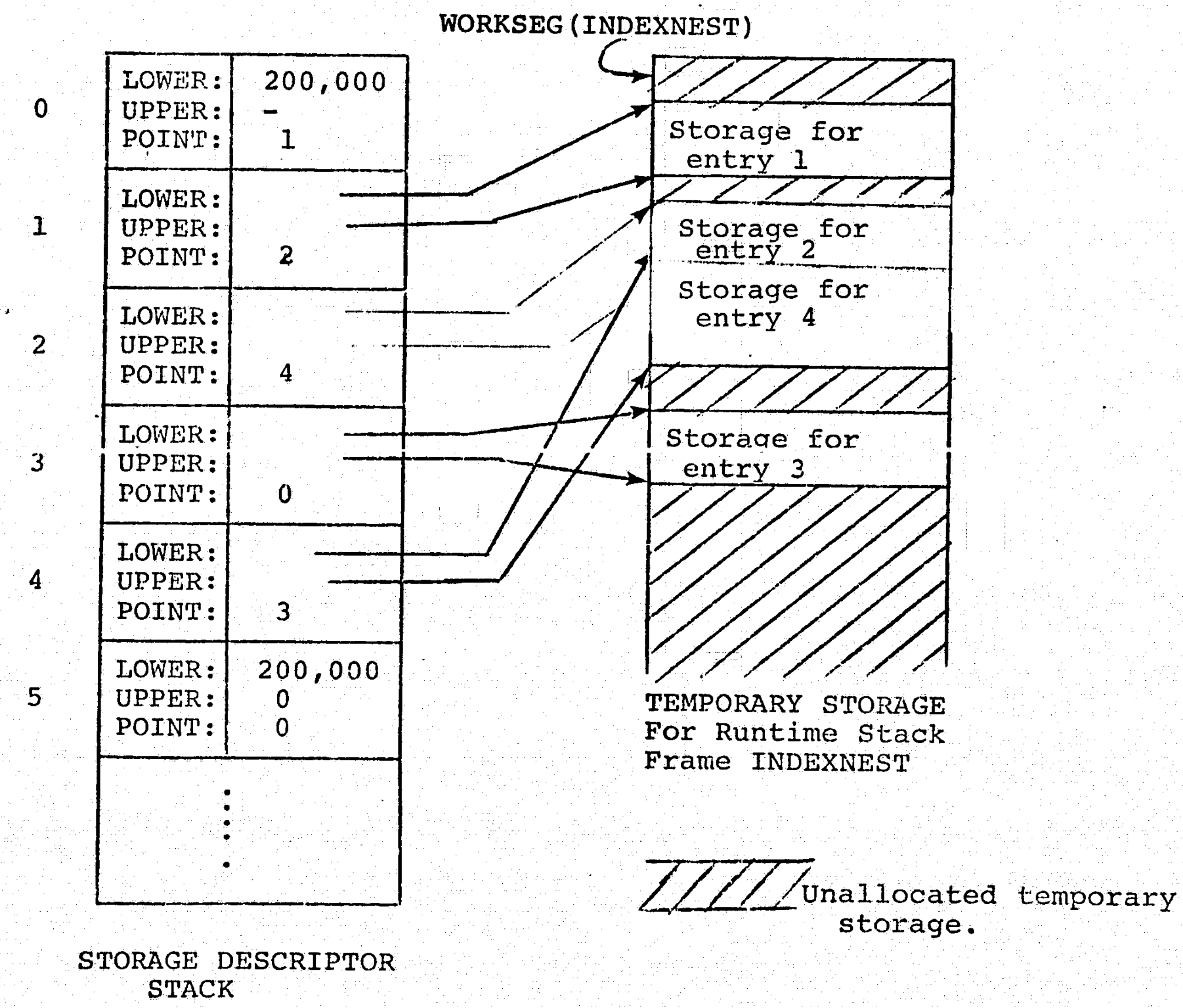
To begin the search, TEMP, a temporary variable, is set to WORKSEG(INDEXNEST) and then normalized to meet alignment requirements. IX2 is used for chaining through the linked list and is initially 0. IX1 is the entry in the Storage Descriptor Stack to be allocated. Now a loop begins to see if TEMP + SIZE < LOWER(POINT(IX2)). If it is the loop is exited. Otherwise, IX2 is set to POINT(IX2), and TEMP is set to a normalized version of UPPER(IX2) and the loop is repeated.

When space has been found, the new Storage Descriptor Stack entry is allocated, and the POINT fields are changed to insert the new entry at the appropriate point in the linked list. If UPPER(IX1) is greater than MAXTEMP(INDEXNEST), the maximum storage needed by the Runtime Stack Frame, this number is changed. WORK_CTR(IX1) is set to the current HALMAT line and WORK_USAGE(IX1) is set to 1 to indicate one user of the Storage Descriptor Stack entry.

A new Indirect Stack entry is set up to represent the Storage Descriptor Stack entry. Its form is WORK to indicate this. The LOC field is set to the Storage Descriptor Stack entry. The BASE of the entry is TEMPBASE since anything in the Runtime Stack is addressed from this register. The DISP field is LOWER(IX1) except for vectors and matrices. For them, DISP is LOWER(IX1) - TYPESIZE because of the addressing conventions used.

GETFREESPACE returns a pointer to the Indirect Stack entry as set up.

Possible configuration of Storage Descriptor Stack and Temporary Storage as shown in the previous diagram after a call to GETFREESPACE:



Entry 2 was the first entry with UPPER ¬> 0, so it was allocated.

GETINTLBL

Function

Purpose:

Create stack entry and statement number for flow number.

Parameters Passed:

LABEL#: a flow number.

Communicates via:

LABEL_ARRAY and indirect stack.

Returns:

Pointer to generated stack entry.

GETSTATNO

Function

Errors Detected:

BS 114: Statement labels all in use.

Purpose:

To get a free statement number to use as a label.

Parameters Passed:

None.

Local Variables:

None.

Value Returned:

A statement number.

Description:

STATNO, the number of statement numbers generated, is incremented and if the result does not exceed STATNOLIMIT, it is returned. Otherwise, ERRORS is called.

Function

Purpose:

To set up an Indirect Stack entry for a generated statement label.

Parameters Passed:

STATNO: A statement number-label.

Local Variables:

PTR: A pointer to an Indirect Stack entry for the statement label.

Value Returned:

PTR: A pointer to an Indirect Stack entry for the statement label.

Description:

PTR is set to the result of calling GET_STACK_ENTRY. The form of the entry is set to STATNO, to show the entry represents a statement number. LOC and VAL of the entry are set to STATNO. PTR is returned.

Procedure

Purpose:

To set up addressing for a symbolic variable. This includes stack walks, dereferencing, external referencing, base register loads, etc.

Parameters:

OP: indirect stack entry for variable to be referenced.

INST: instruction to do the referencing

BY_NAME: if false, dereference pointer variable

NEED_SRS:

Local variables:

R: register to use

PLOC: symbol table pointer for item to be addressed

Function

Purpose:

To convert an integer to external HEX notation.

Parameters Passed:

HVAL: The value to be converted.

N: The length of the hex string to be returned.

Local Variables:

K:. Temporary variables.

B: Temporary variables.

Value Returned:

The external Hex representation of the number.

Function

Purpose:

To generate a readable current location counter.

Parameters Passed:

None.

Local Variables:

None.

Value Returned:

A formatted external hex representation of LOCCTR(INDEXNEST).

INCORPORATE

Procedure

Purpose:

To incorporate integer constants associated with an Indirect Stack entry into the register containing the entry.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

LITOP: A pointer to an Indirect Stack entry for the constants.

OPER: An opcode used for incorporating the constants into the term.

Communicates via:

The Indirect Stack and the Register Table.

Description:

If COLUMN(OP)>0, then OP is an Indirect Stack entry for a bit string that starts at the bit position indicated by the stack entry represented by COLUMN(OP). BIT_SHIFT is called to shift the operand contained in REG(OP) left by the amount represented by COLUMN(OP). Then the register contents are masked according to the length of the bit string, SIZE(OP), by calling BIT_MASK. RETURN_STACK_ENTRY is called to return the entry pointed to by COLUMN(OP). COLUMN(OP) is set to 0 to show that the shift has been incorporated.

If CONST(OP)¬=0, there is a constant term that should be incorporated into the register that will contain OP. GET_INTEGER_LITERAL is called to get an Indirect Stack entry for the constant, and a pointer to it, LITOP. If REG(OP) is negative, then the entry is not contained in a register. FINDAC is called to find a register for OP, and OPER is set to LOAD since the register will be loaded with the term. If OP has a register, OPER will be SUM since the constant will have to be added to the register contents.

ARITH_BY_MODE is called to add or load the constant into the register. R_CON(REG(OP)), the total of all constant terms in the register, is incremented by CONST(OP). CONST(OP) is set to zero since it is incorporated into the register. The Indirect Stack entry for the constant is returned since it is no longer needed.

INITIALISE

Procedure

Purpose:

Initialize phase 2 of compiler, allocate compile time storage, reorganize selected parts of the symbol table, allocate storage for all declared variables.

A collection of flags are set up based on the contents of TOGGLE, PARM_FIELD and OPTION_BITS.

Compile time, storage has already been allocated for the tables inherited from phase 1. Storage is now allocated for the EXTENT array which will be passed to phase 3. After that, storage is allocated for the other six columns of the symbol table which are local to phase 2, for the LABEL_ARRAY, for the LOCATION array, and for the LOCATION_LINK array. This storage will be returned at the end of phase 2. For each non-IGNOREable name in the symbol table perform appropriate initialization actions.

SYT_CLASS=0

This is an impossible value and consequently indicates that all the entries have been processed.

First, ESDs are defined using the appropriate setup routines depending on the type of the program unit. Then the locations (in the stack frame) for the error vector, temporaries, and work areas are laid out for each procedure in the compilation unit.

STORAGE_ASSIGNMENT assigns a location to every variable.

The REGISTERS array is set up to indicate the possible uses of each register; the NOT_MODIFIER, PACKFORM, and SYMFORM arrays are initialized here rather than at their declarations for convenience; the indirect stack is built and finally the procedure returns.

SYT_CLASS=1

The unusual placement of the declaration of procedure VARIABLES here is for historical reasons:

For non-parameters, simple process using VARIABLES.

For parameters, determine type of parameter passed (nb. for arrays, ... parameter is pointer) and size and addressing information on actual parameter.

SYT_CLASS=2 labels

If not NAME or EXTERNAL

statement label -- assign it a statement number

procedure -- PROCENTRY, CHECK_COMPILABLE

task -- PROCENTRY, assign unique task number,
ENTER_ESD, link into list of tasks

program -- PROCENTRY, CHECK_COMPILABLE

compool -- similar to program

external label -- link into list of external labels

if parameter

count argument, PARAMETER_ALLOCATE, SET_PROCESS_SIZE

if EXTERNAL

ENTER, SET_NEST_AND_LOCKS, SET_PROCESS_SIZE

if non-HAL

link into list of non-HALs

otherwise,

PROCENTRY, ENTER_ESD.

SYT_CLASS=3 functions

For regular HAL functions, fill in information about the type of the function in a format similar to a variable of that type after first doing a PROCENTRY and a CHECK_COMPILABLE.

If NAME_FLAG is on, this is a NAME variable which can point to a function (currently illegal).

For non-HALs, link into list of non-HALs and then process like a variable.

SYT_CLASS=7 templates

Guarantee that only the full template is processed by checking for SYT_TYPE=TEMPL_NAME. Perform a complete template walk. For each node or leaf

node -- ENTER, set type to STRUCTURE, remember location in SYT_SORT for ALLOCATE_TEMPLATE

leaf is a structure -- ENTER, set type to STRUCTURE, copy information from template of the leaf.

leaf is name of program or task -- ENTER, SET_PROCESS_SIZE

leaf has a simple type -- VARIABLE, if NAME handle as functions;

When finished with a minor node, ALLOCATE_TEMPLATE.

When finished with whole template, remove it from SYT_SORT, then traverse entire template, relocating sub-trees so that SYT_ADDR of each node becomes the total offset from the beginning of the template. Link template into list of templates.

Procedure

Purpose:

To generate code to perform integer multiply. If both operands are in registers, an attempt is made to perform the multiply without making a copy but this may be impossible if the register pairs are not available. If one operand is a power of two, the multiply is done by shifting. In all other cases, EXPRESSION is called to generate general purpose code. Notice that if EXPRESSION gets an XEXP opcode it performs a non-commutative multiply.

Parameters:

OPCODE: the opcode part of a HALMAT instruction.

Function

Purpose:

To convert a scalar to an integer. The scalar is in DW(0) and DW(1). Since XPL has no scalar data type, the code is written in machine language. The code checks that the scalar is small enough to be represented as an integer.

Parameters:

None.

Returns:

False if the scalar is malformatted or is too large; true otherwise.

If true, DW(3) contains the integer equivalent.

Function Fixed

Arguments Passed:

PTR, a pointer to an indirect stack entry.

Returns:

A fixed point value or NEGMAX.

Procedures Called:

INTEGERIZABLE, INTEGER_VALUED

This routine analyzes an indirect stack to determine if it is a numeric literal. If so, it checks for INTEGER data type, and returns the corresponding VAL if true. Otherwise, it checks if the SCALAR number is both representable as an integer and is a whole number (no fractional digits). If so, the intergerized value is returned. A return of NEGMAX indicates that the stack does not represent an integer valued numeric literal.

LITERAL

Procedure

Purpose:

To set up a stack entry for a literal. This includes any necessary type conversions.

Parameters:

PTR: literal table pointer

LTYPE: type of desired literal

STACK: an indirect stack entry to be filled in.

LOAD_NUM

Procedure

Purpose:

To force a number into a specified register.

Parameters Passed:

R: The register to be loaded.

NUM: The number to be loaded.

FLAG: If bit 1 is non-zero, then R's Register Table entries are unchanged; if bit 3 is one, then the double precision is used; if zero, then single precision.

Local Variables:

LITOP: Pointer to an Indirect Stack entry for NUM.

RT: If the number is in a register, this is the register it is in.

Communicates via:

Register Table.

Description:

GET_INTEGER_LITERAL is called to get an Indirect Stack entry for the number, and a pointer to it, LITOP. The TYPE(LITOP) is modified to indicate the precision specified by FLAG; bit 3 of the type specifies precision (double precision if one; single precision if zero), SEARCH_REGS is called to search the registers for the number. If it is in a register already, it can be loaded into R using an RR instruction. Otherwise, various tests are carried out to determine how to load the number into the register, and the proper code emitting routine is called.

If bit 1 of FLAG is ZERO, the Register Table entry for R is changed as follows:

USAGE(R) = USAGE(R)|1:	the usage is known.
R_CONTENTS(R) = LIT:	the contents are a literal.
R_CON(R) = NUM	The register contents.
R_XCON(R) = 0	The register contents.

The stack entry for the number is returned once it is no longer necessary.

Function

Purpose:

To determine if a STRUCTURE Indirect Stack entry is a major structure.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

None.

Value Returned:

TRUE if OP is a major structure, FALSE otherwise.

References:

The procedures STRUCTFIX and STRUCTURE_DECODE.

Description:

If the operand type is STRUCTURE and LOC2(OP)=SYT_DIMS(LOC(OP)), the struture is a major structure. This is because of the way STRUCTFIX and STRUCTURE_DECODE set up the Indirect Stack entry. LOC(OP) will always point to the Symbol Table entry for the structure reference's major structure. LOC2(OP) points to the Symbol Table entry for a structure node, and the structure template's Symbol Table entry for a major structure.

MAX

Function

Purpose:

To find the maximum of two values.

Parameters Passed:

VAL1, VAL2: Two values.

Local Variables:

None.

Value Returned:

The maximum of VAL1 and VAL2.

MIN

Function

Purpose:

To find the minimum of two values.

Parameters Passed:

VAL1, VAL2: Two values.

Local Variables:

None.

Value Returned:

The minimum of VAL1, VAL2.

MOVEREG

Procedure

Purpose:

To move register attributes from one register to another.

Parameters Passed:

RF: The register the attributes are being moved from.

RT: The register the attributes are being moved to.

RTYPE: Tye operand type of the register contents.

USED: A flag indicating whether the USAGE of RF should be decremented.

Local Variables:

None.

Communicates via:

The Register Table.

Description:

If RTYPE is DSCALAR, RT+1 is loaded with RF+1, and RTYPE is changed to SCALAR. EMITRR is called to load RT from RF. If the contents of RF are known, (its USAGE is odd), the fields in the register table for RT are equated to the corresponding fields of RF. The USAGE of RT is set to 3 to indicate it has one known use. If the contents of RF are unknown, the USAGE of RT is set to 2 to indicate one unknown use. If the USED flag is one, the USAGE of RF is decremented by 2.

Reference:

Opcode construction.

Procedure

Purpose:

To get a new block of HALMAT.

Parameters Passed:

None.

Local Variables:

None.

Communicates via:

The global variables, OPR, CTR, CURCBLK.

Description:

The next block of HALMAT is retrieved from CODEFILE and stored in the OPR array. CURBLK, the current HALMAT block, is incremented. CTR, the pointer into the OPR array, is set to 0. NUMOP is set to the number of operands in the first HALMAT instruction.

NEW_REG

Procedure

Purpose:

To move VAC to a new register.

Parameters Passed:

PTR: A pointer to an Indirect Stack entry.

USED: A flag indicating whether the usage of OP's current register should be decremented.

Local Variables:

RTEMP: The new register.

Communicates via:

Indirect Stack.

Description:

FINDAC is called to find a new index register, RTEMP, for the Indirect Stack entry to use. MOVEREG is called to move the attributes and contents from the stack entry's old register to the new one. The entry's REG field is changed to RTEMP.

Procedure

Purpose:

To clear outdated variable usages from the registers.

Parameters Passed:

OP: A pointer to the Indirect Stack entry for the outdated variable.

FLAG: A flag indicating that UNRECOGNIZABLE should be called in spite of differences between the register and stack entry's indexing constants.

BY_NAME: Variable has NAME attribute.

Local Variables:

I: A do loop temporary.

Communicates via:

Calling UNRECOGNIZABLE.

Description:

The procedure checks each register whose usage is known to see if the register's properties and the stack entry's properties match within a certain tolerance. If they do, UNRECOGNIZABLE is called to indicate that the register's contents are no longer known. The BY_NAME flag is used to help determine which properties to match.

NEXTCODE

Procedure

Purpose:

To position to the next HALMAT operator and decode it.

Parameters Passed:

None.

Local Variables:

None.

Communicates via:

The global variables, CTR, PP.

Description:

PP, the number of HALMAT operators decoded is incremented. CTR, the current HALMAT operator pointer, is incremented to point to what should be the next HALMAT operator. The last bit of this word is tested: a value of 1 indicates that it is an operand word; a 0, an operator word. As long as the test indicates the word is an operand, CTR is incremented. When the next operator is found, DECODEPOP is called to decode it.

Procedure

Purpose:

To condense the intermediate code. The entire intermediate code file is read. All labels are checked for consistency (this is a check on compiler consistency, not source program consistency).

In the FC compiler, an attempt is made to use short form addressing in SRS instructions.

References:

The intermediate code is described in the 360 Compiler Spec, Appendix C.

Purpose:

To translate the intermediate code file to an object module acceptable to the FC or 360 linkage editor respectively.

OBJECT_GENERATOR must output cardimages containing alphabetic and non-alphabetic data. Since XPL I/O is all alphabetic some magic must be performed. Specifically, the cardimage is built in an array (not a character string) and a character string descriptor DUMMY_CHAR is built to allow this cardimage to impersonate a character string. Since it is sometimes convenient to move words and other times convenient to move bytes, the

```
DECLARE CARDIMAGE FIXED, COLUMN(79) BIT(8);
```

equates CARDIMAGE(i) with COLUMNs (4i-4, 4i-3, 4i-2, 4i-1).

NEXT_REC reads the next intermediate language instruction and breaks it down into:

TEMP =

LHS	RHS
16	16

GET_INST_R_X breaks down RHS and returns a properly shifted instruction code:

INST	F	R	IA	IX
8	1	3	1	3

RHS

Notice that the INST (in both compilers) is usually a 360 opcode and consequently must be translated by AP-101INST for the FC compiler.

After emitting the SYM and ESD cards, the routine reads the entire intermediate code file.

RR Type:

If INST < "04" instruction is AP-101 load fixed immediate. OR together the instruction code and two registers and emit it.

RX, RS, RI, SS Types:

Build addressing with FORM_BD, put it all together and emit it.

DELTA:

Add it into ADDRESS_MOD.

Labels & Statement Numbers:

Print the right name.

CSECT:

If this is a different ESD, print it.

Set CURRENT_ESD from instruction and if address specified, set CURRENT_ADDRESS too.

RLDs:

Use EMIT_RLD to make table entry. The actual RLD cards will be issued later by EMIT_RLD_CARDS.

SRS Instructions:

Form base displacement with FORM_BD, put it all together and emit it.

56 = Address Check:

This instruction causes generation of SDF information via EMIT_ADDRS. Specifically, the HAL/S statement number (RHS), first location of the statement (ERRSEG(CURRENT_ESD)), and last location of the statement (STACKSPACE(CURRENT_ESD)) are output. ERRSEG and STACKSPACE are maintained by INST_ADDRS. EMIT_ADDRS is called from INITIALISE to initialize itself and from TERMINATE to clean up.

After handling all instructions on the intermediate code file, RLD cards are issued, and an END card is issued. If this is the main program, a compilation of a program called START is simulated. START simply calls the main program.

OFF_INX

Procedure

Purpose:

To decrement the usage of an index register.

Parameters Passed:

R: The register or a negative pointer to an Indirect Stack entry for the register if it has been checkpointed.

Local Variables:

None.

Communicates via:

The Indirect Stack and the Register Table.

Description:

If R is positive, it is the actual register number. The only thing that needs to be done is to decrement USAGE(R) by 2 to show there is one less claim on the register.

If R is negative, then R = -R to get a pointer to the Indirect Stack entry for a checkpointed register. DEL(R), which corresponds to USAGE of a register, is decremented by 2. If DEL(R) is zero, the value of the checkpointed register is no longer needed. DROPSAVE is called to add the Storage Descriptor Stack entry for the register to the list of no longer needed entries. RETURN_STACK_ENTRY is called to return the stack entry.

OPTIMISE

Procedure

Purpose:

This routine originally did some machine independent optimization on the HALMAT before code generation, hence its name. The optimization function is now performed in phase 1.5. Currently, the routine scans the HALMAT for one source statement doing some bookkeeping.

Parameters Passed:

BLOCK_FLAG: {0 - start scan at next HALMAT instruction
1 - start scan at current HALMAT instruction

Communicates via:

Code emission and assorted global variables.

Description:

Find SMRK and emit intermediate code for it; update first and last statement numbers; if any errors from phase 1, call ERRORS; set flags for I/O statement or in-line function definition. Check for DEBUG directive and take appropriate action.

Procedure

Purpose:

Determine storage locations for formal parameters.

Parameters passed:

OP: symbol table pointer of formal parameter

PTYPE: type of parameter passed

LEN: number of items passed

Communicates via:

symbol table, FIXARG, PTRARG.

Description:

If the parameter can be passed by register it is set up for that; otherwise, it is passed in the area after the REGISTER_SAVE_AREA. FIXARG or PTRARG is updated appropriately.

Procedure

Purpose:

To position a HALMAT block if necessary.

Parameters Passed:

BLK: The HALMAT block to be positioned.

Local Variables:

None.

Communicates via:

Calling NEW_HALMAT_BLOCK if necessary.

Description:

If BLK is not CURCBLK, CURCBLK is set to BLK-1. Then, NEW_HALMAT_BLOCK is called to position the block. CURCBLK is always one greater than the block in OPR.

Function

Purpose:

To determine if an Indirect Stack entry is a constant integer power of two.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

TEST: A temporary variable.

Value Returned:

TRUE if entry is a power of two, FALSE otherwise.

Description:

If the form of the entry is not LITERAL, and the operand type is not INTEGER, the entry cannot be a power of two. If the entry is a positive integer literal, it is tested to see if it is a power of two. If it is a power of two, INX_SHIFT(OP) records the power.

Procedure

Purpose:

To generate argument passing code. The arguments have already been accumulated in ARG_STACK. Consistency is checked for number of arguments, INPUT/ASSIGN type, type. For INPUT arguments, copies are generated where necessary. Code is generated to pass the arguments. If there are not enough registers, the parameter is passed in the stack.

Local Variables:

ARGSTART: point in ARG_STACK of first argument

ARGSTOP: point in ARG_STACK of last argument

ASSIGN_PARM: true if current argument is ASSIGN

NAME_PARM: true if current argument is NAME variable

CONFLICT: true if type conflict between formal and actual parameter

PROCENTRY

Procedure

Purpose:

To do the bookkeeping for initializing tables describing a procedure, task, compool, unlabelled update block, program, or external label. Set up block definition table entry, set up stack frame parameters.

Procedure

Purpose:

To copy array-do-loop entry from outer level to inner level.

Parameters passed:

LEVEL = call stack pointer.

Communicates via:

Array-do-loop stack.

Description:

If this is not outermost level and it is a normal procedure/function call, do the copy; otherwise, initialize to 0.

Procedure

Purpose:

Prints out register status if HALMAT_REQUESTED. In production runs, it is a no op.

Procedure

Purpose:

Called when an error is encountered to clean up various stacks, and reset various stack pointers.

Parameters Passed:

None.

Local Variables:

None.

Communicates via:

Globally declared stack pointers, the Indirect Stack and Storage Descriptor Stack.

Description:

This procedure sets various flags and stack pointers to zero. It also reinitializes the Indirect Stack, clears the Register Table, and frees the Storage Descriptor Stack entries.

Procedure

Purpose:

To resume a given location counter at its last value.

Parameters Passed:

NEST: The number of the CSECT whose location counter is to be resumed.

Local Variables:

None.

Communicates via:

The global variable INDEXNEST.

Description:

The value of INDEXNEST, the CSECT for which code is currently being generated, is checked. If its value is NEST, the procedure returns. Otherwise, INDEXNEST is set to NEST, which automatically ensures the proper location counter is resumed since the location counters are an array indexed by CSECT number. EMITC and EMITW are called to omit intermediate code indicating the CSECT change.

Two variables must be reset as a result of the CSECT change. CCREG must be set to 0 to indicate the condition code is no longer valid. STOPPERFLAG is set to false.

Reference:

Appendix C, Section on CSECT Definition in HAL/S-360 Compiler Spec.

Procedure

Errors Detected:

None.

Purpose:

To release an Indirect Stack entry.

Parameters Passed:

PTR: A pointer to the Indirect Stack entry to be released.

Local Variables:

None.

Communicates via:

Changes linked list of free stack entries.

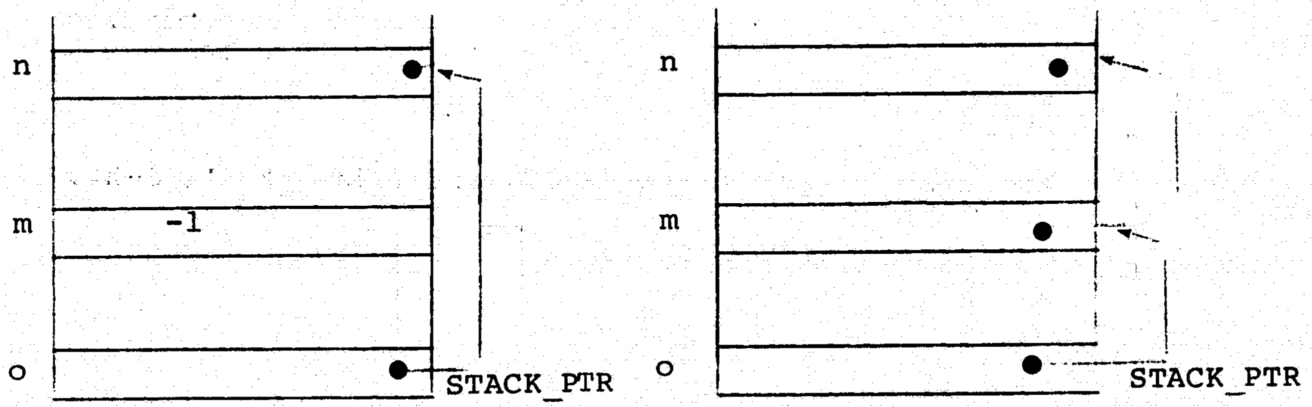
Description:

This procedure adds the stack entry pointed to by PTR to the linked list of free Indirect Stack entries. This is done by setting STACK_PTR(PTR) to STACK_PTR and STACK_PTR to PTR.

Indirect Stack:

before RETURN_STACK_ENTRY(m)

After



References:

SETUP_STACK, RETURN_STACK_ENTRY, GET_STACK_ENTRY together describe the allocation and deallocation of Indirect Stack Entries.

Procedure

Purpose:

Routine to save contents of all floating point registers.

Parameters Passed:

None.

Local Variables:

I: Do Loop temporary.

Communicates via:

Does not affect any variables directly, but it calls CHECKPOINT_REG which does.

Description:

This procedure saves the contents of each of the floating point registers, by calling CHECKPOINT_REG for each register in turn. CHECKPOINT_REG does the actual work involved in saving the register contents.

SAVE_LITERAL

Function

Errors Detected:

BS 109: Constant table overflow.

Purpose:

To add a literal to the Constant Table and the appropriate literal pool.

Parameters Passed:

OP: A pointer to a literal's Indirect Stack entry.

OPTYPE: The literal's type.

Local Variables:

PTR: A pointer to the literal's Constant Table entry.

Value Returned:

PTR: A pointer to the literal's Constant Table entry.

Message Condition:

DIAGNOSTICS

Description:

OPTYPE is set to OPMODE(OPTYPE), the mode associated with the operand type which will be used to determine the literal pool the operand belongs in. FORM(OP) is used to specify the intermediate code qualifier for the literal pool which is determined by adding OPTYPE to CHARLIT. The literal pool qualifiers are consecutive numbers starting at CHARLIT(INITIAL 8), and can be determined in this way.

The procedure then searches the Constant Table to see if OP is in it. Otherwise, it adds it to the table. When the constant has been found, the pointer to its entry is returned and LOC(OP) is set to this pointer.

The Constant Table can be considered to be five linked lists: one for each Literal Pool. The OPMODE of the literal determines the Literal Pool. CONSTANT_HEAD of the OPMODE points to the beginning of a linked list of all iterals in the same pool. Each member of the linked list is a Constant Table entry with the following fields:

CONSTANT_PTR: Pointer to the next Constant Table entry for a literal in the same pool. CONSTANT_HEAD points to the newest entry in the pool. CONSTANT_PTR points to the entry preceding a given entry.

CONSTANTS: The value of the constant. For double precision constants, the entry and subsequent entry together hold the value.

The entries in the Constant Table are allocated consecutively and are not deallocated. CONSTANT_CTR points to the last allocated entry in the Table.

SAVE_REGS

Procedure

Purpose:

To save the contents of specified fixed registers starting with R4, and the contents of all the floating registers or of R2 if desired.

Parameters Passed:

N1: The number of the last fixed register to be saved.

FLT: A flag:

Value	Meaning
0	Only fixed register to be saved
1	Floating registers to be saved
10	R2 to be saved
11	R2 and floating register to be saved.

Local Variables:

I: Do Loop temporary.

Communicates via:

Does not affect any variables directly, but it calls CHECKPOINT_REG and SAVE_FLOATING_REGS which do.

Description:

The routine calls CHECKPOINT_REG to save the contents of the fixed registers from R4 to N1 and of any registers indicated by FLT.

Function

Purpose:

To check if a register contains a specified Indirect Stack entry.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

RC: The register class that could hold OP.

I,J: Temporary variables.

Value Returned:

The number of the register containing the desired information, or -1 if none do.

Description:

To narrow the search, RC, the register class associated with OP, is determined by evaluating RCLASS(TYPE(OP)). Once the register class is determined, RCLASS_START(RC) and RCLASS_START(RC+1) give the range of index in REGISTERS, that contain the register numbers within that class. Every register in the appropriate class is searched until one containing the information is found, or until the registers in the class are exhausted. For each register, the Register Table fields and the Indirect Stack entry's fields are compared in a manner determined by the Stack entry's form. If all the relevant fields match, the register number is returned, otherwise, the search continues.

SET_AREA

Procedure

Purpose:

To establish the area of an Indirect Stack Entry.

Parameters Passed:

PTR: A pointer to an Indirect Stack Entry.

Local Variables:

None.

Communicates via:

The global variable AREASAVE.

Description:

The procedure first checks that the Indirect Stack entry is not a label, and then computes AREASAVE according to the PACKTYPE of the Indirect Stack entry's TYPE.

PACKTYPE		AREASAVE
Value	Description	
0	Vector/Matrix	Number of items in the vector or matrix.
1	Bit	1
2	Character	CSE(SIZE(PTR)+2 unless it is an arrayed character formal parameter where it it SYT_DIMO.
3	Integer/Scalar	1
4	Structure	The size of the template plus the displacement of the template.

Function

Purpose:

To set up an Indirect Stack entry for an unknown array size reference.

Parameters Passed:

OP: A pointer to the Symbol Table entry for the reference.

CON: The extra storage necessary to pass the information.

Local Variables:

PTR: The pointer to the Indirect Stack entry set up for the reference.

Value Returned:

PTR: The pointer to the Indirect Stack entry set up for the reference.

Description:

PTR is set to the result of calling GET_STACK_ENTRY to get an Indirect Stack entry for the reference. The relevant fields of the entry are set: FORM to SYM, LOC to OP, and TYPE to DINTEGER. Since * size arrayness or character strings occur for formal parameters, additional storage is necessary to store this information when parameters are passed. The amount of storage is determined by the parameters CON if it is non-zero. Otherwise, SYT_LEVEL(OP) specifies the amount in fullwords and shifting it by 1, specifies the number of halfwords. This number is stored in INX_CON. The value of PTR is returned.

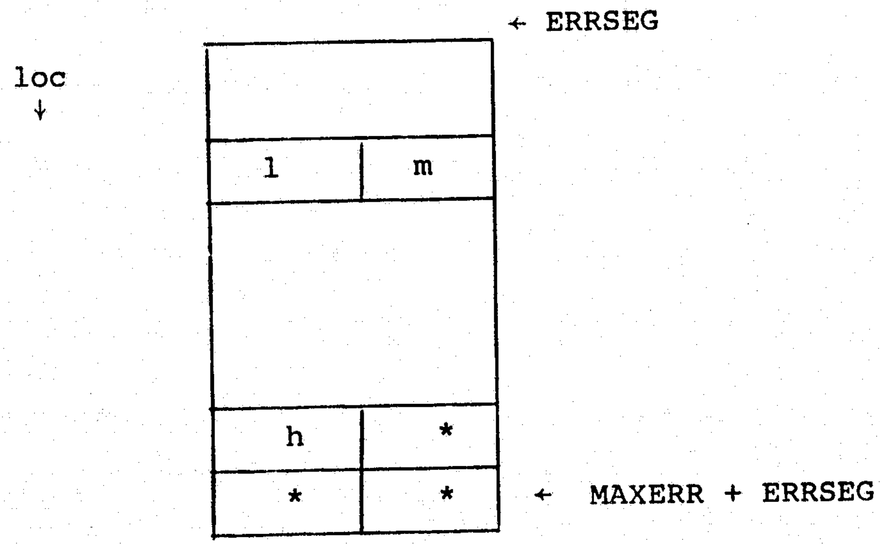
References:

SYT_LEVEL field of Symbol Table, Section 3.1.1.8 of HAL/S-FC Compiler Spec.

Procedure

Purpose:

Assign stack displacement for error number and fill in information in indirect stack entry. The displacements are assigned with the more specific coming first like this:



Parameters:

OP: indirect stack entry for error group number

ERRNUM: integer value of error number

Procedure

Purpose:

To set the location of a specified statement number.

Parameters Passed:

STMTNO: The statement number whose location is to be set.

FLAG1: If 0 indicates that the label may be the destination of a block.

FLAG2: The statement number is for a Phase 2 generated label.

Local Variables:

PAGE: Never referenced.

Communicates via:

LOCATION, LOCATION_LINK, LAST_LABEL.

Message Conditions:

ASSEMBLER_CODE

References:

Appendix C, Section on Label Definition, HAL/S-360 Compiler Spec.

Description:

If FLAG1=0, CLEAR_REGS is called to clear the registers.* CCREG and STOPPERFLAG are reset to 0. The statement number's location, LOCATION(STMTNO), is set to LOCCTR(INDEXNEST), the current location counter. The statement number is added to the linked list of labels within the current CSECT by assigning LASTLABEL(INDEXNEST) to LOCATION_LINK(STMTNO), and by assigning STMTNO to LASTLABEL(INDEXNEST). If the statement number belongs to a phase 2 generated label, the appropriate intermediate code is emitted by calling EMITC.

* This is because the label may be branched to, and by clearing all the registers, the code generation process does not have to worry about different values in the registers depend;ng on the statement branching to the label.

SET_LOCCTR

Procedure

Purpose:

To force the location counter to the desired CSECT and value.

Parameter Passed:

VALUE: The value of the location counter is to be set to.

NEST: The number of the CSECT whose location counter is to be set.

Local Variables:

None.

Communicates via:

The global variables INDEXNEST, LOCCTR(INDEXNEST).

References:

Appendix C, Section on CSECT Definitions, HAL/S-360 Compiler Spec.

Description:

If INDEXNEST, the CSECT for which code is currently being generated, is NEST and LOCCTR(INDEXNEST), its location counter is VALUE, the procedure returns. Otherwise, INDEXNEST is set to NEST, and its location counter is set to VALUE. EMITC and EMITW output intermediate code indicating the changes.

Procedure

Purpose:

To modify an Indirect Stack entry for a label so that its form is EXTSYM, and the entry represents an address constant for the label.

Parameter Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

SY: The Symbol Table entry associated with OP.

IX: The CSECT number used for addressing the label.

Communicates via:

Indirect Stack.

References:

Indirect Stack and Symbol Table.

Description:

If the operand's FORM is neither LBL or SYM, the procedure resturns since only these two forms may need label address constants. If OP's Symbol Table entry has the NAME attribute, its SYT_TYPE is set to SYM, and the procedure returns. Label address constants are not used for variables with the NAME attribute.

The procedure determines how the address constant should be set up. For procedures, variables, and EXTERNAL templates, IX is set to the SYT_SCOPE of SY, the CSECT associated with SY. For programs, tasks, and compools, addressing is carried out using address constants in PCEBASE so IX is set to PCEBASE. INX_CON(OP) will give the offset in PCEBASE where the constant is. The constant is SYT_PARM(SY)*6, where SYT_PARM is a number generated by INITIALIZE uniquely identifying each program, task, or compool.

The form of the stack entry is changed to EXTSYM to show it represents a label address constant. The LOC field of the entry is set to IX, the CSECT used for addressing.

SETUP_BOOLEAN

Procedure

Purpose:

To generate code to jump on success or failure of relational expression.

Parameters Passed:

COND: condition code to branch on

```
FLAG: {0 - if condition fails, jump to VAL(LEFTOP)
       1 - if condition fails, jump to XVAL(LEFTOP)
```

SETUP_PRIORITY

Procedure

Purpose:

Construct SVC argument list for update priority.

Parameters passed:

N: pointer to HALMAT operand specifying priority.

Communciates via:

WORK1, WORK2, emitting code.

SETUP_STACK

Procedure

Purpose:

To set up the Indirect Stack.

Communicates via:

Sets up linked list of Indirect Stack entries.

Description:

This procedure forms a linked list of all the Indirect Stack entries, assuming them all to be free. As a result of the procedure the Indirect Stack looks as shown below:

100	0
99	100
	⋮
5	6
4	5
3	4
2	3
1	2
0	1

SIZEFIX

Procedure

Purpose:

To set up Indirect Stack size parameter for symbols.

Parameters Passed:

PTR: A pointer to an Indirect Stack entry.

OP1: A pointer to the Symbol Table entry associated with it.

Local Variables:

LITOP: A temporary variable.

Communicates via:

The Indirect Stack fields related to the entry's size.

References:

See Symbol Table for a description of SYT_DIMS.

Description:

The procedure sets up the size parameters for a symbol's Indirect Stack entry according to the PACKTYPE of the entry. The information necessary to set up the parameters is in OP1's SYT_DIMs field.

Results of SIZEFIX according to PACKTYPE(TYPE(PTR)):

0: Vector-Matrix: Row: The number of rows in a matrix, or 1 for a vector.

Column: The number of columsn in a matrix or components in a vector.

DEL = 0 to indicate no partition.

SIZEFIX (Con't.)

1: Bit — ROW: The length of the bit string.

COLUMN: Pointer to an Indirect Stack entry representing the position of the first bit in a bit string in a location in core.

2: Character — ROW: The length of the character string.

3: Integer/Scalar

4: Structure — DEL: Pointer to the symbol table entry of the structure's template.

ROW: The size of the template.

(In some cases, ROW is referred to by SIZE which is declared to be "LITERALLY 'ROW'").

Procedure

Parameters Passed:

OP, a pointer to an indirect stack entry.

This procedure is called to record in the R_PARM stack formal parameters which have been set up to be passed via registers, whether for HAL or library calls. BACKUP_REG is set to reflect the corresponding REG entry in the event that the register is subsequently checkpointed before the actual call is issued.

STACK_TARGET

Procedure

Parameters Passed:

OP, a pointer to an indirect stack entry.

This procedure is functionally equivalent to STACK_PARM except that the TARGET_REGISTER specified is reset.

Procedure

Parameters Passed:

R, a register number;

TYP, a corresponding data type (optional).

If TYP is not specified, it is set to the R_TYPE of R. Then a VAC stack entry is created via GET_VAC, specifying register R and data type TYP.

This stack entry is then passed to STACK_PARM. This routine is used when a register parameter is created for which no existing VAC type stack exists, such as character or vector size parameters.

DROP_PARM_STACK

Procedure

This routine is called prior to issuing the actual call to any HAL block or library routine. It passes through the R_PARM stack, reloading any checkpointed values via CHECK_VAC, and then returning the indirect stack entries.

Procedure

Purpose:

To scan ahead in the HALMAT and get line number for next statement.

STORAGE_ASSIGNMENT

Procedure

Purpose:

To determine location for all allocated storage. The symbol table pointers for all variables to be allocated reside in SYT_SORT. This is sorted to allow packing, minimize offsets, and minimize wasted storage for boundary alignments. The values for the base register (SYT_BASE) and displacement from that base (SYT_DISP) are then computed for each variable. Each time a new scope number is encountered, SET_BLOCK_ADDRS allocates space for the proper block header.

STRUCTFIX

Function

Purpose:

To prepare an Indirect Stack entry containing information about a major structure. This entry is set up to do preprocessing associated withthe major structure before modifying the entry to represent a structure node reference. If the major structure has no subscripting, STRUCTFIX is called by GET_OPERAND directly before resolving the node reference. If there is structure subscripting, STRUCTFIX is called by GET_STRUCTOP while the subscript reference is being resolved, and GET_OPERAND does not set up the stack entry again, but obtains a pointer to it, and then resolves node references.

Parameters Passed:

OP: A pointer to a structure's Symbol Table entry.

FLAG: 1 if OP is a SIZE function argument, or a struture that is to be subscripted, 0 otherwise.

Local Variables:

PTR: A pointer to an Indirect Stack entry set up to represent the structure.

Value Returned:

PTR: A pointer to an Indirect Stack entry set up to represent the structure.

References:

Array Reference Stack, the HALMAT EXTN and TSUB operators.

Description:

STRUCTFIX calls GET_STACK_ENTRY to get a pointer, PTR, to an Indirect Stack Entry. The entry's FORM is SYM, and a great deal of STRUCTFIX parallels the case of GET_OPERAND devoted to Symbol Table Entries. STRUCTFIX first sets up the basic information needed by the stack entry:

FORM(PTR) = SYM

TYPE(PTR) = SYT_TYPE(OP)

LOC(PTR) = OP, a pointer to the Symbol Table entry.

LOC2(PTR) = SYT_DIMS(OP), a pointer to the template.

UPDATE_CHECK is called to update any lock group references. SIZEFIX is called to set up the stack entry's size fields.

The second part of STRUCTFIX takes care of preparing for array or subscript processing if the Symbol Table entry has copies. SET_AREA is called XVAL(PTR) and SUBLIMIT(STACK#) are set to AREASAVE. For structures AREASAVE is the size plus the displacement of the template, and its number is used for indexing from one copy of the structure to the next. COPY(PTR) is set to 1, since having copiness is equivalent to one dimension of arrayness. STRUCT(PTR) is set to one to indicate that the major structure has copies since further processing of a structure node will add any arrayness associated with the node to COPY(PTR). (This happens in DIMFIX.) DOPTR and DOTOT of the present call level are reset in case arrayness has been pushed because of a call.

The preparation so far is relevant to array processing and subscripting. If FLAG=1, no more preparation is needed; any indexing necessary for subscripting is taken care of when the subscript reference is resolved. If the structure is an argument of the SIZE function, no indexing is needed. If FLAG=0, STRUCTFIX must check to see if a DO LOOP is necessary to process the structure copies; this is indicated by COCOPY(CALL_LEVEL)>0 which shows there is an array reference. If DOFORM(CALL_LEVEL) is 2, no loop has been set up, so EMIT_ARRAY_DO is called to set up the loop. Ordinarily, if DOFORM is 2, no do loop is necessary since the array reference is for a simple arrayed parameter. These would occur in consecutive storage except for arrayed structure terminals. Since the terminals are not in consecutive locations, EMIT_ARRAY_DO sets up a do loop to do the necessary indexing. If FLAG=0, FREE_ARRAYNESS is called, to emit code for the structure arrayness.

STRUCTFIX returns PTR, the pointer to the Indirect Stack entry.

Procedure

Purpose:

Part of the process of setting up an Indirect Stack entry for a structure node, the procedure is called for each Symbol Table entry that is resolved except the major structure reference and the last reference if the reference is BY_NAME.

Parameters Passed:

PTR: A pointer to an Indirect Stack entry set up for the structure node by STRUCTFIX and modified by calls to STRUCTURE_DECODE.

OP: A HALMAT EXTN operator operand number.

BY_NAME: The operand is part of a NAME pseudo-function.

Local Variables:

R: A register used for setting up Indirect Stack entry for a structure node.

Communicates via:

Indirect Stack.

References:

The HALMAT EXTN operator, the procedure STRUCTFIX.

Description:

DECODEPIP is called to decode the operand word for the next Symbol Table reference, and LOC2(PTR) is set to a pointer to the reference's Symbol Table entry. STRUCT_CON(OP1), the constant associated with structure addressing, is incremented by SYT_ADDR of the Symbol Table entry, the displacement of the node within the structure.

If the BY_NAME flag is false or the node is the last operand and it does not have the name attribute, the way the node's stack entry is addressed must be updated. RESUME_LOCCTR(NARGINDEX) is called if a declaration is in effect so that code will not be emitted in the data CSECT. INX_CON(PTR) is set to STRUCT_CON(PTR) so that SUBSCRIPT_RANGE_CHECK can be called to modify the index register if the adjusted displacement is not addressable.

Register 2 is used for addressing, but if the form of the entry is not CSYM or the register is being used, GET_R must be called to get a register. The register is loaded with the address, and DROP_INX is called to drop the index register. Various fields must be modified:

INX_CON, STRUCT_CON=0 — Since the constants have been incorporated.

FORM-CSYM — Since the entry has its own base and displacement.

DISP=0 — The address is all in the base register.

BASE, BACKUP_REG=R — The register containing the address.

STRUCTURE_WALK

Procedure

Purpose:

To walk a structure template in order to compute the location (INITADDR) of the node. The routine gets to the next terminal node by STRUCTURE_ADVANCE. STRUCTURE_ADVANCE moves down the tree to the terminals using DESCENDENT and to the parent and brothernodes using SUCCESSOR. Once at a terminal node it counts through the items (for vectors, matrices and arrays) in the terminal node ($N \geq 1$) before proceeding to the next terminal node. The process continues until the desired element is found.

Parameters:

WALK#: The number of the item desired. Notice that a terminal node may contain many items.

Other Variables:

INITWALK: The number of items already passed. Initially we are not even at an item so INITWALK starts at -1.

INITDECR: INITWALK

N: Number of items left in terminal node

INITOP: Symbol table pointer for node

INITADDR: Total offset of INITOP

INITTYPE: Type of INITOP

SUBSCRIPT_MULT

Procedure

Purpose:

To multiply an Indirect Stack entry for a subscripting index of a subscript.

Parameters Passed:

OP: A pointer to an Indirect Stack entry for a subscript.

VALUE: If positive, the value the subscript is to be multiplied by; if negative, a negative pointer to the Symbol Table reference for the subscript.

Local Variables:

LITOP: A pointer to the Indirect Stack entry set up for VALUE.

Communicates via:

Calling code emitters.

Description:

INX_MULT, the constant multiplier associated with two dimensional subscript references, is set to one since SUBSCRIPT_MULT will take care of the multiplying if called from SUBSCRIPT2_MULT.

If VALUE is negative, SET_ARRAY_SIZE is called to create a stack entry for the multiplier, and CHECK_ADDR_NEST is called to set up proper addressing. Code is emitted to perform the multiplication, according to whether the AP-101 index register self-alignment feature is in effect.

If VALUE is a literal, GET_INTEGER_LITERAL is called to get a stack entry for the literal. Code is emitted to perform the multiplication according to whether the multiplier is a power of two and whether the compiler SELF_ALIGNING option is in effect.

OP's register is marked unrecognizable since its contents have been modified.

Procedure

Purpose:

To verify if an adjusted displacement is addressable, and to incorporate the adjustment into the index register if it is not.

Parameters Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

INCOP: A pointer to an Indirect Stack entry used for modifying OP's index register.

CON: The indexing constant used for addressing OP.

RANGE: A temporary variable.

REMOTE: A flag indicating whether or not OP has the REMOTE attribute.

Communicates via:

The Indirect Stack.

Description:

If the indexing constant is zero or the Indirect Stack entry does not have the REMOTE attribute or an index register, there is no addressing problem so the procedure returns.

CON, the indexing constant, INX_CON(OP), will be incorporated into OP's displacement for addressing purposes if the resulting displacement is between 0 and 2047. The temporary variable RANGE together with CON are used to test this. If the resulting displacement would be outside of this range or OP has the REMOTE attribute, the indexing constant will have to be incorporated into OP's index register.

If the SELF-ALIGNING compiler option is in effect, that is, the context of the AP-101 index registers will be aligned automatically, the index constant must be modified. It must be divided by the number of halfwords occupied by one item of OP's operand type, BIGHTS(TYPE(OP)). The automatic alignment will multiply the index by that amount during address computation.

GET_STACK_ENTRY is used to get INCOP, a pointer to a free indirect stack entry which will be used for incorporating the constant into the index register's constant. Before doing this, OP's stack entry must be checked to see if it has an index register. If it does not have one, or if it has several users, FINDAC is called to find an index register. In the second case, MOVEREG is called to move the register contents and attributes to the new index. REG(INCOP) is set to the index register, CON(INCOP) to the indexing constant. INCORPORATE is called to add the constant to the register. INX_REG(OP) is set to REG(OP), and INX_CON(OP) is set to 0 since the constant has been incorporated. INCOP's stack entry may be returned.

SUBSCRIPT2_MULT

Procedure

Purpose:

To generate code of form

```
   LEFTOP                       MULT         RIGHTOP
old_index = old-index * dimension + next_subscript
```

The bulk of the routine attempts to find the value already in a register; otherwise, it would be much shorter.

Parameters:

mult = dimension multiplier

Local variables:

I: just a dummy

R: register used for calculation

SYT_COPIES

Procedure

Purpose:

To find the arrayness of Symbol Table variables and record the information in the Array Reference and Array Do Loop Stacks.

Parameters Passed:

OP: A pointer to a Symbol Table entry.

Local Variables:

I,J: Temporary variables.

Communicates via:

Array Reference Stack and SUBLIMIT.

Description:

SYT_COPIES resets the values of DOPTR(CALL_LEVEL) and DOTOT(CALL_LEVEL to their base values which are respectively SDOPTR(CALL_LEVEL) and SDOPTR(CALL_LEVEL)+ DOCOPY(CALL_LEVEL). This is necessary because arrayness is pushed from an outer to an inner level when dealing with invocation references.

If the Symbol Table entry is arrayed, SYT_COPIES also sets up the entries in SUBLIMIT that will contain arrayness information. SUBRANGE is used as a temporary variable in the process. STACK# will be 0 unless OP is a subscript in a subscript reference for a variable with m dimensions of arrayness. In this case, STACK# is m+1. At the end of SYT_COPIES, SUBLIMIT contains the following new information: (Assume OP has n dimensions):

SUBLIMIT(STACK#)	The size of the 1^{st} dimension
⋮	⋮
SUBLIMIT(STACK#+n-1)	The size of the n^{th} dimension
SUBLIMIT(STACK#+n)	AREASAVE

TERMINATE

Procedure

Purpose:

To handle logical control after GENERATE.

GENERATE_CONSTANTS

emit code end intermediate instruction

OBJECT_CONDENSER

Create ESD entries for external labels

Initialize for OBJECT_GENERATOR

OBJECT_GENERATOR

Function Fixed

Parameters Passed:

F, a fixed point value or descriptor.

Values Returned:

F, a fixed point value.

This function is the opposite of the DESC function. The argument passed is a character string descriptor word which is interpreted as a fixed point integer upon return, allowing assignment into a fixed variable. This routine is used during initialization to build an array which can later be referenced using the DESC function, by-passing the 1024 descriptor limitation of XPL.

UNRECOGNIZABLE

Procedure

Purpose:

To mark the contents of a register unknown without decrementing the number of claims on its contents.

Parameters Passed:

R: The register.

Local Variables:

None.

Communicates via:

The global variable USAGE(R).

Description:

The rightmost bit of the register's USAGE is set to 0 to indicate its contents are unknown. This is done because the procedures which search the Register Table for registers with certain properties, only look at the entries for registers whose USAGE is odd. Sometimes, a code emitter will be called to generate code that modifies the register's contents without modifying any of the register's attributes in the Register Table. By marking the register unrecognizable, the register's entry will not be considered when the table is searched.

Procedure

Purpose:

To keep track of all lock groups used within an update block.

Parameters Passed:

OP: A pointer to a symbol table entry.

Local Variables:

None.

Communicates via:

UPDATE_FLAGS LITERALL SYT_CONST(UPDATING).

References:

Description of the Symbol Table, Description of Local Block Data Area, LOCK_ID Field.

Description:

The procedure first checks to see if code for an UPDATE block is being generated. This is indicated by UPDATING > 0; UPDATING is the pointer to the symbol table entry of the UPDATE block. If this is the case, SYT_CONST(UPDATING) is modified to reflect OP's lock group. The purpose of this procedure is to determine the lock groups in the UPDATE block so that BLOCK_CLOSE may set up the block's Local Block Data Area.

Procedure

Purpose:

To verify an Array Index Indirect Stack entry's register is safe.

Parameter Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

RM: Never referenced.

Communications via:

Register Table.

Description:

If OP's register has a claim on it and its contents will be modified, NEW_REG is called to get OP another register. Otherwise, the register's USAGE is incremented by 2 to show it has another claim on it; the register's USAGE_LINE is set to the current line of HALMAT.

VAC_COPIES

Procedure

Purpose:

To set up indexing into shaping function results.

Parameters Passed:

OP: A pointer to an Indirect Stack Entry.

Local Variables:

I: A Do Loop temporary.

Communicates via:

Array Reference stack and SUBLIMIT.

Description:

This procedure parallels the function of SYT_COPIES but instead of working on a stack entry that has just been set up for Symbol Table entry, it uses a stack entry that has previously been set up to represent the results of a shaping function. The first thing the procedure does is to check that the entry has arrayness; if it does not, this procedure is unnecessary.

VAC_COPies starts by resetting DOPTR(CALL_LEVEL) and DOTOT(CALL_LEVEL) to their former values. This is necessary because arrayness is pushed from an outer to an inner level when dealing with invocation references.

Then, the entries in SUBLIMIT that will contain OP's arrayness information are assigned starting at entry STACK#. STACK# will be 0, unless the Indirect Stack entry is a subscript of a variable with m dimensions of arrayness. In this case, STACK# is m+1. VAL(OP) is a pointer to the first entry in SF_RANGE containing information about OP's arrayness.

The results of the assignments are:

	Assigned to	Description
SUBLIMIT(STACK#)	SF_RANGE(VAL(OP))	The size of the 1st dimension
SUBLIMIT(STACK#+1)	SF_RANGE(VAL(OP)+1))	The size of the 2nd dimension
⋮		⋮
SUBLIMIT(STACK#+COPY(OP)-1)	SF_RANGE(VAL(OP)+COPY(OP)-1)	The size of the last dimension
SUBLIMIT(STACK#+COPY(OP)	AREASAVE	This is computed by calling SET_AREA

FREE_ARRAYNESS is called to set up indexing for unsubscripted variables.

Function

Purpose:

To compute space required for a variable and enter it in the symbol table.

Parameters Passed:

OP1 = symbol table pointer.

Value returned:

Size of variable or single element of array.

Procedure

Purpose:

To protect an index register prior to adjusting its contents.

Parameter Passed:

OP: A pointer to an Indirect Stack entry.

Local Variables:

R: An index register.

Communications via:

Register Table and Indirect Stack.

Description:

If OP's index register has only one or no claims on it, it is marked unrecognizable to prevent other users from mistaking the register's contents. If the index register has several users, FINDAC is called to find it another register to use an an index. The new index register is loaded with the contents of the old register, and the USAGE of the old index register is decremented by 2 to show there is one less claim on it.

VMCALL

Procedure

Purpose:

To generate calls to library routines for performing vector-matrix operations. The routine generates code to load all and only those parameters required by the library routine (as determined by array CTRSET) and then calls GENCALL to generate the actual call.

Parameters:

OPCODE: HALMAT style opcode

OPTYPE: true if double precision

OP0: indirect stack entry for result

OP1: indirect stack entry for first operand

OP2: indirect stack entry for second operand

PART: paritition information

6.0 PHASE 1.5 - THE OPTIMIZER

6.1 Introduction

6.1.1 General Description

The HAL/S Optimizer takes HALMAT produced by Phase I and performs the following functions:

- Common subexpressions (CSE's) are recognized.
- Additional constant folding is carried out.
- Unneeded divisions are replaced by multiplications.
- Superfluous matrix transpose operations are eliminated.

Altered HALMAT is then passed to Phase II for object code generation.

6.1.2 Design Comments

The most important design consideration is that the Optimizer does nothing to most HAL/S statements! Thus, the sooner this is recognized, the less time wasted on a statement and the more efficient is the Optimizer. More concretely, the following features are of note:

1. The CSE_TAB doubly linked list drastically reduces the number of Nodes searched for CSE's. This might be compared with FORTRAN H where the previous ten statements are searched for CSE's, even though they may contain no common variable with the present statement.
2. If a Node does not have enough eligible operands for a CSE, no search is made (SEARCHABLE = FALSE).
3. The Optimizer is quite conservative. For example, all user procedure and function calls cause ZAP_TABLES to be invoked.

6.1.3 Optimizations Attempted

This section describes those optimizations presently implemented in the HAL/S OPTIMIZER and corresponding Phase II, and gives appropriate user information.

Optimizations Performed

1. COMMON SUBEXPRESSION ELIMINATIONS

   a. "Cummutative" Operations

      For bits: &, |

      For scalars: +, -, <>, ÷

      For integers: +, -, <>

      For vectors and matrices: +, -

Example 1:

```
        F = A - D + B - C;
        G = D - C - B + A;
```

becomes*:

```
   CSE1 = A - C;
   CSE2 = B - D;
        F = CSE1 + CSE2;
        G = CSE1 - CSE2;
```

Example 2:

```
        F = (A/B) (C/D);
        G = C(B/D) A;
```

becomes:

```
   CSE1 = C/D;
   CSE2 = A/B;
        F = CSE1 CSE2;
        G = CSE1/CSE2;
```

* Often the CSE's are merely retained in registers with no temporaries created.

Example 3:

```
        F = A + B + (C D) + E + (B C A);
        G = D + (D C) + E + A + (A B);
  becomes:
     CSE1 = A + E + (C D);
     CSE2 = (A B);
        F = CSE1 + B + (CSE2 C);
        G = CSE1 + D + CSE2;
```

b. Noncommutative Operations

1. For bits: ||,¬

   Built-in functions: XOR.

2. For scalars and integers: **, negation, conversion to integer or scalar from integer or scalar.

   Built-in functions: ABS, CEILING, FLOOR, ODD, ROUND, SIGN, SIGNUM, TRUNCATE, ARCCOS, ARCCOSH, ARCSIN, ARCSINH, ARCTAN, ARCTANH, COS, COSH, EXP, LOG, SIN, SINH, SQRT, TAN, TANH, DIV, MOD, SHL, SHR, INDEX, LENGTH, MIDVAL, ARCTAN2, REMAINDER.

3. For vectors and matrices*: negation, m v, v m, v*v, v x, x v, v/x, m m, v v, m x, x m, m/x, m**i.

   Built-in functions: ABVAL, DET, INVERSE, TRACE, TRANSPOSE, UNIT.

* i = non-negative integer literal,
x = scalar or integer,
m = matrix, and
v = vector.

Example 4:

```
         X_NEW = X COS(THETA) + Y SIN(THETA);
         Y_NEW = Y COS(THETA) - X SIN(THETA);
  becomes:
          CSE1 = COS(THETA);
          CSE2 = SIN(THETA);
         X_NEW = X CSE1 + Y CSE2;
         Y_NEW = Y CSE1 - X CSE2;
```

Example 5:

```
           R1 = (-B + SORT(B**2 - 4 A C))/2A:
           R2 = (-B - SQRT(B**2 - 4 A C))/2A;
  becomes:
         CSE1 = -B;
         CSE2 = SQRT(B**2 - 4 A C);
         CSE3 = 2 A;
           R1 = (CSE1 + CSE2)/CSE3;
           R2 = (CSE1 - CSE2)/CSE3;
```

2. MATRIX TRANSPOSE ELIMINATIONS

M^T V is changed to V M and V M^T is changed to M V, saving a transpose operation.

Example 6:

$$M = M^T((M1 + M2)^T V);$$

becomes:

$$M = (V (M1 + M2)) M;$$

3. CONSTANT FOLDING

Some constant folding not done by Phase I involving integer and scalar +, -, <>, and ÷ is performed.

Example 7:

$$F = (2A)/(4 B C); \quad \text{(all scalars)}$$

becomes

$$F = (.5A)/(B C);$$

CSE's involving folded constants are found.

4. DIVISION ELIMINATIONS

Terms are rearranged to eliminate unneeded divisions.

Example 8:

$$F = (A/B) (C/D) (E/F);$$

becomes:

$$F = (A C E)/(B D F);$$

6.1.4 Scope of Optimization

Common subexpressions are recognized over approximately basic blocks of code. No CSE's are recognized across:

- labels
- user procedure or function calls
- assignments into name variables
- HALMAT blocks
- inline functions
- GO TO's
- DO CASE's
- DO FOR's
- DO UNTIL's
- END's for above 3
- END's for simple DO END if there is a corresponding EXIT
- beginnings of each case in DO CASE
- Major or Minor Structure Assignments
- READ, READALL, AND FILE I/O instructions
- program organization operators (e.g. PROCEDURE, CLOSE)
- WAIT statements
- ERROR statements
- IF statement conditionals containing more than one boolean comparison
- ends of the true parts in IF THEN's or IF THEN ELSE's
- ends of IF THEN ELSE's.

The presence of any of the following causes the entire statement to be skipped.

- user procedure or function calls
- inline functions
- statements causing array loop generation
- I/O instructions
- shaping functions
- character operations
- bit or character conversion to integer or scalar
- real time statements

Name variables, bit conversions, and SUBBIT's are not presently included in CSE's.

In IF statements, no CSE's may occur in a part of the relational expression which is not always executed, (e.g. the * statement in example 9).

Optimizing stops when a statement containing a Phase 1 error is detected.

Example 9:

```
                B = SIN(A);
                C = SIN(A);
                D = SIN(A) + USER_FUNCT(A);
                E = SIN(A);
                F = SIN(A);
                IF SIN(A) = SIN(A) AND B = SIN(A) THEN DO;
                    G = SIN(A);
                END;
                ELSE H = SIN(A);
                I = SIN(A);
  becomes:
             CSE1 = SIN(A);
                B = CSE1;
                C = CSE1;
                D = SIN(A) + USER_FUNCT(A);
             CSE2 = SIN(A);
                E = CSE2;
                F = CSE2;
      * -----IF CSE2 = CSE2 AND B = SIN(A) THEN DO;
                    G = SIN(A);
                END;
                ELSE H = SIN(A);
                I = SIN(A);
```

6.1.5 Programming Considerations

CSE's and division elimination may alter the order of computation of statements, including parenthesized statements (see Examples 2, 7, 8). If it is necessary to prevent this, the programmer must break up the statements in question into the desired computation using temporaries. Thus, example 8 could be programmed:

```
temp1 = A/B;
temp2 = C/D;
temp3 = E/F;
    F = temp1 temp2 temp3;
```

to insure the computation of the three terms. If the order of multiplication is also important, the last statement could be replaced by:

```
F = temp1 temp2;
F = F temp3;
```

Another trick is insertion of DO; EXIT; END;. This prevents CSE's from being recognized across the insertion.

When a CSE is recognized by the compiler the resulting code is usually better than if the programmer had created a temporary, since the CSE is often retained in a register until use.

Thus:

```
F = A + C D;
G = B - C D;
```

is both more readable and produces better code than:

```
TEMP = C D;
F = A + TEMP;
G = B - TEMP;
```

Compiler Options

By specifying:

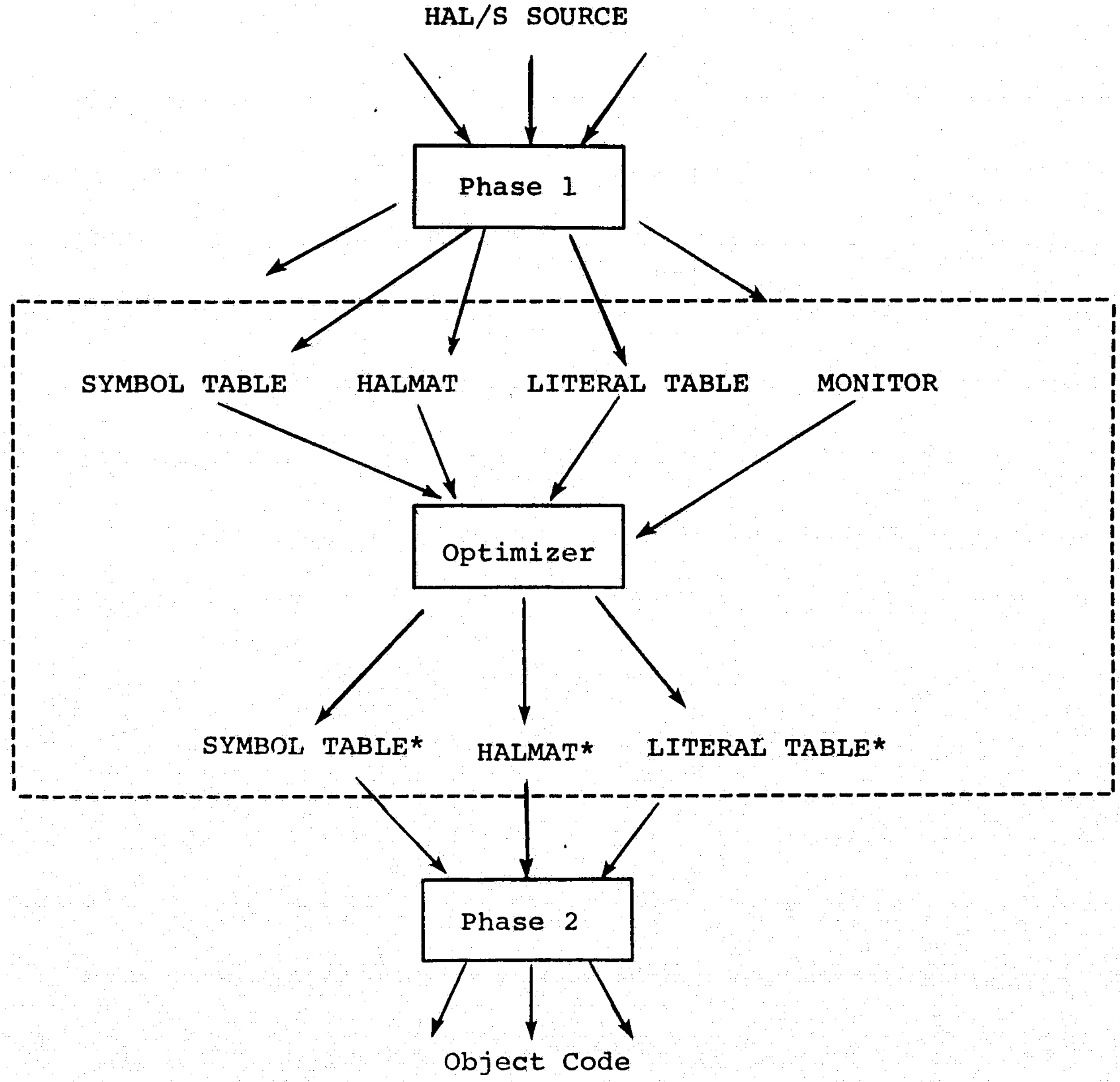
```
OPTION='X6'
```

in the EXEC statement, optimization statistics and timing information will be given.

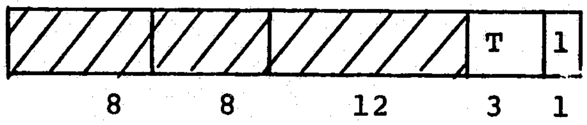
Related Memos

1. IR #127-1, "Common Subexpression Recognition".
2. Shuttle Memo #110-74, "HAL Optimizations".

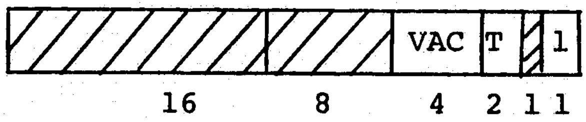
6.2 Functional Description



The HALMAT received by Phase 2 differs from that produced by Phase 1 in the following respects (see HAL/S-360 Compiler System Spec., p. A-2):



Operator Word



Operand Word

1. Except for XXAR operators and as noted below, all operators and VAC operands have tag T = 0.

2. Operators referenced more than once (CSE's) have T = "4". TSUB's may have this bit set, even though referenced once.

3. VAC operands referring to operators which are referenced by later VAC operands have T = "2".

4. The functions previously performed by Phase 2 routine OPTIMISE are now performed by Optimizer routine PREPARE_HALMAT.

The literal table may receive additional entries corresponding to folded constants.

The bit in SYT_FLAGS corresponding to STUB_FLAG (or ARRAY_FLAG) is set in procedures, functions and inline functions which cannot possibly be leaf procedures as an aid to Phase 2.

6.3 Global Flow

General Description

HALMAT statements are processed sequentially. First, PREPARE_HALMAT is called. If optimization has not been disabled and CHICKEN_OUT determines that optimization is allowed, then GROW_TREE builds the NODE list. GET_NODE produces a node, and if it can possibly contain a CSE it is checked with CSE_MATCH_FOUND. Finding a CSE causes REARRANGE_HALMAT to make necessary changes to the HALMAT, and STRIP_NODES to modify the NODE list.

Each node is rescanned until no more CSE's are found, at which time it is entered into the CSE_TAB by TABLE_NODE, thus allowing it to match later CSE's.

Upon completion, statistics are printed if requested by PRINTSUMMARY.

Global Flow Procedures and Data Base

Number	Variable	Use
6.3.1	MAIN_PROGRAM:	
3.1.1	CLOCK	Array of times for PRINTSUMMARY.
3.1.2	STATISTICS	Set by option 'X6'. Prints final statistics.
3.1.3	OPTIMIZING	True until HALMAT finished.
3.1.4	OPTIMIZER_ OFF	Disables optimization. Set by option 'X1' or Phase I bug.
3.1.5	LITCHANGE	True if change to literal file.
3.1.6	WORK3	Saved FREELIMIT.

MAIN_PROGRAM optimizes the HALMAT, block by block.

6.3.2 INITIALIZE:

Number	Variable	Use
3.2.1	TRACE	Option 'X5' or DEBUG H(5) gives dynamic printout of program flow and databases.
3.2.2	WATCH	Option 'X5' or 'X3' or DEBUG H(5) or DEBUG H(3) lists HALMAT changes.
3.2.3	HALMAT_REQUESTED	(Option 'X5' and ¢5) or DEBUG H(6) lists HALMAT as it is processed.
3.2.4	SYT_SIZE	Symbol table size.
3.2.5	LITMAX	Number of literal blocks.
3.2.6	LITSIZE	Literals in a block.
3.2.7	LIT1	First words array of literal block in core.
3.2.8	SYT_USED	Last possible valid symbol.
3.2.9	SYT_WORDS	Index of last word in VALIDITY_ARRAY containing valid bit.

INITIALIZE sets toggles, reads in a literal block, handles based storage, etc.

6.3.3 STORAGE_MGT:

STORAGE_MGT allocates based data.

6.3.4 PRINT_DATE_AND_TIME:

PRINT_DATE_AND_TIME computes date and prints message followed by date.

6.3.5 PRINT_TIME:

PRINT_TIME computes time and prints message followed by time.

6.3.6 NEW_HALMAT_BLOCK:

Number	Variable	Use
3.6.1	OPR	The HALMAT block in core.
3.6.2	CURCBLK	Current HALMAT code block.
3.6.3	CTR	Points to current HALMAT word.

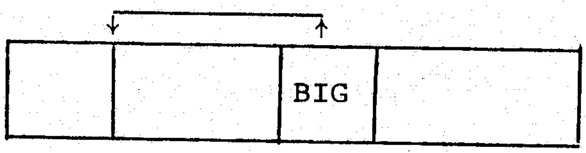
NEW_HALMAT_BLOCK reads in a new HALMAT block and initializes.

6.3.7 PREPARE_HALMAT:

Number	Variable	Use
3.7.1	SMRK_CTR	Index of next SMRK.
3.7.2	LAST_SMRK	Index of last SMRK.

PREPARE_HALMAT extracts inline functions, transports invariant function calls and shaping functions out of arrayed text, and moves array markers (ADLP) to their proper places.

6.3.8 MOVECODE:



Number	Variable	Use
3.8.1	LOW	Start of HALMAT to be moved up.
3.8.2	HIGH	Start of HALMAT to be moved back.
3.8.3	BIG	Number of words moved.
3.8.4	ENTER_TAG	TRUE if references to CSE's may be among words moved.

MOVECODE moves from HIGH to HIGH + BIG - 1 before LOW.

Number	Variable	Use
6.3.9 OPTIMISE:		
3.9.1	STT#	HAL/S statement number.
3.9.2	STILL_NODES	True until no more CSE's can be found with the statement being checked and earlier statements.
3.9.3	SEARCHABLE	False if the node under examination cannot possibly match previously examined nodes.

OPTIMISE governs the flow within a HALMAT block, building tables, checking for CSE's, and changing HALMAT and tables accordingly.

6.3.10 DECODEPOP:

Class 0:

TAG	NUMOP	CLASS 0	OPCODE SUBCODE2		0
8	8	4	8	3	1

Class >0:

TAG	NUMOP	CLASS >0	SUB- CODE	OPCODE		0
8	8		3	5	3	1
			SUBCODE2			

Number	Variable	Use
3.10.1	TAG	See Compiler System Spec., Appendix A, and above.
3.10.2	NUMOP	
3.10.3	CLASS	
3.10.4	OPCODE	
3.10.5	SUBCODE	

DECODEPOP decodes HALMAT operators. (See Compiler System Spec., Appendix A.)

6.3.11 NEXTCODE:

NEXTCODE positions CTR to the next HALMAT operator.

6.3.12 PUT_HALMAT_BLOCK:

PUT_HALMAT_BLOCK writes the changed HALMAT block for Phase II.

6.3.13 PRINTSUMMARY:

3.13.1	CSE#	Number of CSE's processed.
3.13.2	COMPLEX_MATCHES	Number of CSE's which contain other CSE's.
3.13.3	TRANSPOSE_ELIMINATIONS	Number of Matrix Transposes eliminated.
3.13.4	LITERAL_FOLDS	Number of literals folded.
3.13.5	COMPARE_CALLS	Number of calls to COMPARE procedure in CSE_FOUND.
3.13.6	SCANS	Number of times the SCAN routine is used in COMPARE.

Number	Variable	Use
3.13.7	MAXNODE	Largest size of NODE list encountered.
3.13.8	MAX_CSE_TAB	Largest size of CSE_TAB list encountered.
3.13.9	DIVISION_ ELIMINATIONS	Number of Nodes where divides were replaced by multiples.
3.13.10	EXTN_CSES	Number of CSE's which are structure nodes.
3.13.11	TSUB_CSES	Number of CSE's which are structure subscripts.

PRINTSUMMARY prints times and above results.

6.3.14 X_BITS:

X_BITS returns "code optimizer bits" used in PREPARE_HALMAT.

6.3.15 ERRORS:

ERRORS prints error message when error detected in literal collapsing, obtaining storage for phase 1.5, or table overflows.

6.3.16 RELOCATE:

RELOCATE relocates HALMAT after MOVECODE.

6.3.17 DECODEPIP:

DECODEPIP decodes HALMAT operands and prints them if requested.

6.3.18 OPOP:

OPOP returns the operator part of a HALMAT operator.

6.3.19 VAC_OR_XPT:

VAC_OR_XPT returns true if HALMAT operand is a VAC or XPT.

6.4 Stalking The Wild CSE: Table Building

General Description

Each HALMAT statement is checked by CHICKEN_OUT and either allowed, skipped, or both skipped and tables are deleted by ZAP_TABLES. If the statement is allowed, GROW_TREE builds the NODE list. In the process, useless matrix transpose operations are eliminated, additional literals are folded, and unneeded divides are replaced by multiplies.

Stalking Procedures and Data Base

	Number	Variable	Use
6.4.1	CHICKEN_OUT		
	4.1.1	FIRST	First HALMAT operator to be checked.
	4.1.2	LAST	Last HALMAT operator to be checked.
	4.1.3	CLASS0	For class 0 operators: "0" - Statement skipped and ZAP_TABLES called. "1" - Statement skipped. "3" - Statement processed.
	4.1.4	IF_CTR	Index of first CLASS 7 (conditional) operator in sentence or 0.
	4.1.5	ASSIGN_CTR	Index of first assignment or return operator in sentence 0.
	4.1.6	DO_LIST	Stack for simple DO's. Negative if EXIT references corresponding END.
	4.1.7	DO_INX	Index for DO_LIST.
	4.1.8	DO_SIZE	Maximum simple DO nesting permitted.

Number	Variable	Use
4.1.9	DEBUG	Debug toggle set by DEBUG cards.
4.1.10	HALMAT_BLAB	Prints HALMAT block after optimization.
4.1.11	STUB_FLAG	Set in SYT_FLAGS to indicate impossibility of leaf procedure.

6.4.2 ZAP_TABLES

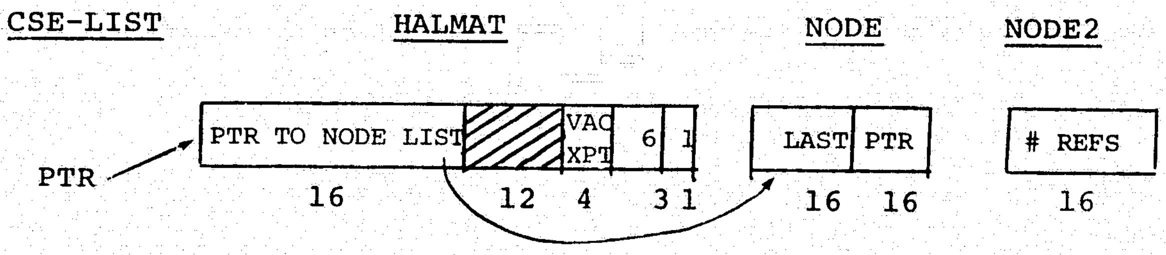
ZAP_TABLES deletes all tables and calls

RELOCATE_HALMAT if CSE has been found.

6.4.3 RELOCATE_HALMAT:

4.3.1	CSE_L_INX	Number of VAC pointers to be relocated. Number of entries in CSE_LIST.
4.3.2	CSE_LIST	Pointers to VAC's that may need relocating. On second pass contains index into NODE list to entry with last reference to CSE so tag can be removed.

RELOCATE_HALMAT relocates certain VAC's.



The NODE pointer replaces the HALMAT pointer. LAST keeps track of last VAC referencing the CSE in question. #REFS is the number of times referenced.

6.4.4 DETAG:

DETAG removes TAG from HALMAT word.

6.4.5 CSE_TAB_DUMP:

CSE_TAB_DUMP prints CSE_TAB, NODE list, and CATALOG_PTR's (parallel to symbol table).

6.4.6 FORMAT:

FORMAT places numbers into N-strings.

6.4.7 CSE_WORD_FORMAT:

CSE_WORD_FORMAT makes NODE list words somewhat readable.

6.4.8 HEX:

HEX converts integer to Hex characters.

6.4.9 EXIT_CHECK:

EXIT_CHECK negates the entry in DO_LIST corresponding to an EXIT. Used to prevent CSE's across simple END's referenced by an EXIT.

6.4.10 ASSIGNMENT:

Number	Variable	Use
4.10.1	PM_FLAGS	Mask to determine if variables can be equated for CSE purposes when they appear in simple assignments.

ASSIGNMENT checks for assignment into a name variable. If present, ZAP_TABLES is called. Otherwise, if a simple assignment (A = B), the variables are marked as identical using CATALOG_PTR and VALIDITY. If not a simple assignment (A = B + ...) then receivers have VALIDITY set to 0, preventing participation in further CSE's.

6.4.11 ST_CHECK:

ST_CHECK verifies that a structure receiver of an assignment contains no name variables.

6.4.12 NAME_CHECK:

NAME_CHECK verifies that a variable is not a name variable.

6.4.13 SYTP:

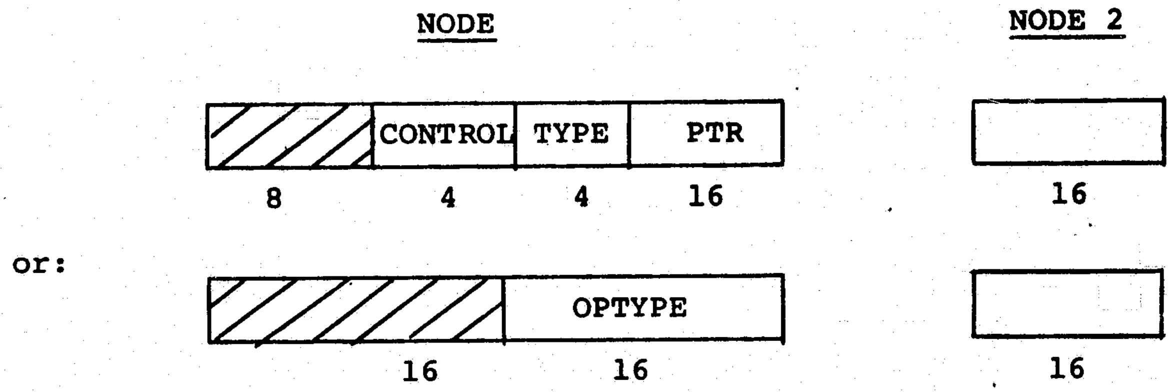
SYTP is true if the HALMAT operand in question is a symbol table pointer.

6.4.14 GROW_TRUE:

Number	Variable	Use
4.14.1	MAX_NODE_SIZE	Size of NODE list.
4.14.2	STILL_NODES	True until all nodes processed in statement in question.
4.14.3	GET_INX	Points to operator word in NODE list of next NODE to be checked for CSE's.

GROW_TREE checks that enough space is available for the Nodes of the statement in question. An END_OF_LIST is placed on the NODE list and BUILD_NODE_LIST is called.

6.4.15 BUILD_NODE_LIST:



4.15.1 NODE — Array of words in CSE word format.

4.15.2 NODE2 — Array of halfwords pointers parallel to NODE.

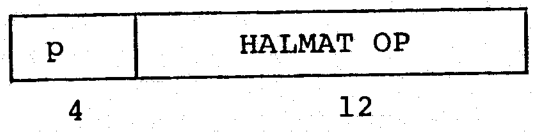
FORMATS

```
                                       CONTROL TYPE PTR       */
 LITERAL                     FIXED INITIAL("2   E   0000"),        /* LAST IN SORT
                                                                   ORDER*/
 IMMEDIATE                   FIXED INITIAL("0   6   0000"),
 TERMINAL_VAC                FIXED INITIAL("0   3   0000"),
 ASTERISK                    FIXED INITIAL("0   7   0000"),
 CSZ                         FIXED INITIAL("0   8   0000"),
 ASZ                         FIXED INITIAL("0   9   0000"),
 OUTER_TERMINAL_VAC          FIXED INITIAL("0   C   0000"),  /*POINTS TO VAC_PTR
                                                                      WORD*/
 VALUE_NO                    FIXED INITIAL("0   B   0000"),
 DUMMY_NODE                  FIXED INITIAL("0   D   0000"),
 SYT                         FIXED INITIAL("0   1   0000"),
 END_CF_NODE                 FIXED INITIAL("F   0   0000"),
 VAC_PTR                     FIXED INITIAL("F   1   0000"),
 END_OF_HALMAT_BLOCK         FIXED INITIAL("F   8   0000"),
 END_OF_LIST                 FIXED INITIAL("F   F   0000"),

 TYPE_MASK                   FIXED INITIAL("0   F   0000"),

 CONTROL_MASK                FIXED INITIAL("F   C   0000"),

 /* FOR DSUB, CONTROL = SHL(ALPHA,1) | EETA */

 PARITY_MASK                 FIXED INITIAL("FFE F 0000"),

 /* FOR TSUB, CONTROL = SHL(ALPHA - 7,1) | BETA */
```

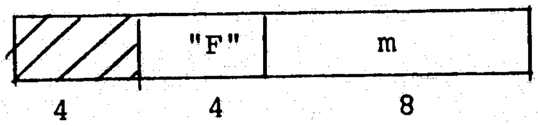
Number	Variable	Use
4.15.3	LITERAL	Literal operand. PTR = 0. NODE2 is a pointer to the literal table. CONTROL = 3 if odd parity.
4.15.4	IMMEDIATE	Immediate operand. PTR = value. NODE2 = 0. CONTROL = 1 if odd parity.
4.15.5	TERMINAL_VAC	VAC or XPT operand which points to different node. PTR is a pointer to the VAC_PTR word in the NODE list of that node. NODE2 is a pointer to the END_OF_NODE word of the NODE containing the TERMINAL_VAC. CONTROL = 1 if odd parity.
4.15.6	OUTER_TERMINAL_VAC	VAC or XPT operand which points to a CSE. PTR is same as for TERMINAL_VAC. NODE2 is a pointer into CSE_TAB for the CSE pointed to. CONTROL = 1 if odd parity.
4.15.7	VALUE_NO	Value number. PTR is a pointer into CSE_TAB. NODE2 is the WIPEOUT#. CONTROL = 1 if odd parity.
4.15.8	DUMMY_NODE	You guessed it. CONTROL = 1 if odd parity.
4.15.9	SYT	Symbol table pointer. Only present between GROW_TREE and GET_NODE. PTR is symbol table pointer. CONTROL = 1 if odd parity.
4.15.10	END_OF_NODE	No more operands for this node. NODE2 points to optype of Node in NODE list.
4.15.11	VAC_PTR	PTR is index of HALMAT operator of Node in question. NODE2, if non-zero, is a pointer into the CSE_TAB (in this case, the Node is a CSE).

Number	Variable	Use
4.15.12	END_OF_HALMAT_BLOCK	Unused.
4.15.13	END_OF_LIST	Last entry in NODE list for the statement in question.
4.15.14	OPTYPE	Internal operator (set by CLASSIFY) derived from HALMAT operator.
4.15.I5	N_INX	Indexes NODE and NODE2.

OPTYPE Formats



Normal format. Commutative operators always become the even paritied operator, e.g. SSUB becomes SADD with PARITY = 1. p is the precision for conversion operators and zero otherwise.



Built-in functions. m is the index of the function.

STRUCTURE OF NODE LIST:

	NODE	NODE2
	END_OF_LIST	0
Increasing	VAC_PTR	0 OR CSE_TAB PTR
N_INX ↓	END_OF_NODE	PTR TO OPTYPE
	OPERANDS:	
	VALUE_NO: (PTR TO CSE_TAB)	WIPEOUT#
NODE:	OUTER_TERMINAL_VAC (PTR TO "VAC_PTR" OF NODE)	CSE_TAB_PTR
	LITERAL	PTR TO LIT TABLE
	TERMINAL_VAC (PTR TO "VAC_PTR" OF NODE)	PTR TO END OF NODE
	SYT(PTR TO SYMBOL TABLE)	0
	OPTYPE	0 OR PTR TO "END_OF_NODE" OF NODE CONTAINING TERMINAL_VAC REFERENCING NODE. IF TOPTAG THEN UNRELIABLE.
NODE:		
	END_OF_LIST	0
MORE NODES:		

Example:

F = A - B C;

produces HALMAT:

```
0.  SSPR

              B(SYT)
              C(SYT)

3.  SSUB

              A(SYT)
              0(VAC)

4.  SASN

              3(VAC)
              F(SYT)
```

After BUILD_NODE_LIST:

	NODE			NODE2	
	Control	Type	Ptr.		
1.	END_OF_LIST		0	0	
2.	VAC_PTR		3	0	Node
3.	END_OF_NODE		0	6	
4.	1	TERMINAL_VAC	8	3	
5.	0	SYT	A	0	
6.	0	0	SADD	0	
7.	VAC_PTR		0	0	Node
8.	END_OF_NODE		0	11	
9.	0	SYT	B	0	
10.	0	SYT	C	0	
11.	0	0	SSPR	3	

Number	Variable	Use
4.15.16	ADD	Stack of operators to be added to the present Node. Indexed by A_INX.
4.15.17	A_PARITY	Stack of parities for corresponding operator. Odd parities for subtracts and divides. Indexed by A_INX.
4.15.18	A_INX	Index for ADD and A_PARITY.
4.15.19	DIFF_NODE	Stack of Nodes in the same statement but different than the one currently being processed. Indexed by D_N_INX.
4.15.20	DIFF_PTR	Used to get TERMINAL_VAC's pointing to VAC_PTR of appropriate Node. Indexed by D_N_INX.
4.15.21	D_N_INX	Index for DIFF_NODE and DIFF_PTR.
4.15.22	TRANSPARENT	HALMAT operator which produces no Node but whose operands may produce Nodes cause TRANSPARENT = TRUE.
4.15.23	BFNC_OK	False for Built-in functions which produce no Nodes and thus partitícpate in no CSE's.
4.15.24	EON_PTR	Points to END_OF_NODE of present NODE.
4.15.25	REF_PTR	Pointer to TERMINAL_VAC referring to the present NODE.
4.15.26	TYPE	HALMAT operand type.

Number	Variable	Use
4.15.27	PRTYEXPN	Parity of the operand in question.
4.15.28	OP	HALMAT operator (of even parity) for present Node.
4.15.29	DIVIDE#	Number of divides in this Node.

BUILT_NODE_LIST adds the Nodes from a statement to the NODE list. Constants are folded and unneeded divisions eliminated.

6.4.16 LIT_CONVERSION:

LIT_CONVERSION replaces a VAC referencing a harmless literal conversion by the literal itself.

6.4.17 CONVERSION_TYPE:

CONVERSION_TYPE checks a literal conversion to see if it is harmless.

6.4.18 CLASSIFY:

Number	Variable	Use
4.18.1	SET_P	True if PARITY is to be set.
4.18.2	FIX_SPECIALS	True if unneeded matrix transposes are to be eliminated.
4.18.3	PARITY	Odd if subtraction or division.

CLASSIY creates the OPTYPE from a HALMAT operator, sets PARITY, and eliminates unneeded matrix transposes.

6.4.19 CHECK_TRANSPOSE:

CHECK_TRANSPOSE changes M^T V to V M and V M^T to M V.

6.4.20 PRINT_SENTENCE(PTR):

PRINT_SENTENCE formats and prints HALMAT from PTR to the next SMRK.

6.4.21 SET_NONCOMMUTATIVE:

Number	Variable	Use
4.21.1	BIT_TYPE	True if bit type.
4.21.2	NONCOMMUTATIVE	True if "Noncommutative".
4.21.3	REVERSE_OP	Odd paritied operator corresponding to OP.

SET_NONCOMMUTATIVE returns NONCOMMUTATIVE and sets BIT_TYPE, TRANSPARENT, and REVERSE_OP.

6.4.22 NO_OPERANDS:

NO_OPERANDS returns the number of HALMAT operands following an operator.

6.4.23 PTR_TO_VAC:

PTR_TO_VAC formats a PTR_TO_VAC word for the NODE list.

6.4.24 FORM_VAC:

FORM_VAC formats a TERMINAL_VAC word for the NODE list.

6.4.25 FORM_TERM:

FORM_TERM formats terminal word for NODE list.

6.4.26 TERMINAL:

Number	Variable	Use
4.26.1	TAG	If TRUE, considers a VAC or XPT pointing to a different operator as terminal.

TERMINAL returns true if the operand in question is an outer node for the tree decomposition of the statement in question.

6.4.27 BUMP_CSE:

BUMP_CSE puts a literal on the CSE list for literal folding.

6.4.28 ELIMINATE_DIVIDES:

ELIMINATE_DIVIDES eliminates all but one divide from a Node.

6.4.29 COLLAPSE_LITERALS:

COLLAPSE_LITERALS folds literals and modifies HALMAT and NODE list accordingly.

6.4.30 COMBINED_LITERALS:

Number	Variable	Use
4.30.1	DW	Common data word for communication with XPL monitor.

COMBINED_LITERALS does lit arithmetic by monitor calls.

6.4.31 FILL_DW:

FILL_DW fills DW with literal.

6.4.32 LIT_ARITHMETIC:

LIT_ARITHMETIC performs monitor call to do literal arithmetic.

6.4.33 SAVE_LITERAL:

SAVE_LITERAL creates a new literal table entry and returns pointer to it.

6.4.34 GET_LITERAL

Number	Variable	Use
4.35.1	LITORG	Smallest index of literal in core.
4.35.2	LITLIM	Largest index of literal in core.
4.35.3	CURLBLK	Literal block in core.

GET_LITERAL guarantees a literal in core.

6.4.35 MESSAGE_FORMAT:

MESSAGE_FORMAT formats a NODE LITERAL word for diagnostics.

6.4.36 VALIDITY:

VALIDITY returns the validity bit of the symbol in question.

6.4.37 SET_VALIDITY:

SET_VALIDITY sets the validity bit of the symbol in question.

4.38.1	VALIDITY_ARRAY	Index i = 1 if symbol is eligible for a CSE.

6.4.38 ASSIGN_TYPE:

ASSIGN_TYPE returns true if operator is regular or structure assignment.

6.4.39 TERM_CHECK:

TERM_CHECK either calls ZAP_TABLES for assignment to major or minor structure or else sets validity false for a structure node receiver.

6.5 Recognition

General Description

GET_NODE gets a node and, if a CSE can possibly exist with a previously processed Node, CSE_FOUND searches for CSE's. GROW_TREE has produced Nodes such that a backward scan of the NODE list by GET_NODE examines Nodes in the proper order.

Recognition Procedures and Data Base

Number	Variable	Use
6.5.1 GET_NODE:		
5.1.1	SEARCH	Stack of NODE list operands which might be part of a CSE. Indexed by SEARCH_INX.
5.1.2	SEARCH2	Same as search, except NODE2 entries go here.
5.1.3	SEARCH_INX	Index for SEARCH and SEARCH2.
5.1.4	GET_INX	Pointer to NODE list rised by GET_NODE.
5.1.5	NODE_BEGINNING	Points to OPTYPE of Node in NODE list.
5.1.6	SYT_POINT	Symbol table pointer.
5.1.7	CATALOG_PTR	Array parallel to symbol table of VALUE_NO's with pointers to CSE_TAB.

Example:

```
F = A + B;
```

Symbol Table	CATALOG_PTR		VALIDITY
	Type	Ptr	
A	B	6	1
B	B	11	0
.			
.			
.			
F	-	-	-

From the above values, we may deduce:

1. The VALUE_NO for A is 6.

2. The VALUE_NO for B is 11.

3. B no longer has a valid VALUE_NO since it must have recently been the receiver in an assignment.

4. No further searching for CSE's need be made since we are left with only one valid operand.

Number	Variable	Use
5.1.9	NODE_SIZE	Number of oprands in SEARCH list for a Node.
5.1.10	PRESENT_NODE_PTR	Points to VAC_PTR word of Node presently being examined by GET_NODE.

GET_NODE takes a Node and places all operands which have been encountered before this Node and after the last assignment or ZAP_TABLES into the SEARCH list. SYT words are replaced by VALUE_NO's in the process. The Node is sorted if not NONCOMMUTATIVE. If appropriate, the SEARCH list is sorted.

6.5.2 TYPE:

TYPE returns type of a word in NODE list format.

6.5.3 CATALOG:

Number	Variable	Use
5.3.1	NEW_OP	True if CSE_TAB entries already exist for the VALUE_NO in question, but not one for the present OPTYPE. False if no CSE_TAB entry at all.
5.3.2	CSE_TAB	Doubly linked array of pointers into NODE list.

CSE TAB FORMATS

CATALOG ENTRY:

```
     #1      --PTR TO FIRST NODE ENTRY IN CSE_TAB FOR THIS OPCODE
     #2      --OPTYPE
     #3      --PTR TO NEXT CATALOG ENTRY FOR DIFFERENT OPTYPE
               BUT SAME VALUE_NO, ETC.  0 FOR LAST CATALOG
               ENTRY FOR THIS VALUE_NO, ETC.
```

NODE ENTRY:

```
     #1      --PTR TO OPTYPE OF A NODE
     #2      --PTR TO NEXT NODE ENTRY IN CSE_TAB FOR THIS
               OPTYPE AND VALUE_NO, ETC.  0 FOR LAST ENTRY.
```

CATALOG sets up a catalog entry and the first node entry for a particular VALUE_NO and a particular OPTYPE in the CSE_TAB.

6.5.4 CATALOG_ENTRY:

CATALOG_ENTRY adds a node entry to CSE_TAB.

6.5.5 GET_FREE_SPACE:

Number	Variable	Use
5.5.1	FREE_BLOCK_BEGIN	Beginning of unused block.
5.5.2	FREE_SPACE	Amount of space in block.

GET_FREE_SPACE gets an unused block in CSE_TAB.

6.5.6 CATALOG_SRCH:

Number	Variable	Use
5.6.1	CSE_INX	See below.

CATALOG_SRCH checks catalog entries in the CSE_TAB for a particular VALUE_NO or OUTER_TERMINAL_VAC.

If a matching OPTYPE is found, CSE_INX is set to the first mode entry in CSE_TAB for that OPTYPE. A pointer to the appropriate catalog entry in CSE_TAB is returned.

Otherwise, CSE_INX is set to the last catalog entry present for the given VALUE_NO, etc., and 0 is returned.

6.5.7 SORTER:

SORTER sorts the NODE list.

6.5.8 SEARCH_SORTER:

SEARCH_SORTER sorts the SEARCH list.

6.5.9 CSE_MATCH_FOUND:

Number	Variable	Use
5.9.1	REVERSE	True if reverse CSE, e.g. F = (A - B) (B - A);

CSE_MATCH_FOUND calls COMPARE and, if appropriate, does a reverse compare.

6.5.10 SETUP_REVERSE_COMPARE:

Number	Variable	Use
5.10.1	SEARCH_REV	Same as SEARCH but with parities changed.
5.10.2	SEARCH2_REV	Same as SEARCH2.

SETUP_REVERSE_COMPARE copies SEARCH and SEARCH2 into SEARCH_REV and SEARCH2_REV changing parities and sorting.

6.5.11 CONTROL:

CONTROL returns control field of a word in Node list format.

6.5.12 COMPARE:

Number	Variable	Use
5.12.1	PREVIOUS_NODE_ OPERAND	Points to first operand of previous Node.
5.12.2	CSE	List of matched operands from SEARCH list. Indexed by CSE_ FOUND_INX.
5.12.3	CSE2	List of matched operands from SEARCH2 list. Indexed by CSE_FOUND_INX.

Number	Variable	Use
5.12.4	CSE_FOUND_INX	Indexes CSE, CSE2.
5.12.5	PREVIOUS_NODE	Pointer to OPTYPE of previous Node.
5.12.6	PRESENT_HALMAT	Points t○ HALMAT for present Node.
5.12.7	PREVIOUS_NODE_ PTR	Points to VAC_PTR word of previous match's Node.
5.12.8	PREVIOUS_HALMAT	Points to HALMAT for previous match.

COMPARE checks if a Node has a CSE with a previous Node.

6.5.13 COMPARE_LITERALS:

COMPARE_LITERALS compares 2 literals, returning true if equal.

6.6 Bringing Home the Bacon: HALMAT Rearranging

General Description

After a CSE is found, the HALMAT is rearranged so that the CSE is in fact computed in both the previous and present Node. The previous Node is tagged. All references are tagged and their HALMAT VAC pointers replaced by pointers into the NODE list, to the appropriate VAC_PTR. Such relocation is necessary since the CSE may be moved due to a later CSE.

The second (or present) computation of the CSE is now replaced by NOP's (except for the case of TSUBS where its CSE TAG is set). In some cases, the negative or reciprocal of the CSE is called for, and the HALMAT for the present node is accordingly modified.

Bacon Procedures and Data Base

6.6.1 SETUP_REARRANGE:

Number	Variable	Use
6.1.1	PNPARITY0#	Number of parity 0 operands in previous Node.
6.1.2	PNPARITY1#	Number of parity 1 operands in previous Node.
6.1.3	FNPARITY0#	Number of parity 0 operands in forward Node.
6.1.4	FNPARITY1#	Number of parity 1 operands in forward Node.
6.1.5	M PARITY0#	Number of parity 0 operands in match (CSE).
6.1.6	M PARITY1#	Number of parity 1 operands in match (CSE).
6.1.7	NEW_MODE_PTR	Points to VAC_PTR word in NODE list for the new CSE.
6.1.8	TOTAL_MATCH_PRES	TRUE if CSE = present Node.

SETUP_REARRANGE sets the above variables needed by REARRANGE_HALMAT.

Number	Variable	Use
6.6.2	REARRANGE_HALMAT:	
6.2.1	FORWARD_UNMATCHED_PLUS	TRUE if there is a parity 0 operand in the present (or forward) Node which is not in the CSE.
6.2.2	FORWARD_MATCHED_MINUS	TRUE if forward Node has parity 1 operand in the CSE.
6.2.3	FORWARD_MATCHED_PLUS	TRUE if forward Node has parity 0 operand in the CSE.
6.2.4	FORWARD	TRUE if forward (= present) Node being processed.
6.2.5	TOPTAG	TRUE if previous Node was already a CSE.
6.2.6	TOTAL_MATCH_PREV	TRUE if previous Node = CSE.
6.2.7	MULTIPLE_MATCH	TOPTAG & TOTAL_MATCH_PREV.
6.2.8	HALMAT_PTR	Last HALMAT operator in CSE.
6.2.9	HALMAT_NODE_START	First HALMAT operator in the Node.
6.2.10	ALTER_HALMAT	True unless TSUB CSE where HALMAT is not NOP'ed.

REARRANGE_HALMAT rearranges, flags, NOP's, etc., HALMAT to create a CSE with its references.

6.6.3 SET_HALMAT_FLAG:

SET_HALMAT_FLAG sets CSE TAG in HALMAT.

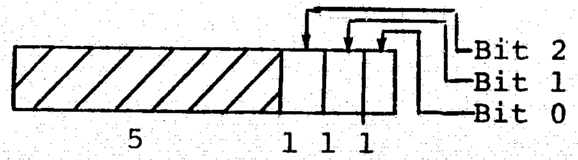
6.6.4 COLLECT_MATCHES:

Number	Variable	Use
6.4.1	ELIMINATE_DIVIDES	= 1 unless COLLECT_MATCHES called to eliminate unneeded divisions.
6.4.2	LAST_INX	Pointer to the last HALMAT operator written during processing of this Node.
6.4.3	H_INX	Pointer to HALMAT to keep track of scan of Node.
6.4.4	INVERSE	TRUE if generated HALMAT operators are to be of odd parity.
6.4.5	P0	Number of even parity operands not in CSE.
6.4.6	P1	Number of odd parity operands not in CSE.
6.4.7	POINT1	Pointer to a partial computation of a Node.

COLLECT_MATCHES groups HALMAT for the CSE computation at the beginning of the Node in question.

6.6.5 FLAG_NODE:

6.5.1	FLAG	Array of flags parallel to the HALMAT.



Bit 0 is TRUE if corresponding HALMAT is an operator in the Node in question.

Bit 1 is TRUE if corresponding HALMAT is an operand in the CSE in question.

Bit 2 is the parity of the corresponding HALMAT operator or operand in the Node.

FLAG_NODE sets bits 0 and 2 in the FLAG array for the Node in question.

6.6.6 HALMAT_FLAG:

HALMAT_FLAG returns the CSE tag for the HALMAT operator or operand in question.

6.6.7 SET_FLAG:

SET_FLAG sets a bit in a given FLAG word.

6.6.8 FLAG_MATCHES:

FLAG_MATCHES sets bit 1 in the FLAG array for a Node.

6.6.9 FLAG_V_N:

FLAG_V_N flags bit 1 in the FLAG array of a Node corresponding to a given VALUE_NO.

6.6.10 FLAG_VAC_OR_LIT:

FLAG_VAC_OR_LIT flags bit 1 in the FLAG array of a Node corresponding to a given OUTER_TERMINAL_VAC or LITERAL.

Number	Variable	Use

6.6.11 SET_WORDS:

6.11.1	OPPARITY	Parity of HALMAT operators generated.
6.11.2	MATCHED_OPS	TRUE if non VAC operands are to be in the CSE.
6.11.3	TERMINAL#	Number of non VAC operands.
6.11.4	TAG	True if CSE tag to be set on operator.
6.11.5	SPECIAL	Special case.

SET_WORDS creates a HALMAT operator with two operands of desired characteristics.

6.6.12 NEXT_FLAG:

NEXT_FLAG finds the next HALMAT word with the specified FLAG bit set.

6.6.13 FORM_OPERATOR:

FORM_OPERATOR forms a HALMAT operator word.

6.6.14 FORCE_MATCH:

FORCE_MATCH forces a CSE operand into the operand in question. What was there already is switched with the new operand.

6.6.15 SWITCH:

SWITCH interchanges two HALMAT operands and their FLAGS. If either was tagged, it is entered into the CSE list for later relocation. If a VAC reference is moved below its pointer, HALMAT is shifted by MOVE_LIMB.

6.6.16 ENTER:

ENTER puts a pointer into the CSE_LIST for possible relocation later.

6.6.17 MOVE_LIMB:

MOVE_LIMB moves and relocates HALMAT and relocates the NODE list correspondingly.

6.6.18 FORCE_TERMINAL:

FORCE_TERMINAL forces a terminal HALMAT operand of correct parity to the given spot.

6.6.19 PUSH_OPERAND:

PUSH_OPERAND forces a terminal operand forward into a harmless slot.

6.6.20 SET_VAC_REF:

SET_VAC_REF creates a HALMAT VAC or XPT operand.

6.6.21 PUT_NOP:

PUT_NOP replaces the CSE computation with NOP's.

6.6.22 REFERENCE:

REFERENCE finds the VAC referencing a given HALMAT operator.

6.6.23 BOTTOM:

BOTTOM finds where a limb joins the tree so the limb can be moved.

Number	Variable	Use
6.6.24 GET_LIT_ONE:		
6.24.1	PREVIOUS_CALL	TRUE if literal 1 already generated.

GET_LIT_ONE generates a literal 1 and returns its pointer.

6.7 Table Updating

General Description

After finding a CSE and rearranging HALMAT, the NODE list and CSE_TAB are modified. A new Node for the CSE is created if needed. CSE operands are removed from the previous and present Node if needed. Resorting is sometimes required.

When no CSE is found, TABLE_NODE modifies CSE_TAB so that later Nodes can match with the present Node.

Updating Procedures and Data Base

Number	Variable	Use
6.7.1 STRIP_NODES:		
7.1.1	NEW_NODE_OP	Pointer to NODE list operator word of CSE.
7.1.2	PREV_TREE_TOP	TRUE if previous Node = CSE and it has no predecessor Node.
7.1.3	PREV_REF	Pointer to NODE operator which has operand referencing CSE, if CSE = previous Node.
7.1.4	PREV_REF_OF_VAC	Pointer to Node operand referencing CSE, if CSE = previous Node.
7.1.5	PRES_REF_OF_VAC	Pointer to NODE operand referencing CSE if CSE = present Node.
7.1.6	COMPLEX_MATCH	TRUE if CSE contains OUTER_TERMINAL_VAC.

STRIP_NODES removes CSE operands from Nodes and creates a Node for the CSE if necessary. Sorting, parity changing, and CSE_TAB modification are done where appropriate.

6.7.2 SET_O_T_V:

SET_O_T_V finds the TERMINAL_VAC referencing a Node and returns its index in the NODE list. Where appropriate, it is set to an OUTER_TERMINAL_VAC.

6.7.3 TABLE_NODE:

TABLE_NODE puts references into the CSE_TAB for NODE operands not in a CSE.

6.7.4 CATALOG_VAC:

CATALOG_VAC sets up initial entries for OUTER_TERMINAL_VAC's in CSE_TAB.

6.7.5 REVERSE_PARITY

REVERSE_PARITY switches parity of a NODE list operand.

6.8 HAL/S Option Specifications and Compiler Directives

Following are toggles recognized by the OPTIMIZER.

HAL/S Option Specifications

X1	Optimizer off.
X3	WATCH. HALMAT changes are printed.
X5	TRACE. Prints program flow and data bases.
X6	STATISTICS. Prints timing and other statistics.

Compiler Directives

By inserting a statement:

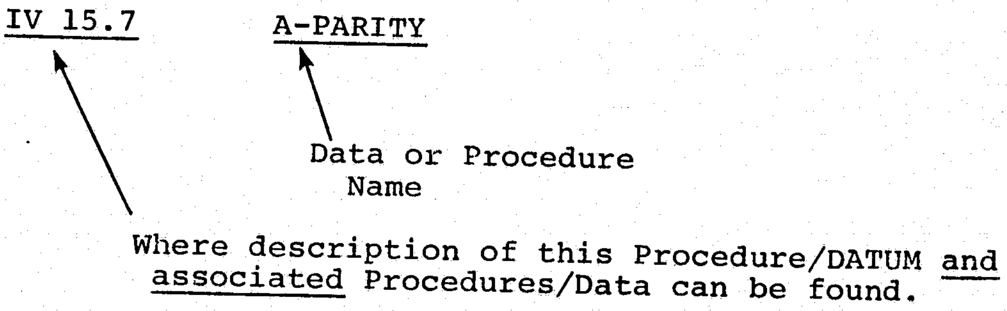
```
DEBUG H(#)
```

starting in column 1, the following actions occur for different values of #:

DEBUG H(1)	Optimizer off until next such statement encountered. No CSE's recognized across the pair of DEBUG's.
DEBUG H(2)	Same as above, but CSE's may be recognized across the pair.
DEBUG H(3)	WATCH status changed.
DEBUG H(5)	TRACE status changed.
DEBUG H(6)	HALMAT_REQUESTED status changed.
DEBUG H(7)	HALMAT_BLAB status changed.
DEBUG H(64)	Set VALIDITY TRACE status changed.

6.9 Alphabetical Index of Names Used in Phase 1.5

Example:



IV 15.7

A-PARITY

Data or Procedure Name

Where description of this Procedure/DATUM and associated Procedures/Data can be found.

By grouping like data and procedures in the previous sections, it is hoped that the time needed to understand procedures in the Optimizer will be greatly reduced.

The algorithm used for CSE recognition is contained in "Common Subexpression Recognition", IR #127-2.

4.15.18	A_INX	
4.15.17	A_PARITY	
4.15.16	ADD	
6.2.10	ALTER_HALMAT	
4.1.5	ASSIGN_CTR	
4.39	ASSIGN_TYPE	label
4.10	ASSIGNMENT	label
4.15.23	BFNC_OK	
4.21.1	BIT_TYPE	
4.23	BOTTOM	label
4.27	BUMP_CSE	label
5.3	CATALOG	label
5.3.1	NEW_OP	
5.4	CATALOG_ENTRY	label
5.6	CATALOG_SRCH	label
7.4	CATALOG_VAC	label
4.1	CHICKEN_OUT	label
9.1.1	FIRST	
9.1.2	LAST	
2.10.3	CLASS	
4.18	CLASSIFY	label
4.18.1	SET_P	
4.18.2	FIX_SPECIALS	
3.1.1	CLOCK	
4.29	COLLAPSE_LITERALS	label
6.4	COLLECT_MATCHES	label
6.4.1	ELIMINATE_DIVIDES	
4.30	COMBINED_LITERALS	label
3.13.5	COMPARE_CALLS	
5.13	COMPARE_LITERALS	label
7.1.6	COMPLEX_MATCH	
3.13.2	COMPLEX_MATCHES	
5.11	CONTROL	label
4.17	CONVERSION_TYPE	label

3.15	ERRORS	label
4.9	EXIT_CHECK	label
3.13.10	EXTN_CSES	
4.32	FILL_DW	label
6.5.1	FLAG	
6.8	FLAG_MATCHES	label
6.5	FLAG_NODE	label
6.9	FLAG_V_N	label
6.10	FLAG_VAC_OR_LIT	label
6.1.3	FNPARITY0#	
6.1.4	FNPARITY1#	
6.14	FORCE_MATCH	label
6.18	FORCE_TERMINAL	label
6.13	FORM_OPERATOR	label
6.25	FORM_TERM	label
6.24	FORM_VAC	label
3.6	FORMAT	
6.2.4	FORWARD	
5.5.1	FREE_BLOCK_BEGIN	
5.5.2	FREE_SPACE	
5.5	GET_FREE_SPACE	label
4.14.3	GET_INX	
6.24	GET_LIT_ONE	label
4.35	GET_LITERAL	label
5.1	GET_NODE	label
4.14	GROW_TREE	label
6.4.3	H_INX	
4.1.10	HALMAT_BLAB	
6.6	HALMAT_FLAG	label
6.2.9	HALMAT_NODE_START	
6.2.8	HALMAT_PTR	
3.2.3	HALMAT_REQUESTED	
4.8	HEX	

6.1.7	NEW_NODE_PTR	
6.12	NEXT_FLAG	label
3.11	NEXTCODE	label
4.22	NO_OPERANDS	label
4.15.1	NODE	
5.1.5	NODE_BEGINNING	
5.1.9	NODE_SIZE	
4.15.2	NODE2	
4.21.2	NONCOMMUTATIVE	
3.10.2	NUMOP	
6.12.1	NUMOP_FOR_REARRANGE	
4.15.28	OP	
3.10.4	OPCODE	
3.18	OPOP	label
3.6.1	OPR	
3.9	OPTIMISE	label
3.1.3	OPTIMISING	
3.1.4	OPTIMIZER_OFF	
4.15.14	OPTYPE	
4.15.6	OUTER_TERMINAL_VAC	
4.18.3	PARITY	
4.10.3	PM_FLAGS	
6.1.1	PNPARITY0#	
6.1.2	PNPARITY1#	
6.4.7	POINT1	
3.7	PREPARE_HALMAT	label
5.12.6	PRESENT_HALMAT	
5.1.10	PRESENT_NODE_PTR	
6.24.1	PREVIOUS_CALL	
5.12.8	PREVIOUS_HALMAT	
5.12.5	PREVIOUS_NODE	
5.12.1	PREVIOUS_NODE_OPERAND	
5.12.7	PREVIOUS_NODE_PTR	
3.4	PRINT_DATE_AND_TIME	label

3.1.6	WORK3	
3.14	X_BITS	label
4.2	ZAP_TABLES	label

7.0 RUNTIME LIBRARIES

The HAL/S compilers generate calls to an extensive runtime library. The library routines: implement all of the functions described in Appendix C of the HAL/S Language Specification; implement the HAL/S input/output facilities; implement most of the matrix/vector operations; augment the in-line code generation of the compiler in several other special cases. HAL/S-360 does not provide genuine real time facilities but does simulate them via a collection of runtime routines collectively called the Real Time Executive.

The runtime library for HAL/S-FC is described in great detail in Chapter 5 of the HAL/S-FC Compiler System Specification. The FC descriptions, augmented by Chapter 5 of the HAL/S-360 Compiler System Specification, serve to define that part of the library which is common to both HAL/S-360 and HAL/S-FC. In addition to this common library, HAL/S-360 requires:

- the real time executive described in Chapter 10 of this document,

- SDL interfaces described in the HAL/SDL ICD.

8.0 HALLINK

8.1 General Comments

HALLINK is the generic name for two programs, HALLINK and HALLKED, which together link edit object decks produced by the HAL compiler. The HALLINK program invokes the IBM OS linkage editor, checks the condition codes returned by the OS linkage editor, and invokes the HALLKED program. HALLKED examines the load modules produced by OS to supply additional information to the OS link editor for a second invocation.

8.2 Description of the HALLINK Program

HALLINK first determines whether a PARM field is present. If a PARM field exists, HALLINK scans the field, looking for the character "slash" (/). That portion of the PARM field which precedes the slash is passed to the second link edit, with NCAL appended to it. The characters following the slash are interpreted as PARMs to HALLINK and HALLKED, and are decoded and stored in a table as information to be passed to HALLKED. The available PARMS and the action taken for each are described in the HAL/SDL ICD.

The program then determines whether it is being passed an alternate DD list. If so, HALLINK modifies its own internal alternate DD lists, which it then passes to the link phases and HALLKED to reflect the user's wishes.

If the option 'PRIVLIB' is specified, HALLINK attempts to invoke the link editors and HALLKED from a library pointed to by a DD card with a DD name of 'LINKLIB'. If PRIVLIB is not specified or if LINKLIB cannot be opened, the invoked programs are sought in the STEPLIB, JOBLIB, or system libraries, as defined in the OS JCL manual.

After the first invocation of the link editor, HALLINK checks the link editor's resultant condition code. If this code is greater than 8, the step is aborted immediately, and the system condition code is set equal to the link editor's condition code. If the code is less than 8, HALLKED is invoked. If, upon return from HALLKED, the condition code is greater than 4, the step is aborted; otherwise the second link edit step is invoked. On return from the second link edit, the system return code is set equal to the second link editor's return code, unless HALLKED has returned a condition code of 4, in which case the system code is set to one greater than the second link editor's code.

Lines		
From	To	Description
279	294	Invoke the link editor a second time.
296	312	Return to calling program.

Variable Usage.

Variable	Usage
RCODE	Store return code from HALLKED (right shifted 2 bits).
MVCP1	Move instruction executed to move PARM field to first link edit.
MVCP3	Move instruction executed to move PARM field to second link edit.
CLC	Compare instruction executed in parsing PARM field for HALLKED PARMS.
PNOGO	PARM field passed to link edit if 'OSLOAD' or 'NOGO' parms.
PARMFLD1	PARM field for first link step.
TESTP	PARM field optionally passed to first link edit.
PARMFLD3	PARM field for second link edit.
NCAL	PARM field passed to second link edit.
SAVE	Save area for OS calls.
DDLIST1 (and variables until LIST1END)	Alternate DD list for first link edit.
ZERO (and variables until LIST2END)	Alternate DD list for HALLKED.
DDLIST3 (and variables until LIST3END)	Alternate DD list for second link edit.
NAME1	Name of first link editor to be invoked.
NAME2	Name of HALLKED to be invoked.

8.2.1 Detailed Description of the Functioning of HALLINK

Lines		
From	To	Description
2	19	Macro used to generate table with HALLINK options.
21	32	Set up OS calling sequences.
34	101	Check for parmfield, and if present scan for slash. Lines 50 through 111 parse the HALLINK PARMS. The algorithm used is a linear search through the valid options, and all character strings not found are ignored. If a match is detected, a byte is set in the table named options (on line 70), corresponding to the option used. Starting on line 75, the program determines whether or not to pass the PARM field to both link edits, depending on the 'BOTH' HALLINK parm, and also whether to pass the option 'TEST' to the first, which is triggered by the absence of the 'SDL' HALLINK PARM.
103	210	Check for presence of alternate DD list, and if there, pass any overrides on to the routines who are to receive them. The DD list format is described in the HAL/SDL ICD.
212	224	Check if 'PRIVLIB' specified. If so, then try to open. In unable to open, then ignore the option.
225	228	Check if a load library was being constructed. If so, then skip the first invocation of the link editor and also of HALLKED.
229	250	Invoke the link editor for the first time, then check return code. If greater than 8, skip to end of program.
252	277	Invoke HALLKED. Check return code. If greater than 4, skip to end of program. If load module member name had been supplied to the first link edit step, then also pass the same name to the second.

Variable	Usage
NAME3	Name of second link editor to be invoked.
OPTABL	List of options recognized, preceded by the byte count minus 1 of the number of characters in the name of the option.
LINKLIB	DCB opened when private library to be used for primary source of loaded code.

8.3 General Comments and Warnings Regarding HALLKED

HALLKED is the 'real' HALLINK, in the sense that it does all the actual processing of the load modules produced by the OS linkage editor and constructs the necessary object decks needed to complete the load module.

There are five functional portions to the program:

1) Initialization of DD names from the alternate DD list, the opening of the files and the acquisition of core for tables.

2) The cross checking of the version numbers of the routines in the load module.

3) The reading of the load module and the construction of the tables for use by part 4).

4) Computation of stack sizes, output of necessary object decks and link editor directive cards, possibly merging user-supplied directives.

5) Closing of files and of freeing all space used for tables and I/O buffers.

The five portions of the program are highly independent of each other and are treated separately.

Warning: There is some rather obscure coding here. For example, instead of finding code such as:

```
ALPHA         BAL      15,BETA
               .
               .
               .
               B   ALPHA
```

you will find:

```
ALPHA         BAL       15, BETA
              .
              .
              .
              .
              B    BETA
```

if register 15 still contains the return address of ALPHA+4.

8.4 Description of the Initialization Phase

Lines 1 through 527 constitute the initialization phase of the program. In it, the alternate DD lists are moved in if available, the PARM field is examined to determine what options to perform, the files are opened, and core is GETMAINed.

Lines	Function
9 - 24	Set up save areas and base registers. Registers 11 and 12 are the base registers throughout the program, and R13 points to the OS save area for I/O calls.
25 - 50	Check for PARM field and modify some instructions, depending on the desired options.
51 - 88	Check for alternate DD list, modify DD names accordingly.
90 - 92	Check if PDS member name supplied by user. If so, skip code that opens PDS as sequential data set.
94 - 133	Open PDS directory, and pick off the first name in it. Use that name as input member name.
134 - 171	Open the other files. Check for successful open. If any unsuccessful, return to caller, passing back condition code HEX'6C' indicating reason for abort.
173 - 457	Save areas, DCBs and some small data areas.
459 - 473	Try to obtain space for tables. First try to get 32-64K, but if unable, halve the request. If at the end of the fourth try (4-8K), give up and return to caller.

Lines	Function
476 - 514	Attempt to open a DCB with DD name of LINKIN. If able to, copy all the records on it to the DCB with DD name of STACKOBJ . Afterwards, close LINKIN DCB and free up buffer space.
515 - 527	Compute the maximum length of each of the tables to be built by the other phases of the program.

8.5 Table of CSECT Version Numbers

The code between lines 529 and 719 is designed to make sure that the version numbers of the various compilation units in the load module are the same as the numbers were when the units were compiled. The version numbers are passed to the link editor on SYM cards, and are retained in the load module because the 'TEST' option is automatically passed to the first link step by HALLINK (except when the SDL option is used).

The HAL compiler provided SYM cards are coded in a special way to prevent other language translator's SYM information from interfering with the checking process by providing extraneous information which other translators will not use.

The HAL compiler emits version information by specifying the CSECT names of the compilation units on the SYM cards with addresses corresponding to the version numbers. This information is sandwiched between two invalid control section names (HALS/S at the front, HALS/E at the end of the version information). The program waits for the HALS/S CSECT appearing on a SYM card before it attempts to process the information contained on the card. The HALS/S must appear as the first byte of information in the SYM card. The version information is then extracted until the HALS/E delimiter is detected.

The first CSECT name encountered after the HALS/S defines the version of the compilation unit, whereas all those following it (if any) until the HALS/E are the versions of the compilation units it references.

The program builds a table (described by the DSECT SYMCELL on lines 1691 - 1695) and processes the data after the end of the SYM cards. (All symbol information appears before the CESD records in a load module.) The table resides in GETMAINed core. As entries are added to it, a check is made to ensure that the size of the table does not exceed the storage available.

8.5.1 Usage of Variables in the Table

SN Control Section name.

FP (Father Pointer.)
Byte 0: Indicates whether the entry defines a version number (def node), a reference to a version (ref node), or is a dummy entry placed there because a ref was made to a CSECT which did not yet have a def entry (undef node).
Bytes 1-3: If byte 0 is ref, bytes 1-3 contain the address of the entry which referred to it. If byte 0 is not a ref, bytes 1-3 are null.

BP (Brother Pointer.)
Byte 0: Contains the version number if ref or def node. Null if undef.
Bytes 1-3: Contains the address of the next entry which is a ref to the CSECT contained in SN in this entry. Null if last or only entry containing this CSECT.

The program will create in this table a def node for each CSECT name at the first entry in which it appears in the table. The BP of this entry will point to the next entry containing a ref node of the same CSECT name. That entry's BP will, in turn, point to the next entry with the same CSECT name, and so on until all entries containing the same CSECT name are linked together. The last entry in the table containing a given name will have a null in BP.

The FP of a ref entry points to the entry containing the CSECT which made the reference to that compilation unit. In the event of a version mis-match, an error message is issued.

8.5.2 An Example of the Construction of the CSECT Version Number Table

Assume that the user linked together four compilation units: A, B, C, D. The version of A was 10, B was 20, C was 30, and D was 40. Assume compilation unit A referenced B and C, compilation unit C referenced B and D, while B and D did not refer to any other compilation units.

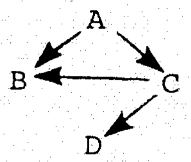


Figure 8-1

Assume these units appeared in the order A, C, D, B.

HALLKED considers each CSECT in the order of its appearance, and will perform as follows in constructing and modifying the entries in the table.

The first CSECT to appear is A. Since it is not referenced by another program, it is labelled DEF in the first byte of FP. The version number, 10, is stored in the first byte of BP. The rest of BP is null because there are no ref nodes referring to A.

The next two entries are for CSECT B. The first of these two is a dummy entry into which a DEF entry will eventually be placed. In byte 0 of FP it has undef, and in bytes 1-3 of BP it has the address of entry three in the table, where B also occurs as a ref. Entry three has the address of the referring program, in this case A, the version number in byte 0 of BP, and nothing in bytes 1-3 of BP. This space will be filled when further ref entries for B are added to the table.

Entries 4 and 5 are similarly constructed. At the end of the first five entries the table will contain these values:

Entry #	SN	FB 0	bytes 1-3	BP 0	bytes 1-3
1	A	DEF		10	
2	B	UNDEF			3
3	B	REF	1	20	
4	C	UNDEF			5
5	C	REF	1	30	

Figure 8-2

The program next considers compilation unit C. FP byte 0 of entry four is changed to a Def. (Having the dummy in this byte has insured that the first occurrence of C in the table will be the father pointed to by those units referenced by C.) Entry six contains data pertaining to B as referenced by C. Byte 0 of FP is Ref., bytes 1 through 3 point to entry four, byte 0 of BP is the version number, in this case, twenty, and bytes 1 through 3 of BP contain a null. At the same time, bytes 1 through 3 of BP of entry three are changed to contain the address of the next reference to B, or six. Entries seven and eight are constructed in a manner similar to the construction of entries two and three.

As the program finally reaches D and B, it changes the dummy first occurrences of those units in the table to definitions. The final appearance of the table is as follows:

		FP		BP	
Entry #	SN	Byte 0	1-3	Byte 0	1-3
1	A	Def		10	0
2	B	Def		20	3
3	B	Ref	1	20	6
4	C	Def		30	5
5	C	Ref	1	30	0
6	B	Ref	4	20	0
7	D	Def		40	8
8	D	Ref	4	40	0

Figure 8-3

The search of the table for discrepancies is straightforward. The program looks for each def node, and follows the pointers contained in column BP to find all ref nodes to the same CSECT. The version numbers of def and ref nodes are compared. When the BP of a ref node is null, the program seeks the next def node. The occurrence of a node which has been referenced but not defined causes an error.

8.5.3 Version Number Cross Referencing

Lines	Function
530 - 535	Read next load module record, check if SYM record. If CESD, save address pointer and drop through. **NOTE** Watch for line 534, I warned you about it before.
537 - 581	Last stage of version processing, verify that all defs and refs are the same.
537 - 544	Look for def nodes.
545 - 550	Check all ref nodes for same version number.
551 - 569	Print error message for version mis-match.
570 - 576	Print error for undef node.
577 - 581	Get address of next entry, skip to CESD if no more.
583 - 590	Check for SYM card image on record, ignore if not SYM. If no more images on this record, read next.
592 - 593	Determine whether the node will be def or ref. Search flag on in variable SYMS if looking for def.
594 - 599	See if HALS/S on card. If not, ignore it.

Lines	Function
604 - 617	Move data from card to internal storage. Check if HALS/S on card, and if so, reset to def mode.
618 - 621	Check for more information on card.
623 - 626	Check if def/ref expected.
627 - 630	Turn off search def flag to indicate ref mode.
631 - 634	Look for entry with same CSECT name.
635 - 648	No such entry, create def node.
650 - 653	Ascertain whether def/ref/undef entry found.
654 - 656	Undef node changed to def.
657 - 658	Def node encountered. Check if two versions are same. (This should not happen, since there should not be two defs for the same CSECT name. Indicates that the compilation units' names not unique in the first 6 characters.)
659 - 674	Print error message about conflicting def versions for the same CSECT name.
676 - 693	Create ref entry.
694 - 697	Search for an entry with same CSECT name.
698 - 709	None found. Create undef entry.
711 - 719	Link entry into chain of refs.

8.5.4 Composite External Symbol Dictionary and Relocation Dictionary

The Composite External Symbol Directory and the Relocation Dictionary are constructed by HALLKED for the purpose of determining the maximum size of stack which will be necessary at any one time in the running of the program being link-edited. To do this, it is first necessary to compute the maximum stack size required by each CSECT, and, in a series of passes, to add to each CSECT stack size the maximum stack size which can be required at any one time by all the CSECTS which it calls. Using the example before, where A calls B and C, and C calls B and D, the stack size required by C will be the sum of its own stack size and the maximum of the stack sizes required by B and D. The stack size required by A will be the sum of its own stack size and the maximum of the stack sizes required by B and C.

HALLKED must construct a dictionary to tell who calls whom. This is the RLD. Each CSECT has one entry in the CESD. If that CSECT calls any other CSECTs, there will be a pointer in its CESD entry to an RLD entry. At that RLD entry, there will be two pointers, one pointing to the CESD entry for the routine which has been called by the original CSECT, and one pointer pointing to the RLD entry which points to the next subroutine called by the original CSECT.

The tables, constructed for the case in Figure 8-1, appear as follows:

Entry No.	CSECT Name	Address of First Byte	Pointer into RLD Table	Unused	Length of CSECT	Indicators
Bytes	8/	12/	14/	16/	18/	20/
1	A		1			
2	C		3			
3	D		-			
4	B		-			

RLD Table

	Pointer Back to CESD Table	/	Pointer to Next Entry in RLD Table
1	4		2
2	2		-
3	4		4
4	3		-

The partial stack size of each routine -- that is to say, the stack size required by each routine alone, and not including the stack size required by any routine it may call -- is supplied by the compiler or assembler. On subsequent passes by HALLKED, the partial stack sizes of the innermost routines (innermost in terms of level of nesting) are added to the partial stack sizes of those CSECTS which reference them to obtain a new partial stack size for the next outer layer of subroutines. This process continues until all stack sizes are either "complete" or until recursion is found.

Description of the individual lines of program follows.

Lines	Function
720-729	Set up registers.
731-739	Determine if ESDID number is out of range indicating lack of core. If so, ABEND. Get address of core buffer for control section information.
741-752	Construct entry in CESD. Determine if it is null, SD (Segment Definition), or LD (entry point other than beginning of routine). If LD, then move the ESDID number of the SD containing it to the length field.
754-766	Move name to table. If CSECT name is HALSTART, ignore NOHALSTART option. Determine if CSECT is HAL program, task, or stack.
768-770	Process all the CESD entries on the record.
772-775	Read the next record. If it is a CESD record, repeat process in lines 731-770, otherwise, continue processing the remaining types of records.

8.5.5 External Reference Table

This section sets up external reference tables and HAL procedure NAME tables (if XREF has been specified by the user).

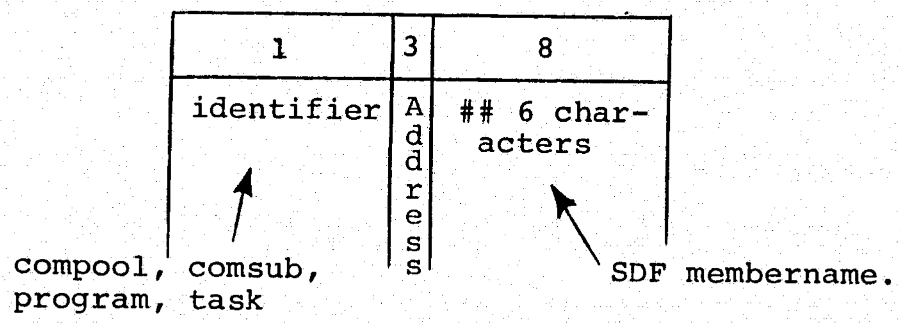
Lines	Function
777-792	Compute addresses of the other tables; RLD table, and (optionally) the control section's HAL-name.
794-795	ABEND if there is insufficient space for auxiliary tables.
797-815	If a stack control section has been read in, or if the first byte of the control section was not on record, do not examine contents of the text record.
835-860	Check the first few bytes of the text of the control sections to see if each was produced by HAL compiler, or is a library member.
861-862	Round stack size to next higher double word (making sure stack size is multiple of eight bytes).
863-875	If control section is a HAL program's internal procedure, indicate this in its CESD table entry.
877-880	If XREF was specified as a HALLINK parameter, move the actual user name for the procedure to a table.
881-887	Indicates in CESD table if this is HAL program, comsub, or task. If entry is a library member, it indicates this.
890	If control section is not a HAL-type section, exit.
892-895	Repeat lines 731-890 for next CESD item on text record.

Lines	Function
897-898	If End of Module switch is set, skip to the next phase of processing (line 1022).
899-917	Read the next record from the load module, determine its type, and branch to appropriate processing routine.
919-930	If Control Relocation Dictionary has been read in, move control information to control buffer, and move relocation information to relocation buffer.
932-938	If Control record has been read in, set up control buffer.
940-945	If RLD record has been read in, set up RLD buffer.
947-948	Set up registers for RLD processing.
949-950	If POS and REF flags were omitted from this entry, use previous entry's POS and REF values.
951-958	Pick up previous POS and REF if they were omitted from this item.
959-967	Pick POS and REF flags from record.
968-975	If the entry is not of consequence, like a null CSECT, ignore it.
975-978	If the current entry has already been linked to its calling entry, do not link it again.
989-997	If a new RLD entry has to be added, the last entry in the table with the same name has a pointer at the current entry appended to it.
998-1012	If the current entry has POS and REF, save POS and REF flag fields.
1013-1012	Resolve entry points into the middle of CSECTS into references to those entries themselves.

Lines	Function
1022-1070	In this section, the stack size required by the program is computed (see description in Section 8.5.4). For the purposes of this description, A is a CSECT which calls B. We start out processing A.
1022-1024	Set up a switch which will indicate that no changes have occurred in the table since the last pass.
1026-1027	If either the complete stack size has already been computed, or should not be computed, to to 1069.
1028-1031	If the CSECT being examined calls another routine (B), go to 1036.
1032-1034	If not, set a bit saying his stack size is valid ("complete").
1036-1040	Find the entry in the RLD table to which the CSECT entry currently under consideration points.
1042-1048	If the called routine is entered from a point other than the beginning of the called routine, reset the register to point to the encompassing SD.
1050-1051	If the routine "B" called by A is a stack or non-HAL, go to 1061 to see if A refers to anybody else.
1052-1054	If stack size of B has not been computed, set indicator for "uncompleted".
1057-1059	If stack size computation has been completed, test it against the current max of routines referenced by A. If it is greater than the current max, replace that number with B's completed stack size.
1061-1062	Determine what other entries are referenced by A.
1064-1067	If A calls nobody else, add current move to CESD value for entry now being processed. Drop through to 1069.

Lines	Function
1069-1070	Points to next entry in CESD table.
1072-1074	If there are no uncomputed stack sizes, go to 1128.
1075-1078	If there are uncomputed stack sizes, and there has been a change in the table on the last pass, make another pass.
1078-1125	If there are unresolved references and no change in the table on the most recent pass, there is recursion. These lines determine where the recursion has occurred and send a message to the user.
1078-1079	Sets up registers.
1081-1082	If stack size is computed, go to 1124.
1083-1084	If this is not a main program go to 1124.
1086-1090	Print message, and this main program has some recursion.
1092-1104	At first node which has no stack size computed for it, start looking for recursion.
1106-1114	Find which of routines which A calls is the one whose stack size is uncomputed. Continue following the pointer.
1116-1118	If the flag which indicates this spot has been visited before it is set, go to 1094 to print out a message which CSECTs are recursing.
1124-1125	Go to next entry in CESD, in case there is more than one recursion .
1125-1158	If TREE has been specified, print out which routines call which other ones.
1160-1179	Compute max stack size for PROGINT and TIMEINT.
1181-1194	Unless NOHALSTART was specified, punch out: INCLUDE SYSLIB(HALSTART).

Lines	Function
1195-1275	Unless NOHALMAP was specified, this section produces a control section called HALMAP which points to every program, task, and Simulator Data File member name.
1195-1201	Produce card to LKED which provides HALMAP CSECT name.
1202-1207	Set up registers.
1208-1221	Determine whether entry in CESD is program, task, compool, or comsub.
1222-1230	Create card pointing at it and identifying it as one of the above.
1231-1241	Output text card of form:



1242-1254	The second thru fourth bytes is a V-type address constant.
1255-1256	Go through each entry in CESD table to see if it is a program, task, or compool. (Go to 1208).
1257-1268	Place in the first four locations of the control section of HALMAP, a count of how many entries in the table there will be. CSECTs for the stack puts out control sections which will be stack for each program and task.
1277-1281	Set up registers.

Lines	Function
1283-1285	Determine if entry has stack associated with it (i.e. it is a PROGRAM or TASK). If there is no stack, go to 1303.
1286-1301	Put out card specifying to the link editor how big the stack is after adding in PROGINT and TIMEINT stack sizes.
1303-1304	Repeat 1283-1301, looking for valid stack candidates.
1308-1313	Put out card saying END.
1315-1354	INCLUDE TEMPLOAD(TEMPNAME). If membername of load module is TEMPNAME, put out no card, if name is not TEMPNAME, put out a card which has NAME user-specified-name(R) on it.
1356-1364	If TREE is in effect, print out max of stack size of PROGINT and TIMEINT.
1366-1389	If XREF has been specified, print out user-supplied HAL names and corresponding CSECT name.
1391-1406	Determine return code to pass back to HALLINK. 0-OK 4-programmer allowed recursion. ≥100 is fatal error.
1408-1470	Gives OS back all its space. Close off files.
1471-1476	Return to calling program.
Internal Subroutines	
1478-1488	Print out name and length of stack in hex on left side of page, return.
1490-1507	Print out up to nine subroutines on the right hand side of the page.
1509-1534	Subroutine from OS for going to next line on printer or skipping to top of next page.
1535-1559	Reads in load module.

BASES	2A	Base addresses.
OSSAVE	18A	Register Save Area.
FINDNAME	D	PDS member name.
SYSLINBL	A	Length of PDS directory buffer.
SYSLINBA	A	Address.
DOUBLE	7D	Register Save Area.
SYMNA	A	Address of next SYM record.
SYMCA	A	Address of current SYM record.
RCODE	F	Return code.
PADDR	A	Address of print buffer.
ARLD	A	Address of RLD table.
ABUFF	2A	Address and size of GETMAINed core.
ACHARS	A	Address of buffer containing the programmer supplied procedure names.
OLDRP	A	Provides REF/POS flags.
SIZES	2A	Size of region to be requested.
CCW	A	Address of portion of CCW.
PAGECT	A	Number of pages printed.
LINECT	A	Number of lines printed on current page.
MAXESD	H	Largest ESDID encountered.
SYMCOUNT	H	Number of SYM table entries.
SYMMAX	H	Number of SYM table entries which can fit in core.
#CTL	H	Number of bytes of CESD information.
#RLD	H	Number of bytes of RLD information.

NEXTRLD	H	Index of next available entry in RLD table.
ADDED	H	Stack size for recursive programs.
#TIME	H	Index into CESD table of TIMEINT.
#PROGINT	H	Ditto for PROGINT.
MAXESDID	H	Largest ESDID which can be placed in CESD table.
MAXRLD	H	Largest RLD which can be placed in RLD table.
S	X	Some general switches.
FLAGS	X	Flag field of RLD item.
BLANKS	8C	Character string blanks.
CTLTABLE	236X	Control data from control or control/RLD record.
RLDTABLE	236X	RLD data from RLD or control/RLD record.
TRCHAR	16C	Translates unpacked decimal number to PRINTABLE character.
SYSLIB	8C	Name of library.
ESDCARD	80C	ESD card to link editor.
INCLUDE	80C	INCLUDE card to link editor.
MEMBER	8C	PDS member name.
HEADER	31C	ESD card defining HALMAP.
HALESD	13C	ESD card.
HALTXT	17C	TXT card.
HALRLD	21C	RLD card.
REGCMSG	C	Error message.

HEADING	C	Heading when "TREE" was specified.
XREFH	C	Heading when "XREF" specified.
VALIDCHR	256C	Valid HAL names for procedures.
O2SYM	4C	Start of SYM card.
SYMSTART	C	Indicator that SYM record produced by HAL compiler.
SYMS	X	General switches.
SYMBUFFV	X	Version Number.
SYMBUFFN	8C	Control section name.
SYME1		Error Messages.
SYME2		Error Messages.
SYMV1		Error Messages.
SYMV2		Error Messages.
SYMDUP		Error Messages.
SYMDUPV1		Error Messages.
SYMDUPV2		Error Messages.
PATCH	10D	Area in which to patch code.

9.0 THE HAL/S SUBMONITOR

The HAL/S submonitor is an augmented version of the standard XPL submonitor. Its primary function is to act as an interface between the compiler and OS/360. The requirements of the HAL/S compiler for interacting with OS/360 take several forms:

1) Loading the various phases of the compiler into core and placing them into execution.

2) Sequential string input and output

3) Direct access input and output

4) Obtaining external information(e.g., DATE)

5) Obtaining information common to the phases.

6) Performing compile-time computations (e.g., SINE)

In addition to this compiler support function, capabilities are built into the submonitor to provide such interface support for the HALSTAT program and dynamic invocation of the compiler.

This section describes how these requirements are met by the HAL/S submonitor. Familiarity with IBM OS/360, job control language, and OS/360 assembler language is assumed in this discussion.

9.1 Compiler Execution

The HAL/S submonitor is the program which is initially loaded into core or called and given control. The submonitor then proceeds to:

1) process any dynamic invocation parameters.

2) process any user specified options for the HAL/S submonitor.

3) setup for parallel file accessing.

4) setup for interrupt handling.

5) setup for compiler timing.

6) open initially needed files.

7) obtain space in memory for Phase I.

8) load Phase I into core.

9) give Phase I control.

Steps 7, 8, and 9 are repeated for each succeeding phase of the compiler with the exception that an additional linking process occurs between each phase. When all phases are complete, control returns to the submonitor where cleanup is performed and control given back to OS/360.

During the execution of any phase of the compiler, the submonitor may be called upon to provide one of the services described in Section 9.0; thus, the submonitor serves in two distinct modes:

-as a caller (overseer) to the HAL/S compiler

-as a co-routine to the compiler

A map of the submonitor as it might reside in core along with flow of program control is provided in Figure 9.1.1. Each of the modes of the submonitor will be discussed separately.

9.2 As an Overseer

The basic functions of the submonitor as an overseer were listed in Section 9.1. Each of these functions will be discussed in turn.

9.2.1 Processing Dynamic Invocation Parameters

OS/360 provides a facility through which DDNAME overrides may be passed to a program to be run. When dynamically invoked, the submonitor may in fact be provided with such an override list. (See the HAL/SDL ICD, Section 2.2.1.1.1 for a description of the override conventions). In addition, a field may be provided in which the name of the control section generated by the compiler may be returned. The submonitor searches through the alternate DDNAME list and moves the override DDNAME for any file into its corresponding area in the submonitor's DCB data area. The CSECT name option, if it exists, is saved for later use by the compiler when returning the desired CSECT name.

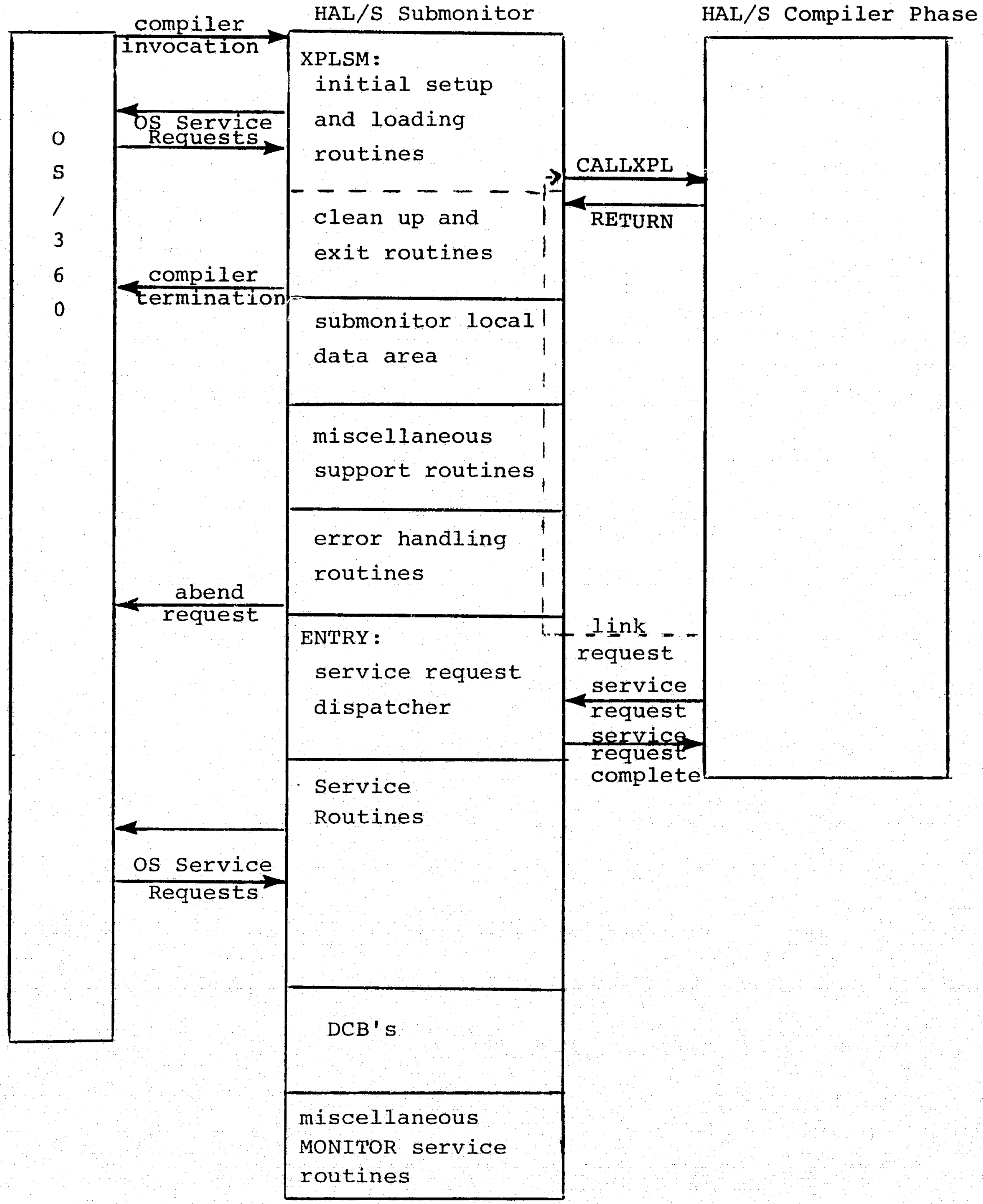


Figure 9.1.1

Compiler Execution

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

9.2.2 Processing of User Specified Options

Upon completing processing of any dynamic invocation parameters, the submonitor proceeds to load the 'MONOPT' options processor and call it.

When a HAL/S system options processor is called, the result is a pointer (OPTADDR) to an option table which describes the values of all Type 1 and Type 2 options. An example of the option table and its associated data is illustrated in Figure 9.2.1. The options processor returns a pointer to a six word list. The first word in the list (options) is the flag field correspinding to the values (default or specified) of the Type 1 options. The fullwords from OPTIONS+4 to OPTIONS+20 contain pointers to further lists. These lists are described below.

CON (referenced via OPTIONS+4) - A series of XPL descriptors which point to character data. The character data show the value of an option as it is currently in effect. Thus, if NODUMP had been specified or defaulted, a descriptor pointing to the characters 'NODUMP' would exist. If DUMP had been ON, then a descriptor pointing to the characters 'DUMP' would exist. A zero descriptor indicates the end of the list.

PRO (referenced via OPTIONS+8) - A series of XPL descriptors which point to character data. The character data correspond to the order of the options described by the CON descriptors. The characters show the state of the option NOT in effect. Thus, if DUMP had been ON, a descriptor in CON would point to 'DUMP' and a descriptor at the corresponding point in PRO would point to 'NODUMP'. A zero descriptor indicates the end of the list.

DESC (referenced via OPTIONS+12) - A series of XPL descriptors which point to character forms of the Type 2 options. The list is terminated by a zero descriptor.

VALS (referenced via OPTIONS+16) - A series of fullwords which contain the value of the corresponding Type 2 option in the DESC table. Thus, if PAGES=1000 had been coded, a descriptor in DESC

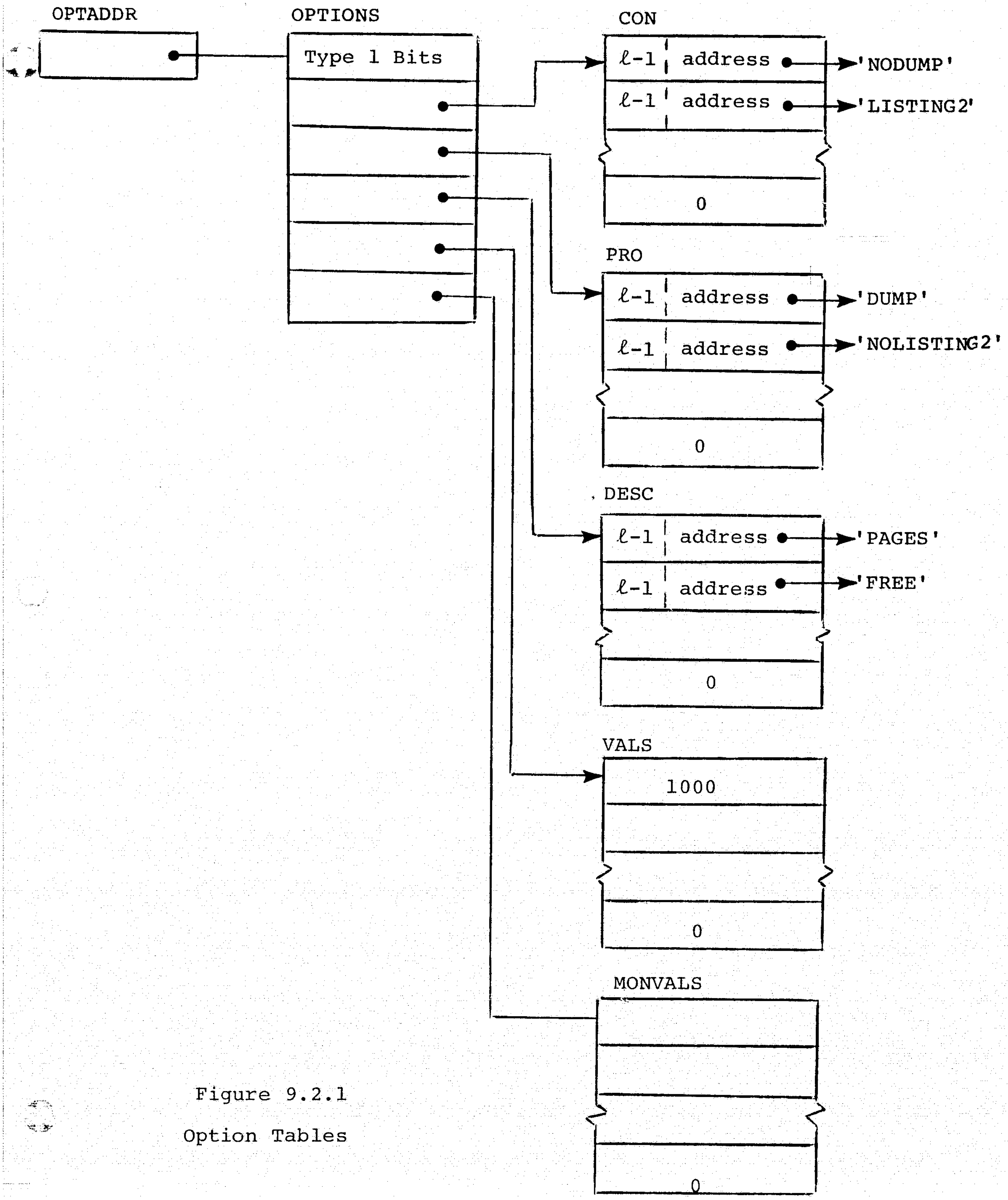


Figure 9.2.1

Option Tables

would point to 'PAGES' and the corresponding entry in VALS would contain the value 1000. Some entries in VALS may be descriptors if the value of the corresponding option is character data (e.g., TITLE).

MONVALS (referenced via OPTIONS+20) - A series of fullwords containing values of options in the same way as VALS. These values correspond to options which are internal to the compiler system and therefore do not have a descriptor allocated in DESC.

Upon return from the call to the MONOPT options processor, the submonitor transfers the information tabularized by the options processor to its local data area.

9.2.3 Parallel File Accessing

The HAL/S compiler requires the capability to simultaneously access the template library in two different manners. The first is as an INCLUDEd input, the second is for template checking purposes. In addition, the compiler requires the capability to reference both the INCLUDE and OUTPUT6 DDNAMEs to find a member. Therefore, the submonitor moves the INCLUDE DDNAME to both INPUT4 and INPUT7 DCB's and copies the DDNAME for the INCLUDE file into INCLNAME and the DDNAME for OUTPUT6 into OUT6NAME for future reference.

9.2.4 Interrupt Handling

The submonitor traps floating point overflow and underflow which might occur during floating point compile time computations. It returns the maximum positive floating point number for an overflow and floating zero for an underflow. These interrupts are trapped by issuing a SPIE macro for interrupts 12 and 13.

9.2.5 Compiler Timing

The submonitor issues a task STIMER macro with an interval of one hour. The resultant timer may be accessed by a subsequent MONITOR call.

9.2.6 Opening Initially Needed Files

The submonitor initially opens the files INPUT0(SYSIN), OUTPUT0(SYSPRINT), PROGRAM(compiler object code), and INPUT5(ERROR). If the LISTING2 option has been requested (known via the options processor call), the LISTING2 file (OUTPUT2) is opened. If any of the OPENs on OUTPUT2, PROGRAM, or OUTPUT0 fails, a 100 abend is forced.

9.2.7 Initial Compiler Phase Execution

The loading of Phase I of the compiler is performed in much the same manner that is explained by McKeeman et. al. for the standard XPL submonitor; however, the HAL/S compiler requires that certain common information be retained in core for passing of data between phases. The resulting layout of a phase of the compiler in core memory is shown in Figure 9.2.2. It differs from the standard XPL layouts in that a COMMON area exists which remains in core between phases (as does the submonitor). This COMMON area must be the same for each phase which references it. The length of the COMMON area may be zero. It also differs from standard XPL layout in that the local descriptor area appears before the code area instead of following it.

The submonitor has been modified to obtain and initialize this COMMON area from the XPL object code for the compiler. From the compiler's object code, the submonitor obtains information about the size of COMMON and entities called common array initialization pairs. These pairs are two fullwords, the first of which is an offset, the second of which is a pointer value. For Phase I, COMMON is initially zeroed, then for each initialization pair, the pointer value is stored at the relative offset from the start of COMMON.

9.2.8 The Linking Process

In addition to this COMMON area, a phase of the compiler may generate certain strings which should be passed to the next phase. These strings reside in the free string area and their descriptors in a GETMAINed area of core. These strings must be retained during the process of loading the next phase. This is impossible to do with the standard XPL submonitor and the HAL/S submonitor has been upgraded to provide this service, henceforward referred to as linking.

An area in COMMON known as descriptor-descriptor (DESCDESC) contains the information necessary for passing of the COMMON strings. A layout of DESCDESC and its associated

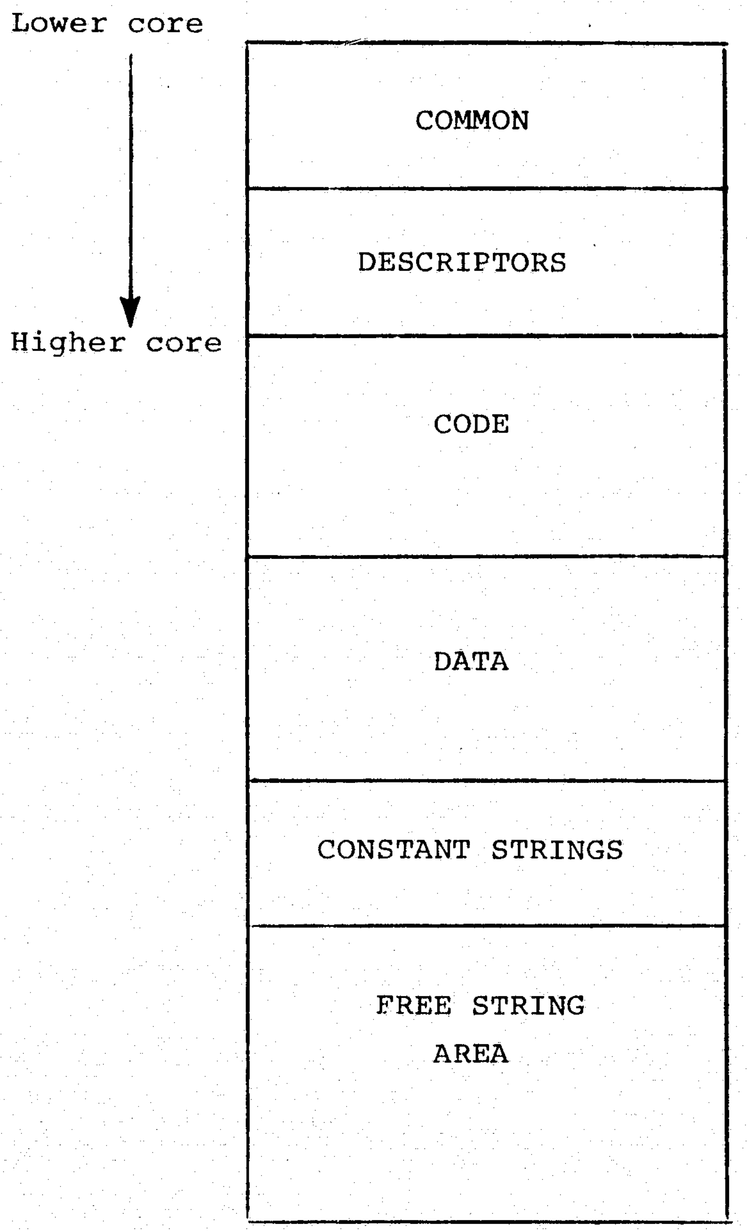


Figure 9.2.2
HAL/S Compiler Phase in Core

data is shown in Figure 9.2.3.

When a phase is done processing, it sets the second element of DESCDESC to zero, indicating that no local strings exist and calls the XPL COMPACTIFY routine (provided by the XPL compiler with each phase). The result of this call is a compressed set of string data in the free string area. A submonitor service request to link is then issued. This request has as its parameters

-the address of DESCDESC

-the start of the COMMON strings

-the top of core

(top of core is passed as a parameter since this may change as a result of compiler dynamic allocation of buffers, etc.).

The submonitor then proceeds to move the COMMON strings to the top of core. Loading of the next phase is done as with Phase I and the COMMON strings are moved back to the start of the free string area. This three step process is illustrated in Figure 9.2.4. The offset between the previous location of the COMMON strings and their new location is computed and this offset added to each of the descriptors. The result is that through DESCDESC the newly loaded phase may access the COMMON strings produced by the previous phase in its own free string area.

9.2.9 Returning to OS

When control is finally returned to the submonitor after completion of all the phases, the submonitor saves the return code of the XPL program, gives memory back to OS, deletes the current options processor, closes all files, restores the old interrupt exit routine and returns to OS.

9.3 As a Co-routine

There is an ENTRY entry in the submonitor which is called for various services requested by the HAL/S compiler during the execution of a particular phase. Various XPL constructs are recognized as being calls to this entry with a specified service code dependent upon the XPL construct. Table 9.3.1 gives a list of the service codes, their interpretation, and an example of the XPL construct which invokes the service routine. On the basis of the request code, the submonitor branches to a subroutine which performs

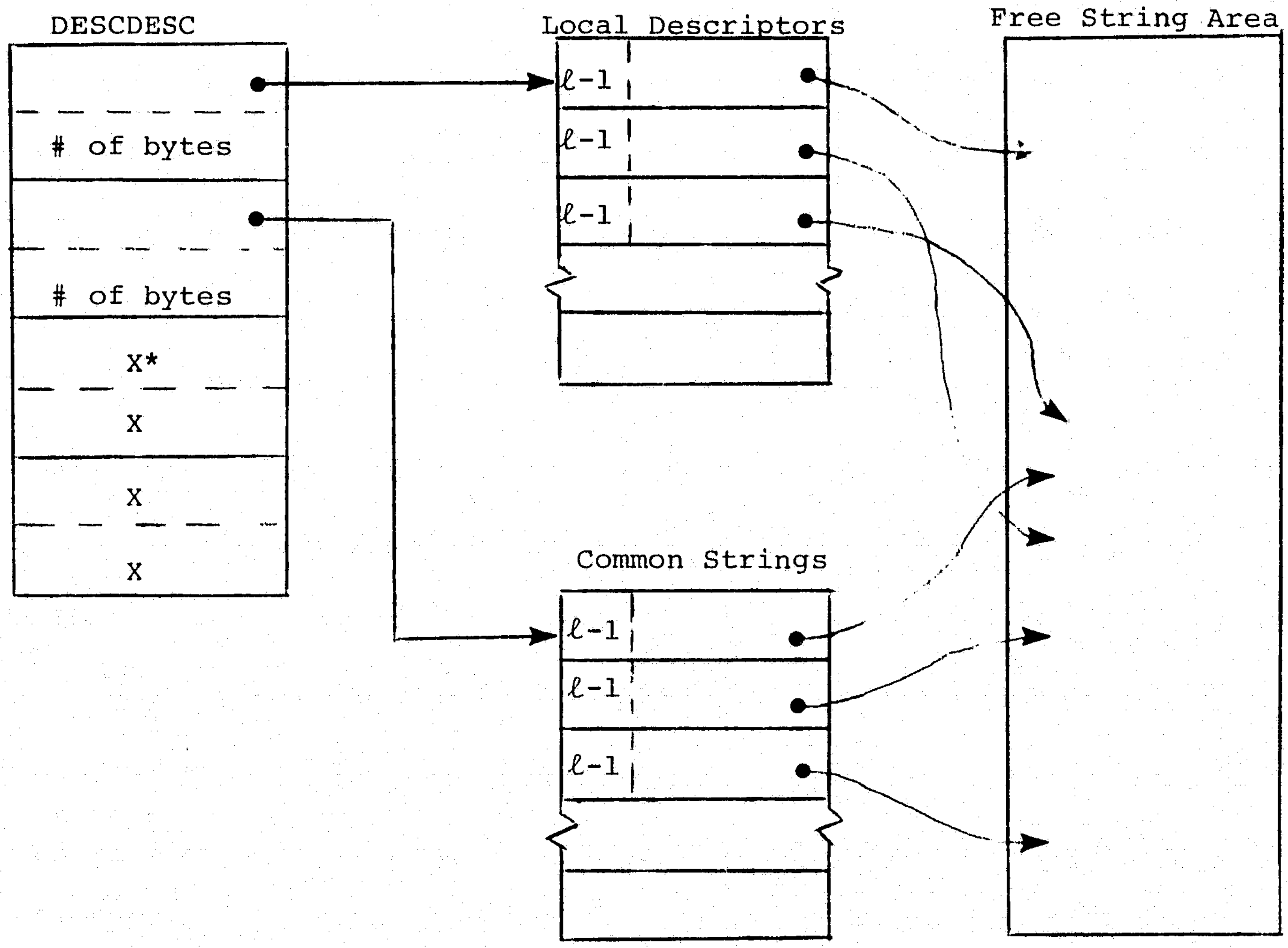
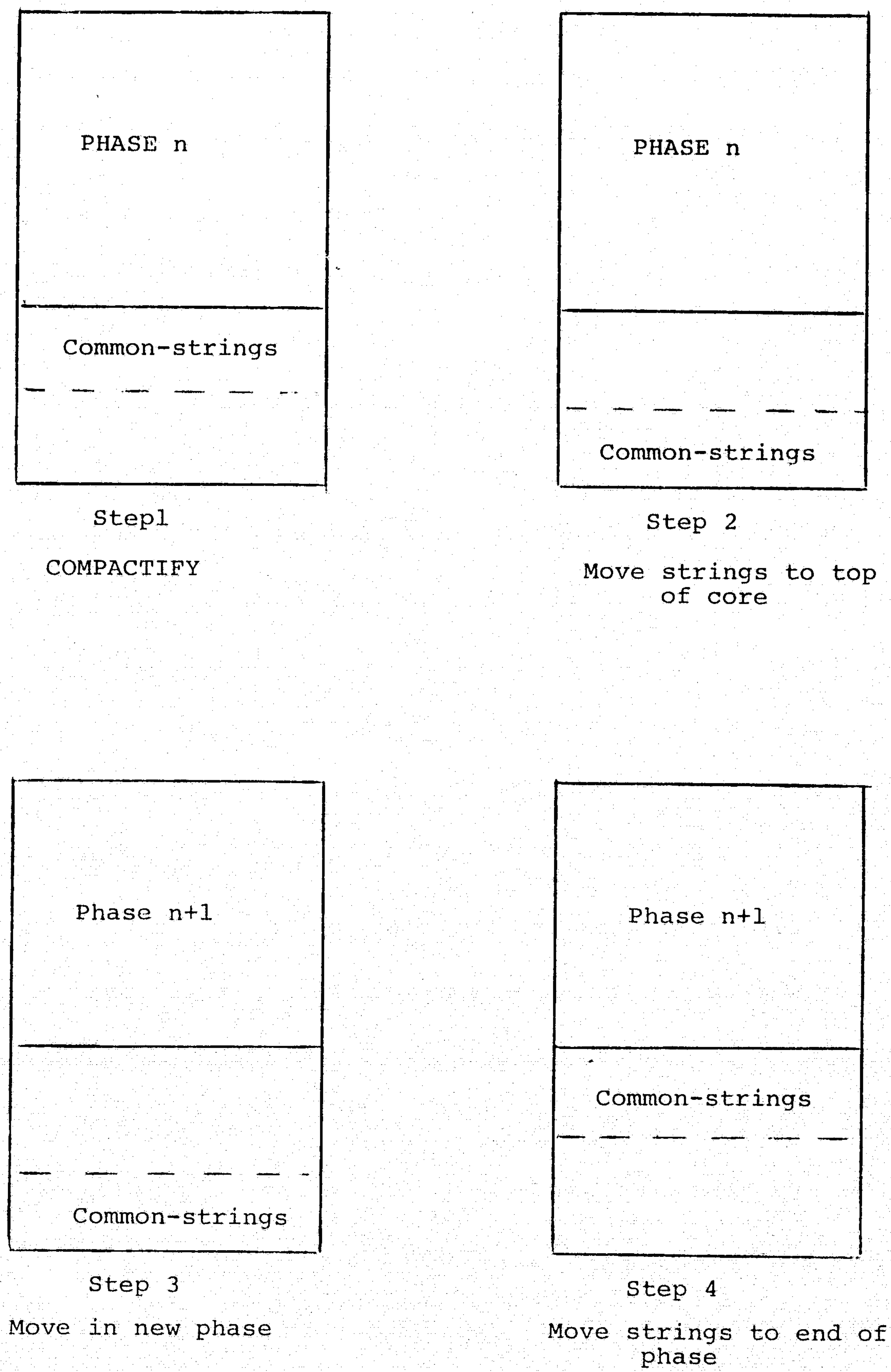


Figure 9.2.3

Descriptor-Descriptor Layout

* For purposes of the HAL/S compiler, the last 4 entries of DESCDESC are unused.



Stepl

COMPACTIFY

Step 2

Move strings to top of core

Step 3

Move in new phase

Step 4

Move strings to end of phase

Figure 9.2.4

Steps in Interphase Linking

Service Code	Interpretation	XPL Construct
4	Sequential string input	<string>=INPUT(I);
8	Sequential string output	OUTPUT(I)=<string>;
12	Return line count	<variable>=LINECOUNT;
16	Set line count limit	CALL SET_LINE_LIM(<value>);
20	Force immediate exit	CALL EXIT;

24	Return time & date	<variable>= TIME;
		<variable>= DATE;

28	Unused	
32	Link to next phase	CALL LINK;
36	Return parameter field	<string>=PARM_FIELD;
40	MONITOR	CALL MONITOR(<parm>);
44	Unused	
48	Unused	
52	Read from FILE1	<variable>=FILE(1,I);
56	Write to FILE1	FILE(1,I)=<variable>;
60	Read from FILE2	<variable>=FILE(2,I);
64	Write to FILE2	FILE(2,I)=<variable>;
68	Read from FILE3	<variable>=FILE(3,I);
72	Write to FILE3	FILE(3,I)=<variable>;
76	Read from FILE4	<variable>=FILE(4,I);
80	Write to FILE4	FILE(4,I)=<variable>;
84	Read from FILE5	<variable>=FILE(5,I);
86	Write to FILE5	FILE(5,I)=<variable>;
92	Read from FILE6	<variable>=FILE(6,I);
96	Write to FILE6	FILE(6,I)=<variable>;

Table 9.3.1

ENTRY Service Dispatch

the necessary steps to satisfy the request and returns control to the compiler.

Each of these services is now discussed.

9.3.1 Sequential String Input (GET)

The INPUT pseudovariable is used for sequential string input by the HAL/S compiler. It has as its value the string represented by the next record on the input file selected by the subscript of the pseudovariable (INPUT(I), I=1,2,3,4,5,6,7,8). Arguments supplied by the HAL/S compiler to the submonitor for this service are the pointer to the next available byte in the free string area (FREEPOINT) and the index indicating which input file is to be accessed (I in INPUT(I)).

When the submonitor is entered with an INPUT service request, it first determines whether the file number supplied is a valid one. If not, the submonitor forces a 1400 abend. Next the submonitor checks whether the dataset currently associated with the specified file is a partitioned or sequential one.

If the dataset is sequentially organized, the submonitor checks to see if the file has been permanently closed. This condition would occur if the compiler opened the file and subsequently closed the file, e.g. after receiving an end of file indication from the submonitor. If the file is found to be permanently closed, the submonitor forces a 1200 abend. The submonitor then checks to see that the file is in fact open. If not, it attempts to open the file. If the attempt to open the file fails, the submonitor immediately returns an end of file indication to the compiler. Having determined that the file is open, the submonitor issues a locate mode GET macro. This macro returns the address of the next input record. This record is moved to the free string area as indicated by the FREEPOINT pointer passed along with the service call. A string descriptor of the new record along with an updated FREEPOINT is then returned to the HAL/S compiler.

If the dataset organization is a partitioned one, the submonitor first checks to see that the file is in fact open. If not, the submonitor forces a 2100 abend since partitioned input may only be performed after a FIND service request has been issued. FIND leaves the DCB in an open state. The submonitor then checks to see whether the input buffer associated with that file contains any records which have not been processed. If not, the submonitor issues a READ macro and a CHECK macro on the file specified. The next record is then moved to the free string area as indicated by

FREEPOINT and the buffer pointer is updated to indicate that one more record has been processed. A string descriptor to the new record along with an updated FREEPOINT is then returned to the compiler.

9.3.2 Sequential String Output (PUT)

The OUTPUT pseudovariable is used for string output by the HAL/S compiler. A descriptor of the string to be output and an index specifying the output file selected (I in OUTPUT(I)) are passed to the submonitor as arguments.

In order to simplify printed processing, the submonitor adopts some arbitrary conventions. If the file specified is OUTPUT0, the submonitor automatically appends a carriage control character of blank (EBCDIC HEX'40') to the beginning of the string to be output.

If OUTPUT1 is specified, the submonitor assumes that the compiler has supplied a carriage control character as the first character of the string to be output. In addition to the standard FBA type control characters, the characters 'H' and '2' have special meaning to the submonitor. These characters indicate a heading line and a subheading line respectively.

Both OUTPUT0 and OUTPUT1 have associated with them page processing. (They actually refer to the same output file (SYSPRINT) but imply different carriage control processing). This processing includes keeping track of the number of pages which have been output and forcing a 600 abend if the page count limit is exceeded. It also includes keeping track of the number of lines printed so far for a page and issuing a page eject with appropriate heading and subheading lines if any are specified.

On output files two through eight (OUTPUT2, OUTPUT3, . . . OUTPUT8), the submonitor assumes that no carriage control and no page processing is required.

In all cases, the submonitor assumes that any strings less than the record length of the dataset associated with the file are to be padded with blanks to the record length and that any strings of length greater than that record length are to be truncated to that record length and only that remaining part output.

When the submonitor is entered with a sequential string output request, it first checks to see that the file is a valid one. If not, the submonitor forces a 900 abend. If the file is valid, the submonitor then checks to see whether the dataset associated with the file has a partitioned type of organization.

If the dataset organization is partitioned, the submontior checks to see that the file is open. If not, it issues an OPEN macro on the file. If the OPEN macro returns a failure indication, the submonitor forces an 1800 abend. Having determined that the file is open, the submonitor issues a GETBUF macro which returns a buffer address. The buffer is used to accumulate individual lines (logical records) into one physical record (BLKSIZE). Logical records are moved into the buffer for each OUTPUT request, padded or truncated to LRECL as necessary. If the buffer is full, WRITE and CHECK macros are issued and the buffer pointer is set back to the start of the buffer in preparation for re-filling.

If the dataset is sequentially organized, the submonitor first checks that the file has been opened. If not, the submonitor attempts to open the file. If the OPEN attempt fails, the submonitor forces an 800 abend. Having determined that the file is open, the submonitor issues a PUT macro in locate mode. The PUT macro returns the address of the next output buffer. The submonitor moves the string to this output buffer area, performing any necessary manipulations on the string as described by the aforementioned conventions.

9.3.3 Current Line Count

The HAL/S compiler may request the current line count for the page on SYSPRINT (OUTPUT0 and OUTPUT1). The submonitor merely returns the value it currently has in its local data area.

9.3.4 Setting SYSPRINT Lines per Page

When a SET_LINE_LIM call is issued by the compiler, the monitor service routine called stores the value passed into its LINELIM location in its local data area.

9.3.5 Forcing an Immediate Exit

If at any point, the compiler has enough information (or lack thereof) to determine that there is no hope in continuing processing, it may CALL EXIT, which forces a 4000 abend and a dump if the DUMP option was specified and a

SYSUDUMP DD card had been provided.

9.3.6 Obtaining TIME and DATE Information

When a TIME or DATE request is issued by the compiler, the submonitor invokes a binary TIME macro. The time is returned as is. The date, returned by the TIME macro in packed decimal form is converted to binary and returned. The compiler itself saves whichever of the results is desired.

9.3.7 Linking

This service request is described in Section 9.2.8.

9.3.8 PARM Field Accessing

The HAL/S compiler may request from the submonitor the string which is the PARM field received from OS. The submonitor moves the string into the free string area, builds a descriptor to that string and updates FREEPOINT. The new descriptor and new FREEPOINT are returned to the compiler. If no PARM field exists, a null descriptor is returned.

9.3.9 The Monitor Call

The monitor call provided by the XPL language is a means through which the capabilities of the XPL system may be extended without requiring changes to the XPL compiler.

The HAL/S submonitor, upon receiving a monitor service request, essentially invokes a monitor within a monitor. At least one parameter is provided and it is interpreted as the MONITOR service request. The current service codes and their interpretation by the MONITOR call are described in Section 13.3.

9.3.10 Direct Access Input and Output (READ and WRITE)

Direct access input and output is performed by the compiler for work areas used for temporary and intra-phase communication.

When the compiler issues an input request on such a direct access file, the appropriate service request is issued, passing the record number and the address of the memory location into which the record is to be placed. The submonitor first checks to see that the specified file is open. If it is not, the submonitor issues an OPEN macro. If the OPEN is not successful, the submonitor forces a 2000 abend. Having determined that an open file does exist for access, the submonitor checks to see whether the file is on magnetic tape. If not, it forms the TTR address of the record desired. The submonitor then issues a POINT macro, a READ macro, and a CHECK macro on the file. It then returns to the compiler.

When the compiler issues an output request for a direct file, the appropriate service request is issued with the record number and the address of the variable to be output as parameters. The submonitor processing for direct access output is similar to the processing for the direct access input request except that the READ macro is replaced by a WRITE macro.

9.4 OS Accessible Code

There exist pieces of code in the submonitor which are invoked by neither the compiler nor the submonitor but by OS directly.

One of these is an interrupt exit routine for floating point overflow and underflow. These interruots may occur during the process of compile time computations. (See Section 9.2.4)

OS provides for an exit routine to be used in the event of an OPEN on a DCB. The HAL/S submonitor takes care of supplying default values for

1) Block Size

2) Record Length

3) Number of buffers

4) Record Format

when these attributes remain unspecified after the OPEN. There are six types of defaults provided. These are listed in Table 9.4.1.

9.5 Error Handling

DEFAULT	BLKSIZE	LRECL	BUFNO	RECFM
1	1680	80	1	FB
2	3458	133	2	FBA
3	400	80	1	FB
4	1680	1680	0	FB
5	3458	133	1	FBM
6	256	256	1	U

Table 9.4.1

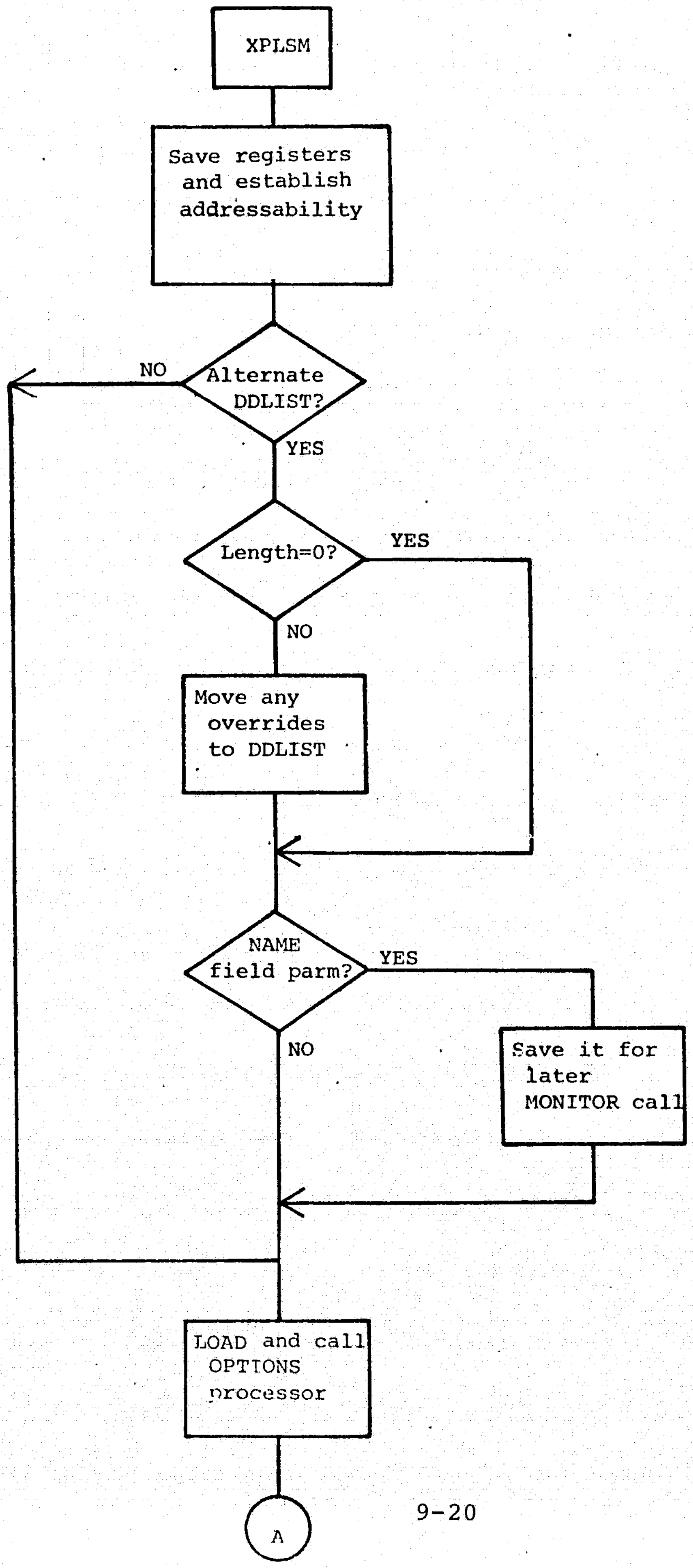
Compiler DCB Defaults

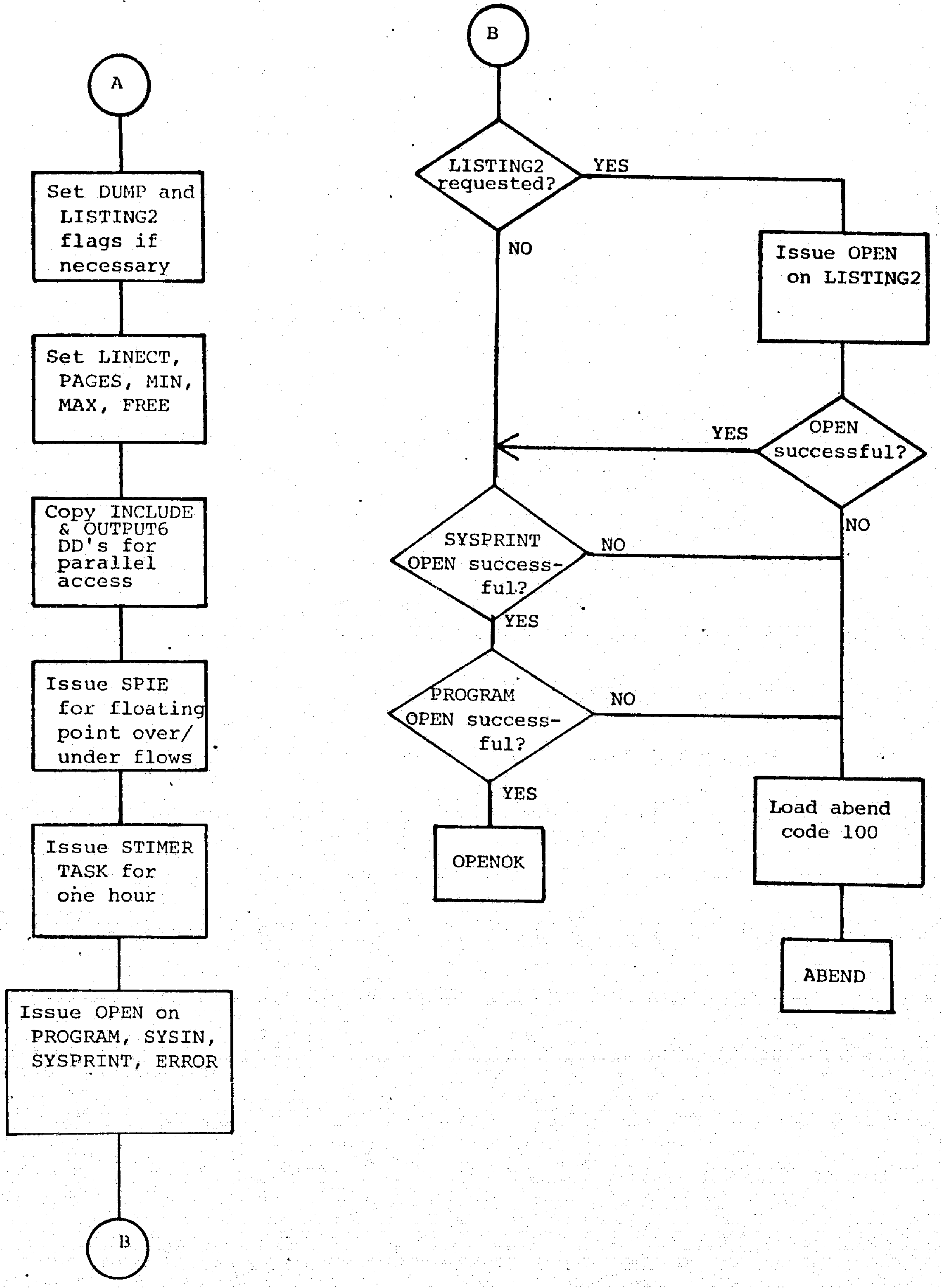
In general, any error condition detected by the submonitor results in abnormal termination of the program through execution of an ABEND macro. A list of abend codes and their interpretation may be found in the HAL/S-360 Users Manual, Appendix F. The abend processing routine saves relevant general registers and attempts to close files before executing the ABEND macro. A dump is performed under the same conditions as described in Section 9.3.5.

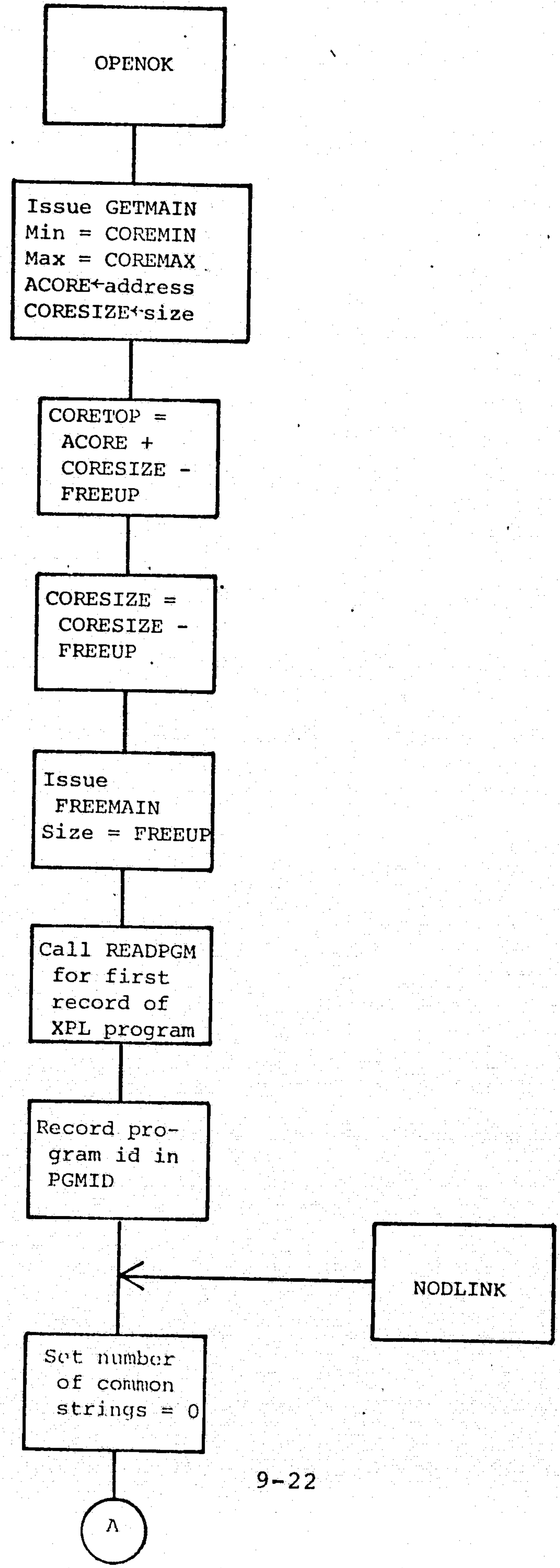
9.6 Flowcharts

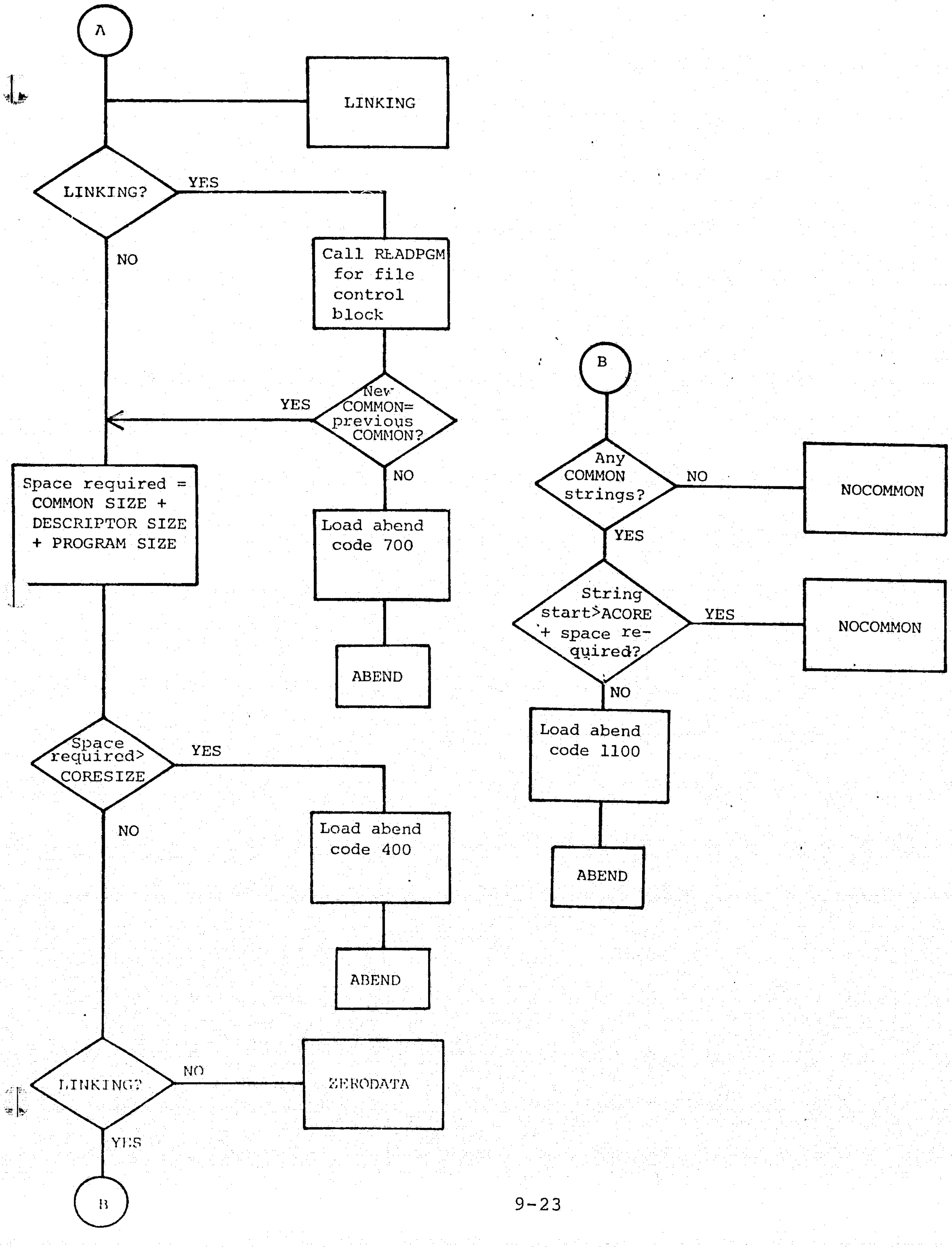
The remainder of this section contains program flow charts describing the operation of the submonitor.

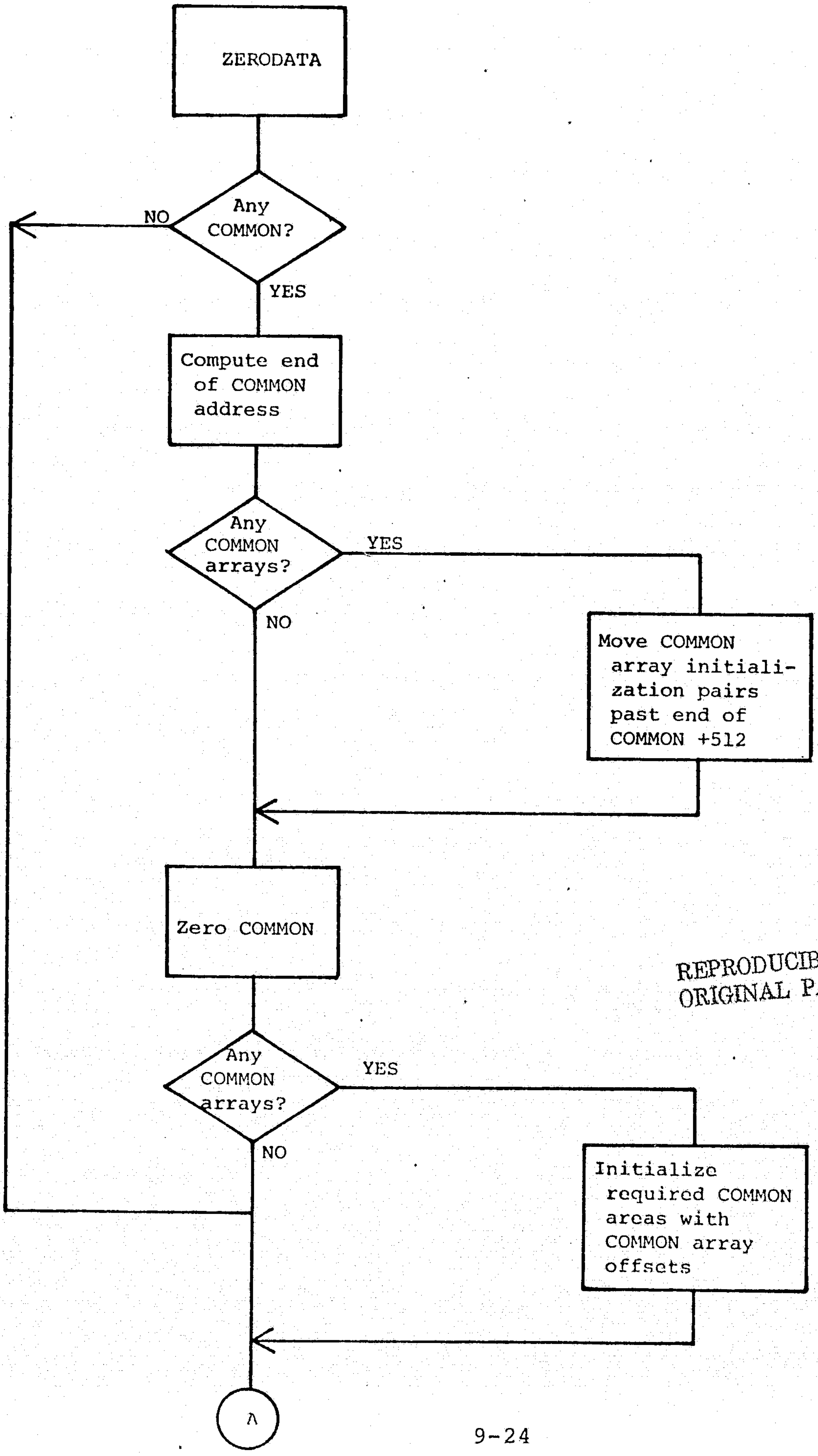
In the flowcharts, a large rectangle represents a processing step and a diamond represents conditional control transfer. A small rectangle represents a location in the code of the submonitor. An arrow into such a rectangle implies transfer of control to that location. An arrow out of such a rectangle denotes the point of definition of that location.


XPLSM
Save registers and establish addressability
Alternate DDLIST?
NO
YES
Length=0?
YES
NO
Move any overrides to DDLIST
NAME field parm?
YES
NO
Save it for later MONITOR call
LOAD and call OPTIONS processor
A

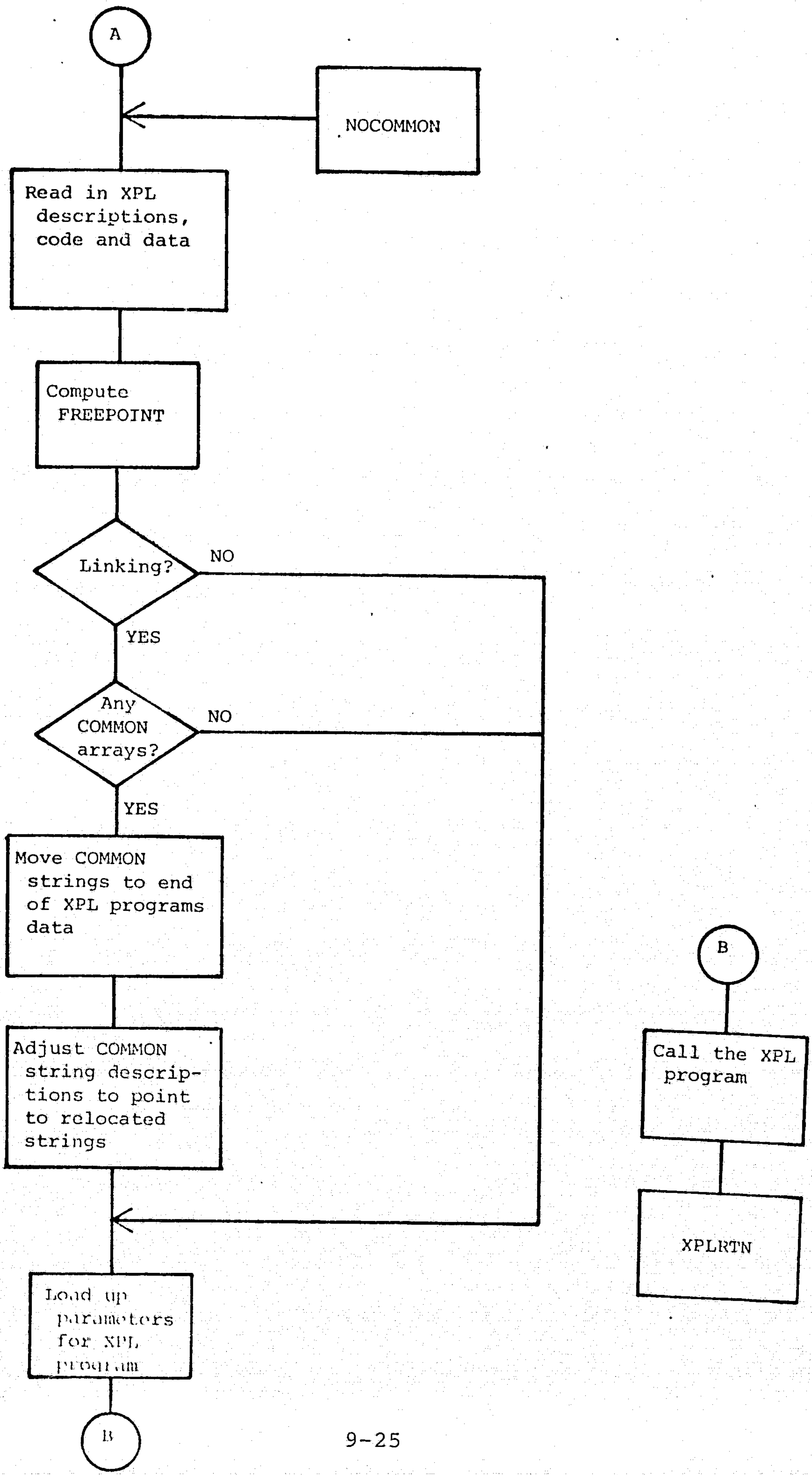

A
Set DUMP and LISTING2 flags if necessary
Set LINECT, PAGES, MIN, MAX, FREE
Copy INCLUDE & OUTPUT6 DD's for parallel access
Issue SPIE for floating point over/ under flows
Issue STIMER TASK for one hour
Issue OPEN on PROGRAM, SYSIN, SYSPRINT, ERROR
B
B
LISTING2 requested?
YES
NO
Issue OPEN on LISTING2
OPEN successful?
YES
NO
SYSPRINT OPEN successful?
NO
YES
PROGRAM OPEN successful?
NO
YES
OPENOK
Load abend code 100
ABEND

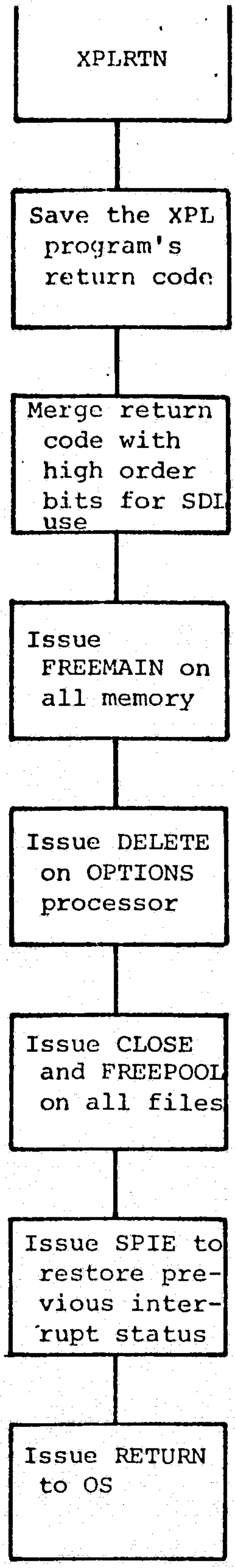

OPENOK
Issue GETMAIN
Min = COREMIN
Max = COREMAX
ACORE←address
CORESIZE←size
CORETOP = ACORE + CORESIZE - FREEUP
CORESIZE = CORESIZE - FREEUP
Issue FREEMAIN
Size = FREEUP
Call READPGM for first record of XPL program
Record program id in PGMID
NODLINK
Set number of common strings = 0
A

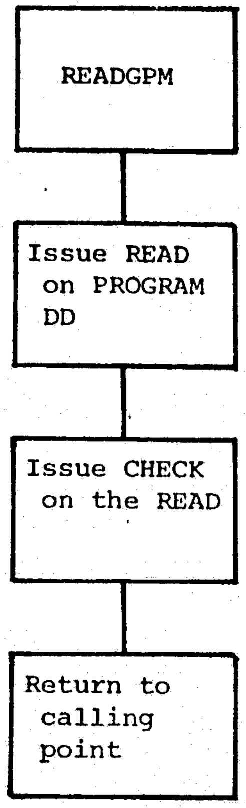

A
LINKING
LINKING?
YES
NO
Call READPGM for file control block
New COMMON= previous COMMON?
YES
NO
Load abend code 700
ABEND
Space required = COMMON SIZE + DESCRIPTOR SIZE + PROGRAM SIZE
Space required> CORESIZE
YES
NO
Load abend code 400
ABEND
LINKING?
NO
ZERODATA
YES
B
B
Any COMMON strings?
NO
NOCOMMON
YES
String start>ACORE + space required?
YES
NOCOMMON
NO
Load abend code 1100
ABEND

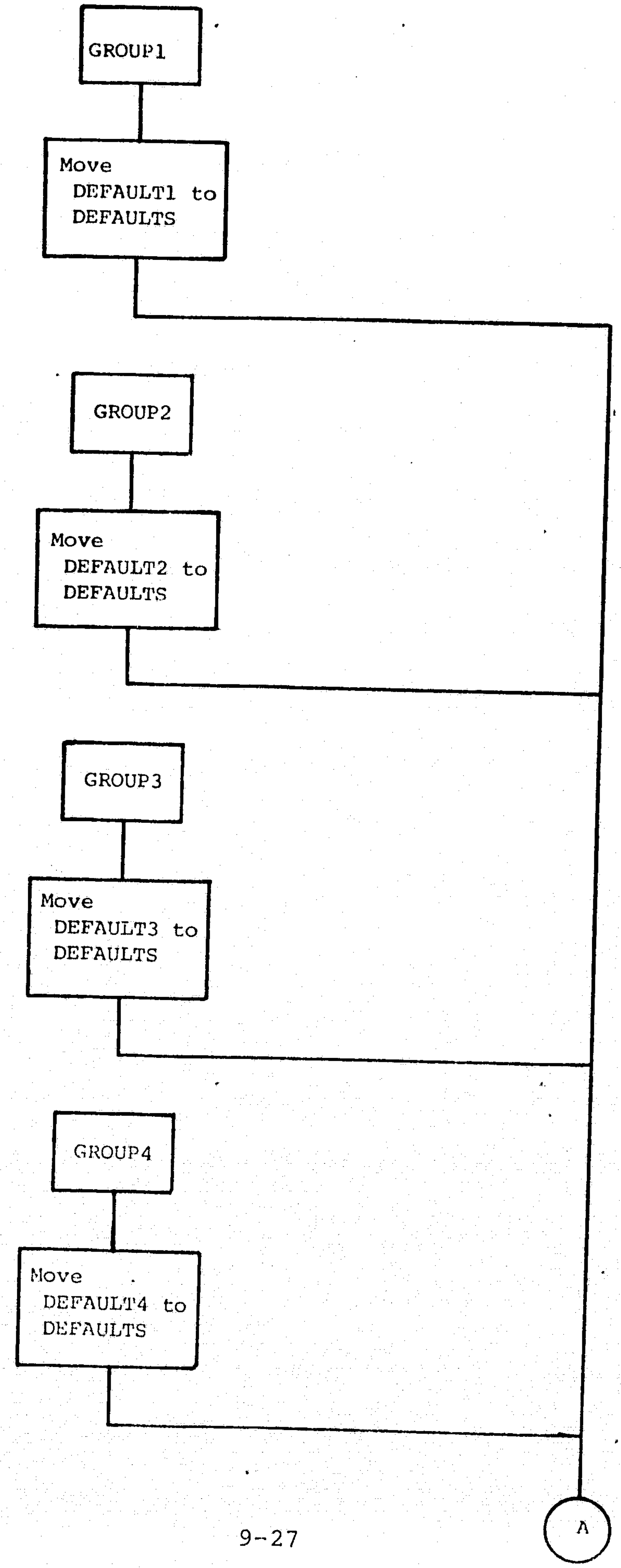

ZERODATA
Any COMMON?
NO
YES
Compute end of COMMON address
Any COMMON arrays?
YES
NO
Move COMMON array initialization pairs past end of COMMON +512
Zero COMMON
Any COMMON arrays?
YES
NO
Initialize required COMMON areas with COMMON array offsets
A

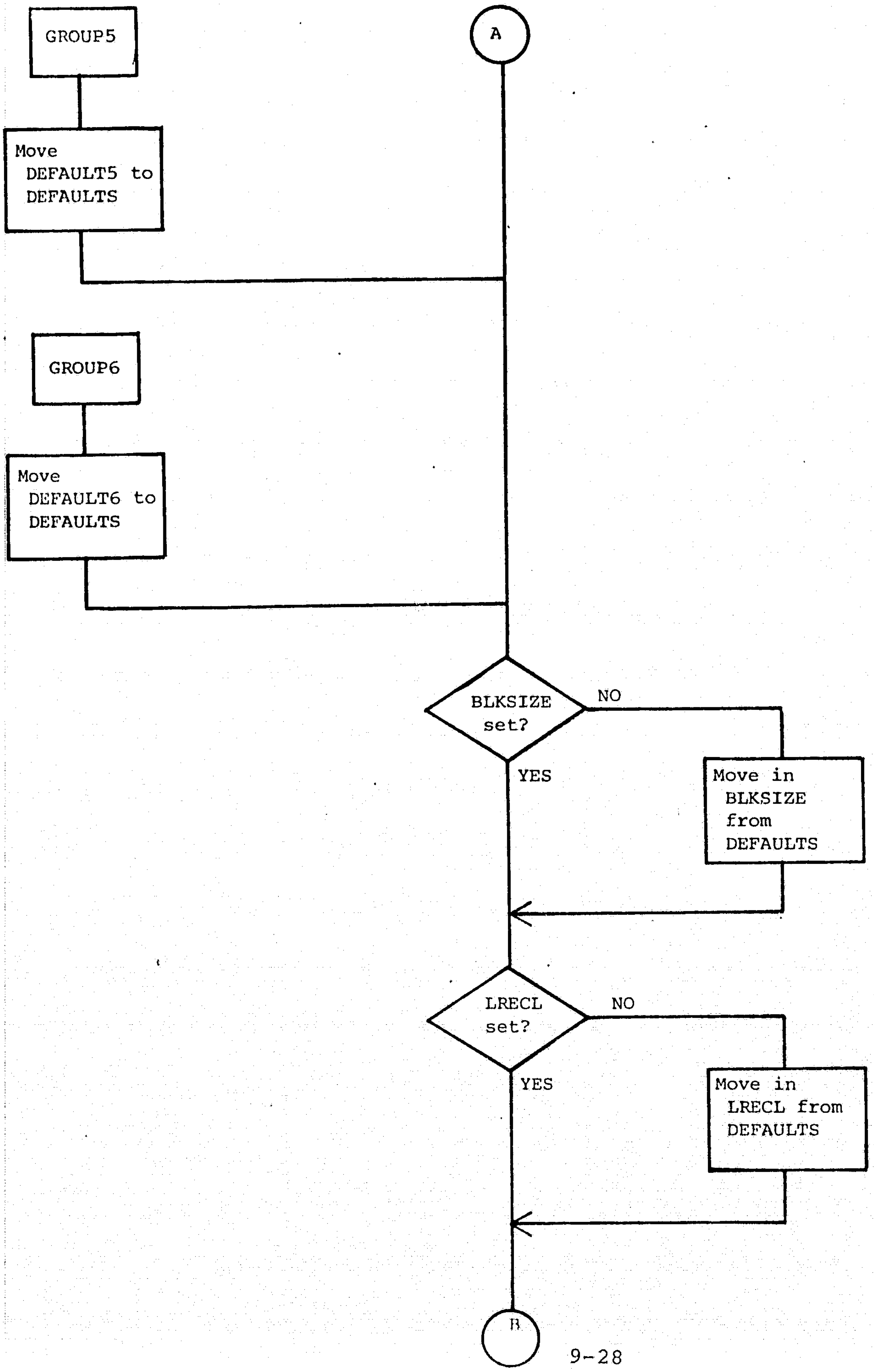
REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

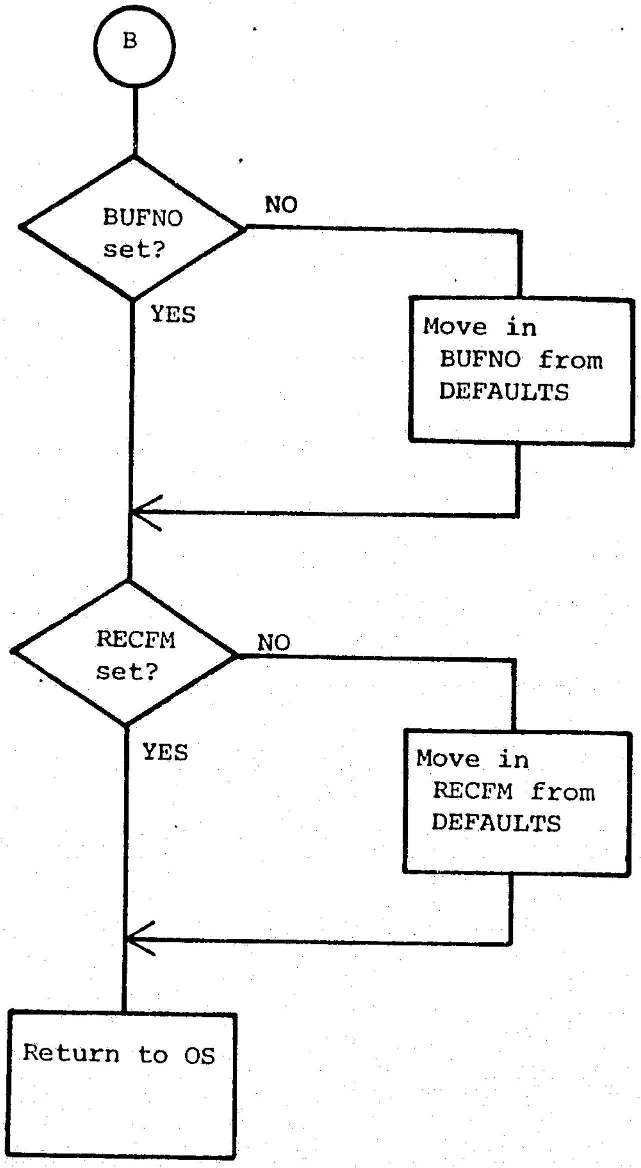

A
NOCOMMON
Read in XPL descriptions, code and data
Compute FREEPOINT
Linking?
NO
YES
Any COMMON arrays?
NO
YES
Move COMMON strings to end of XPL programs data
Adjust COMMON string descriptions to point to relocated strings
Load up parameters for XPL program
B
B
Call the XPL program
XPLRTN

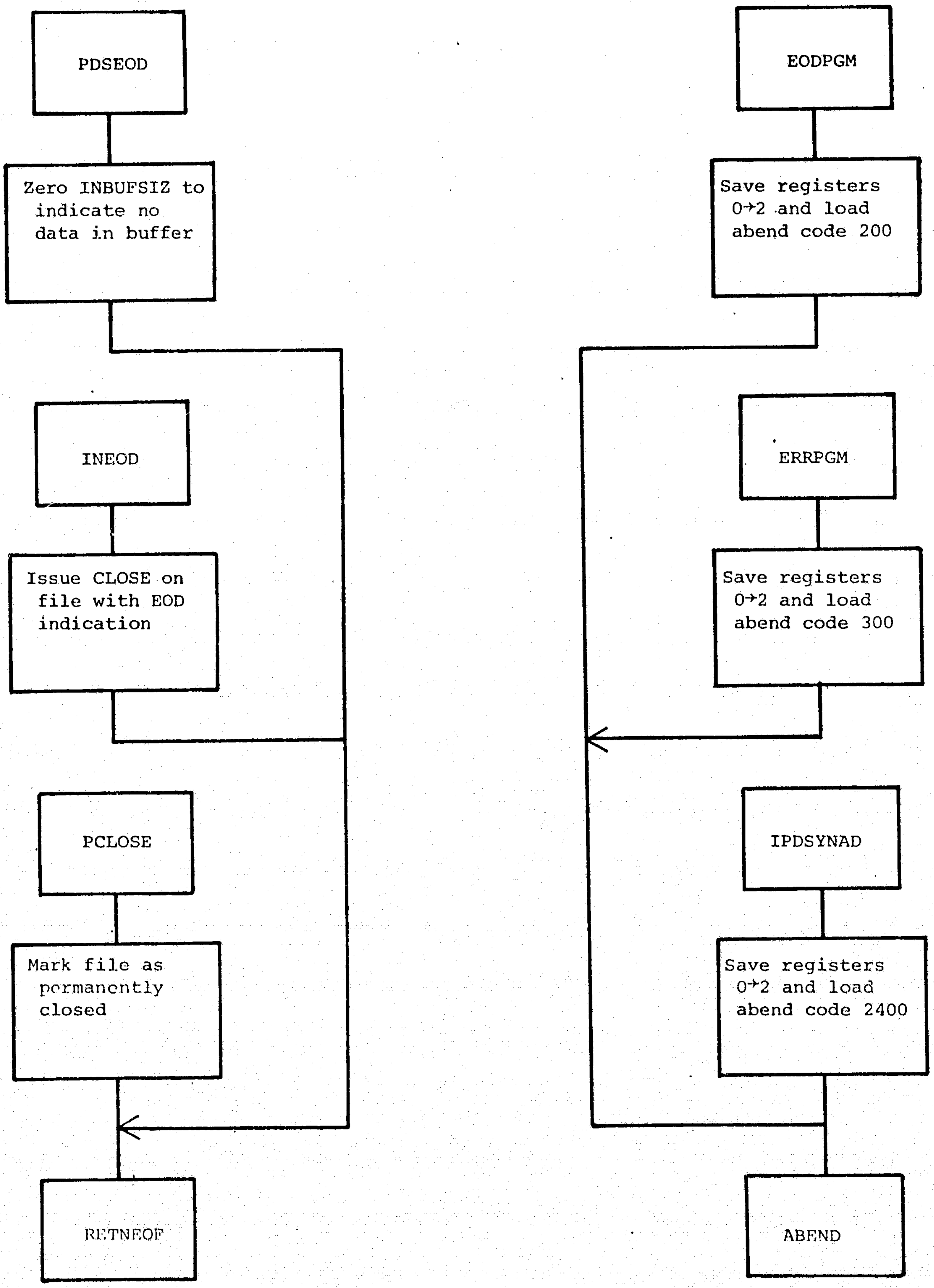

XPLRTN
Save the XPL program's return code
Merge return code with high order bits for SDI use
Issue FREEMAIN on all memory
Issue DELETE on OPTIONS processor
Issue CLOSE and FREEPOOL on all files
Issue SPIE to restore previous interrupt status
Issue RETURN to OS

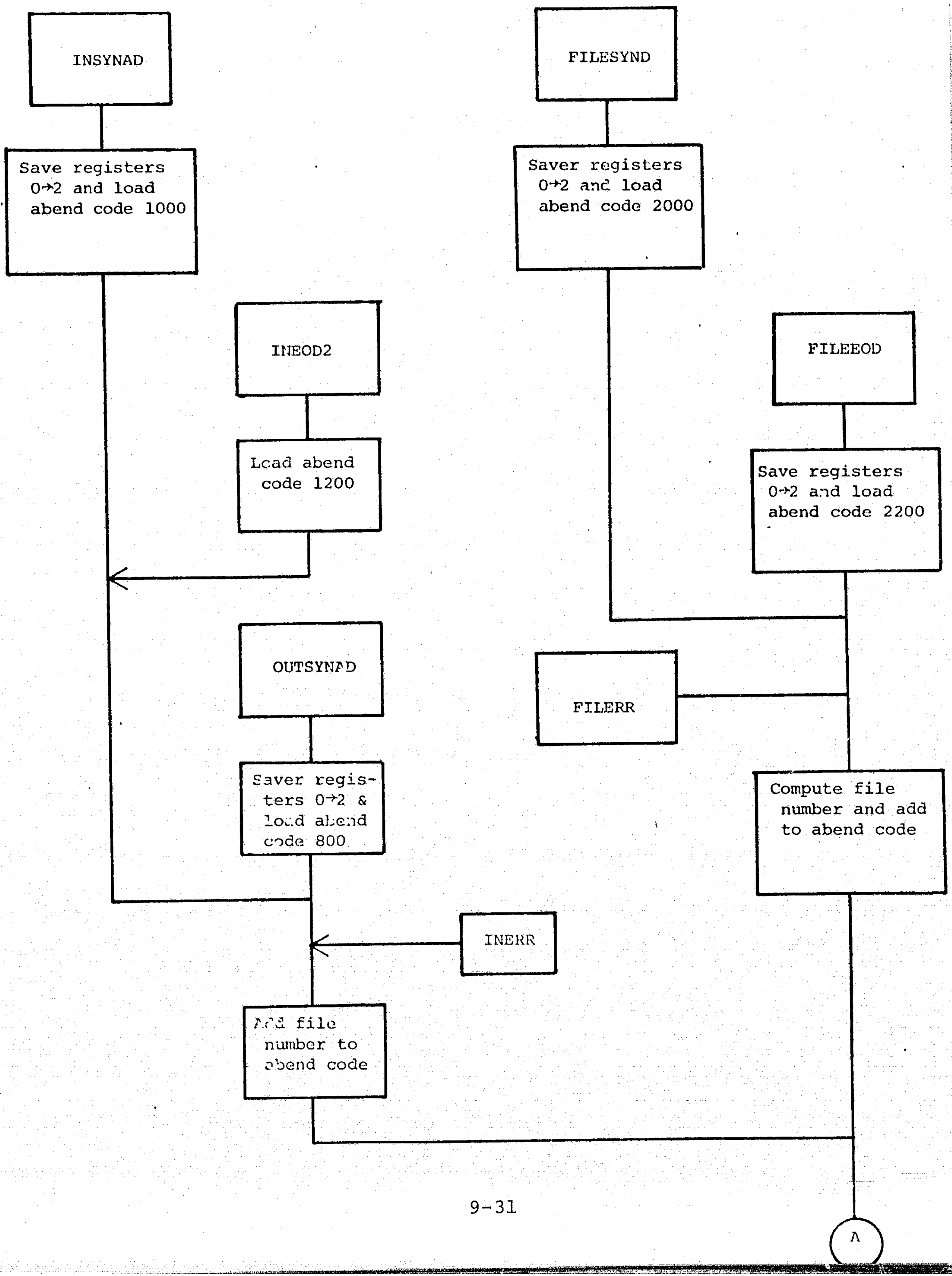

READGPM
Issue READ on PROGRAM DD
Issue CHECK on the READ
Return to calling point

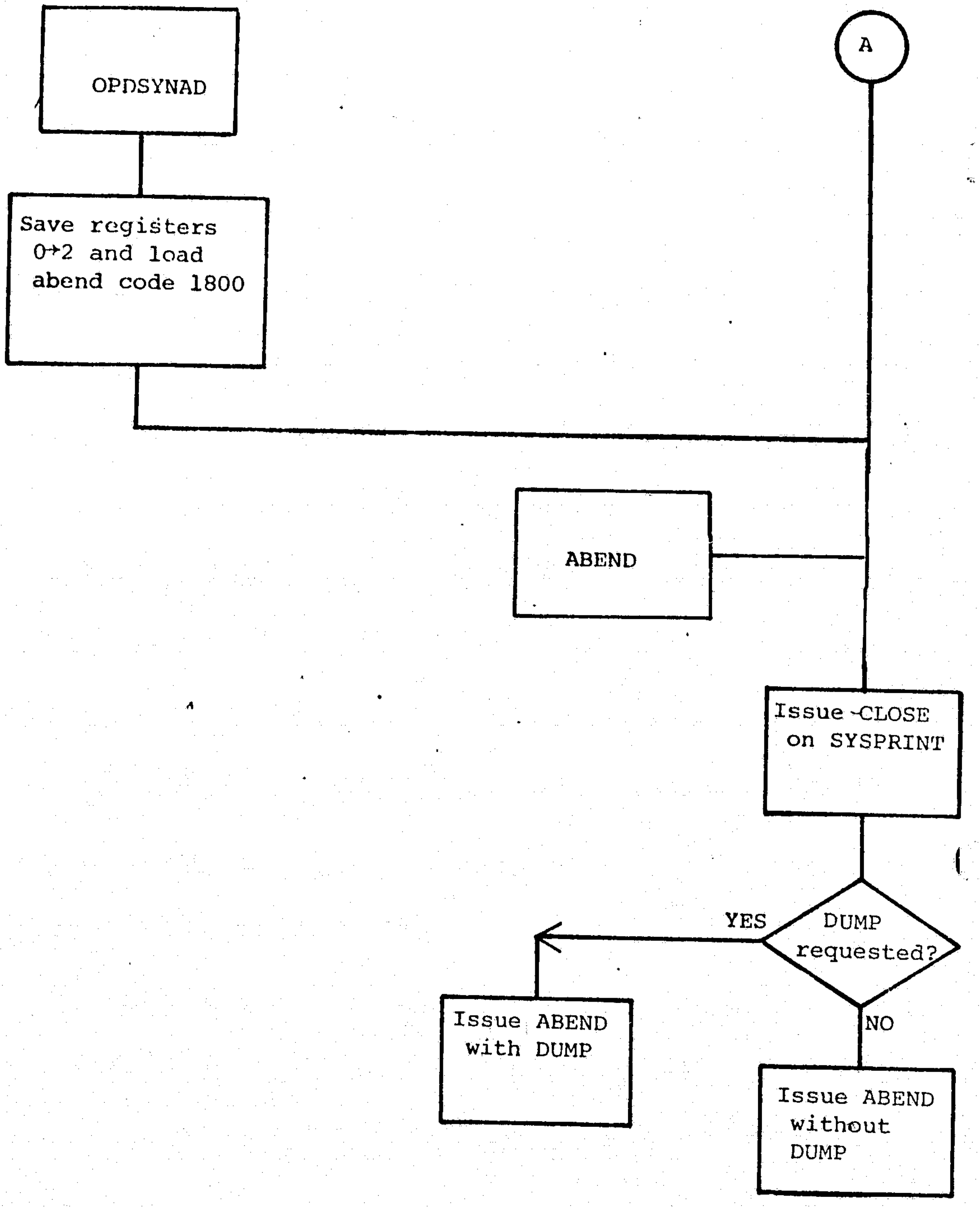

GROUP1
Move DEFAULT1 to DEFAULTS
GROUP2
Move DEFAULT2 to DEFAULTS
GROUP3
Move DEFAULT3 to DEFAULTS
GROUP4
Move DEFAULT4 to DEFAULTS
A

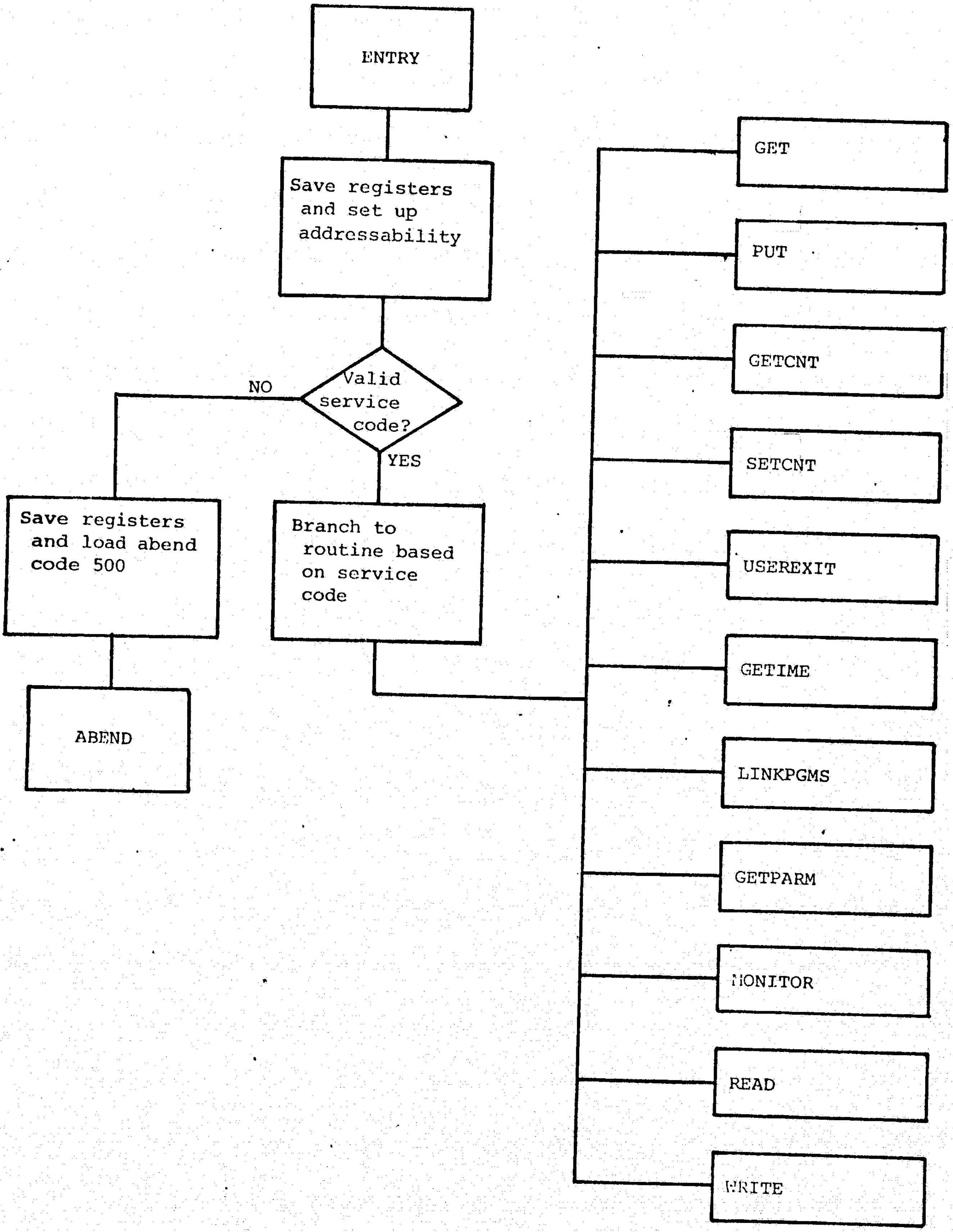

A
GROUP5
Move DEFAULT5 to DEFAULTS
GROUP6
Move DEFAULT6 to DEFAULTS
BLKSIZE set?
NO
YES
Move in BLKSIZE from DEFAULTS
LRECL set?
NO
YES
Move in LRECL from DEFAULTS
B

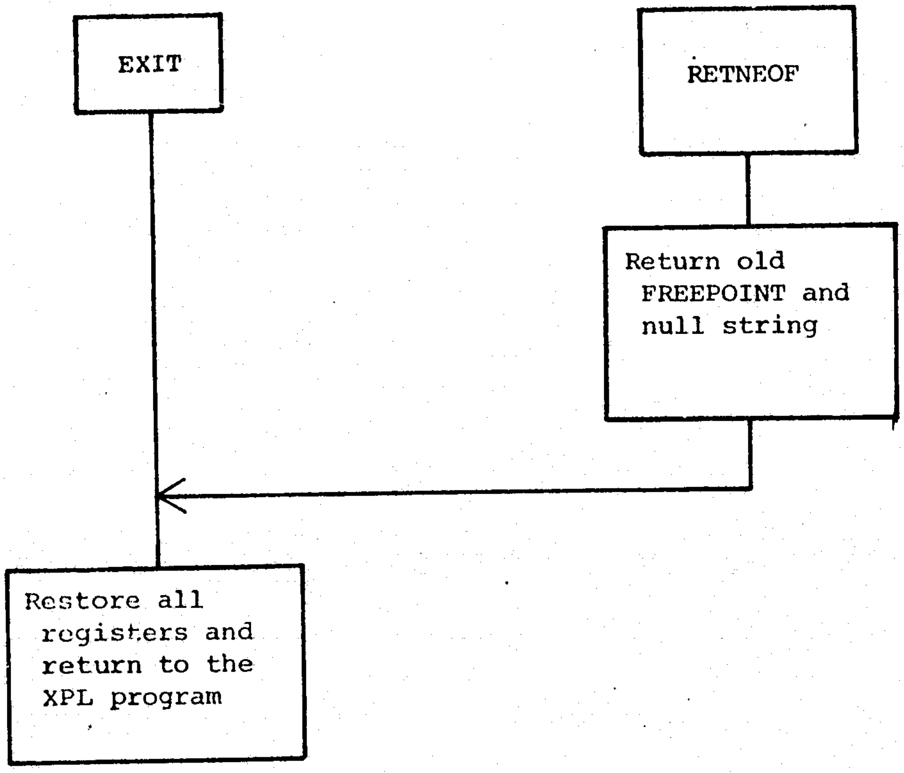

B
BUFNO set?
NO
YES
Move in BUFNO from DEFAULTS
RECFM set?
NO
YES
Move in RECFM from DEFAULTS
Return to OS

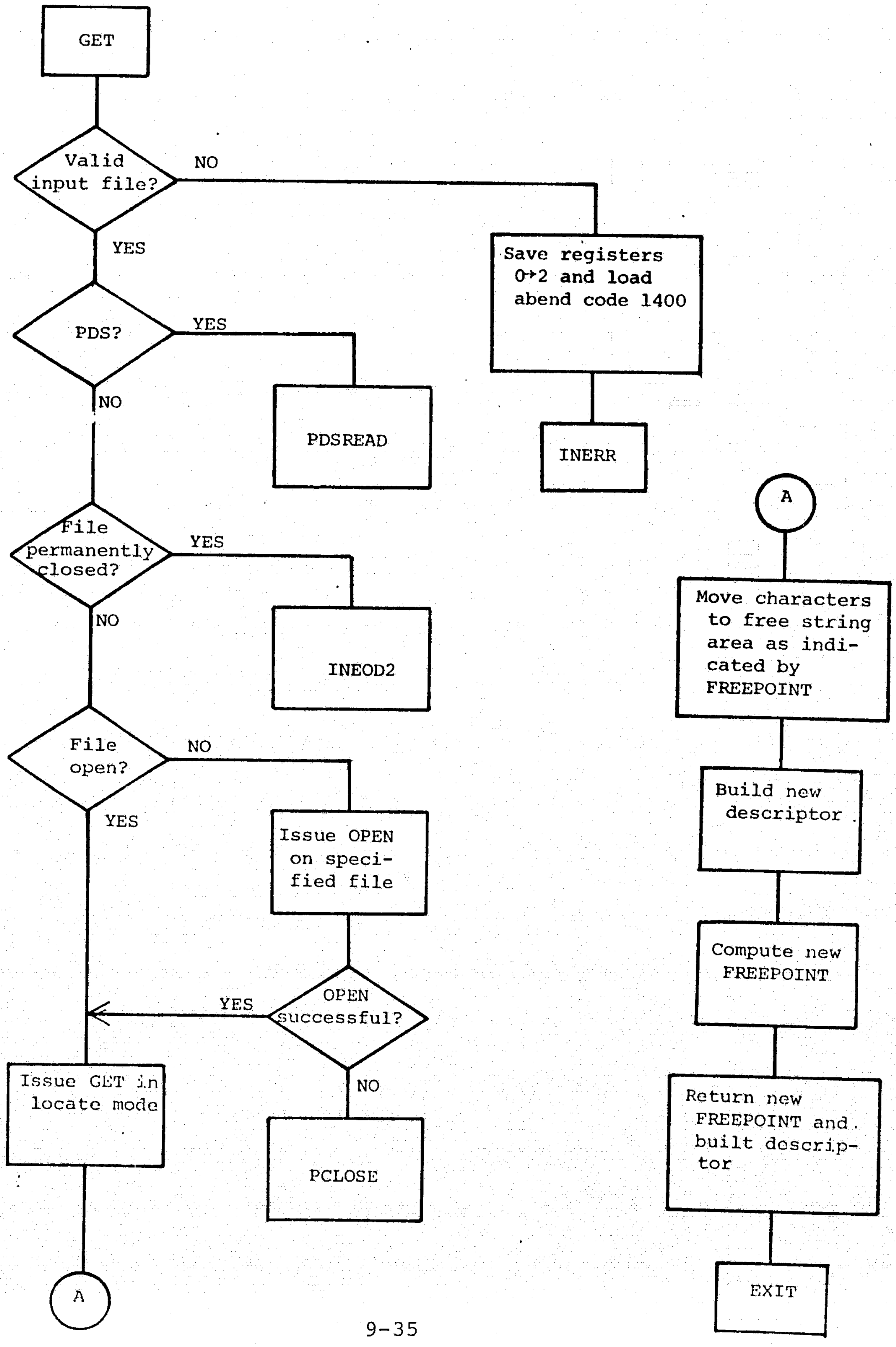

PDSEOD
Zero INBUFSIZ to indicate no data in buffer
INEOD
Issue CLOSE on file with EOD indication
PCLOSE
Mark file as permanently closed
RETNEOF
EODPGM
Save registers 0→2 and load abend code 200
ERRPGM
Save registers 0→2 and load abend code 300
IPDSYNAD
Save registers 0→2 and load abend code 2400
ABEND

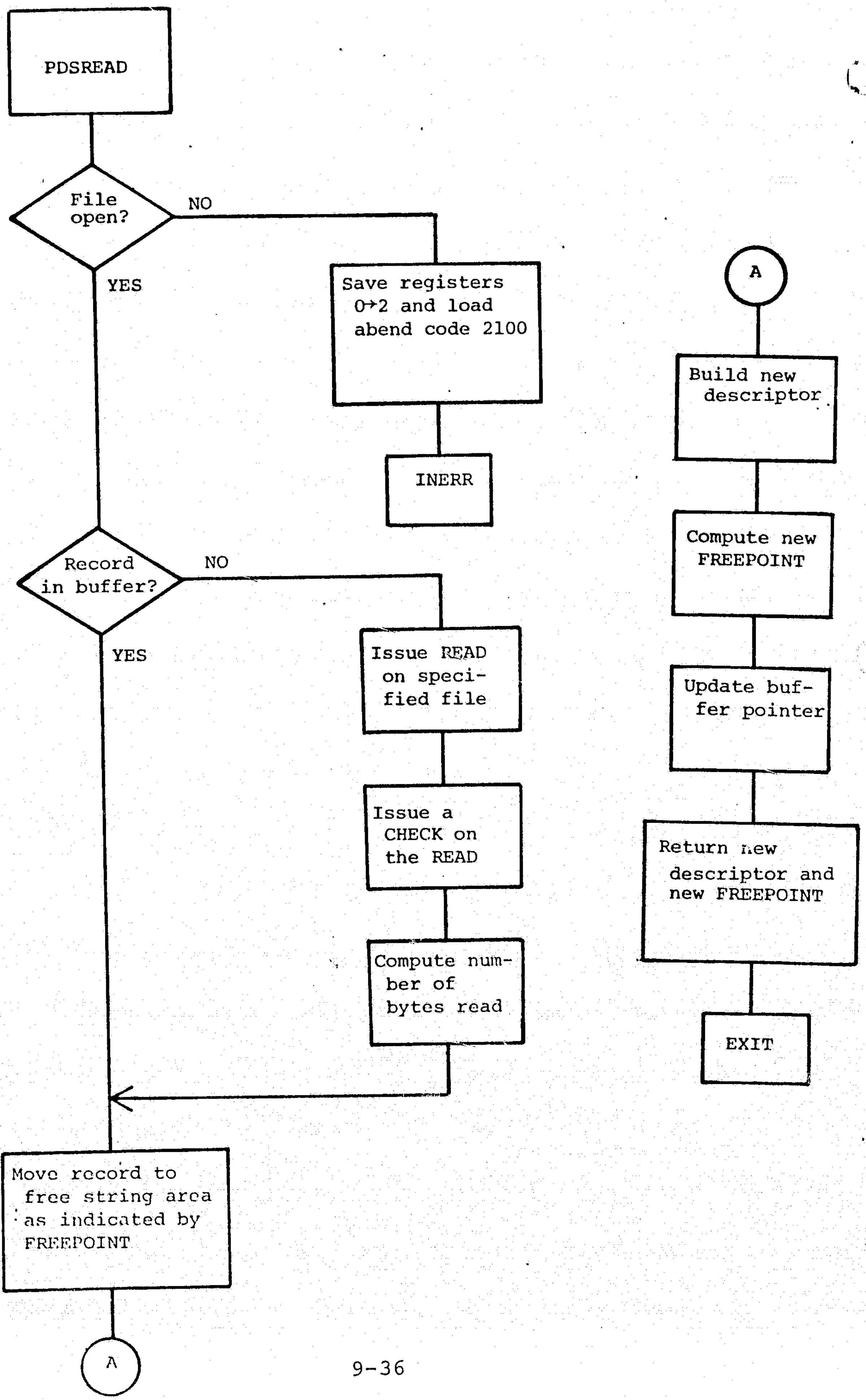

INSYNAD
Save registers 0→2 and load abend code 1000
FILESYND
Saver registers 0→2 and load abend code 2000
INEOD2
Load abend code 1200
FILEEOD
Save registers 0→2 and load abend code 2200
OUTSYNAD
FILERR
Saver registers 0→2 & load abend code 800
Compute file number and add to abend code
INERR
Add file number to abend code
A

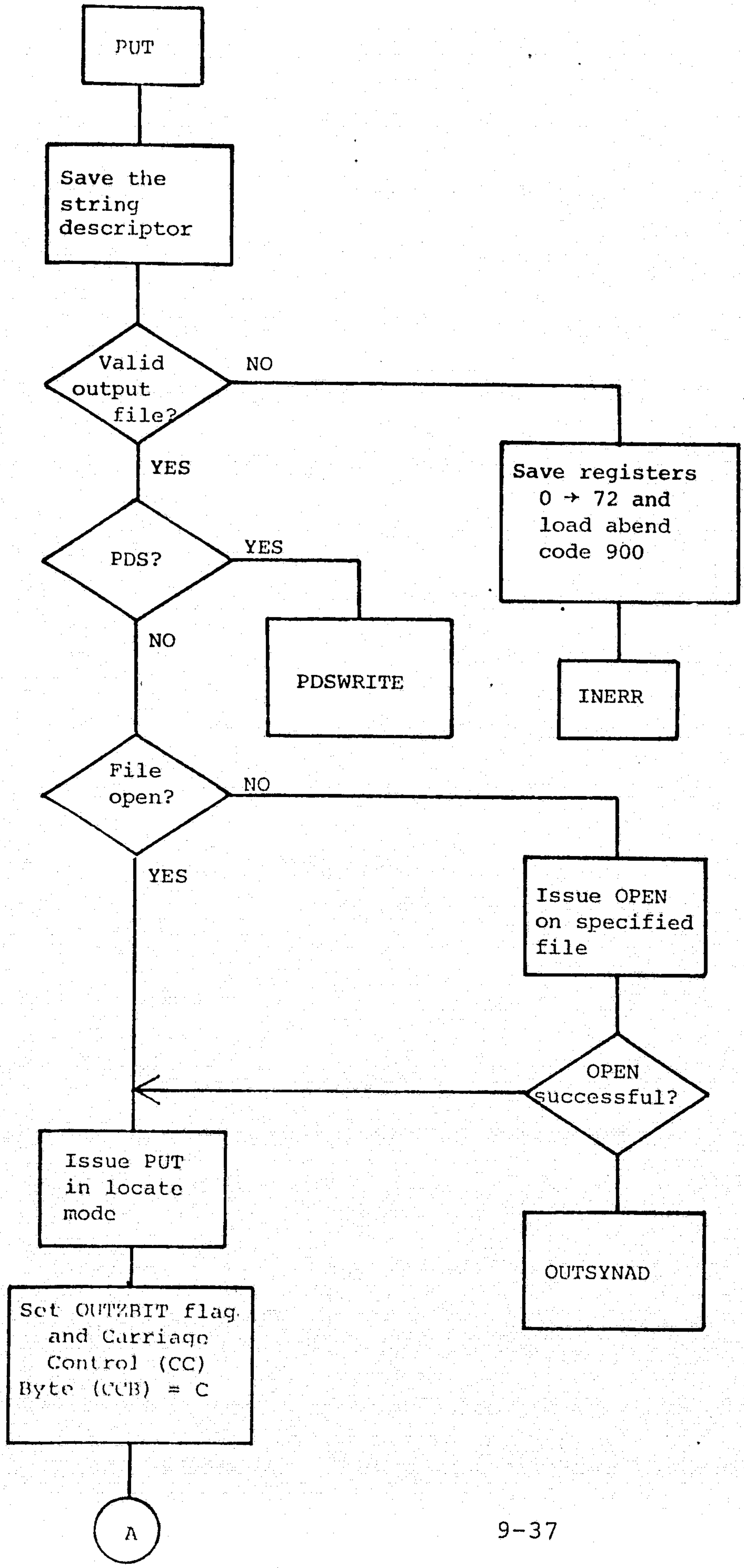

OPDSYNAD
A
Save registers 0→2 and load abend code 1800
ABEND
Issue CLOSE on SYSPRINT
DUMP requested?
YES
NO
Issue ABEND with DUMP
Issue ABEND without DUMP

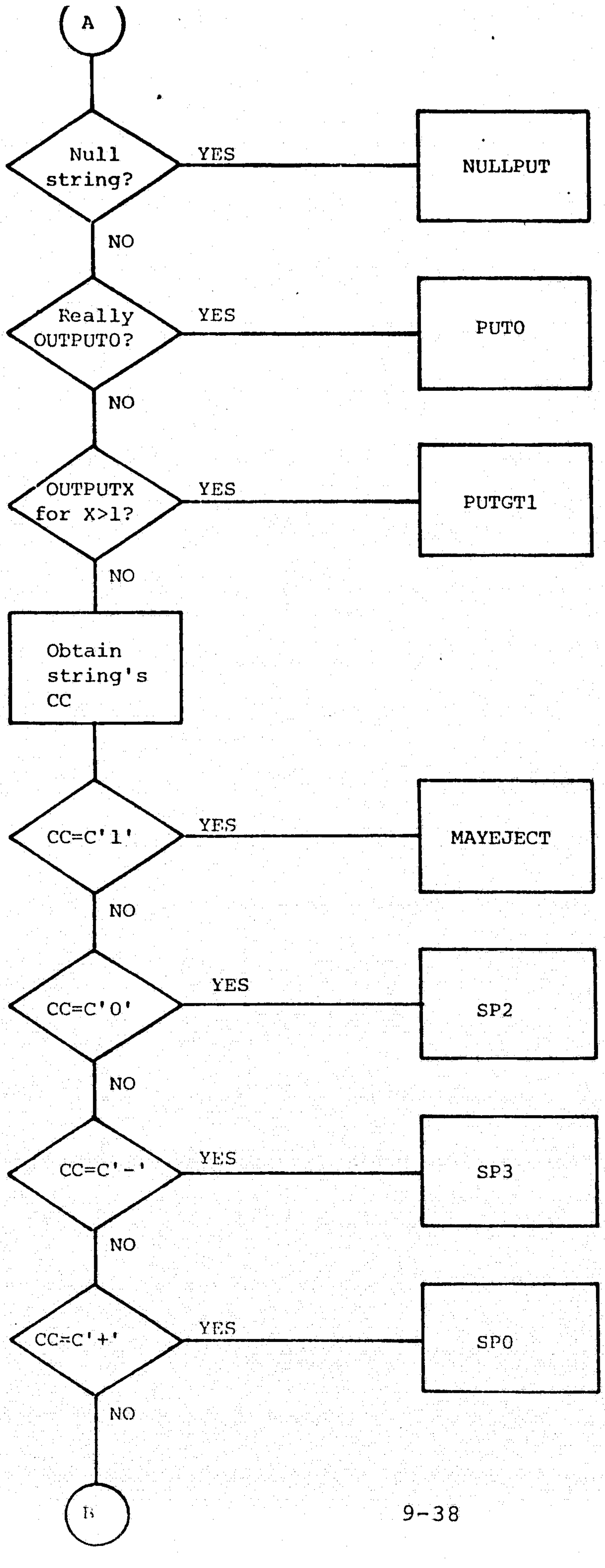

ENTRY
Save registers and set up addressability
Valid service code?
NO
YES
Save registers and load abend code 500
ABEND
Branch to routine based on service code
GET
PUT
GETCNT
SETCNT
USEREXIT
GETIME
LINKPGMS
GETPARM
MONITOR
READ
WRITE

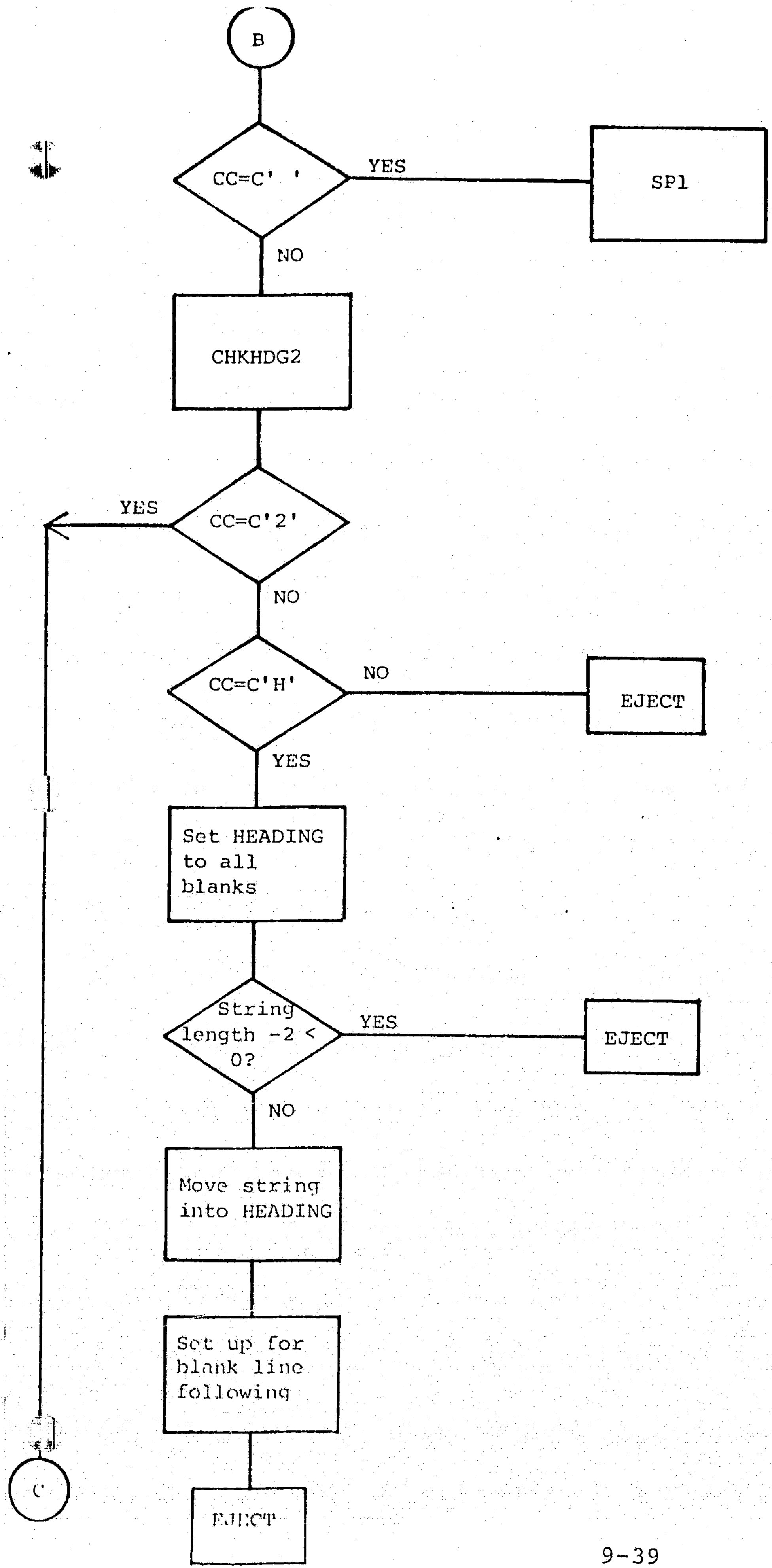

EXIT
RETNEOF
Return old FREEPOINT and null string
Restore all registers and return to the XPL program

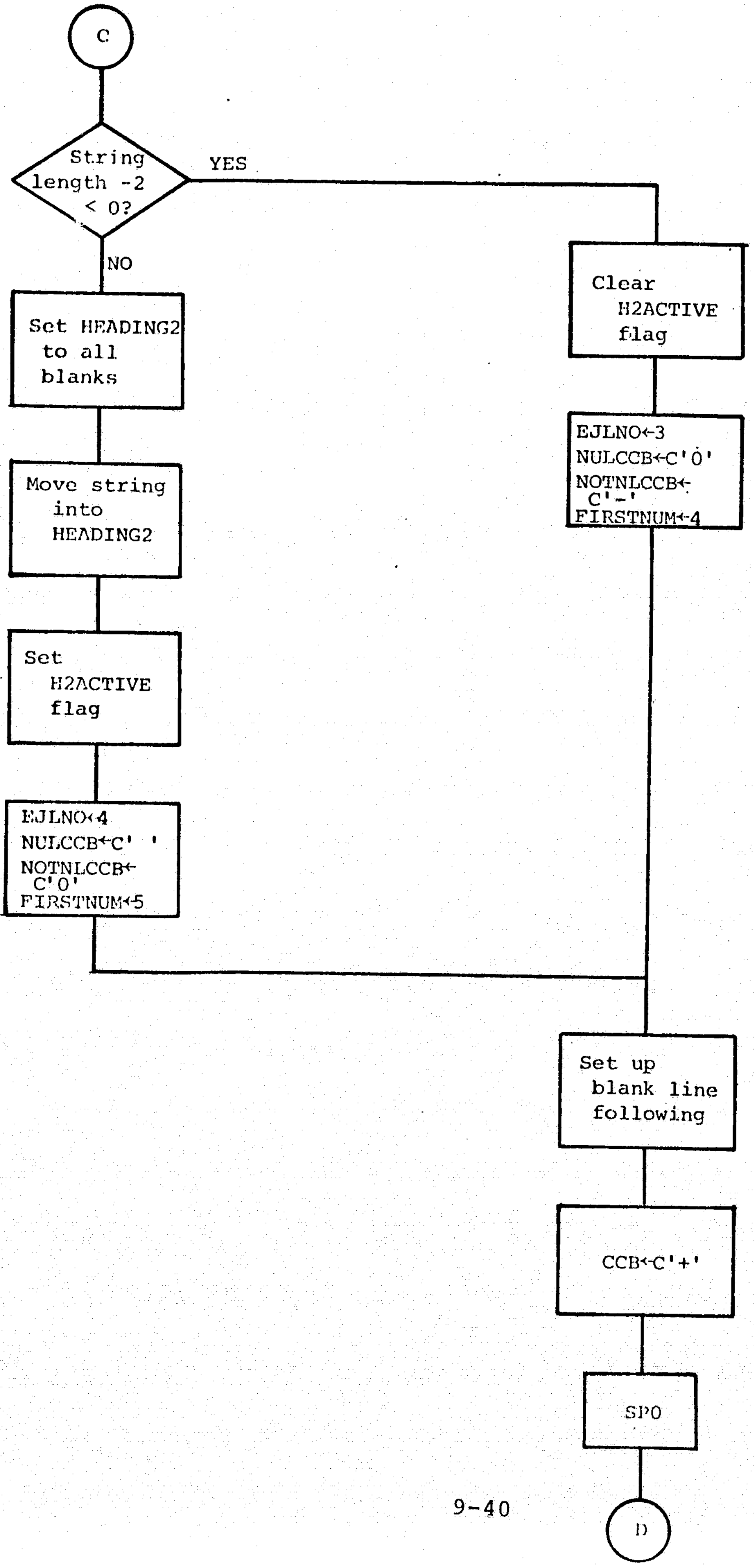

GET
Valid input file?
NO
YES
Save registers 0→2 and load abend code 1400
INERR
PDS?
YES
NO
PDSREAD
File permanently closed?
YES
NO
INEOD2
File open?
NO
YES
Issue OPEN on specified file
OPEN successful?
YES
NO
PCLOSE
Issue GET in locate mode
A
A
Move characters to free string area as indicated by FREEPOINT
Build new descriptor
Compute new FREEPOINT
Return new FREEPOINT and built descriptor
EXIT

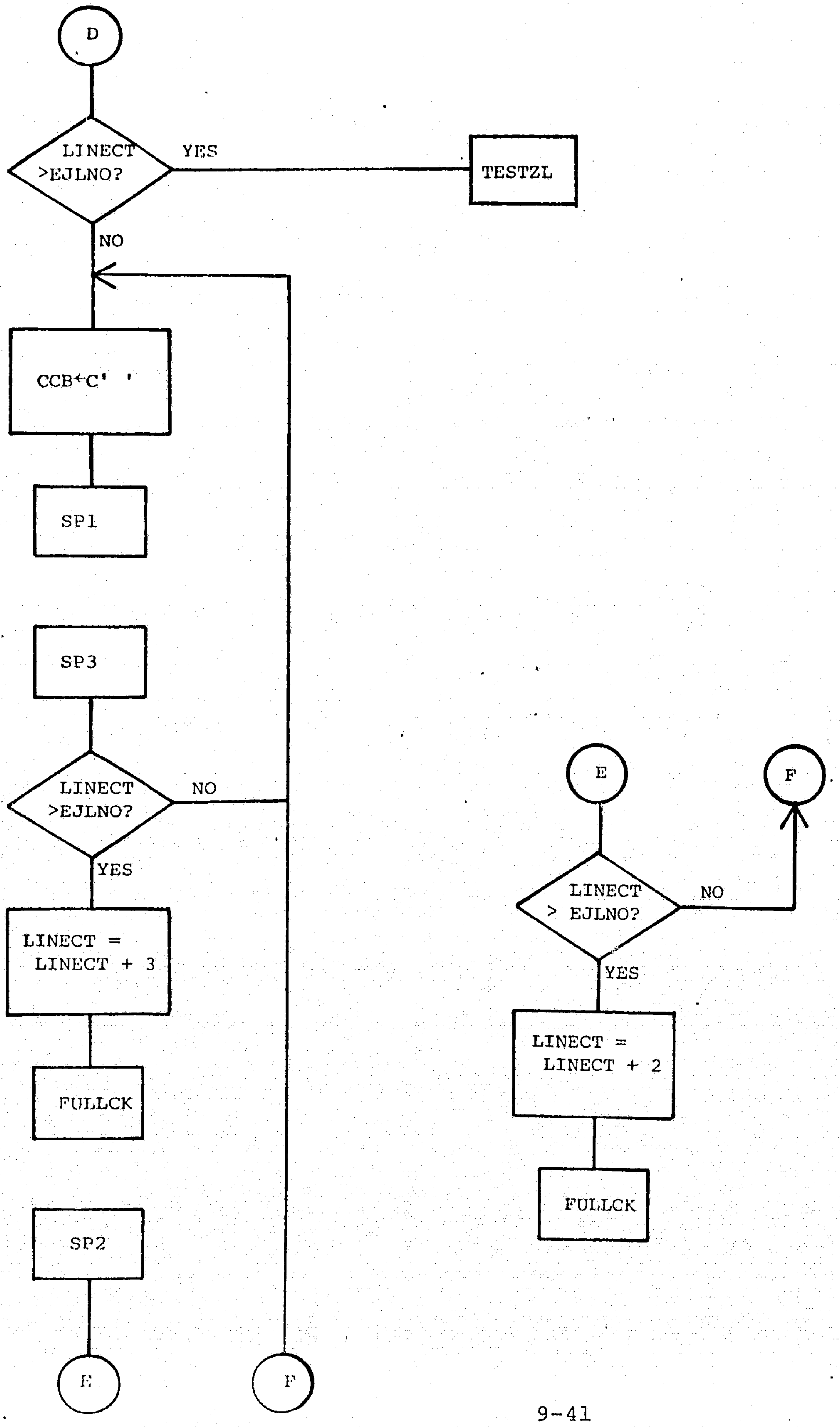

PDSREAD
File open?
NO
YES
Save registers 0→2 and load abend code 2100
INERR
Record in buffer?
NO
YES
Issue READ on specified file
Issue a CHECK on the READ
Compute number of bytes read
Move record to free string area as indicated by FREEPOINT
A
A
Build new descriptor
Compute new FREEPOINT
Update buffer pointer
Return new descriptor and new FREEPOINT
EXIT

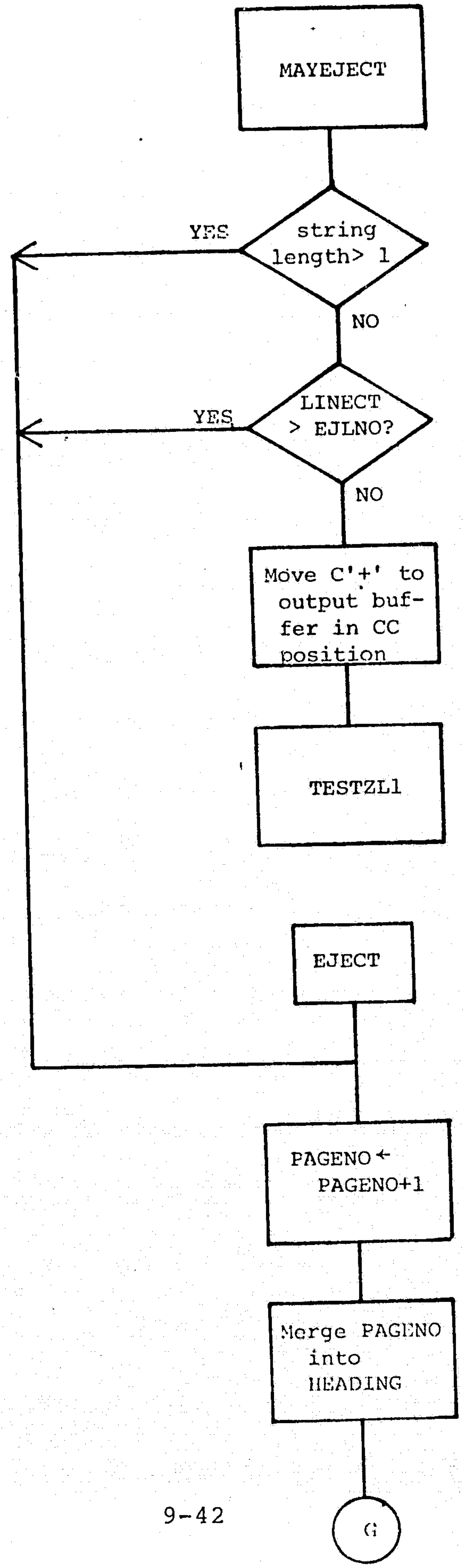

PUT
Save the string descriptor
Valid output file?
NO
YES
Save registers 0 → 72 and load abend code 900
INERR
PDS?
YES
PDSWRITE
NO
File open?
NO
YES
Issue OPEN on specified file
OPEN successful?
OUTSYNAD
Issue PUT in locate mode
Set OUTZBIT flag and Carriage Control (CC) Byte (CCB) = C
A

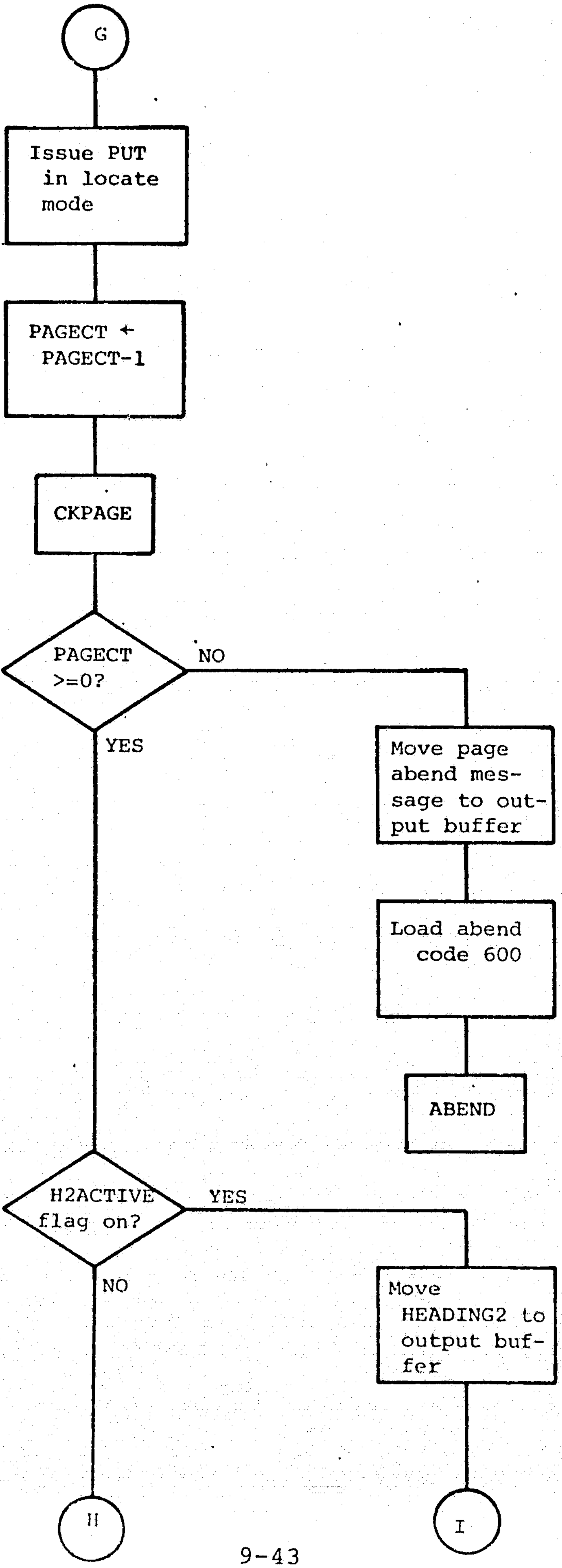

A
Null string?
YES
NULLPUT
NO
Really OUTPUTO?
YES
PUTO
NO
OUTPUTX for X>1?
YES
PUTGT1
NO
Obtain string's CC
CC=C'1'
YES
MAYEJECT
NO
CC=C'O'
YES
SP2
NO
CC=C'-'
YES
SP3
NO
CC=C'+'
YES
SPO
NO
B

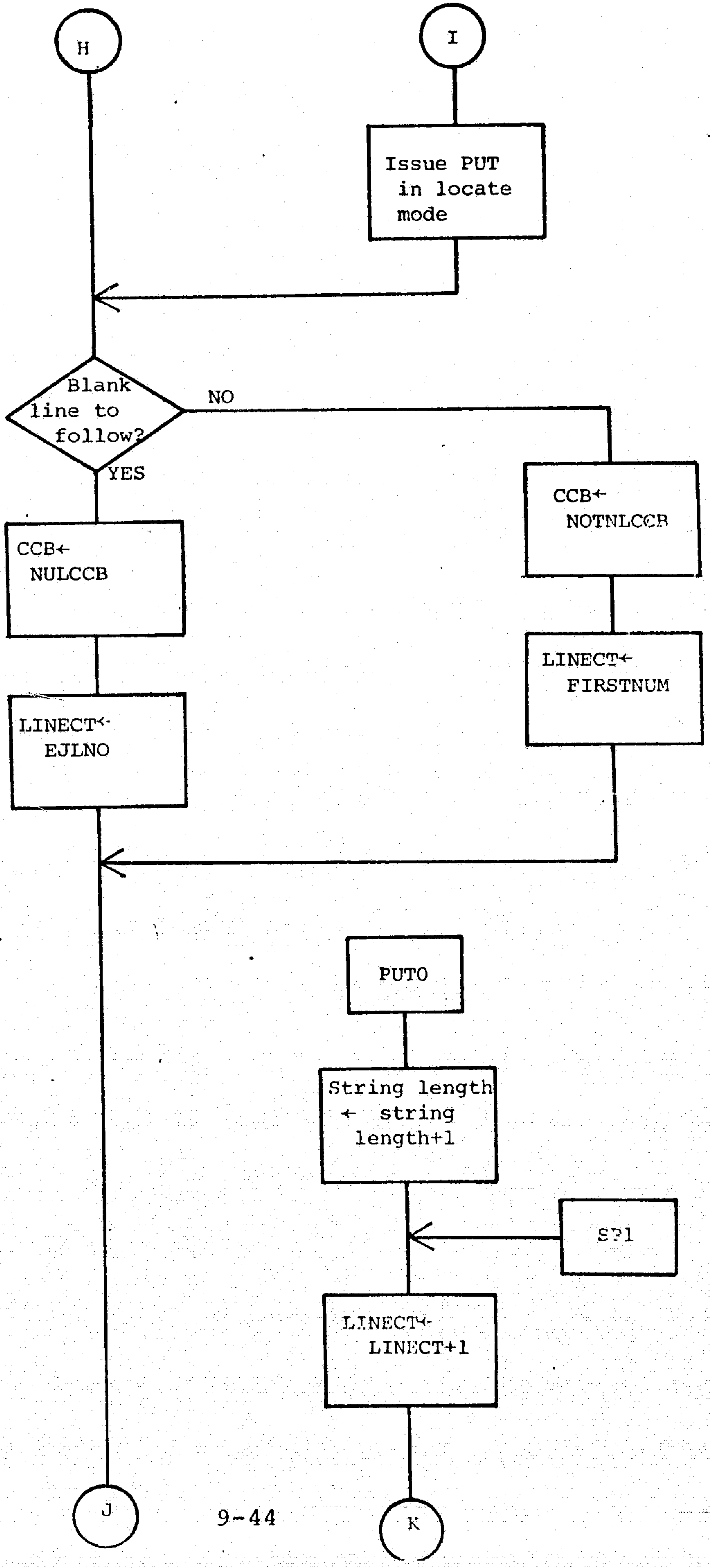

B
CC=C' '
YES
SP1
NO
CHKHDG2
YES
CC=C'2'
NO
CC=C'H'
NO
EJECT
YES
Set HEADING to all blanks
String length -2 < 0?
YES
EJECT
NO
Move string into HEADING
Set up for blank line following
C
EJECT

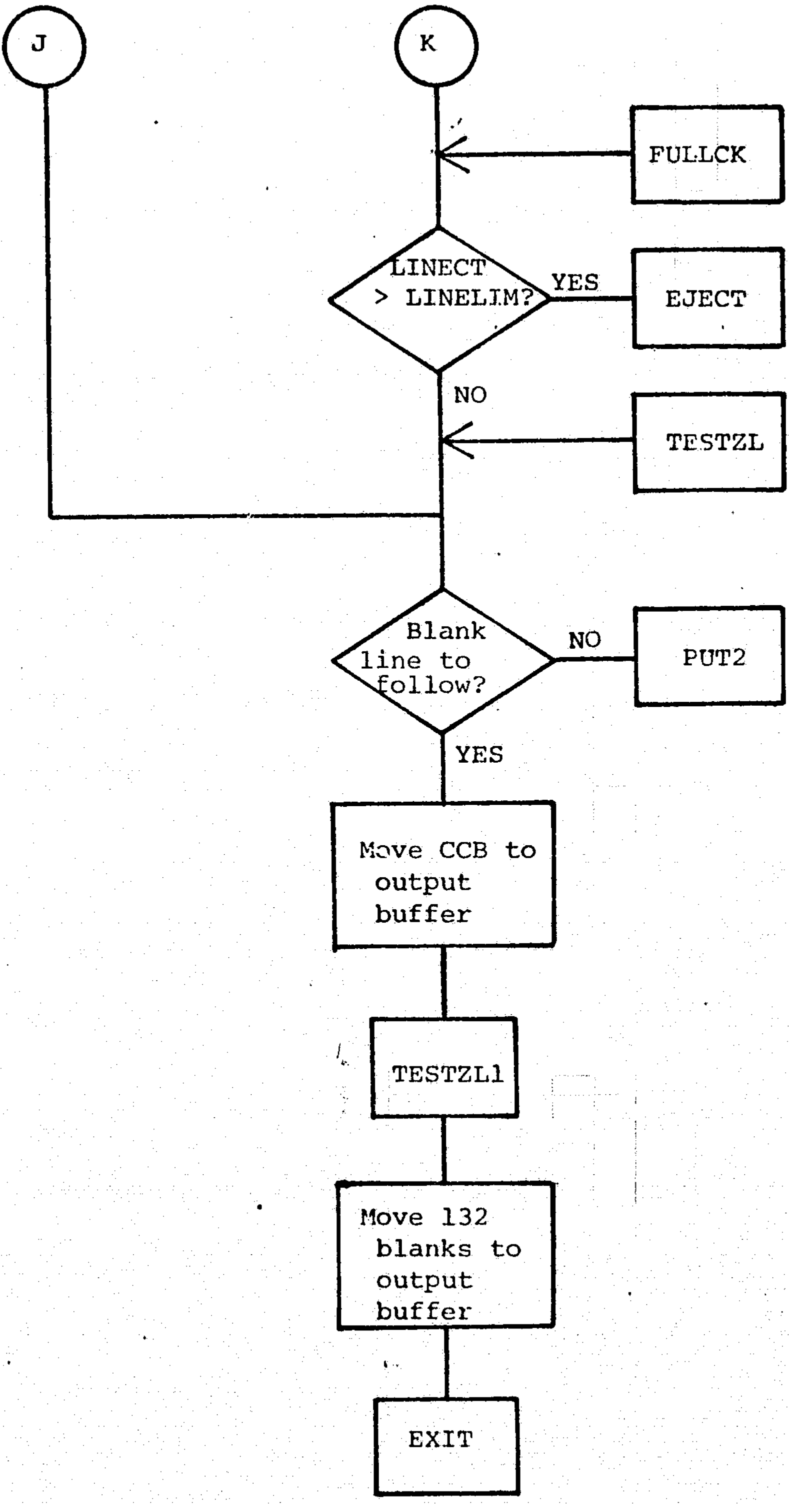

C
String length -2 < 0?
YES
NO
Set HEADING2 to all blanks
Move string into HEADING2
Set H2ACTIVE flag
EJLNO←4
NULCCB←C' '
NOTNLCCB← C'0'
FIRSTNUM←5
Clear H2ACTIVE flag
EJLNO←3
NULCCB←C'0'
NOTNLCCB← C'-'
FIRSTNUM←4
Set up blank line following
CCB←C'+'
SPO
D

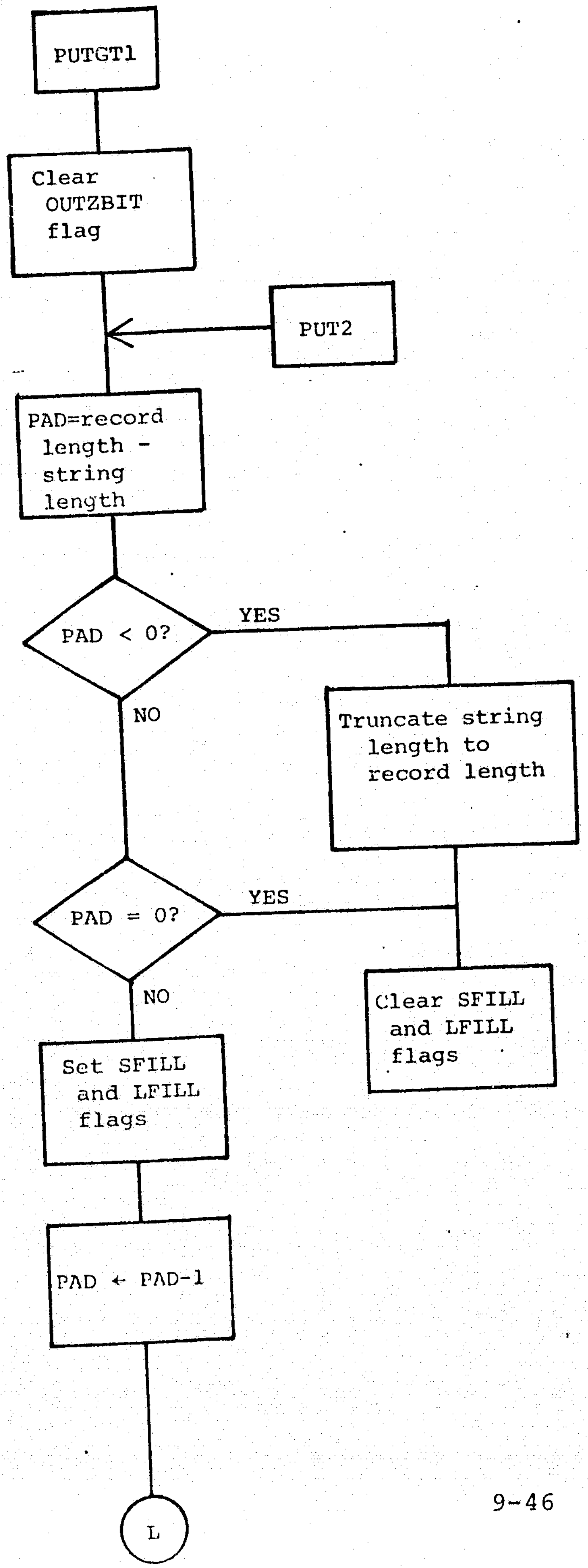

D
LINECT >EJLNO?
YES
TESTZL
NO
CCB←C' '
SP1
SP3
LINECT >EJLNO?
NO
YES
LINECT = LINECT + 3
FULLCK
SP2
E
F
E
F
LINECT > EJLNO?
NO
YES
LINECT = LINECT + 2
FULLCK

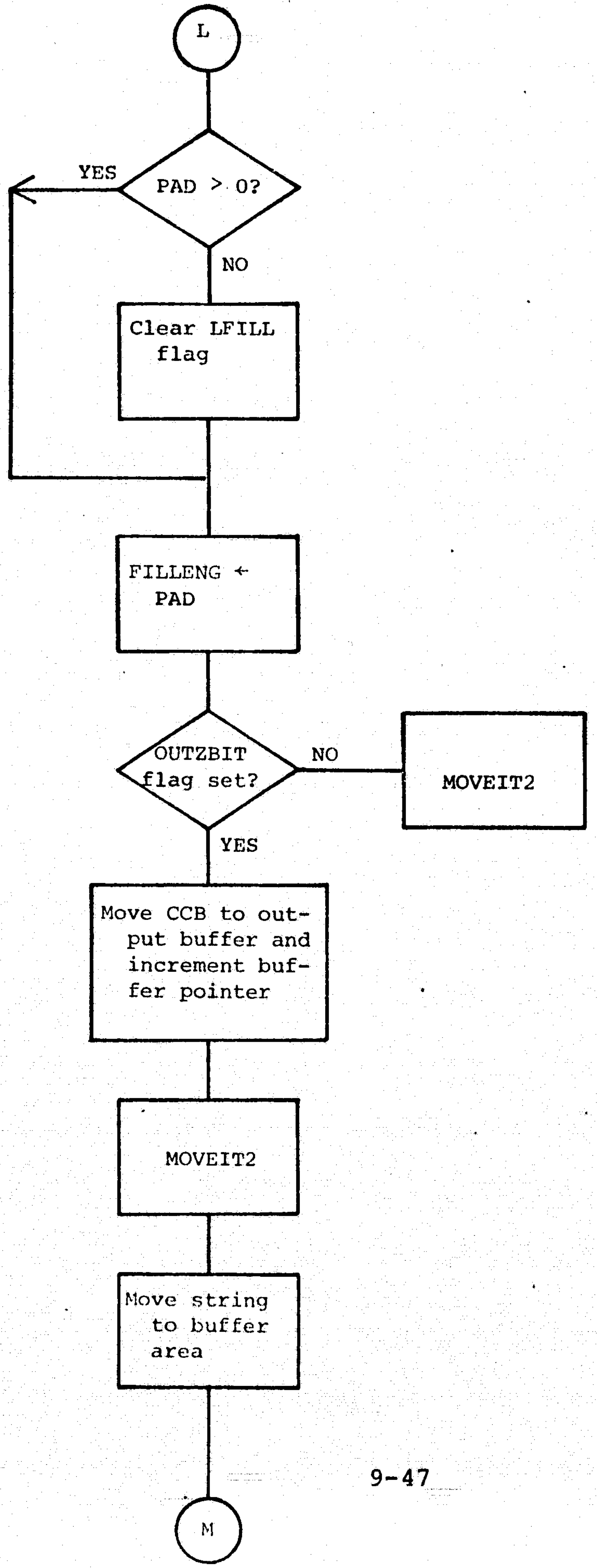

MAYEJECT
string length> 1
YES
NO
LINECT > EJLNO?
YES
NO
Move C'+' to output buffer in CC position
TESTZL1
EJECT
PAGENO ← PAGENO+1
Merge PAGENO into HEADING
G

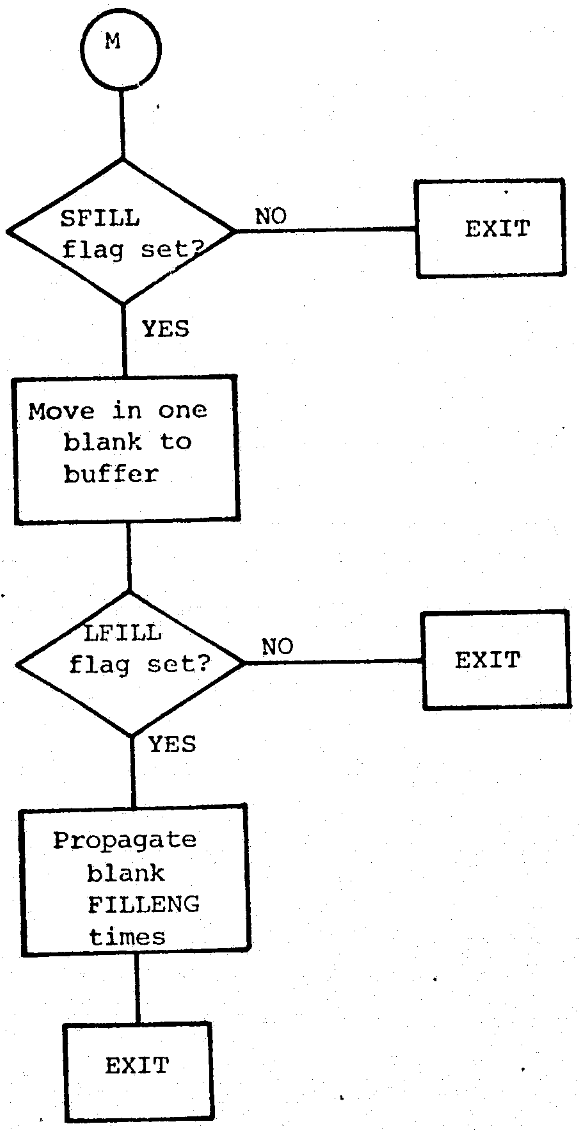

G
Issue PUT in locate mode
PAGECT ← PAGECT-1
CKPAGE
PAGECT >=0?
NO
YES
Move page abend message to output buffer
Load abend code 600
ABEND
H2ACTIVE flag on?
YES
NO
Move HEADING2 to output buffer
H
I

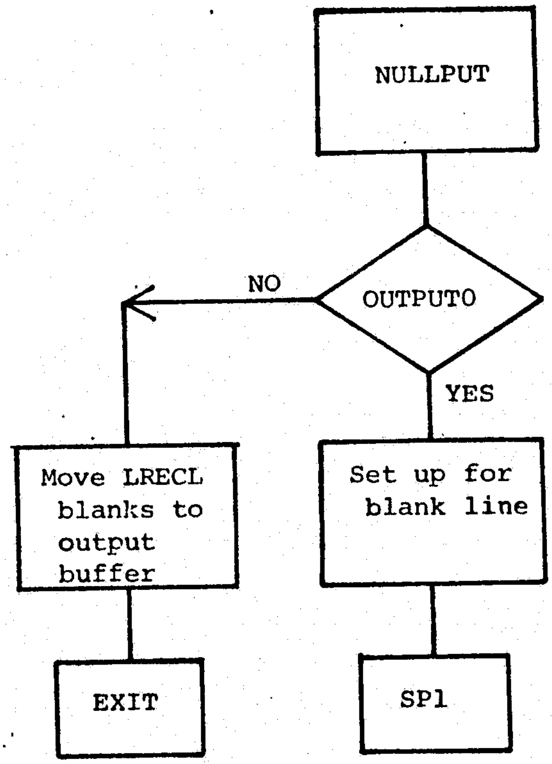

H
I
Issue PUT in locate mode
Blank line to follow?
NO
YES
CCB← NULCCB
CCB← NOTNLCCB
LINECT← EJLNO
LINECT← FIRSTNUM
PUTO
String length ← string length+1
SP1
LINECT← LINECT+1
J
K

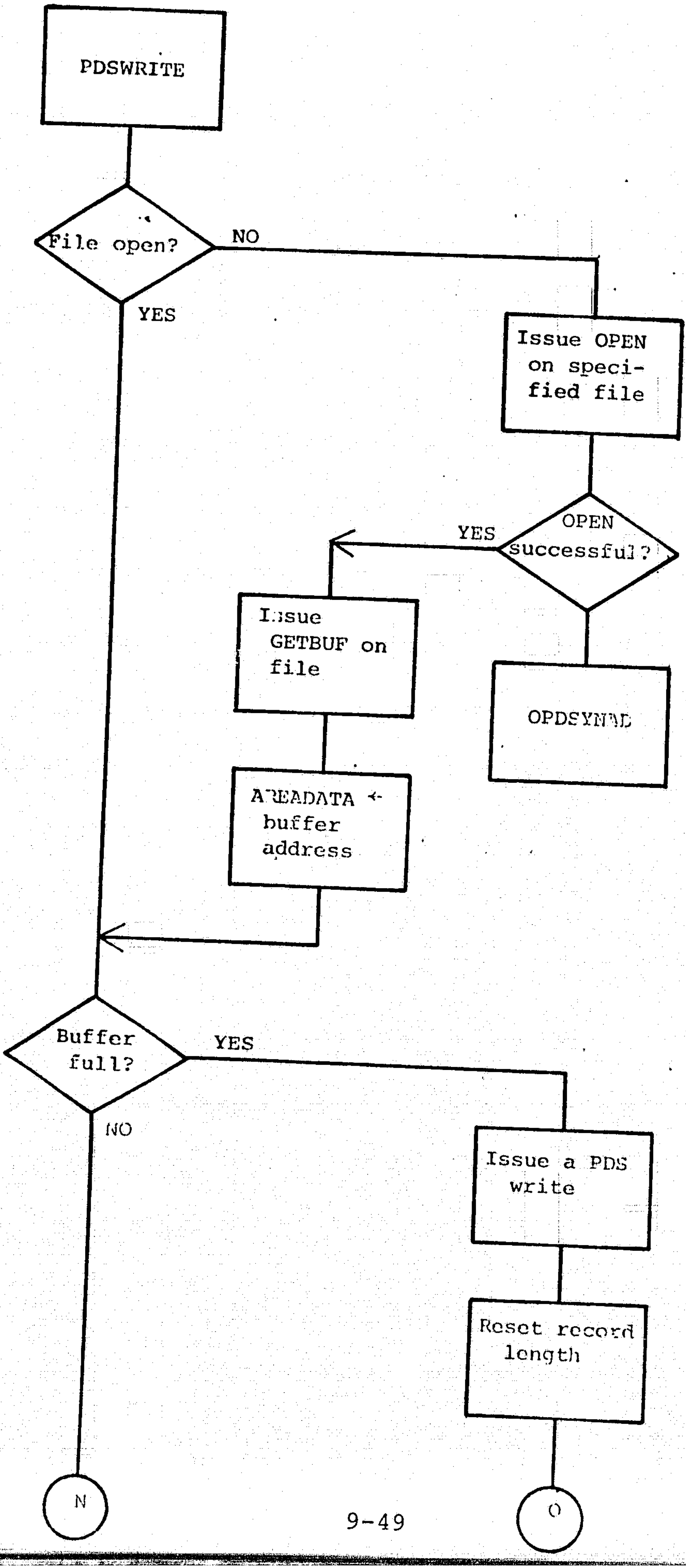

J
K
FULLCK
LINECT > LINELIM?
YES
EJECT
NO
TESTZL
Blank line to follow?
NO
PUT2
YES
Move CCB to output buffer
TESTZL1
Move 132 blanks to output buffer
EXIT

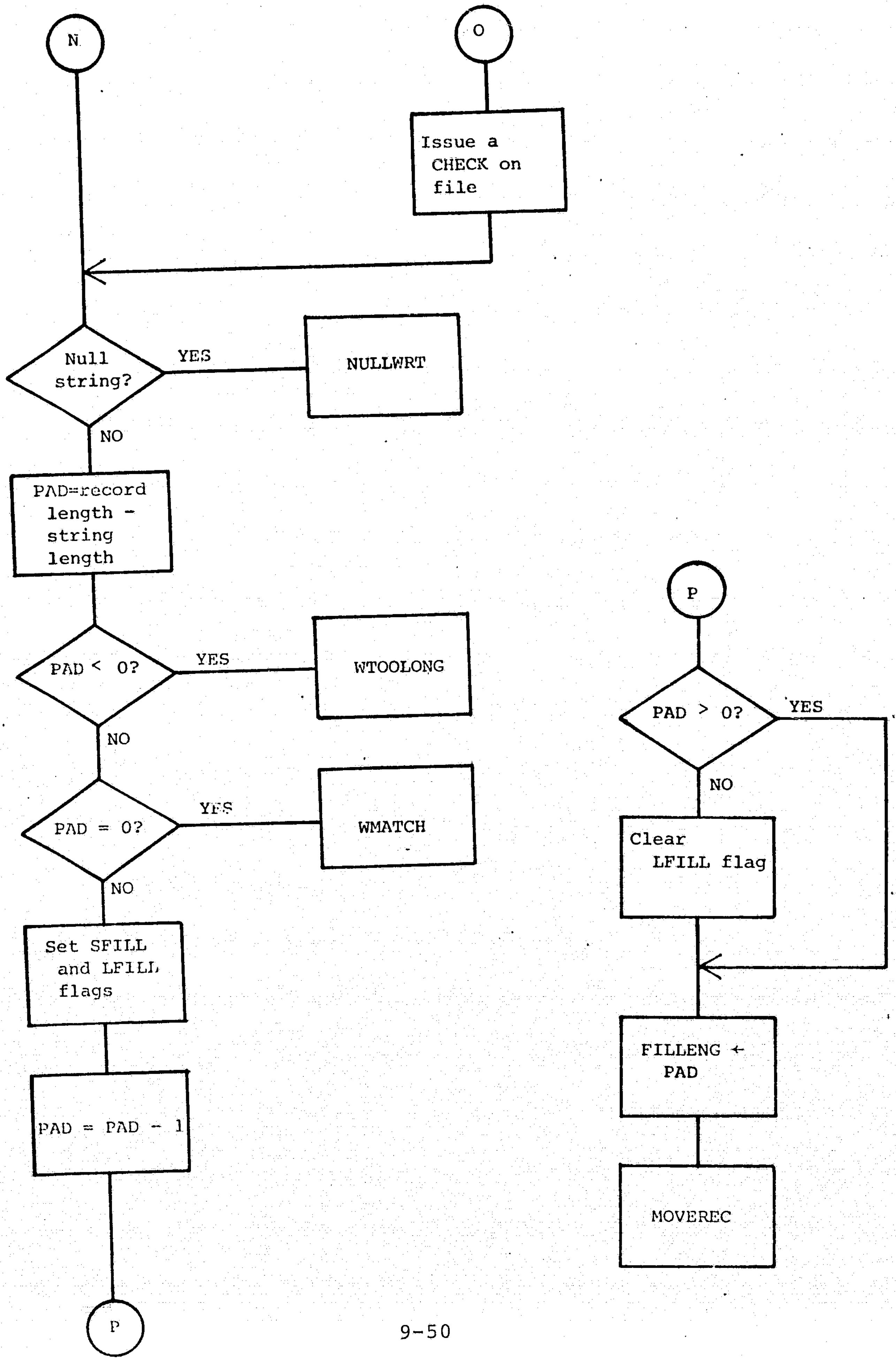

PUTGT1
Clear OUTZBIT flag
PUT2
PAD=record length - string length
PAD < 0?
YES
NO
Truncate string length to record length
PAD = 0?
YES
NO
Clear SFILL and LFILL flags
Set SFILL and LFILL flags
PAD ← PAD-1
L


L
PAD > 0?
YES
NO
Clear LFILL flag
FILLENG ← PAD
OUTZBIT flag set?
NO
MOVEIT2
YES
Move CCB to output buffer and increment buffer pointer
MOVEIT2
Move string to buffer area
M

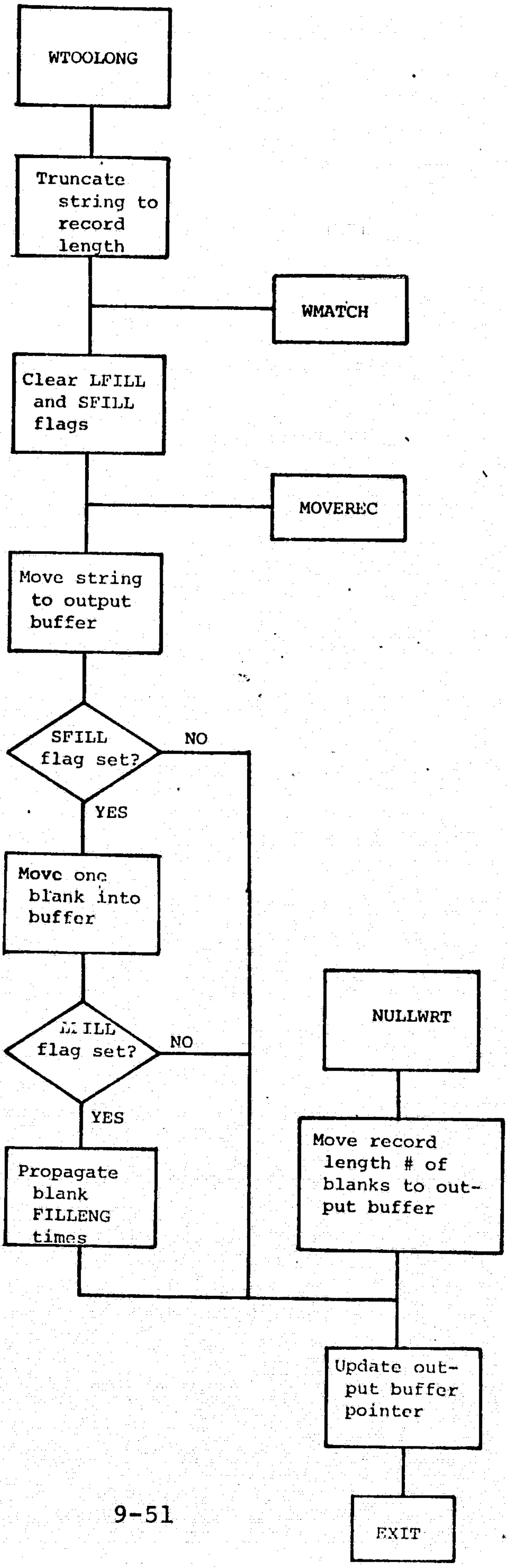

M
SFILL flag set?
NO
EXIT
YES
Move in one blank to buffer
LFILL flag set?
NO
EXIT
YES
Propagate blank FILLENG times
EXIT

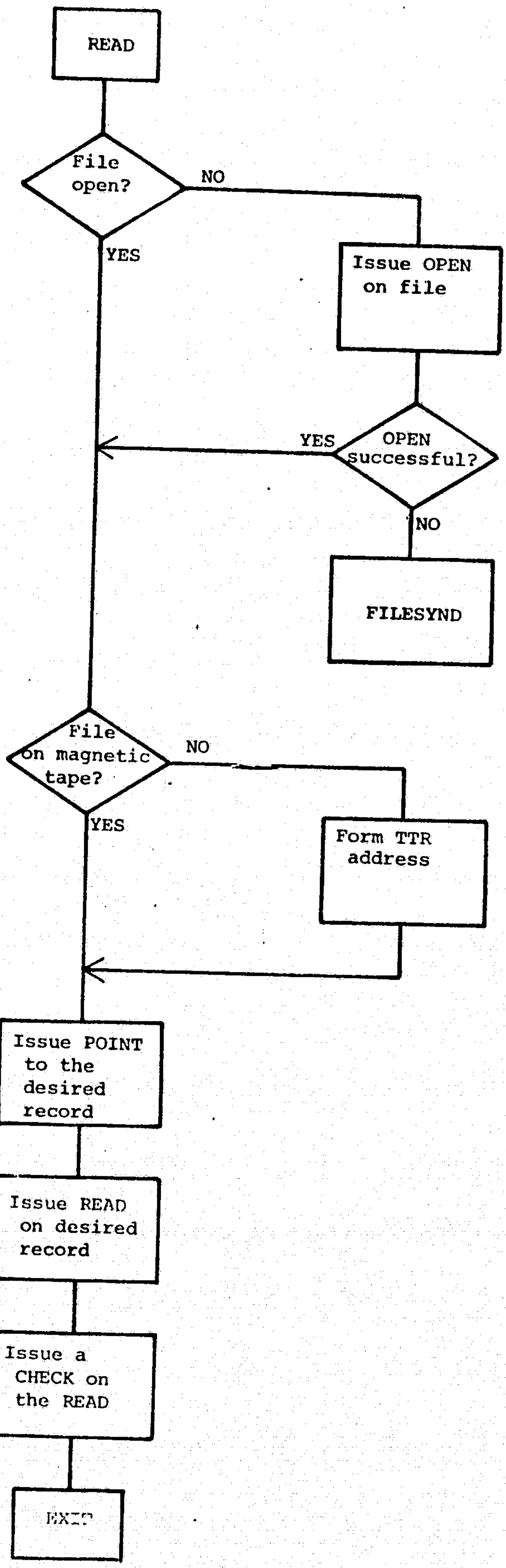

NULLPUT
OUTPUTO
NO
YES
Move LRECL blanks to output buffer
Set up for blank line
EXIT
SP1

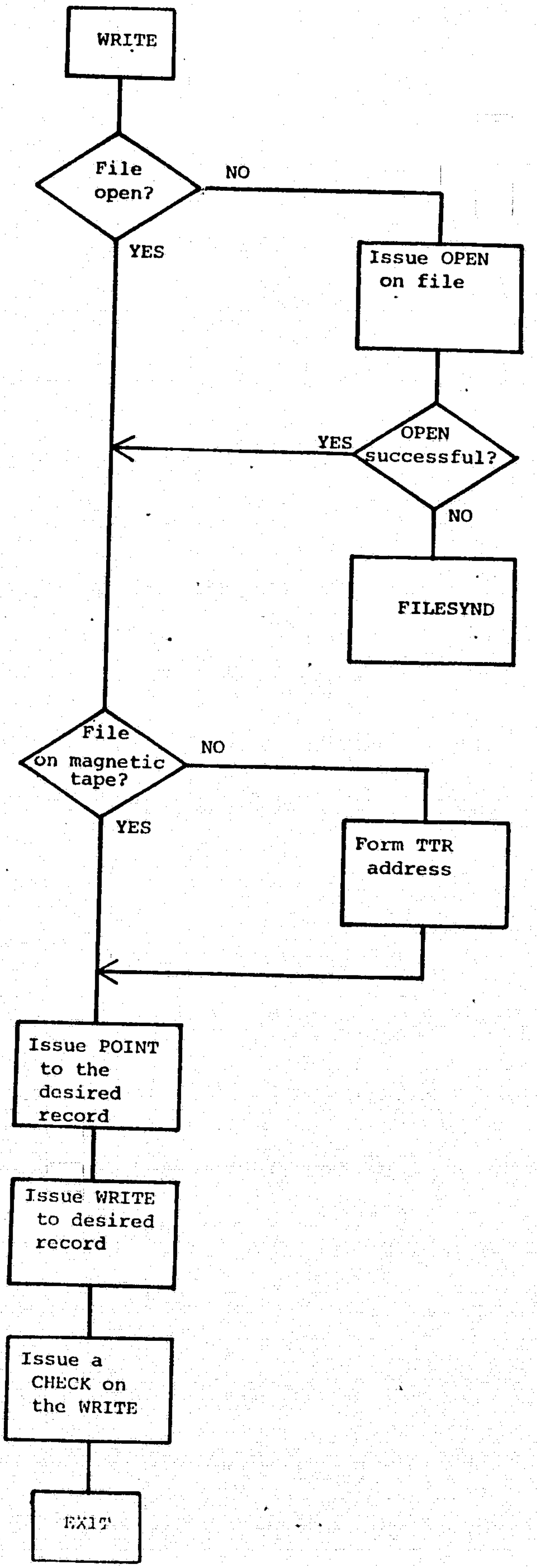

PDSWRITE
File open?
NO
YES
Issue OPEN on specified file
OPEN successful?
YES
Issue GETBUF on file
OPDSYNAD
AREADATA ← buffer address
Buffer full?
YES
NO
Issue a PDS write
Reset record length
N
O

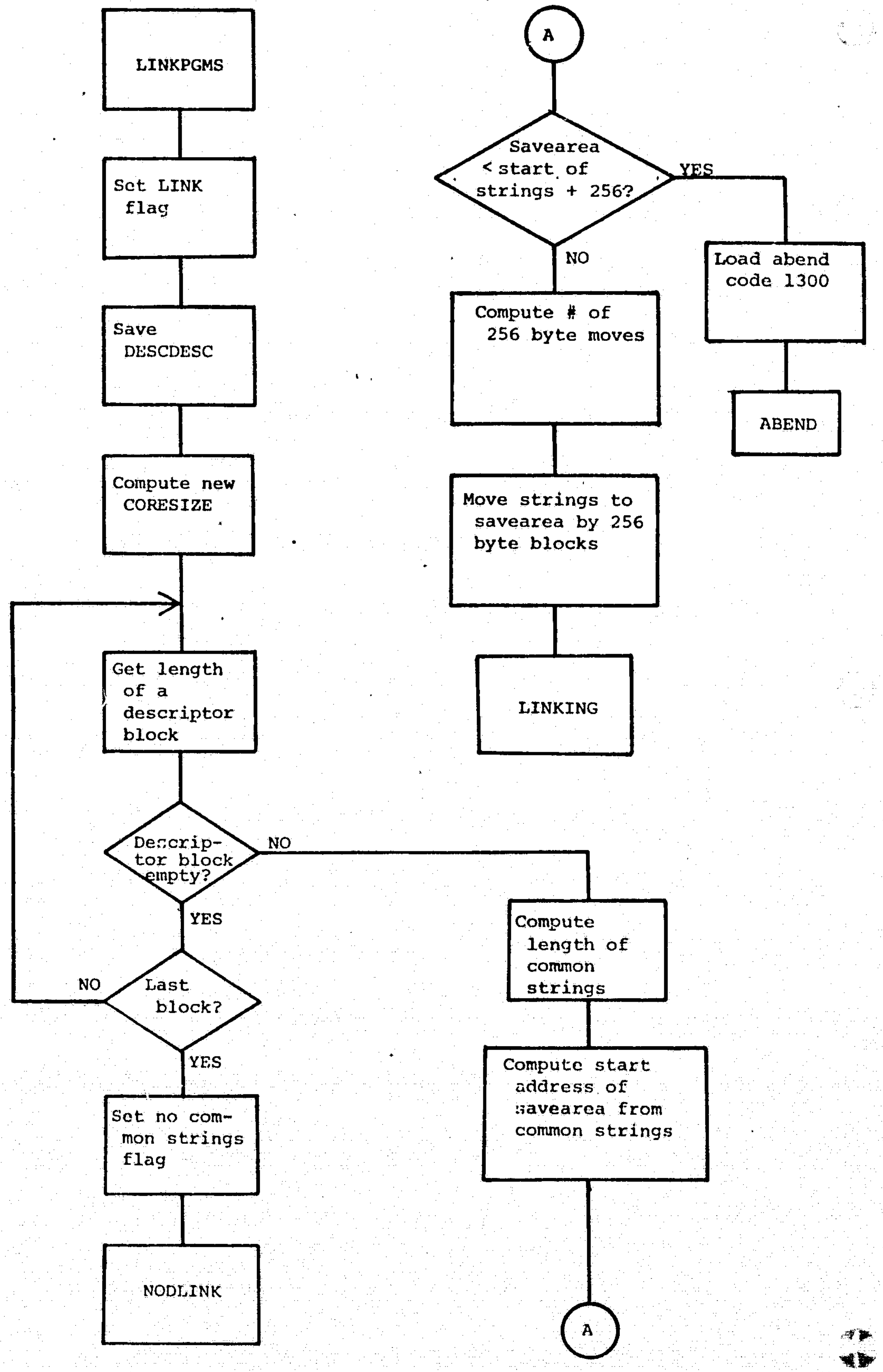


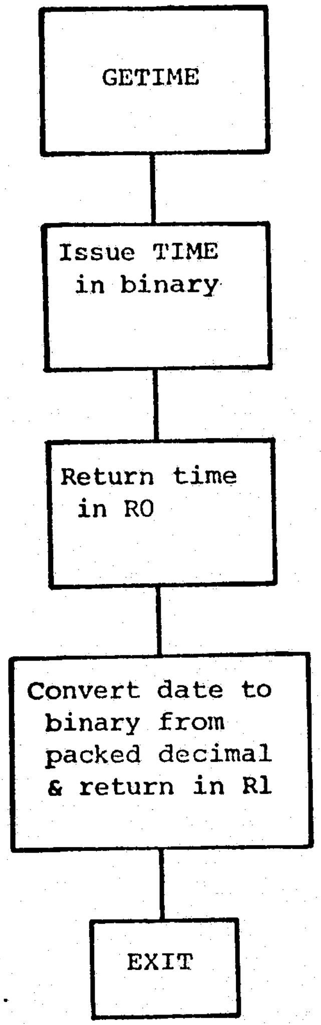
N
O
Issue a CHECK on file
Null string?
YES
NULLWRT
NO
PAD=record length - string length
PAD < 0?
YES
WTOOLONG
NO
PAD = 0?
YES
WMATCH
NO
Set SFILL and LFILL flags
PAD = PAD - 1
P
P
PAD > 0?
YES
NO
Clear LFILL flag
FILLENG ← PAD
MOVEREC

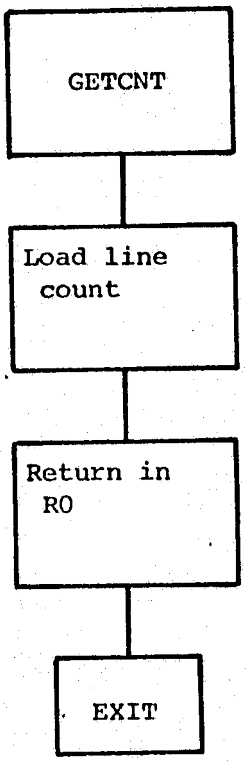

WTOOLONG
Truncate string to record length
WMATCH
Clear LFILL and SFILL flags
MOVEREC
Move string to output buffer
SFILL flag set?
NO
YES
Move one blank into buffer
LFILL flag set?
NO
YES
Propagate blank FILLENG times
NULLWRT
Move record length # of blanks to output buffer
Update output buffer pointer
EXIT

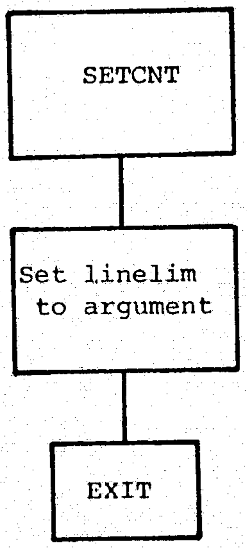

READ
File open?
NO
YES
Issue OPEN on file
OPEN successful?
YES
NO
FILESYND
File on magnetic tape?
NO
YES
Form TTR address
Issue POINT to the desired record
Issue READ on desired record
Issue a CHECK on the READ
EXIT

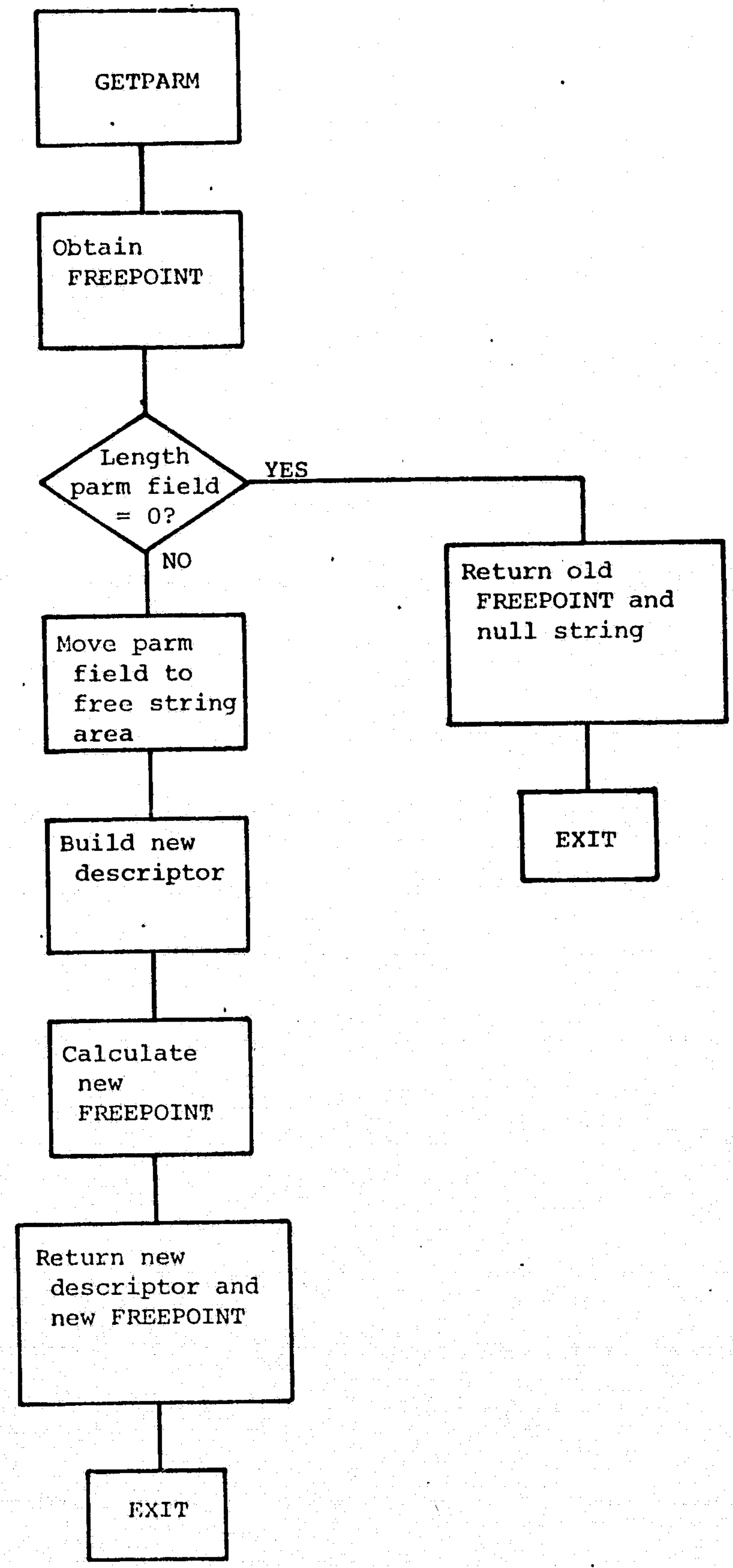

WRITE
File open?
NO
YES
Issue OPEN on file
OPEN successful?
YES
NO
FILESYND
File on magnetic tape?
NO
YES
Form TTR address
Issue POINT to the desired record
Issue WRITE to desired record
Issue a CHECK on the WRITE
EXIT

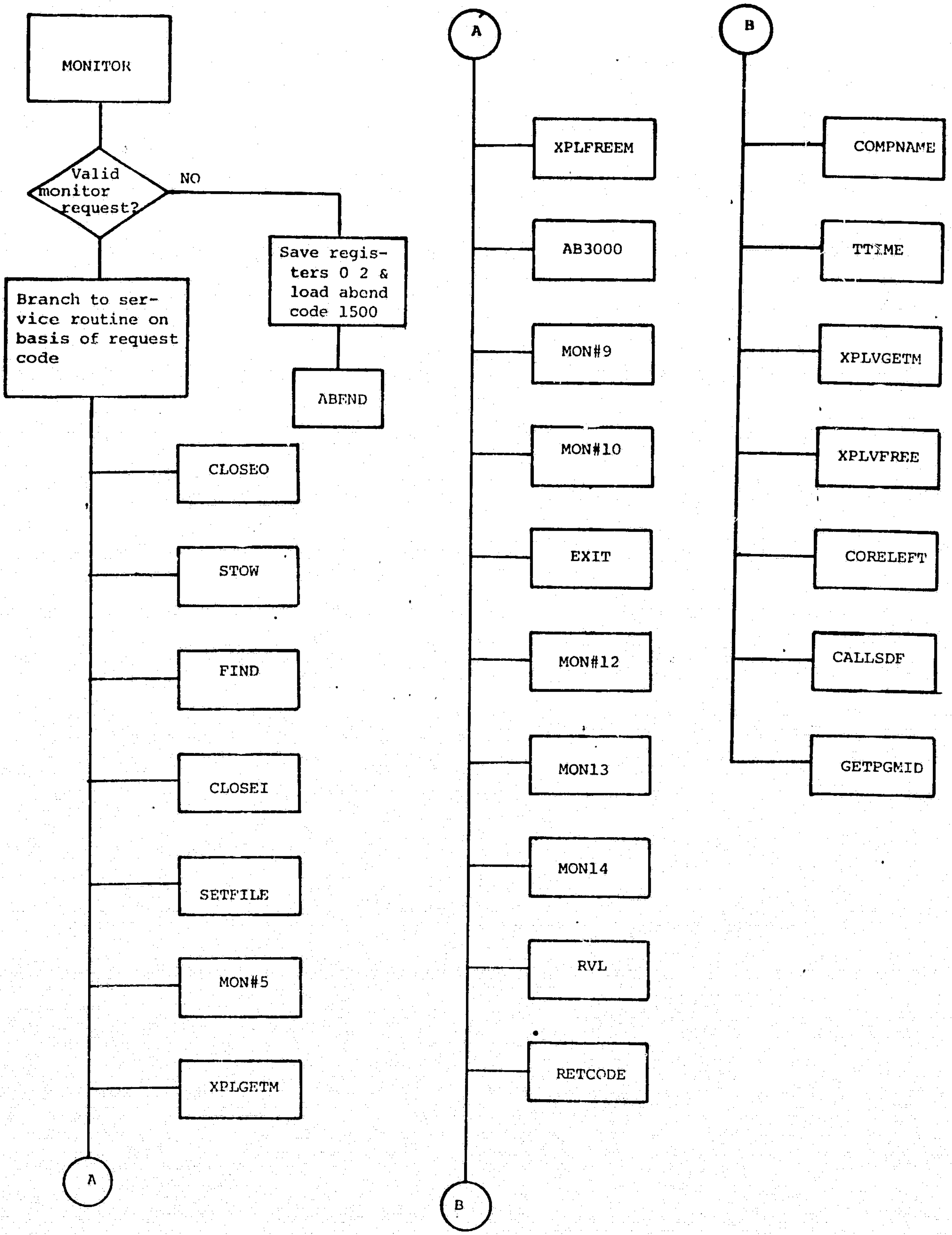

LINKPGMS
Set LINK flag
Save DESCDESC
Compute new CORESIZE
Get length of a descriptor block
Descrip-tor block empty?
NO
YES
Last block?
NO
YES
Set no com-mon strings flag
NODLINK
Compute length of common strings
Compute start address of savearea from common strings
A
A
Savearea < start of strings + 256?
YES
NO
Load abend code 1300
ABEND
Compute # of 256 byte moves
Move strings to savearea by 256 byte blocks
LINKING

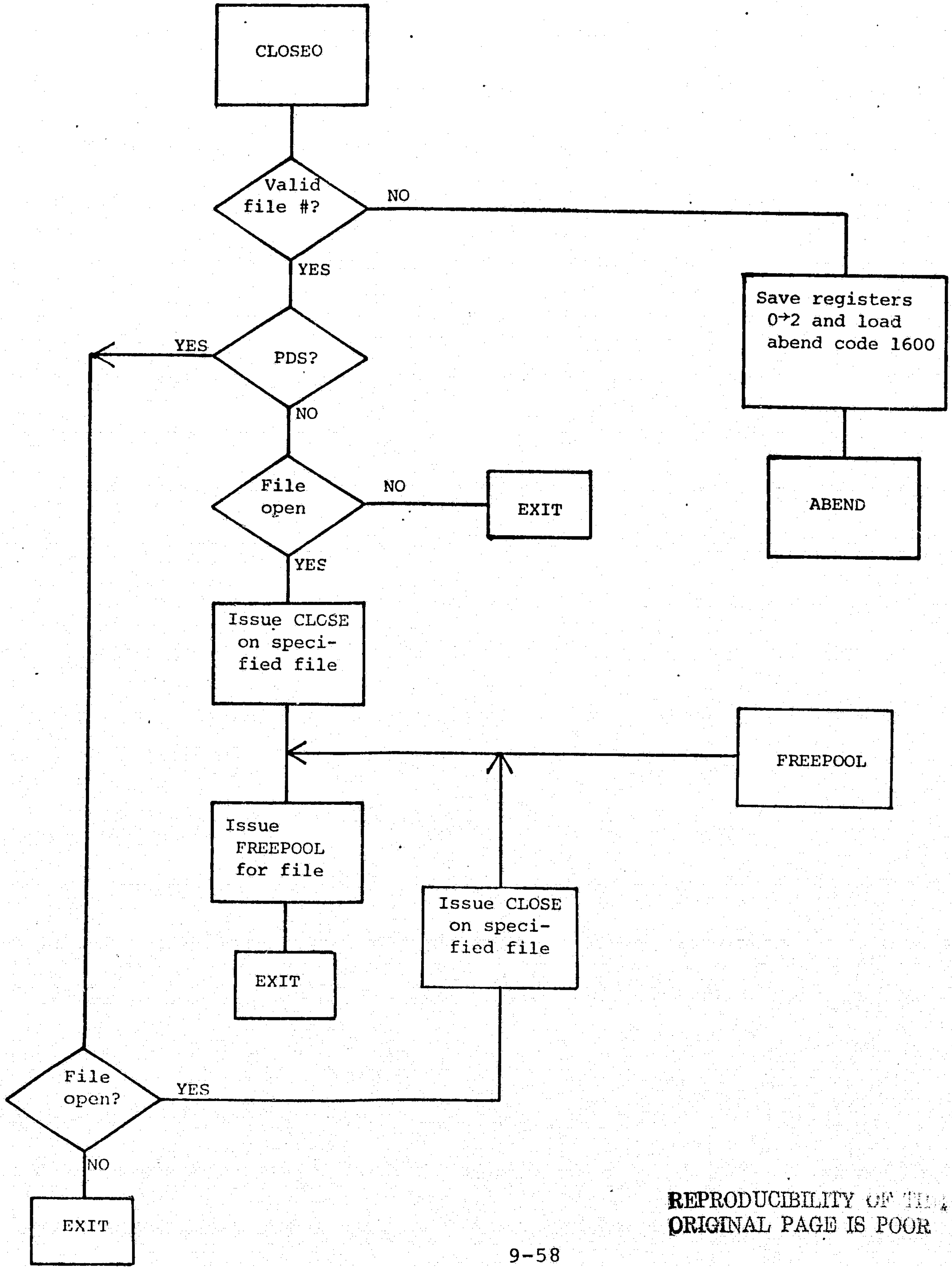

GETIME
Issue TIME in binary
Return time in RO
Convert date to binary from packed decimal & return in Rl
EXIT


GETCNT
Load line count
Return in RO
EXIT

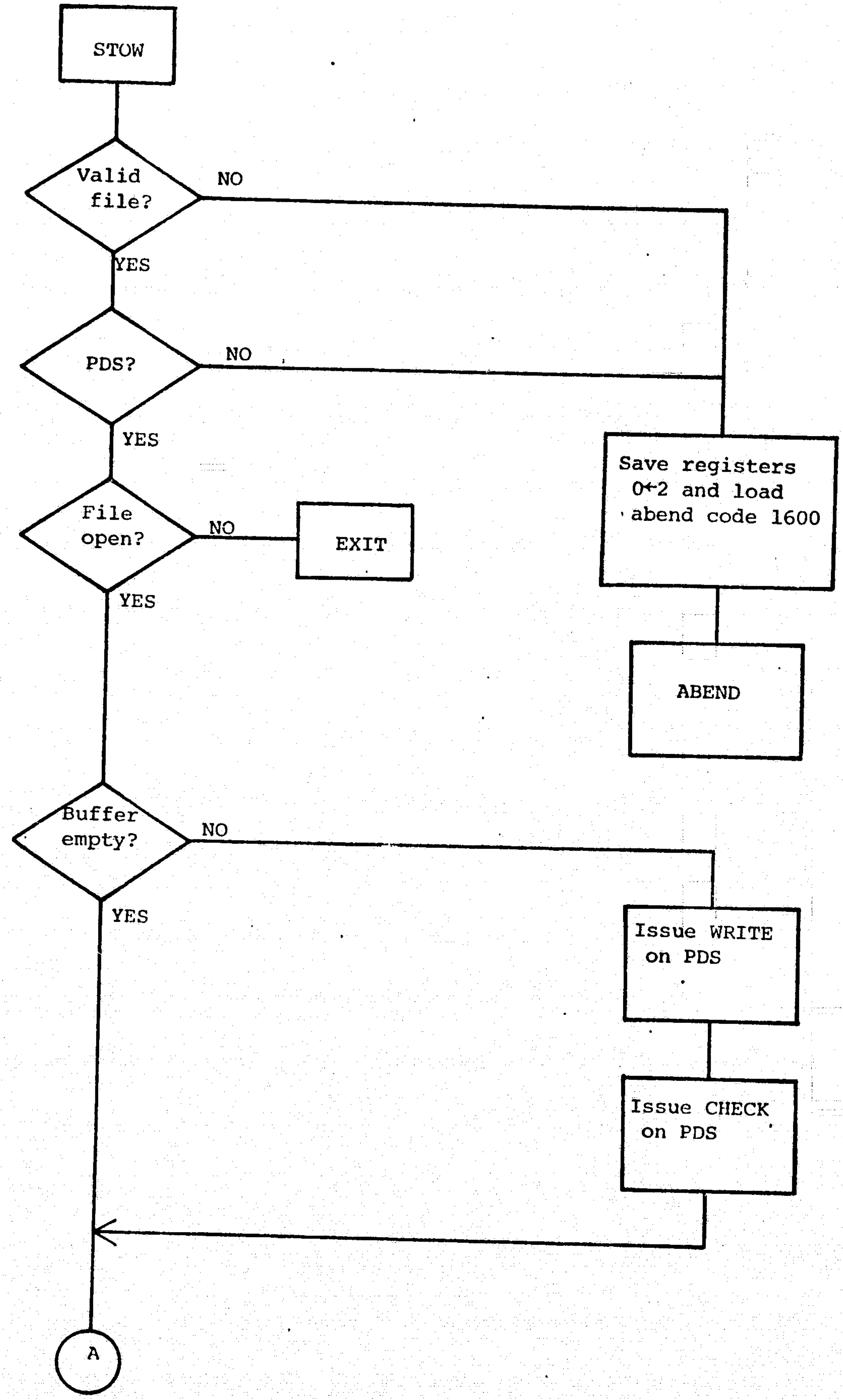

SETCNT
Set linelim to argument
EXIT

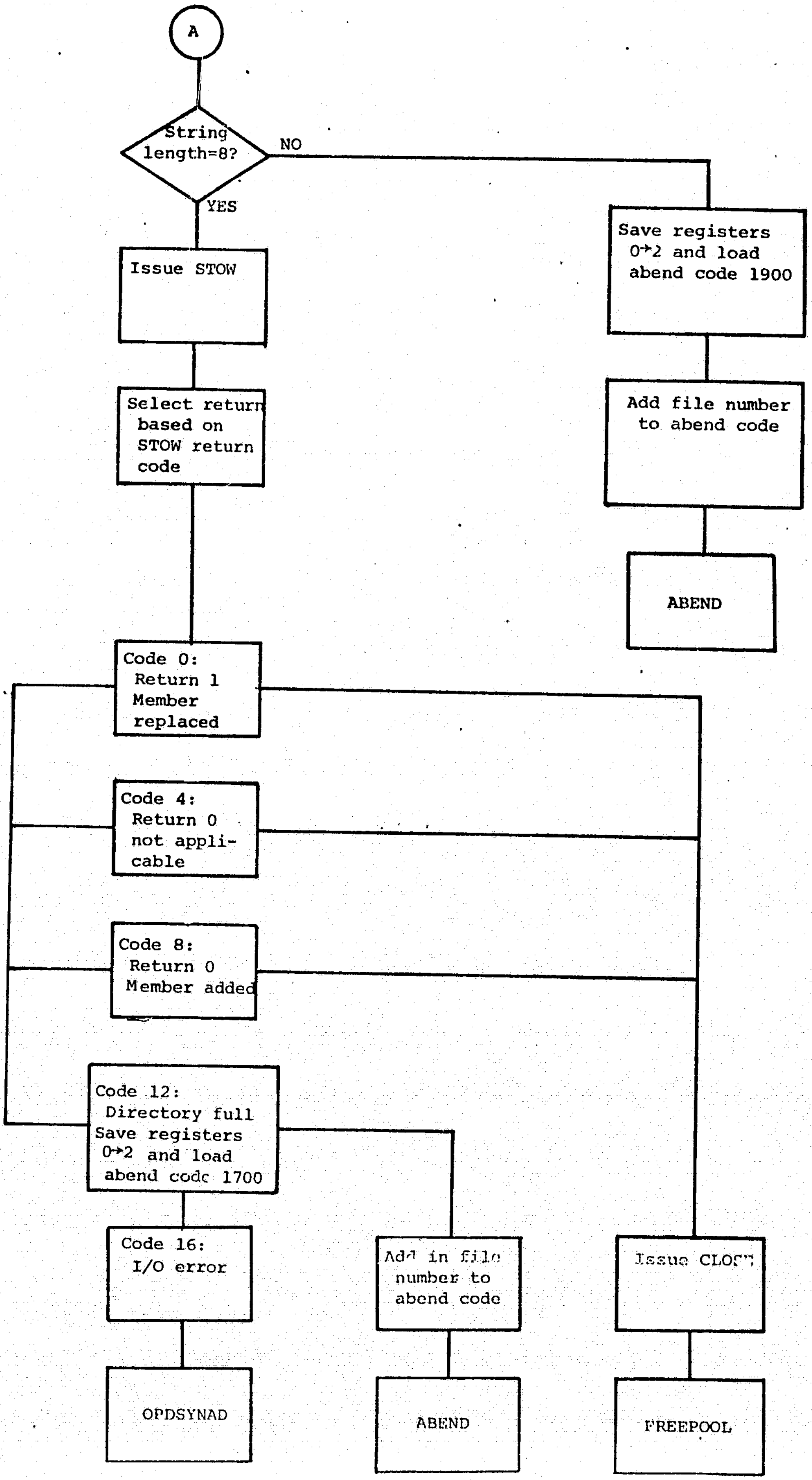

GETPARM
Obtain FREEPOINT
Length parm field = 0?
YES
NO
Return old FREEPOINT and null string
EXIT
Move parm field to free string area
Build new descriptor
Calculate new FREEPOINT
Return new descriptor and new FREEPOINT
EXIT

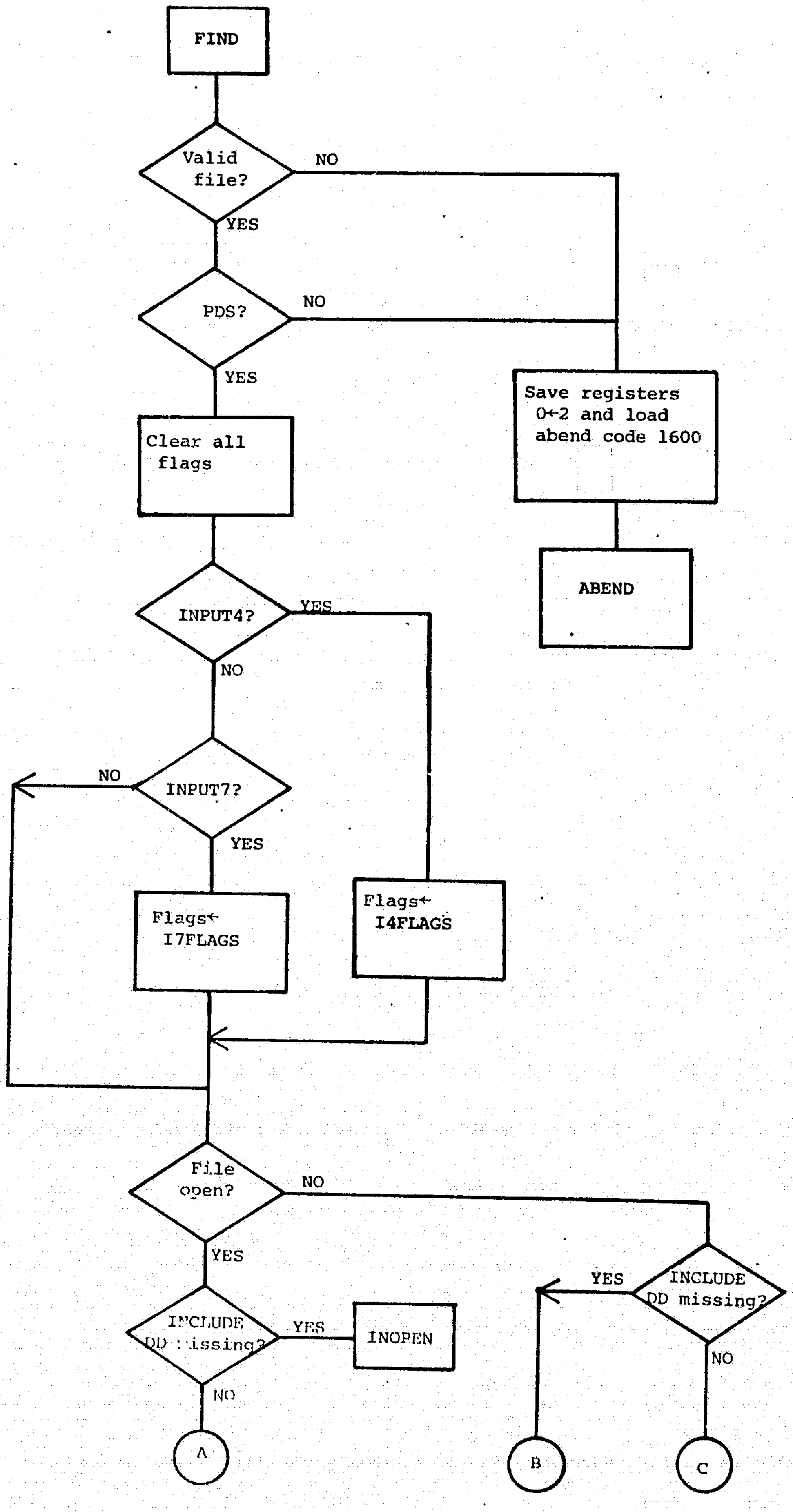

MONITOR
Valid monitor request?
NO
Save registers 0 2 & load abend code 1500
ABEND
Branch to service routine on basis of request code
CLOSEO
STOW
FIND
CLOSEI
SETFILE
MON#5
XPLGETM
A
A
XPLFREEM
AB3000
MON#9
MON#10
EXIT
MON#12
MON13
MON14
RVL
RETCODE
B
B
COMPNAME
TTIME
XPLVGETM
XPLVFREE
CORELEFT
CALLSDF
GETPGMID

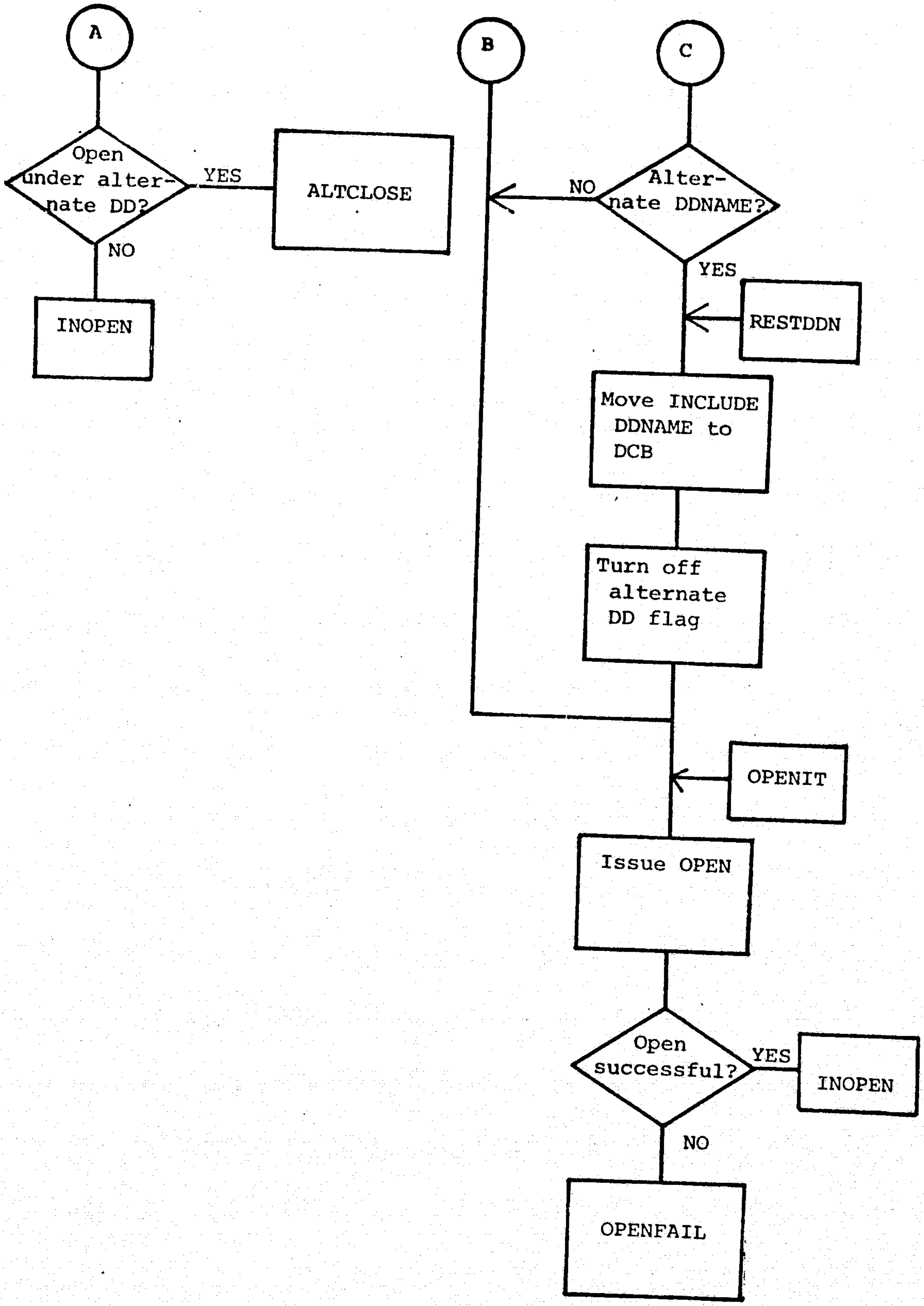


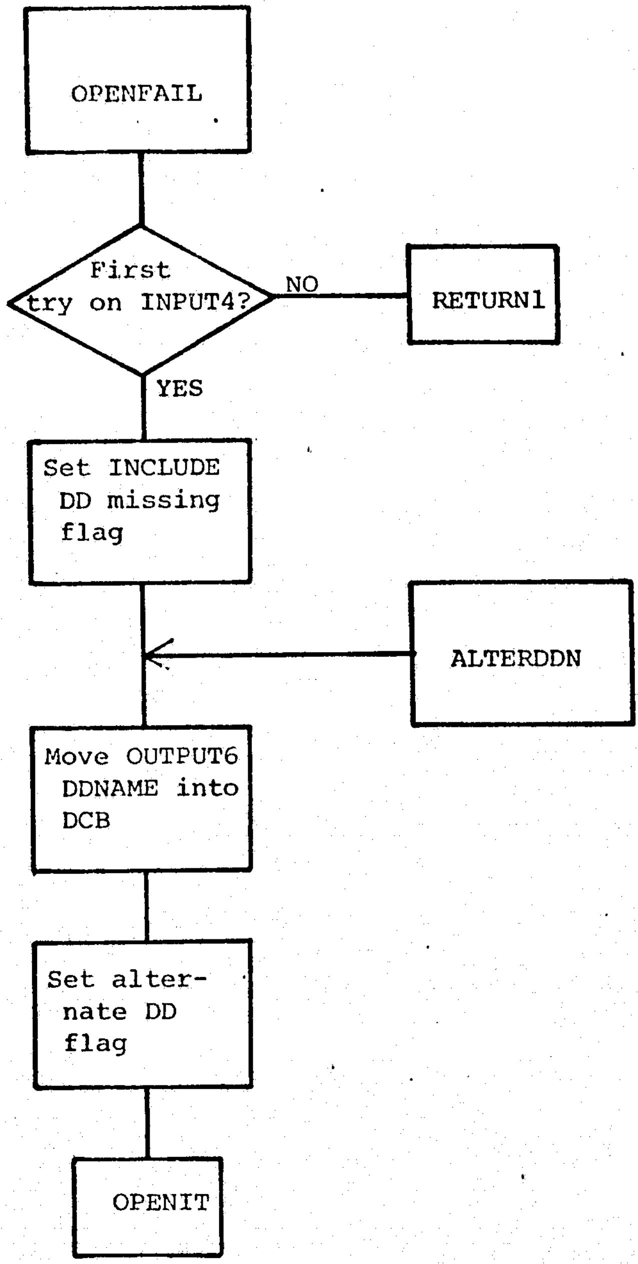
REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

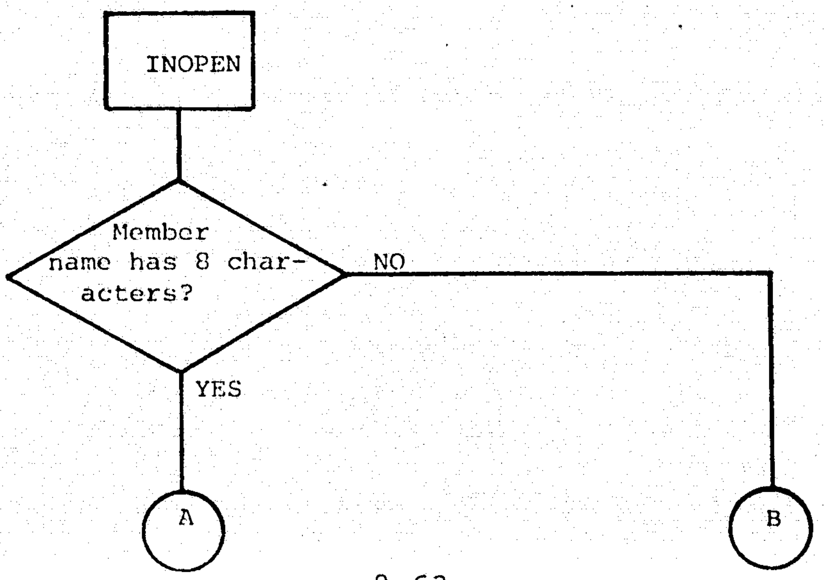

STOW
Valid file?
NO
YES
PDS?
NO
YES
File open?
NO
EXIT
YES
Save registers 0←2 and load abend code 1600
ABEND
Buffer empty?
NO
YES
Issue WRITE on PDS
Issue CHECK on PDS
A

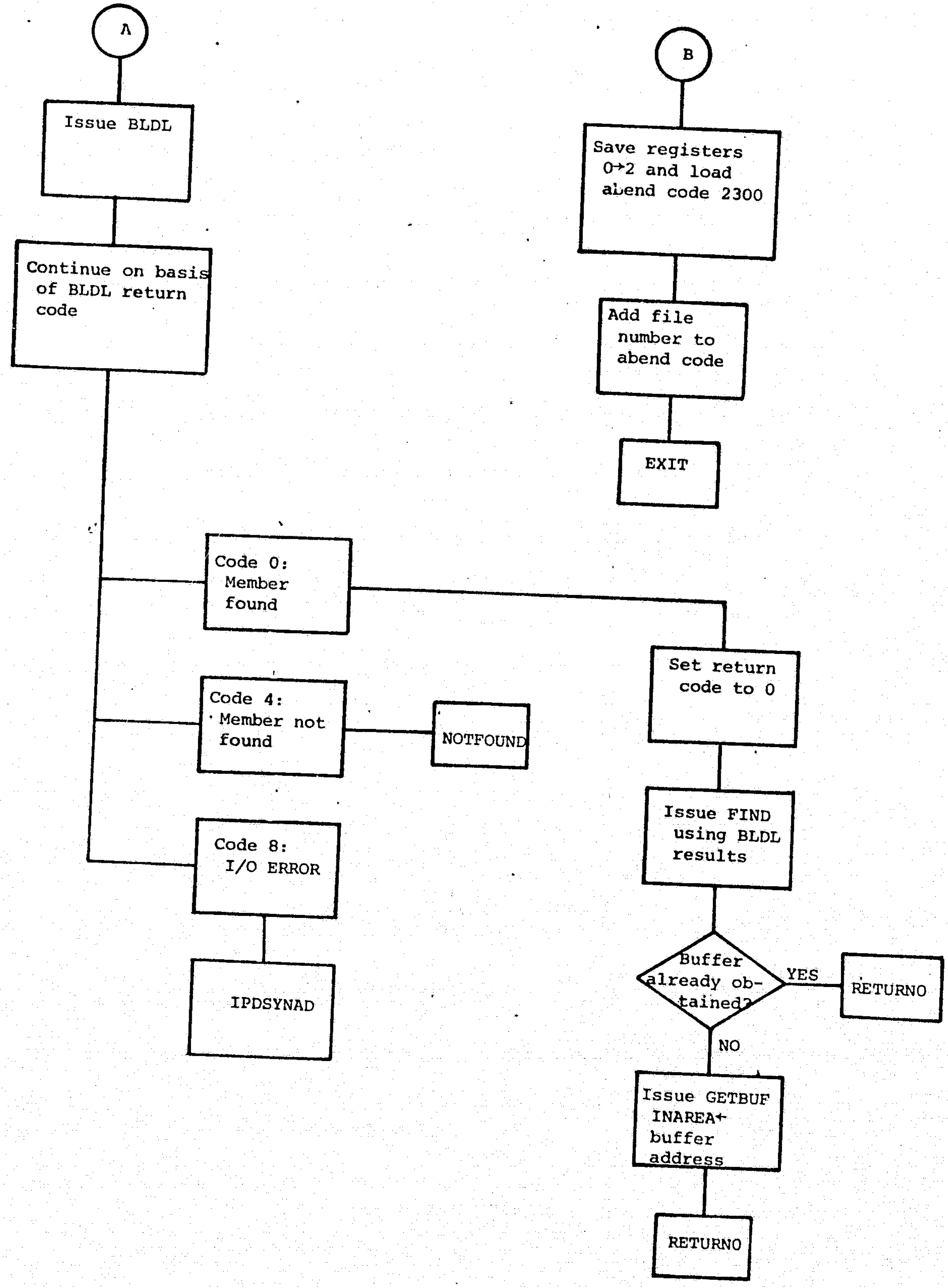

A
String length=8?
NO
YES
Issue STOW
Select return based on STOW return code
Save registers 0→2 and load abend code 1900
Add file number to abend code
ABEND
Code 0: Return 1 Member replaced
Code 4: Return 0 not applicable
Code 8: Return 0 Member added
Code 12: Directory full Save registers 0→2 and load abend code 1700
Code 16: I/O error
Add in file number to abend code
Issue CLOSE
OPDSYNAD
ABEND
FREEPOOL

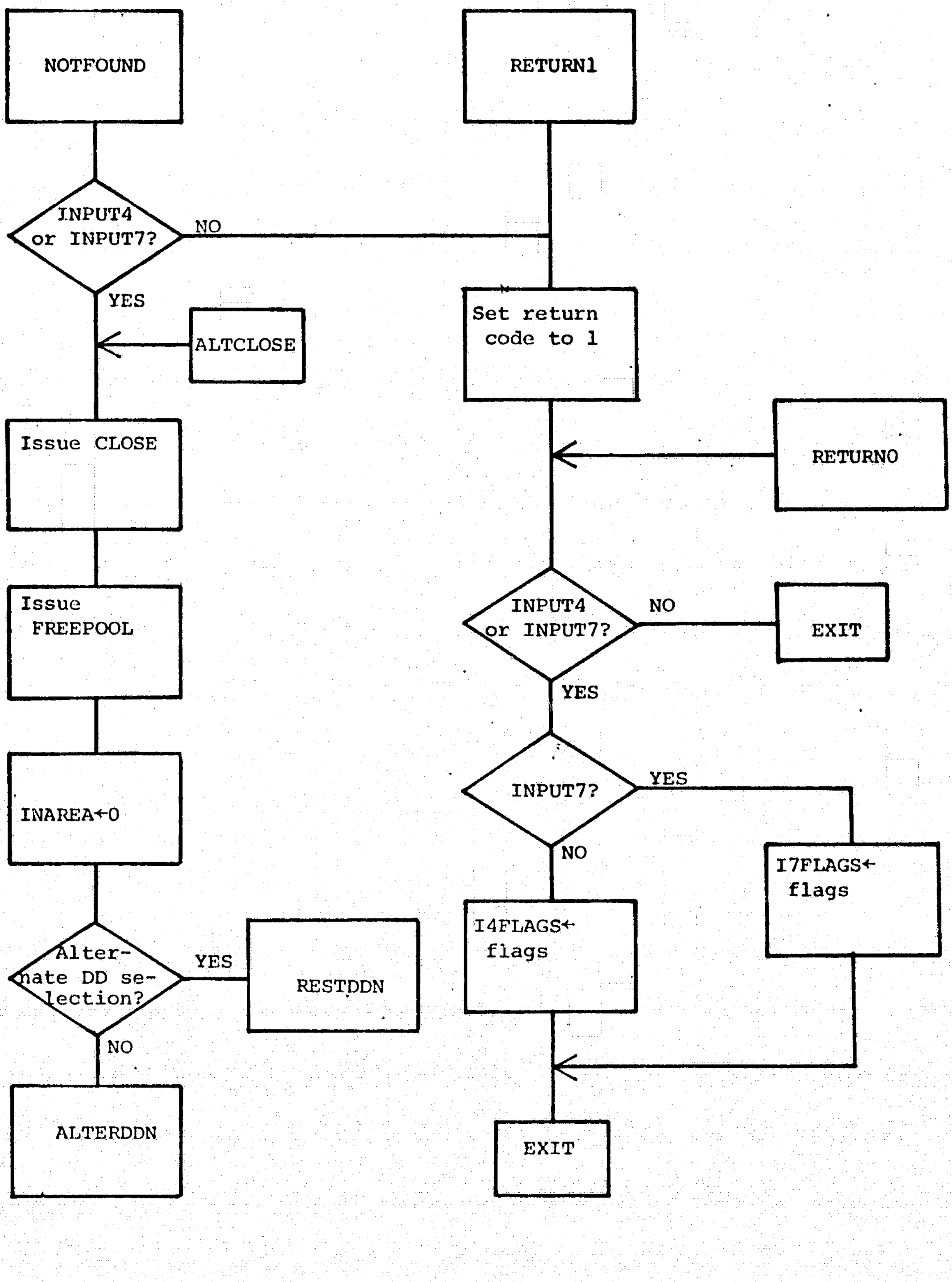

FIND
Valid file?
NO
YES
PDS?
NO
YES
Save registers 0←2 and load abend code 1600
ABEND
Clear all flags
INPUT4?
YES
NO
INPUT7?
NO
YES
Flags← I7FLAGS
Flags← I4FLAGS
File open?
NO
YES
INCLUDE DD missing?
YES
INOPEN
NO
A
INCLUDE DD missing?
YES
NO
B
C

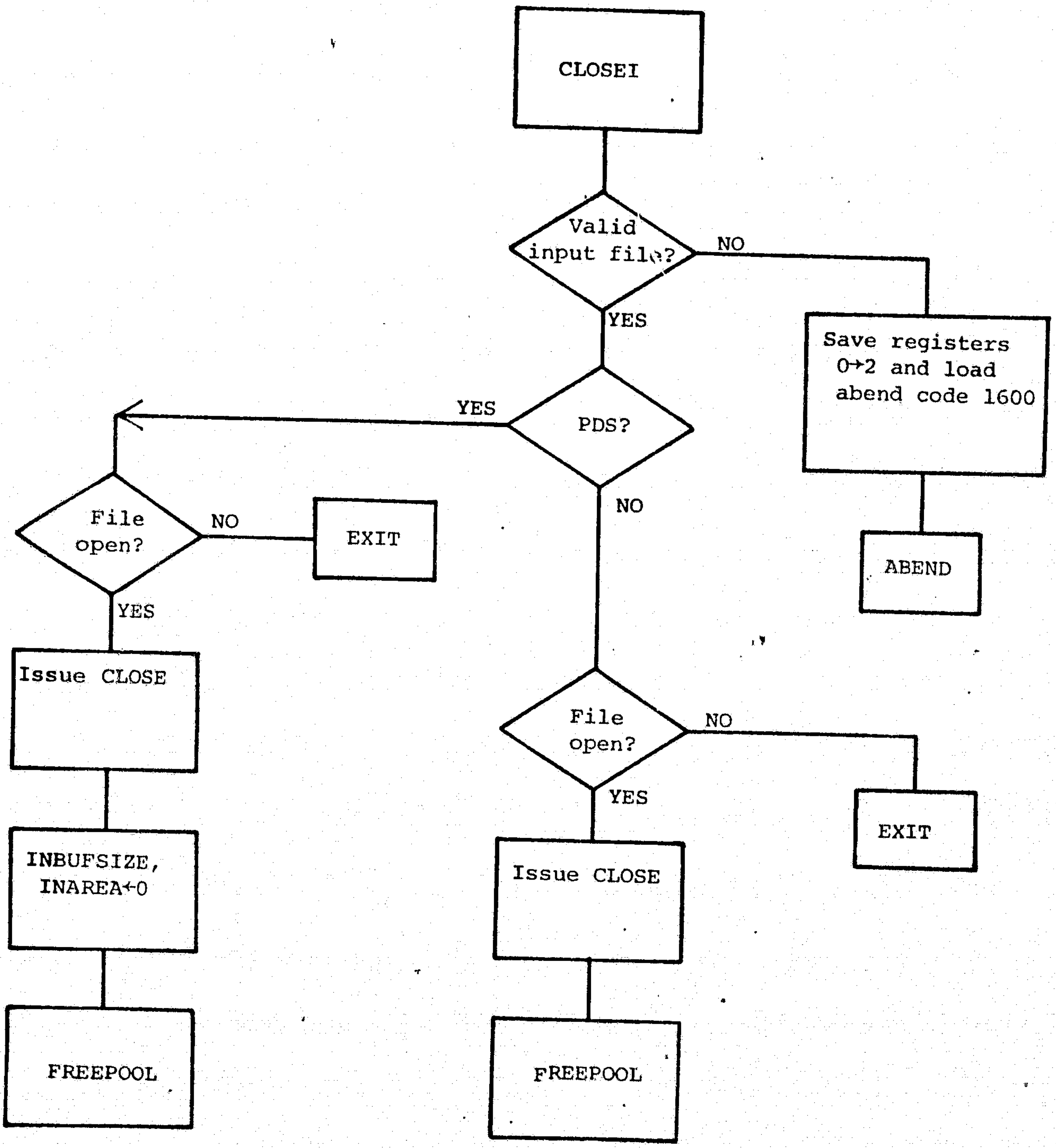

A
B
C
Open under alternate DD?
YES
ALTCLOSE
NO
INOPEN
Alternate DDNAME?
NO
YES
RESTDDN
Move INCLUDE DDNAME to DCB
Turn off alternate DD flag
OPENIT
Issue OPEN
Open successful?
YES
INOPEN
NO
OPENFAIL

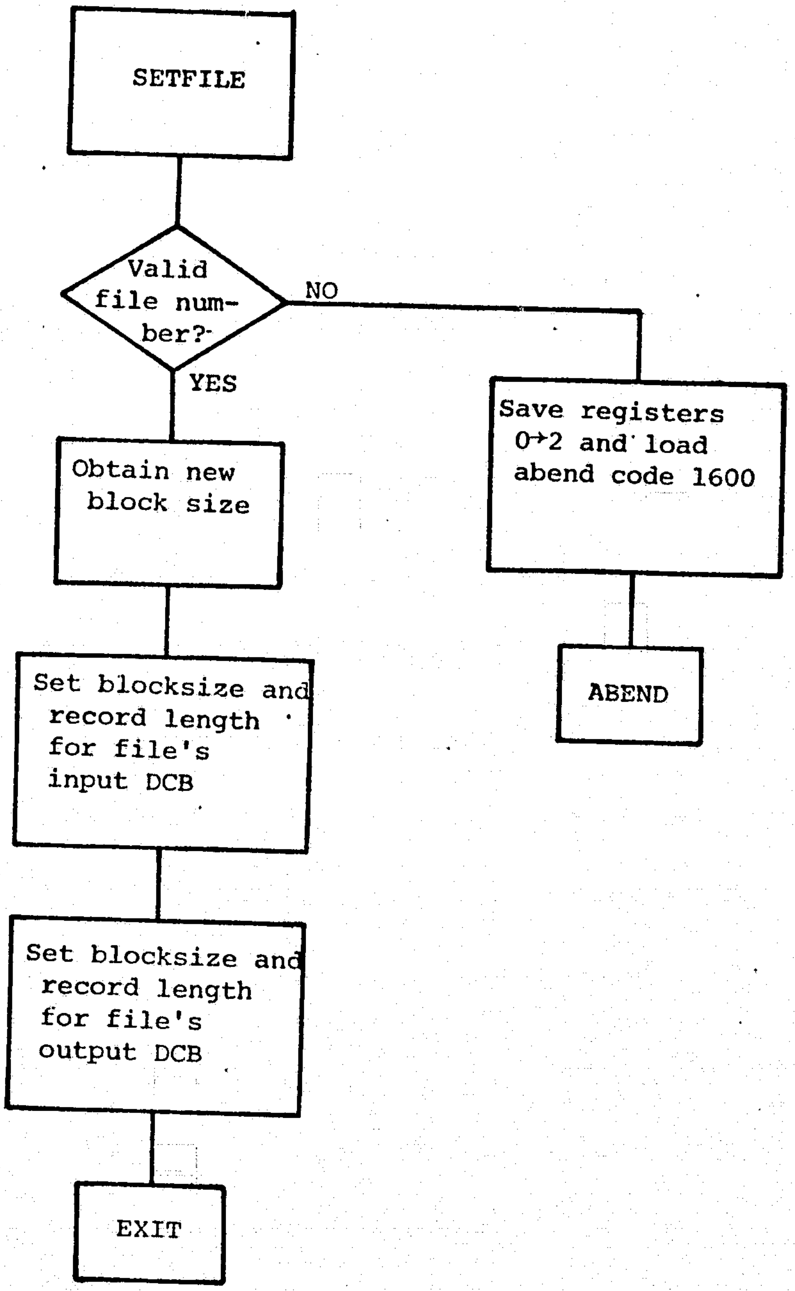

OPENFAIL
First try on INPUT4?
NO
RETURN1
YES
Set INCLUDE DD missing flag
ALTERDDN
Move OUTPUT6 DDNAME into DCB
Set alternate DD flag
OPENIT

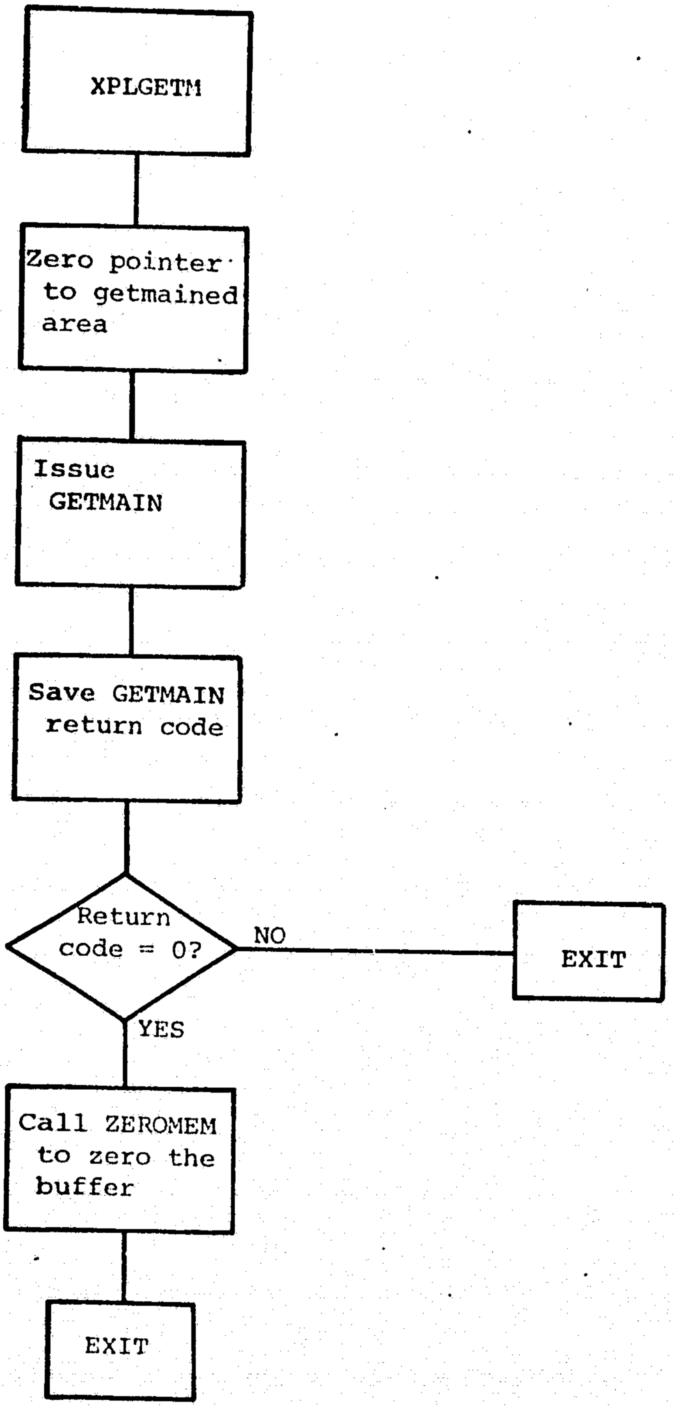

INOPEN
Member name has 8 characters?
NO
YES
A
B

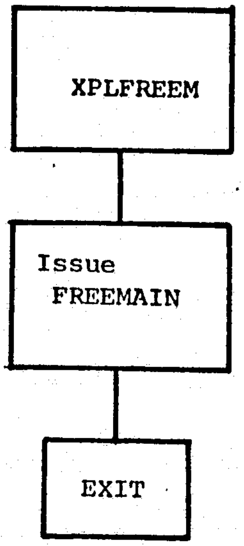

A
Issue BLDL
Continue on basis of BLDL return code
Code 0: Member found
Code 4: Member not found
NOTFOUND
Code 8: I/O ERROR
IPDSYNAD
B
Save registers 0→2 and load abend code 2300
Add file number to abend code
EXIT
Set return code to 0
Issue FIND using BLDL results
Buffer already obtained?
YES
RETURNO
NO
Issue GETBUF INAREA← buffer address
RETURNO

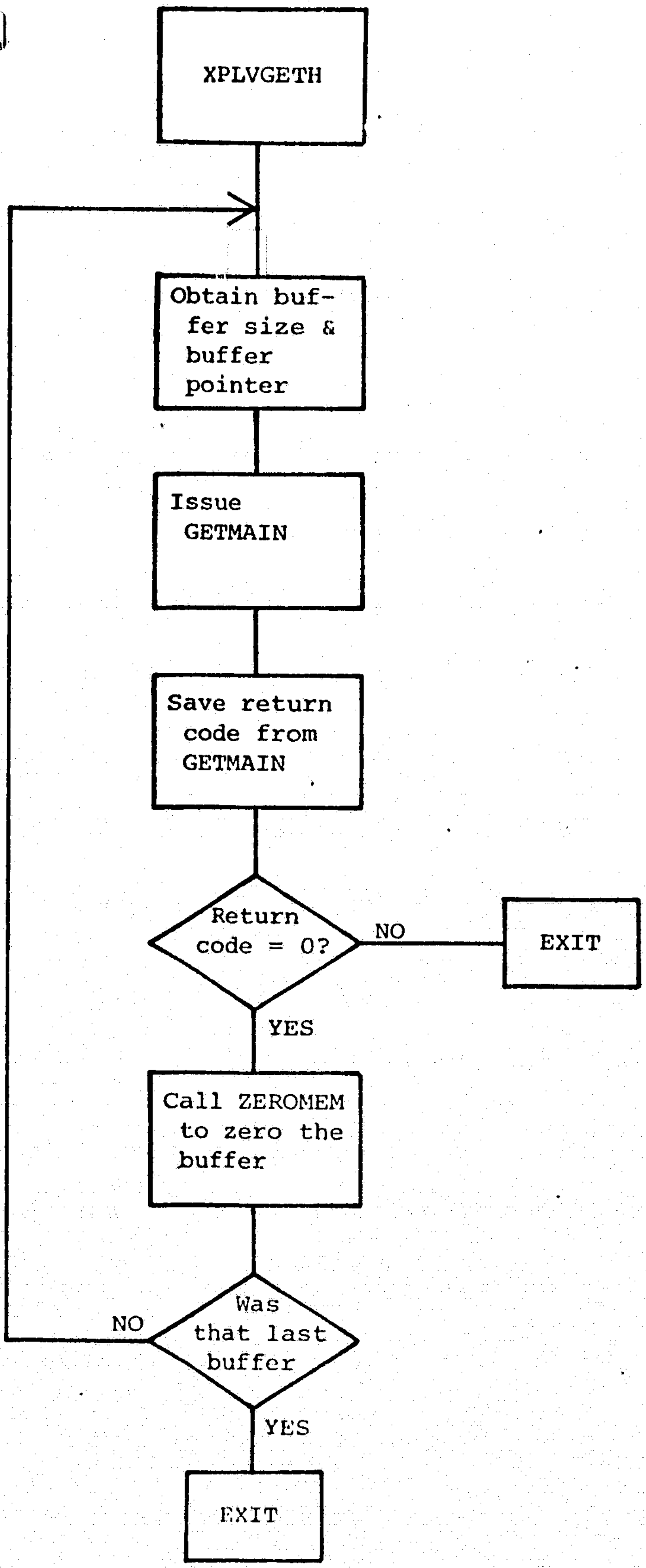

NOTFOUND
RETURN1
INPUT4 or INPUT7?
NO
YES
ALTCLOSE
Set return code to 1
RETURN0
Issue CLOSE
Issue FREEPOOL
INPUT4 or INPUT7?
NO
EXIT
YES
INPUT7?
YES
NO
I7FLAGS← flags
INAREA←0
I4FLAGS← flags
Alter-nate DD se-lection?
YES
RESTDDN
NO
ALTERDDN
EXIT

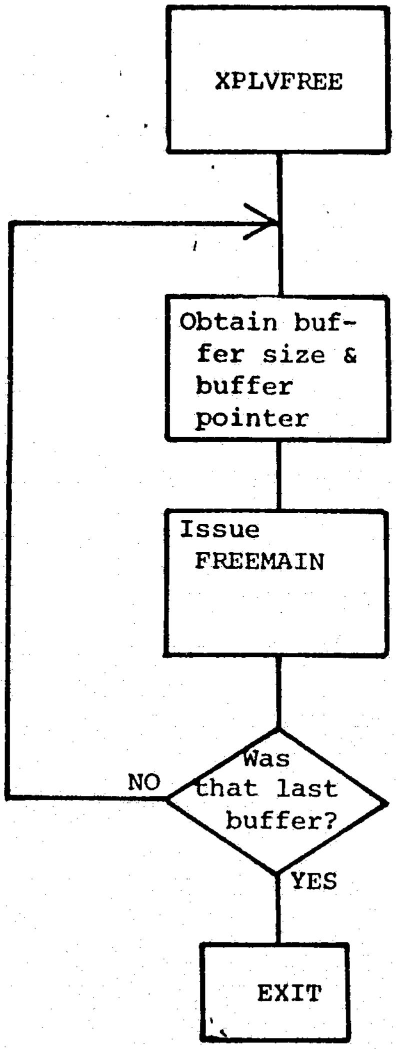

CLOSEI
Valid input file?
NO
YES
Save registers 0→2 and load abend code 1600
ABEND
PDS?
YES
NO
File open?
NO
EXIT
YES
Issue CLOSE
INBUFSIZE, INAREA←0
FREEPOOL
File open?
NO
EXIT
YES
Issue CLOSE
FREEPOOL

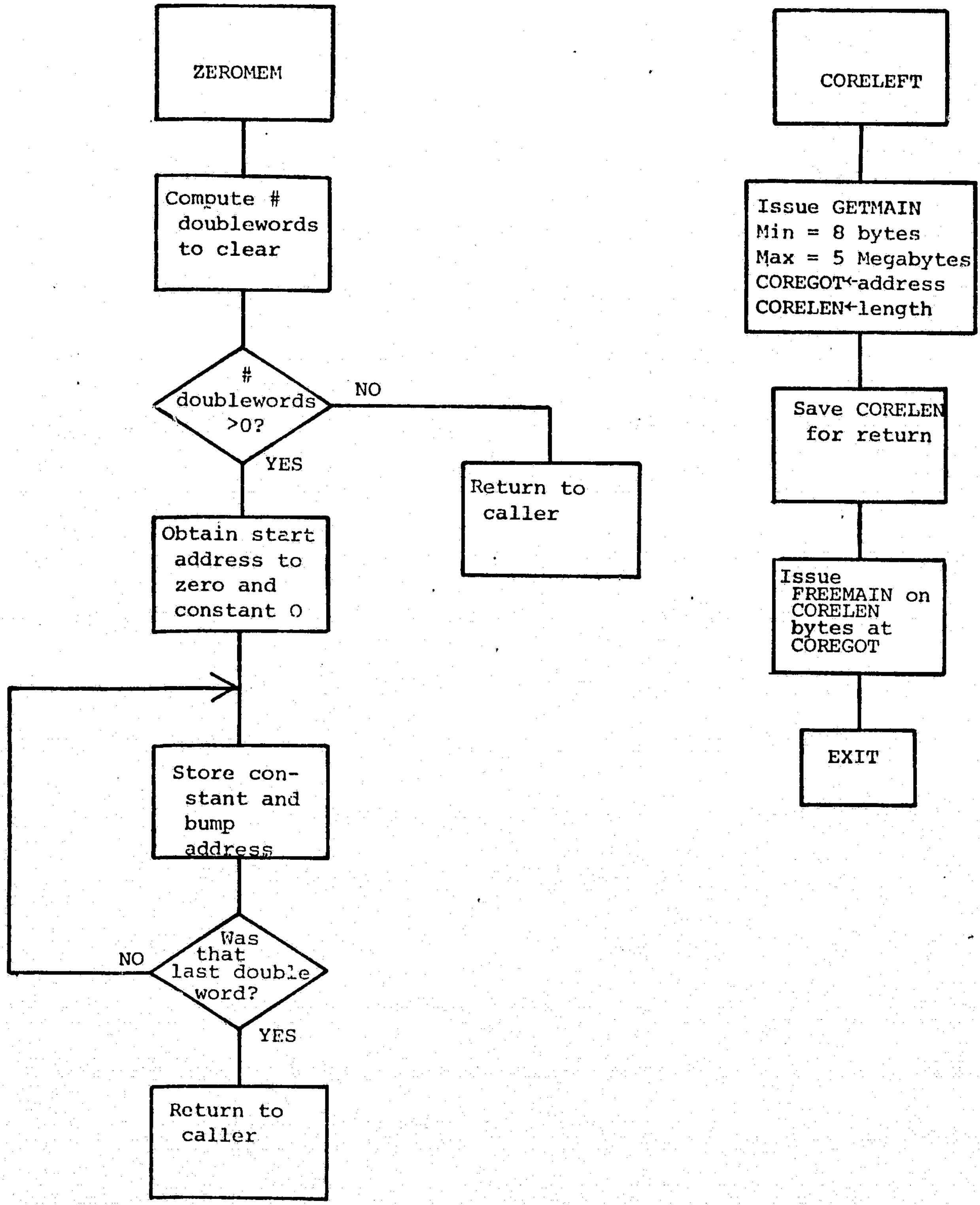

SETFILE
Valid file number?
NO
YES
Obtain new block size
Save registers 0→2 and load abend code 1600
ABEND
Set blocksize and record length for file's input DCB
Set blocksize and record length for file's output DCB
EXIT

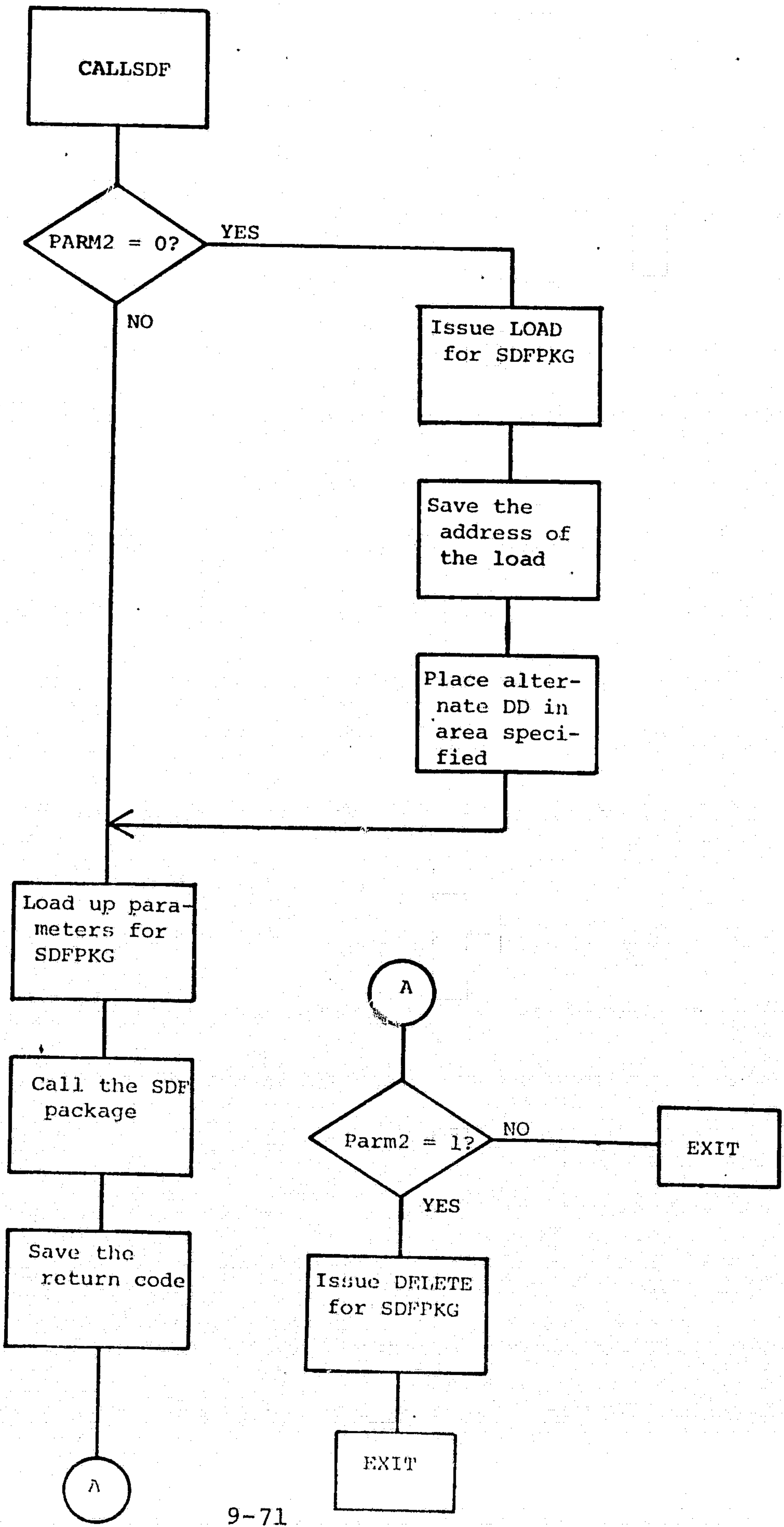

XPLGETM
Zero pointer to getmained area
Issue GETMAIN
Save GETMAIN return code
Return code = 0?
NO
EXIT
YES
Call ZEROMEM to zero the buffer
EXIT

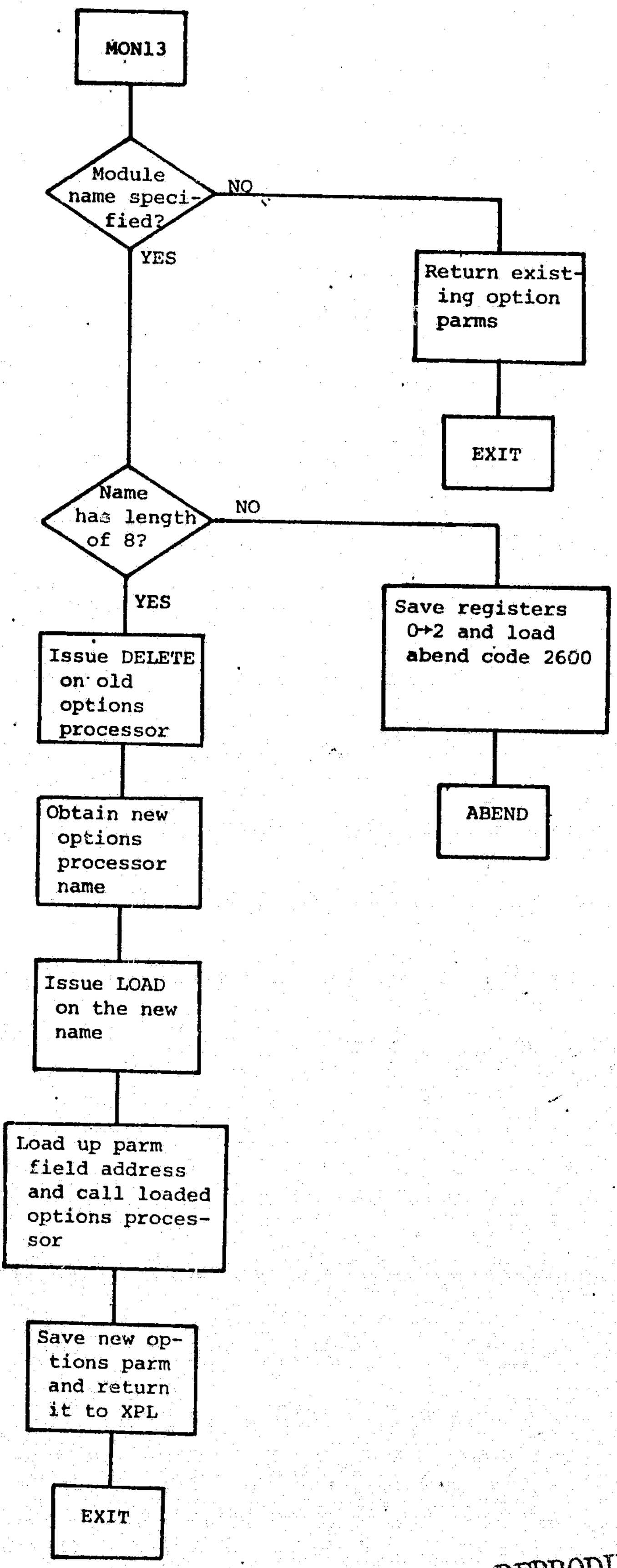

XPLFREEM
Issue FREEMAIN
EXIT

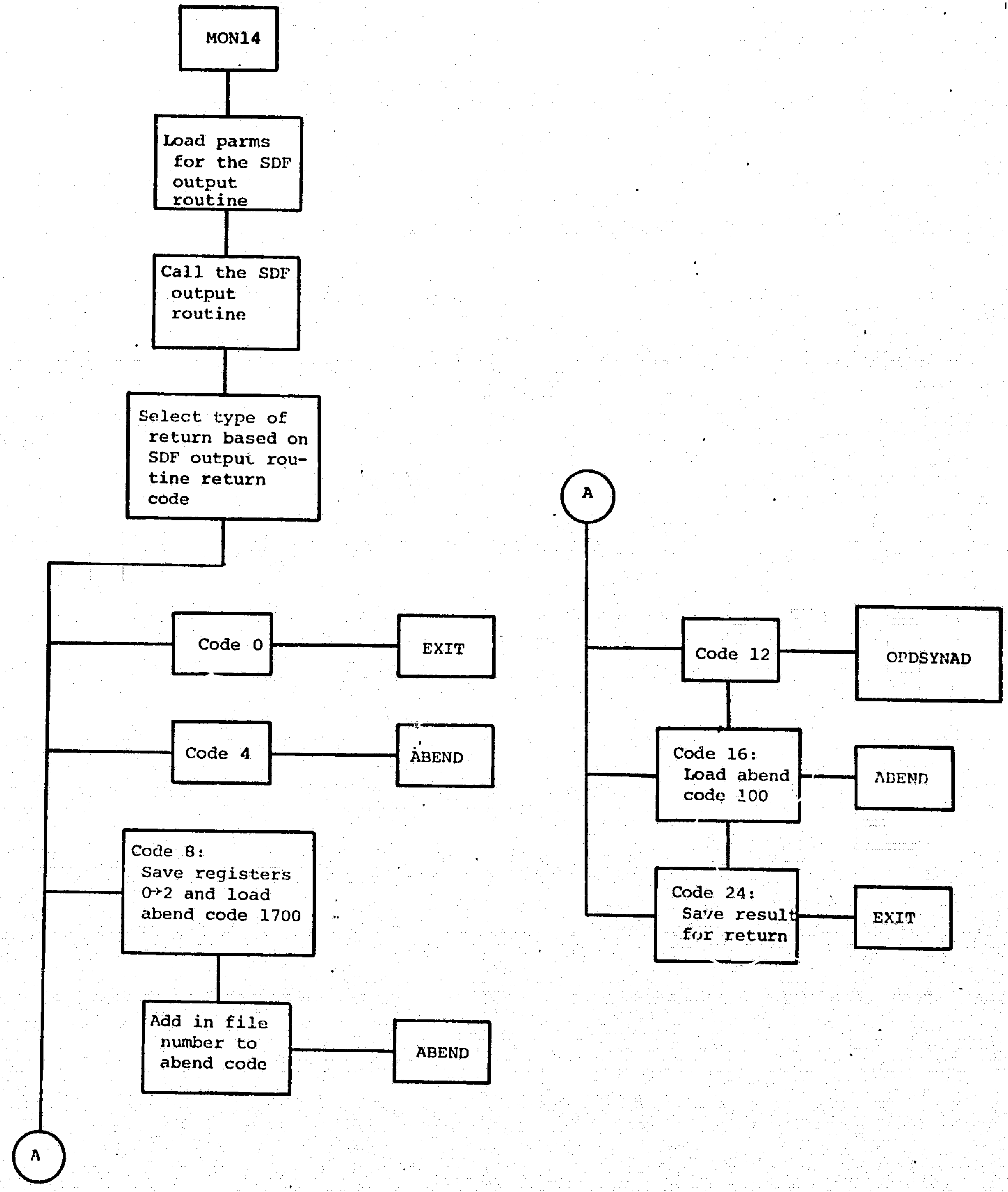

XPLVGETH
Obtain buffer size & buffer pointer
Issue GETMAIN
Save return code from GETMAIN
Return code = 0?
NO
EXIT
YES
Call ZEROMEM to zero the buffer
Was that last buffer
NO
YES
EXIT

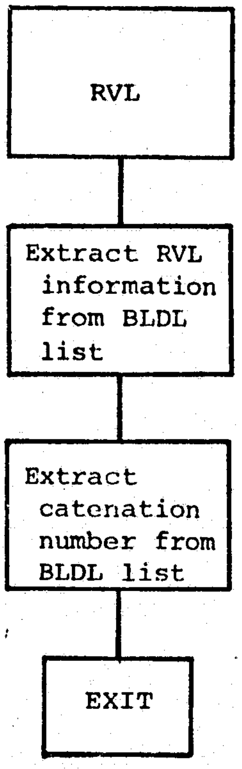

XPLVFREE
Obtain buffer size & buffer pointer
Issue FREEMAIN
Was that last buffer?
NO
YES
EXIT

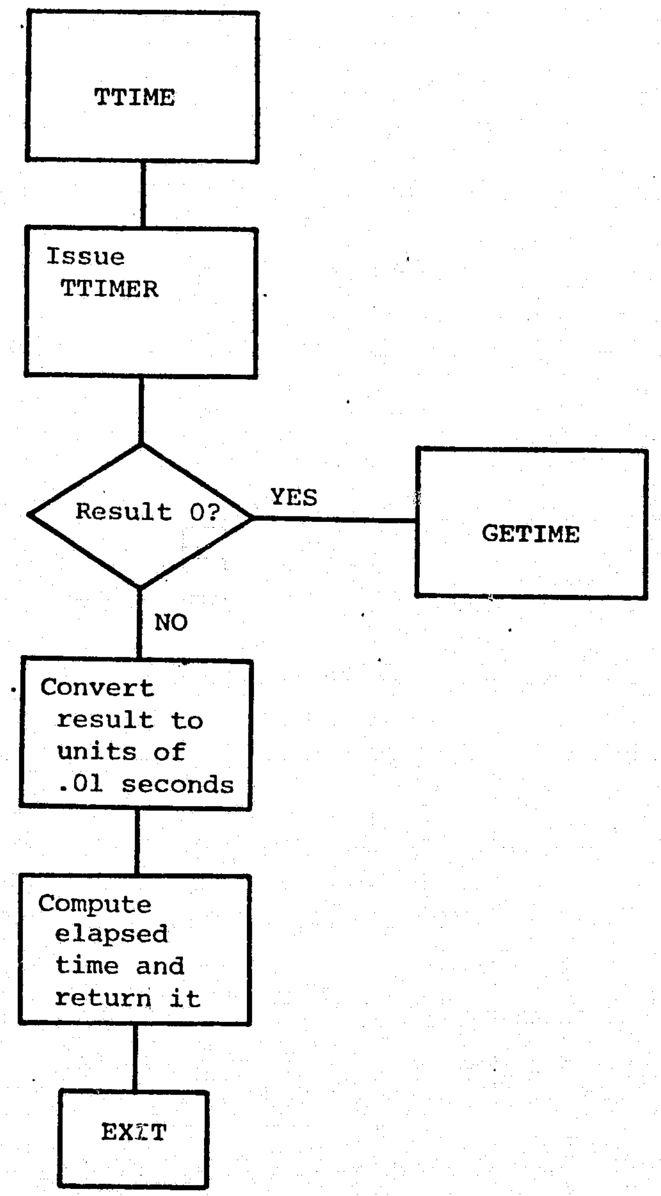

ZEROMEM
Compute # doublewords to clear
# doublewords >0?
NO
Return to caller
YES
Obtain start address to zero and constant 0
Store con-stant and bump address
Was that last double word?
NO
YES
Return to caller
CORELEFT
Issue GETMAIN
Min = 8 bytes
Max = 5 Megabytes
COREGOT←address
CORELEN←length
Save CORELEN for return
Issue FREEMAIN on CORELEN bytes at COREGOT
EXIT

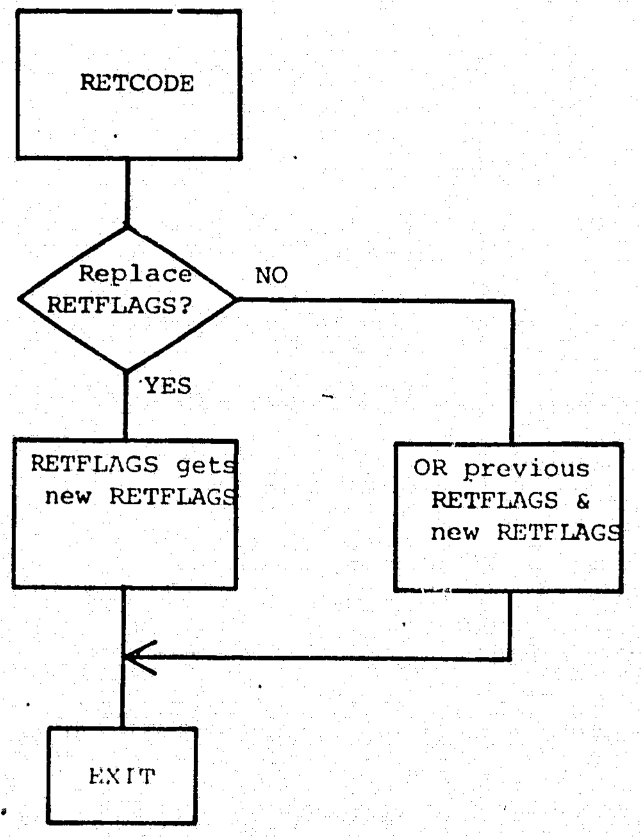

CALLSDF
PARM2 = 0?
YES
NO
Issue LOAD for SDFPKG
Save the address of the load
Place alternate DD in area specified
Load up parameters for SDFPKG
Call the SDF package
Save the return code
A
A
Parm2 = 1?
NO
EXIT
YES
Issue DELETE for SDFPKG
EXIT

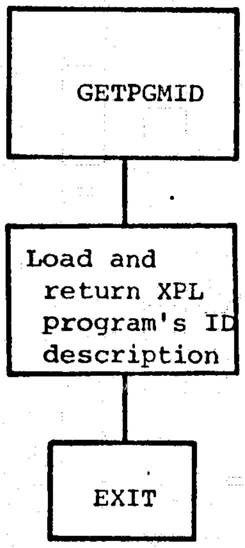

MON13
Module name specified?
NO
YES
Return existing option parms
EXIT
Name has length of 8?
NO
YES
Save registers 0→2 and load abend code 2600
ABEND
Issue DELETE on old options processor
Obtain new options processor name
Issue LOAD on the new name
Load up parm field address and call loaded options processor
Save new options parm and return it to XPL
EXIT

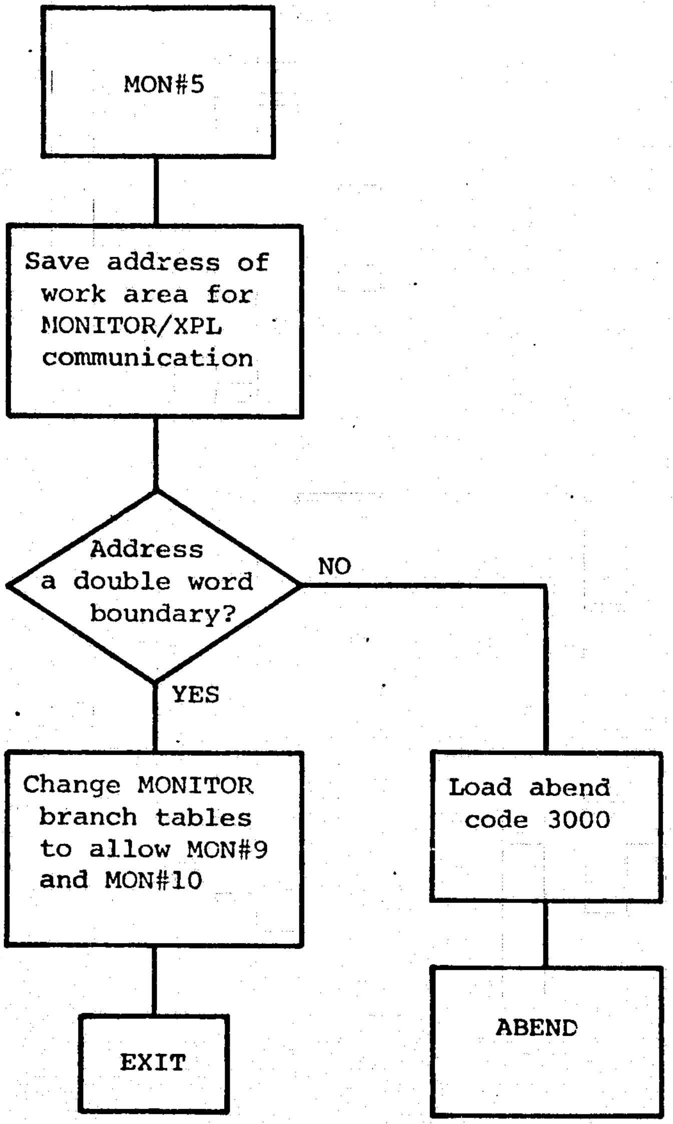

MON14
Load parms for the SDF output routine
Call the SDF output routine
Select type of return based on SDF output routine return code
Code 0
EXIT
Code 4
ABEND
Code 8: Save registers 0→2 and load abend code 1700
Add in file number to abend code
ABEND
A
A
Code 12
OPDSYNAD
Code 16: Load abend code 100
ABEND
Code 24: Save result for return
EXIT

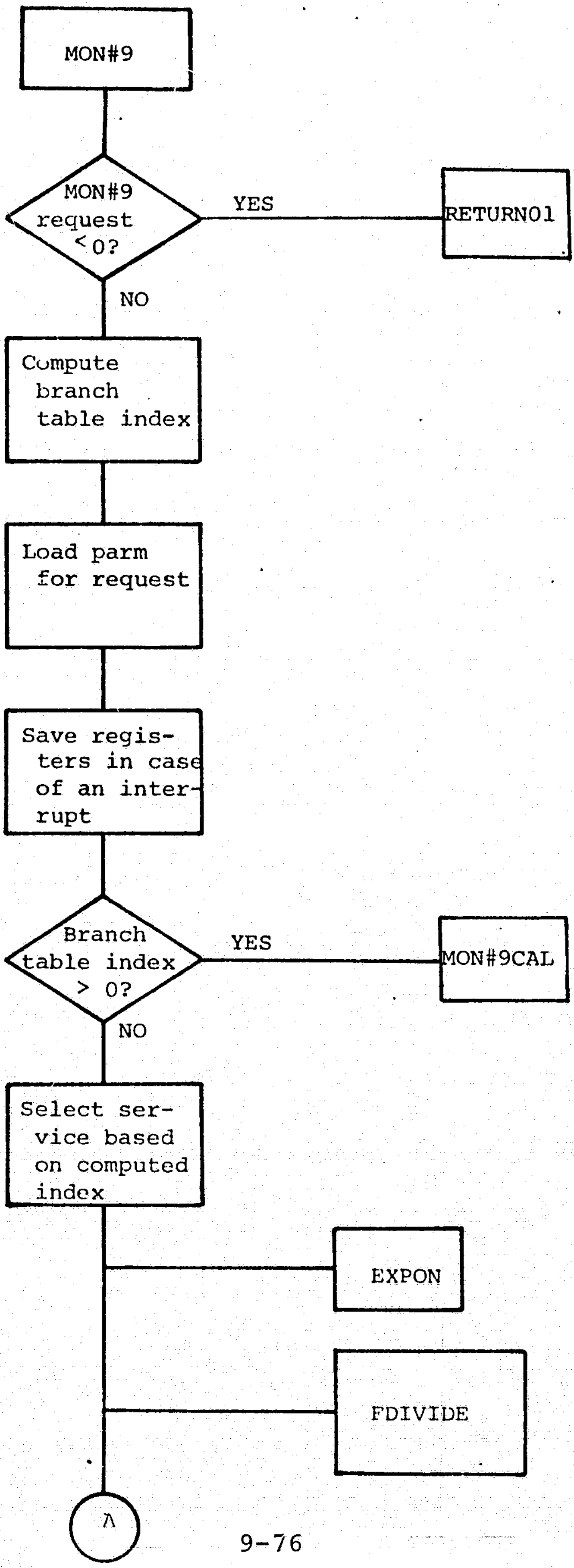

RVL
Extract RVL information from BLDL list
Extract catenation number from BLDL list
EXIT

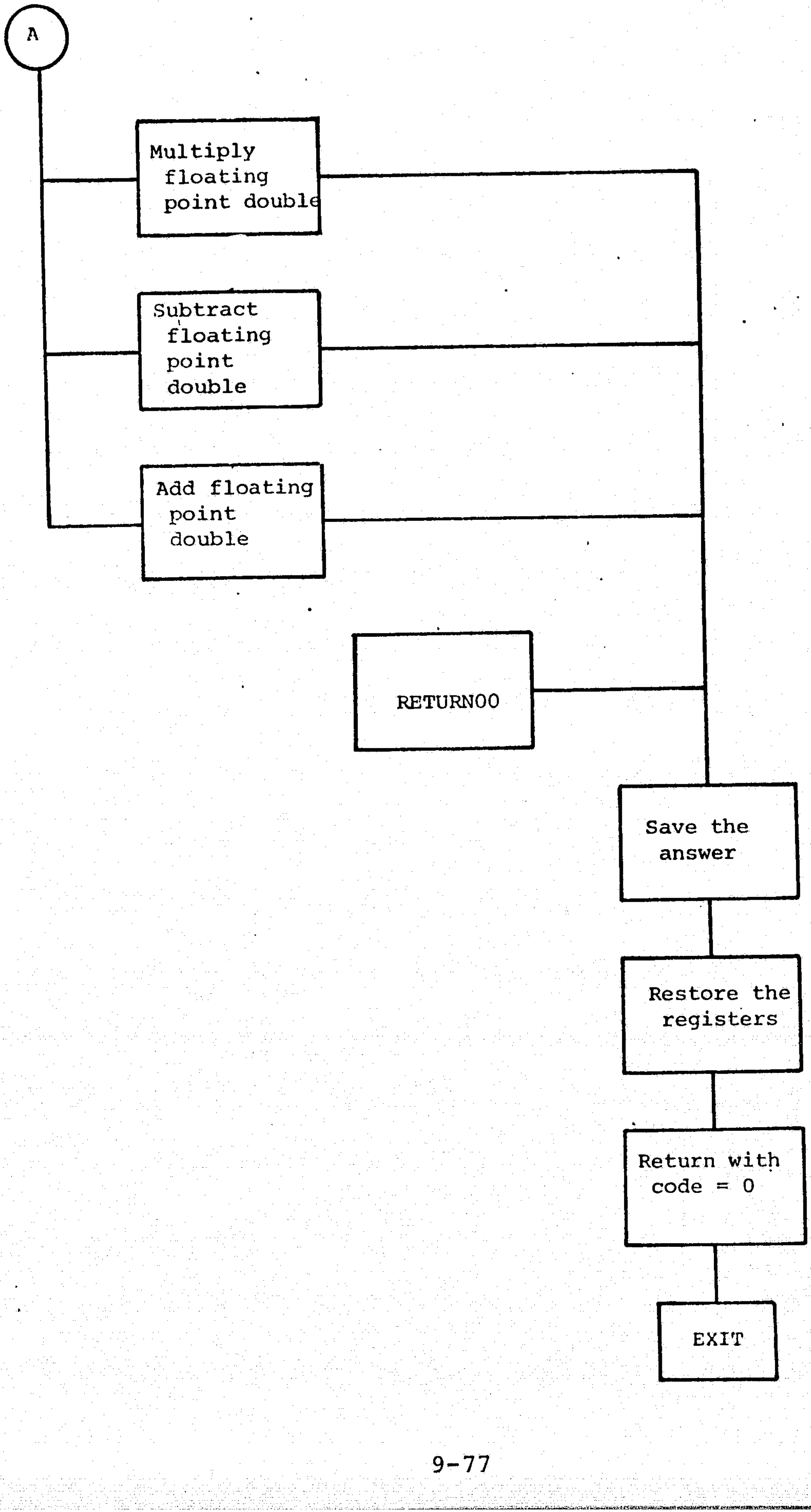

TTIME
Issue TTIMER
Result 0?
YES
GETIME
NO
Convert result to units of .01 seconds
Compute elapsed time and return it
EXIT

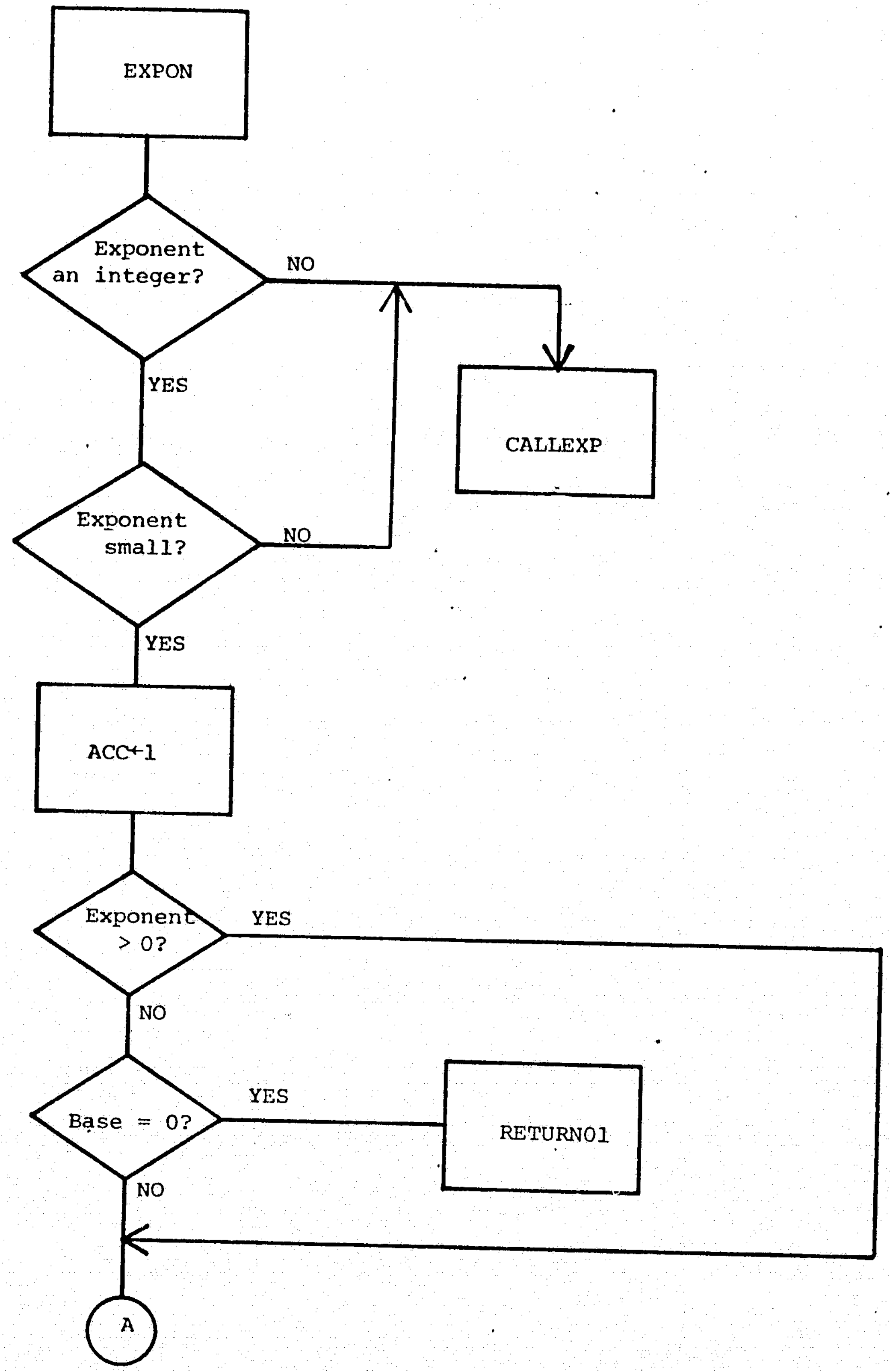

RETCODE
Replace RETFLAGS?
NO
YES
RETFLAGS gets new RETFLAGS
OR previous RETFLAGS & new RETFLAGS
EXIT

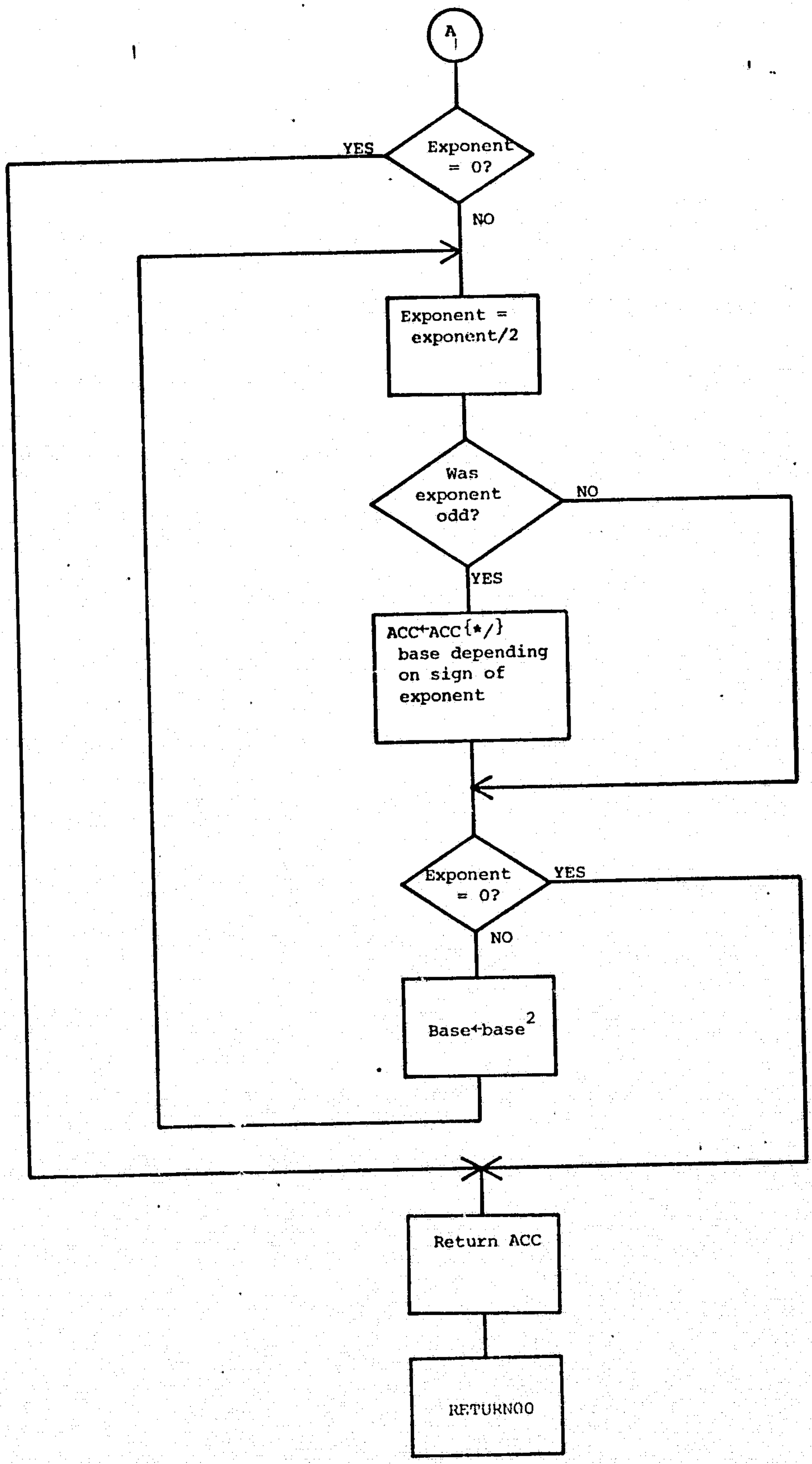

GETPGMID
Load and return XPL program's ID description
EXIT

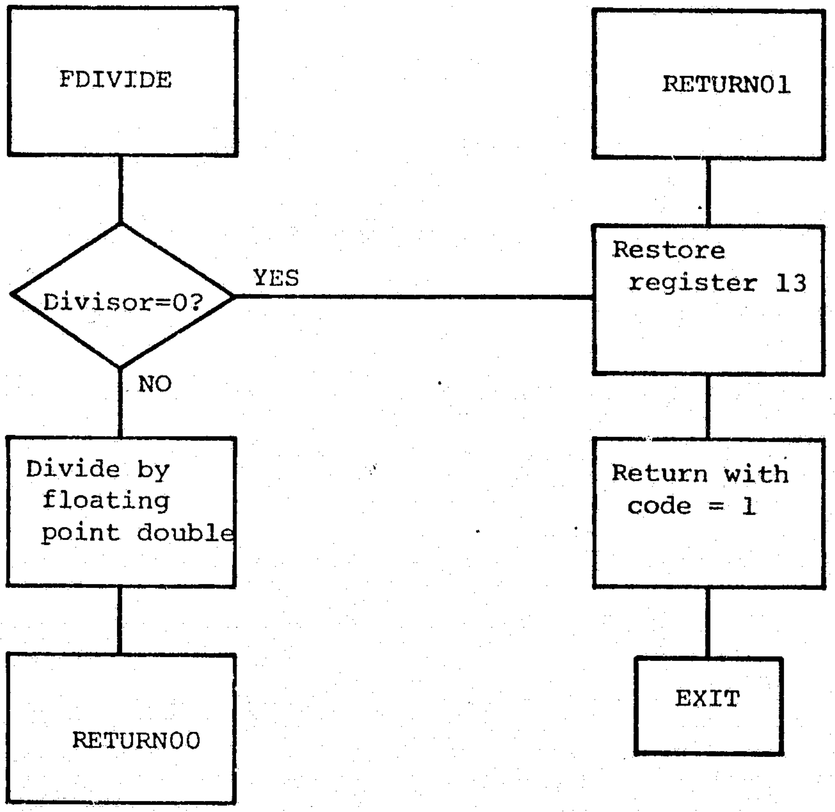

MON#5
Save address of work area for MONITOR/XPL communication
Address a double word boundary?
NO
YES
Change MONITOR branch tables to allow MON#9 and MON#10
Load abend code 3000
EXIT
ABEND

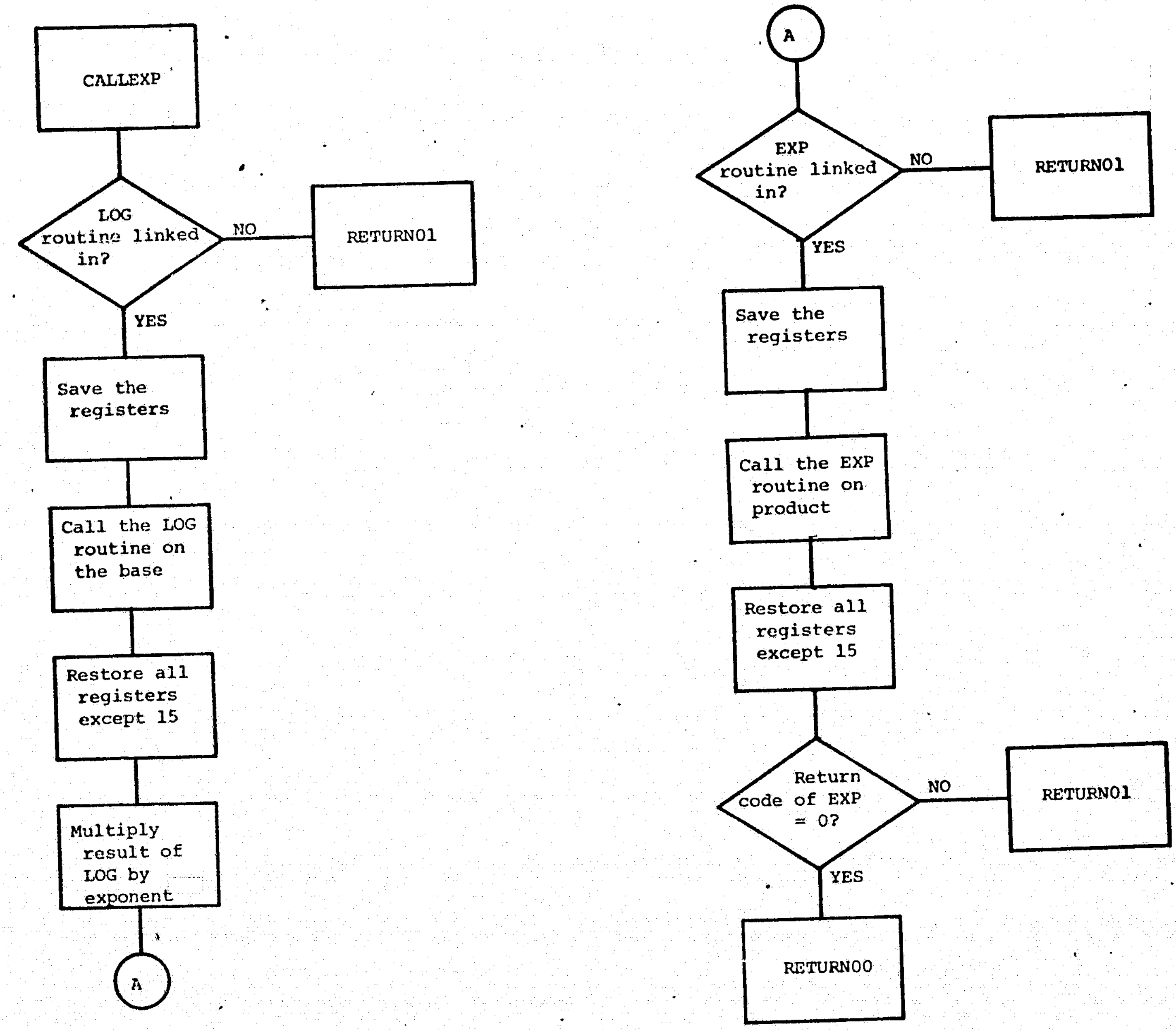

MON#9
MON#9 request < 0?
YES
RETURN01
NO
Compute branch table index
Load parm for request
Save registers in case of an interrupt
Branch table index > 0?
YES
MON#9CAL
NO
Select service based on computed index
EXPON
FDIVIDE
A

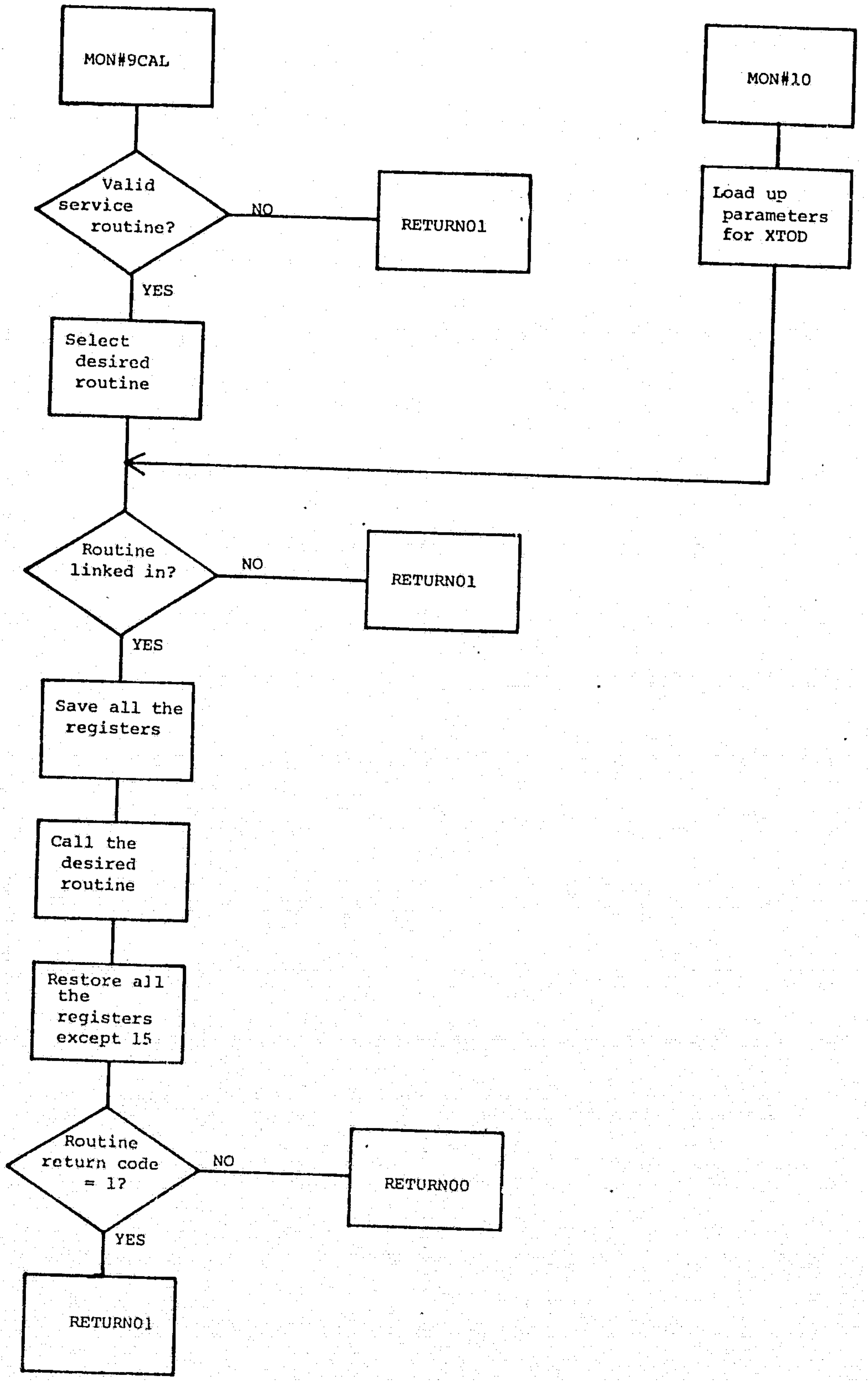

A
Multiply floating point double
Subtract floating point double
Add floating point double
RETURN00
Save the answer
Restore the registers
Return with code = 0
EXIT

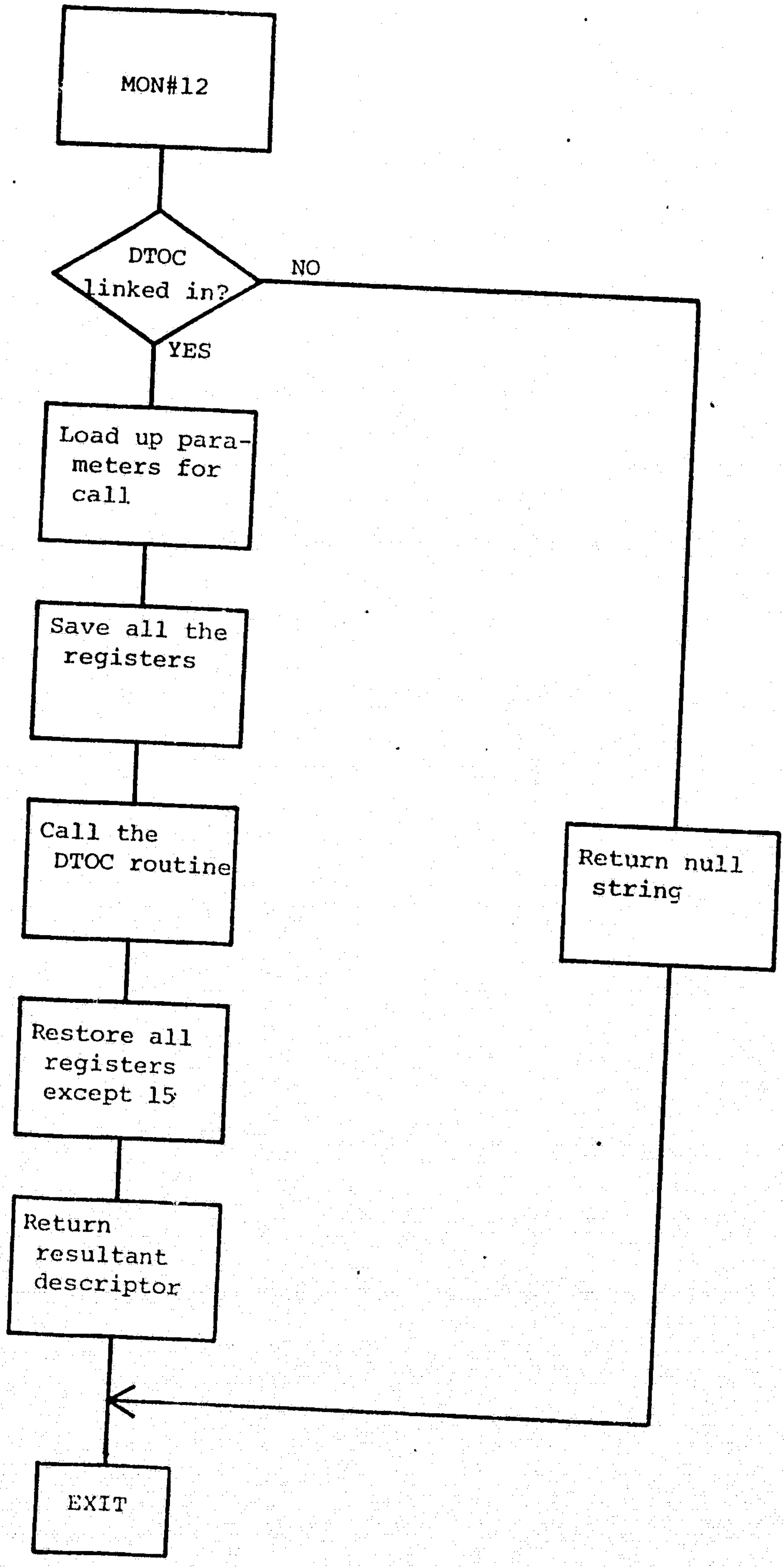

EXPON
Exponent an integer?
NO
YES
CALLEXP
Exponent small?
NO
YES
ACC←1
Exponent > 0?
YES
NO
Base = 0?
YES
RETURN01
NO
A

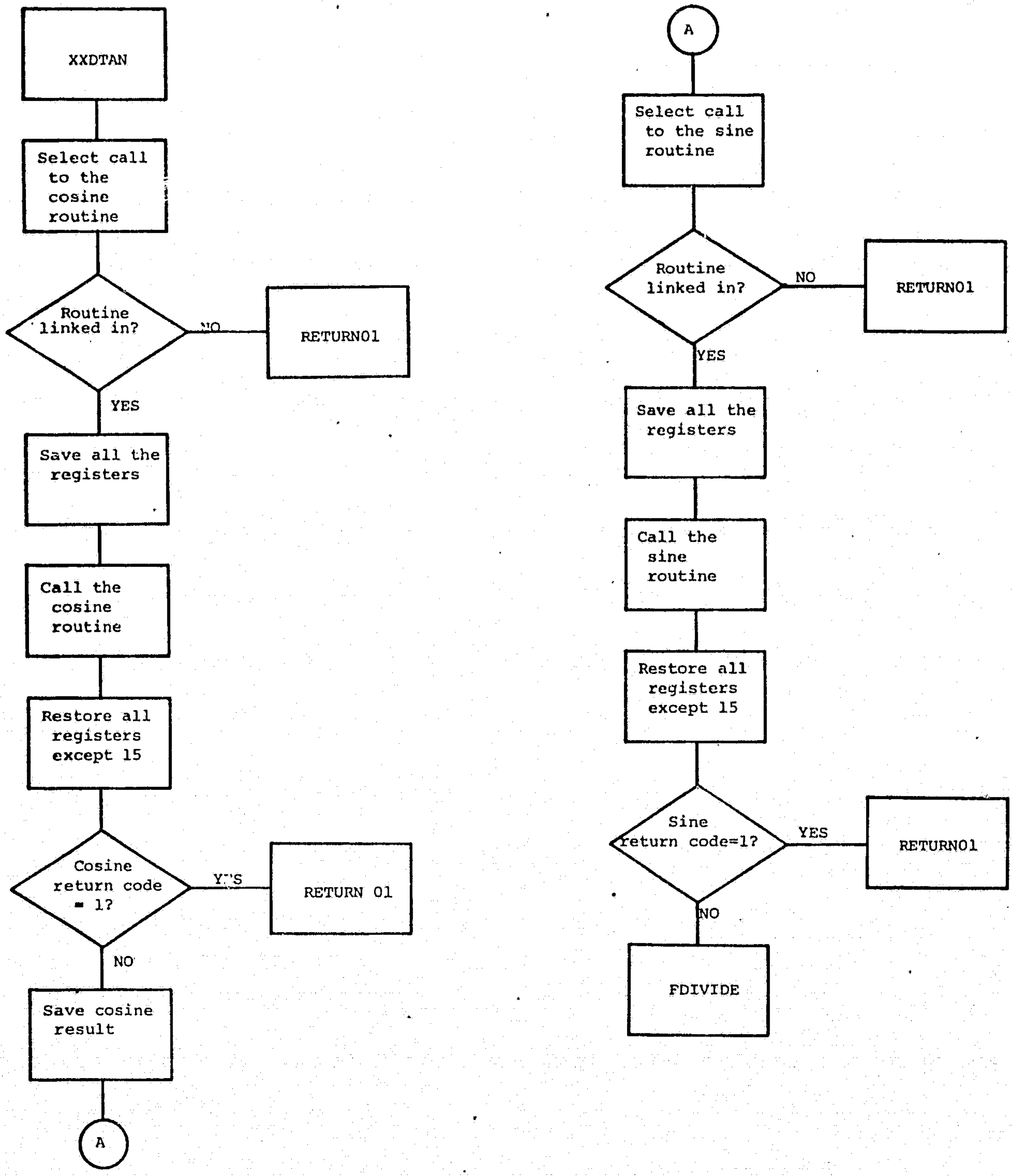

A
Exponent = 0?
YES
NO
Exponent = exponent/2
Was exponent odd?
NO
YES
ACC←ACC{*/} base depending on sign of exponent
Exponent = 0?
YES
NO
Base←base^2
Return ACC
RETURNOO

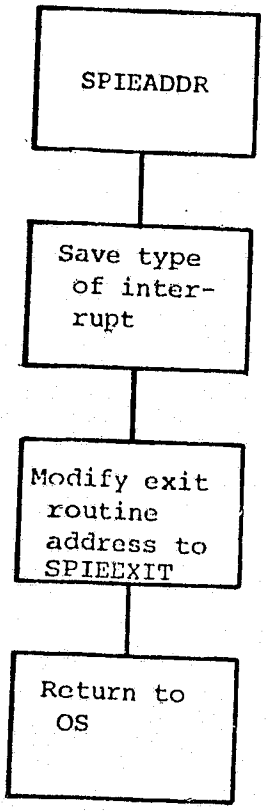

FDIVIDE
Divisor=0?
YES
NO
Divide by floating point double
RETURN00
RETURN01
Restore register 13
Return with code = 1
EXIT

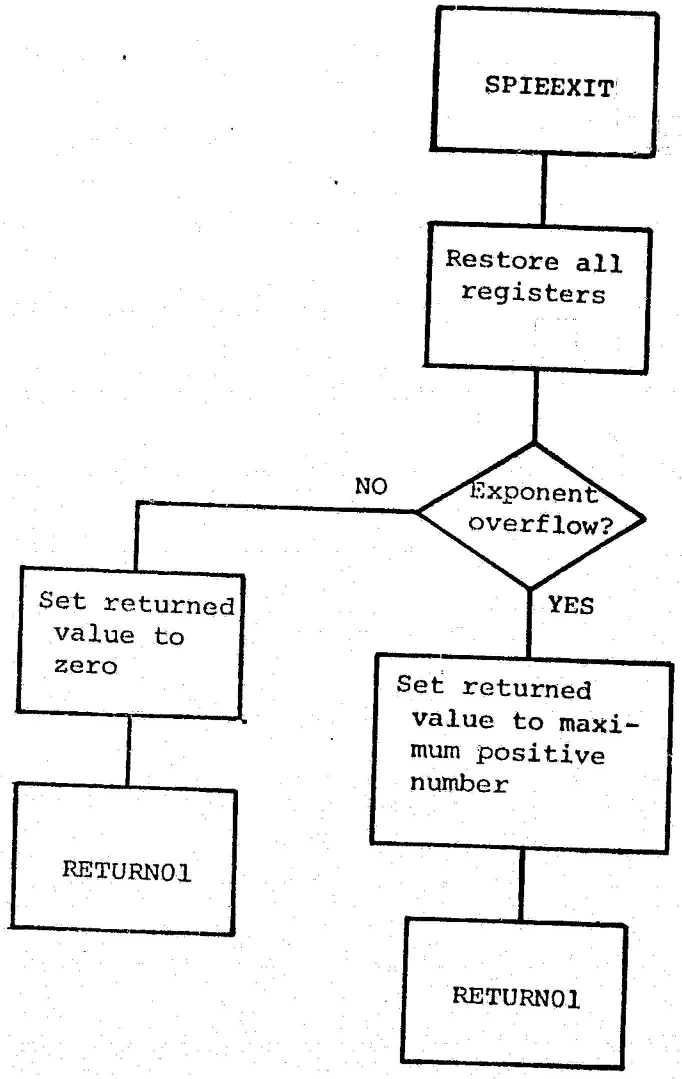

CALLEXP
LOG routine linked in?
NO
RETURN01
YES
Save the registers
Call the LOG routine on the base
Restore all registers except 15
Multiply result of LOG by exponent
A
A
EXP routine linked in?
NO
RETURN01
YES
Save the registers
Call the EXP routine on product
Restore all registers except 15
Return code of EXP = 0?
NO
RETURN01
YES
RETURN00


MON#9CAL
Valid service routine?
NO
RETURN01
YES
Select desired routine
MON#10
Load up parameters for XTOD
Routine linked in?
NO
RETURN01
YES
Save all the registers
Call the desired routine
Restore all the registers except 15
Routine return code = 1?
NO
RETURN00
YES
RETURN01


MON#12
DTOC linked in?
NO
YES
Load up parameters for call
Save all the registers
Call the DTOC routine
Restore all registers except 15
Return resultant descriptor
Return null string
EXIT


XXDTAN
Select call to the cosine routine
Routine linked in?
NO
RETURN01
YES
Save all the registers
Call the cosine routine
Restore all registers except 15
Cosine return code = 1?
YES
RETURN 01
NO
Save cosine result
A
A
Select call to the sine routine
Routine linked in?
NO
RETURN01
YES
Save all the registers
Call the sine routine
Restore all registers except 15
Sine return code=1?
YES
RETURN01
NO
FDIVIDE


SPIEADDR
Save type of interrupt
Modify exit routine address to SPIEEXIT
Return to OS


SPIEEXIT
Restore all registers
Exponent overflow?
NO
YES
Set returned value to zero
RETURN01
Set returned value to maximum positive number
RETURN01

10.0 REAL TIME EXECUTIVE

10.1 Design Overview

10.1.1 HAL/S-360 Real Time Implementation Summary

a) The HAL/S-360 real time package is implemented as a "self contained" system which executes as a single task/job step under OS-360. A load module is created by a "HAL Link Step" using the 360 linkage editor. The load module contains all HAL/S compiled program/tasks, external procedures, and compool blocks which are pertinent to the run, together with a collection of run time routines. This load module or HAL/S system is then loaded and executed under OS as a single task.

b) All HAL/S process management functions, that is control over the scheduling and dispatching of HAL/S program and task blocks, are implemented through HAL run time routines. The HAL/S real time control statements (i.e. SCHEDULE, TERMINATE, WAIT, CANCEL, SIGNAL) are interfaced from the compiler directly to HAL/S run time routines and not to OS-360. The HAL/S run time routines utilize internally defined process queues. The process states and state transitions are controlled by HAL/S compiler run time routines. The compiler generates "branch and link" commands to the appropriate HAL/S routine to implement execution of its real time statements. All HAL/S event tables, event queues and the processing of event expressions are performed by HAL/S run time routines. There is no interaction with OS-360 for servicing event variables.

A timer queue and HAL/S process interaction with timed events is controlled by HAL/S run time routines. The logical implementation of these routines is presented in later sections of this chapter.

c) OS-360 control and OS task interaction is limited to supervision of the HAL/S system load module. It is unaware of the existence of multiplicity of HAL/S processes and queues.

In summary, HAL/S interacts with OS-360 only at the "HAL/S load module level" or system level as a single OS task and not at the statement level or HAL program/task block level (i.e. a HAL process).

d) The HAL/S-360 implementation does not execute in "real-time" on the 360. HAL/S pseudo time is maintained in "machine units" by HAL/S run time routines. Internal pseudo clock registers are updated in machine units which are decremented by a "clock tick" HAL run time routine after the execution of each HAL/S statement. The effect is to model the estimated execution time of each HAL statement for a specific Shuttle flight computer on the 360, and to maintain simulated flight computer time as HAL statements are executed on the 360. This allows the testing of flight software by direct execution on the 360 without requiring simulation. The HAL/S-360 system does not utilize the real time OS-360 clocks.

e) In HAL/S-360, the compiler inserts "hooks" between the code generated for each HAL/S statement to enable recording of variables, implementation of diagnostics, clock updating, process control, and other functions. These HAL/S-360 hooks may be used to interface to an external simulation facility to enable Shuttle avionics environment updates and diagnostics.

f) HAL error control statements ON & SEND are implemented by HAL run time routines. OS-360 is utilized only to trap some 360 error conditions. Process reactivation or termination is accomplished via HAL run time software.

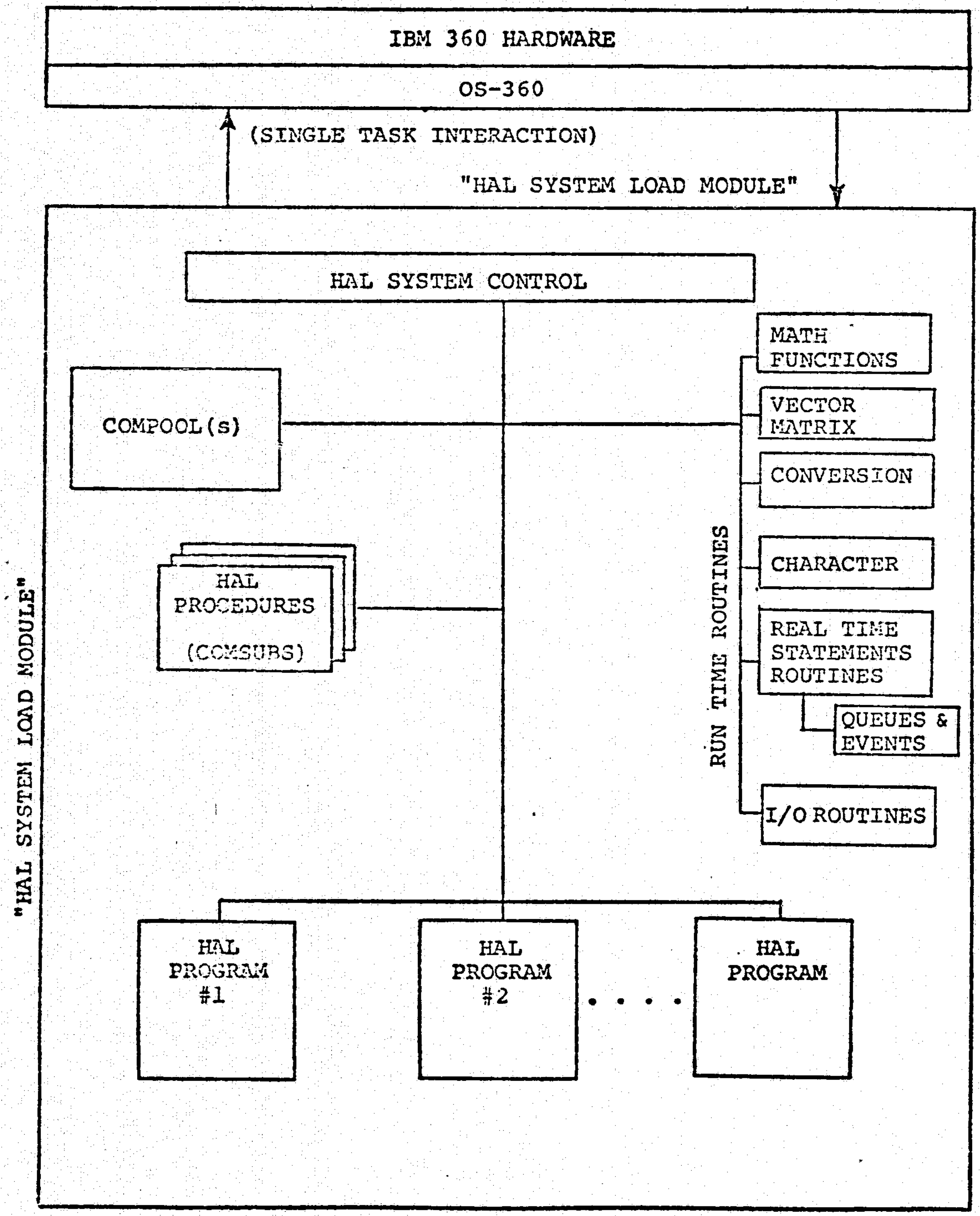
10.1.2 HAL System Load Module

A general overview of the static organization of HAL/S on the 360 is illustrated in Figure 10-1. The HAL/S run time system for the IBM 360 is operated as a single task under OS-360 control. HAL/S source statements are compiled, the separately compilable units linked together into a single HAL/S system load module and executed as a single job step task.

The HAL system load module consists of the code and data blocks for each compilable unit as output by the HAL compiler, together with a collection of HAL run time routines automatically brought in by the linkage editor. These run time routines consist of math routines, I/O routines, conversion routines, built-in functions, and routines to implement the HAL real time statements. On the 360 this is termed the "HAL run time executive" or "process manager". The functions and logic of these HAL run time routines (i.e. process management) is described in this chapter.

Figure 10-1

HAL SYSTEM ORGANIZATION FOR THE IBM 360



O.S. FUNCTIONS

- HAL/S SYSTEM LOAD MODULE

  EXECUTION CONTROL (NO MULTI-PROG)
- I/O SERVICES
- TRAP FIELDING

HAL FUNCTIONS

- ALL HAL PROCESS MANAGEMENT (i.e., TASKING)
- HAL EVENTS/SERVICES
- HAL TIME/SERVICES
- HAL ERROR CONTROL
- HAL I/O

10.1.3 HAL/S Process Management & Control

Processing is controlled by the HAL/S Process Manager. It controls the execution of all processes in the process queues by giving control to the processes which are ready for execution on the basis of priority. The highest priority ready process is given control.

Processes are scheduled for execution by other processes. They are inserted into the process queues by the execution of a HAL/S SCHEDULE statement. Processes may be scheduled for execution by several options:

a) Scheduled at a particular time.

b) Scheduled at a particular event or combination of events.

c) Scheduled immediately.

The scheduler may also be requested through the options of the HAL/S SCHEDULE statement to continue execution of a process on a time iterative or cyclic basis and/or until a particular event or time condition is met.

A process is allocated the CPU on the basis of priority and remains running until: a) it is completed; b) it voluntarily releases the CPU by entering a wait state; or c) it reaches a point where a higher priority process is ready to execute.

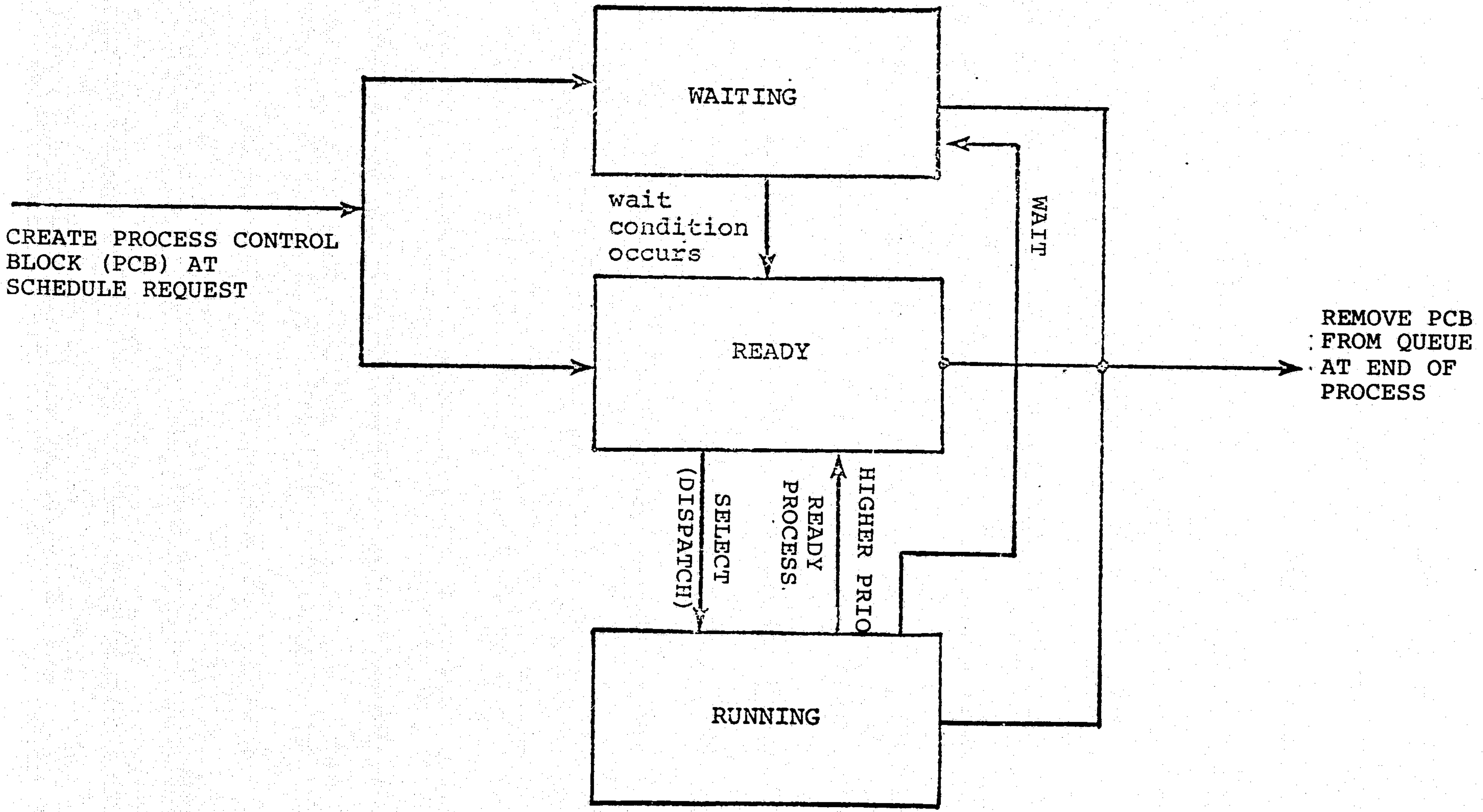
10.1.4 Process State Transition

A simplified version of the transition of process states and their conditions is illustrated in Figure 10-2. Processes are scheduled into either the wait or ready state depending on the conditions supplied in the statement. A waiting process is placed into the ready state only after the condition it was waiting for occurs. Once a process is in the ready state it is allocated the CPU on the basis of priority by a "process selector" function. The selector is entered at the end of a process, at a swap point (if a higher priority process exists) or if a process voluntarily removes itself from the running state via a WAIT statement. Only one process can be in the running state, and it remains running until it ends, or issues a WAIT, or a higher priority ready process exists.

A process may be completed and its PCB removed from the queue from any of these states.

Figure 10-2

SIMPLIFIED PROCESS STATE TRANSITION


WAITING
wait condition occurs
READY
RUNNING
CREATE PROCESS CONTROL BLOCK (PCB) AT SCHEDULE REQUEST
WAIT
SELECT (DISPATCH)
READY PROCESS.
HIGHER PRIO
REMOVE PCB FROM QUEUE AT END OF PROCESS

10.1.5 The Process Control Block (PCB)

A PCB is an element in the process priority queue. It is associated with a single process. It is inserted into the queue when a process enters an active state (i.e. when it is scheduled) and is removed from the queue when the process is terminated.

Each PCB is fixed in size but the number of PCB's on the queue varies. The method of PCB allocation is to create, initialize, and place on a "free PCB" queue the maximum number of PCB's ever required.

The information required in a PCB is illustrated in Figure 10-3 and described functionally below.

a) Priority Queue Linkage

This field contains a pointer to the next PCB in the priority queue.

b) Priority

Process priority assigned in SCHEDULE statement.

c) Process State Information

This field contains the following information:

- READY/WAIT - Is process ready for execution?
- WAIT ON DEPENDENT PROCESS - Is process waiting for dependents?
- INTER-CYCLE WAIT - Is process cyclic and between cycles?
- INITIATED - Has process begun execution (at least once if cyclic)?
- CANCELLED - Is process to be terminated at the end of its current cycle (if cyclic), i.e. has a CANCEL statement been issued for this process?

d) Task/Program

Is process a task or program?

e) Entry Address

Pointer to the program entry for this process.

Figure 10-3

PROCESS CONTROL BLOCK (PCB) INFORMATION

PRIORITY QUEUE LINKAGE
PRIORITY
PROCESS STATE INFORMATION
TASK/PROGRAM FLAG
ENTRY ADDRESS
PROCESS DEPENDENCY LINKAGE (FATHER, SON, BROTHER)
CYCLIC CONTROLS
SAVE AREA
LAST ERROR GROUP CODE
LAST ERROR NUMBER CODE

f) Process Dependency Linkage

This field contains:

- Pointer to PCB of father process (a null pointer indicates an independent program process).
- Pointer to PCB of one son process (a null pointer indicates a process with no dependent processes).
- Pointer to next PCB in a chain of "brother" PCB's.

g) Cyclic Controls

This field contains:

- CYCLIC - A flag indicating whether or not the process is cyclic.
- TYPE - This indicates whether the cyclic type is REPEAT AFTER, REPEAT EVERY, or immediate (from SCHEDULE statement).
- VALUE - A scalar indicating inter-cycle wait time if TYPE is AFTER or complete cycle time if EVERY.

h) Save Area

This field is for the process stack pointer which is used to save and restore the machine environment across process swaps.

i) Last Error Group Code

This field saves the information returned by the ERRGRP built-in function.

j) Last Error Number Code

This field saves the information returned by the ERRNUM built-in function.

10.2 Mechanization and Structure of HAL/S-360 Real Time

The purpose of this section is to describe the overall structure and control of the HAL/S-360 run time system. Figure 10-4 illustrates the organization of the system. There are basically four major sections:

1) A HAL/S Start Routine which gains control from OS-360 and initializes the HAL/S run.

2) A HAL/S Process Manager which performs the selection (dispatching) and initiation of all HAL/S processes in the process queues. It is the central control element.

3) A HAL/S statement processor which is invoked after execution of each HAL/S statement. It performs a series of functions at each statement such as: updating simulated clocks, checks for higher priority processes, determines when a process swap is required and performs tracing and diagnostics when required.

4) A set of HAL/S process management service routines which are called by the process on the execution of a SET, RESET, SCHEDULE, CANCEL TERMINATE<ID>, SIGNAL event statements.

As an overview, a process is given control by the process manager when it is the highest priority ready process. During execution it calls the HAL/S statement processor after each statement. It keeps track of time and diagnostic requests. A process may schedule, cancel, or terminate other processes during execution. This is done by the compiler inserting code to call the appropriate HAL/S process service routine.

Details of the interfaces between the compiler and the process service routines are given in the HAL/S-360 Compiler System Specification (IR #60-4).

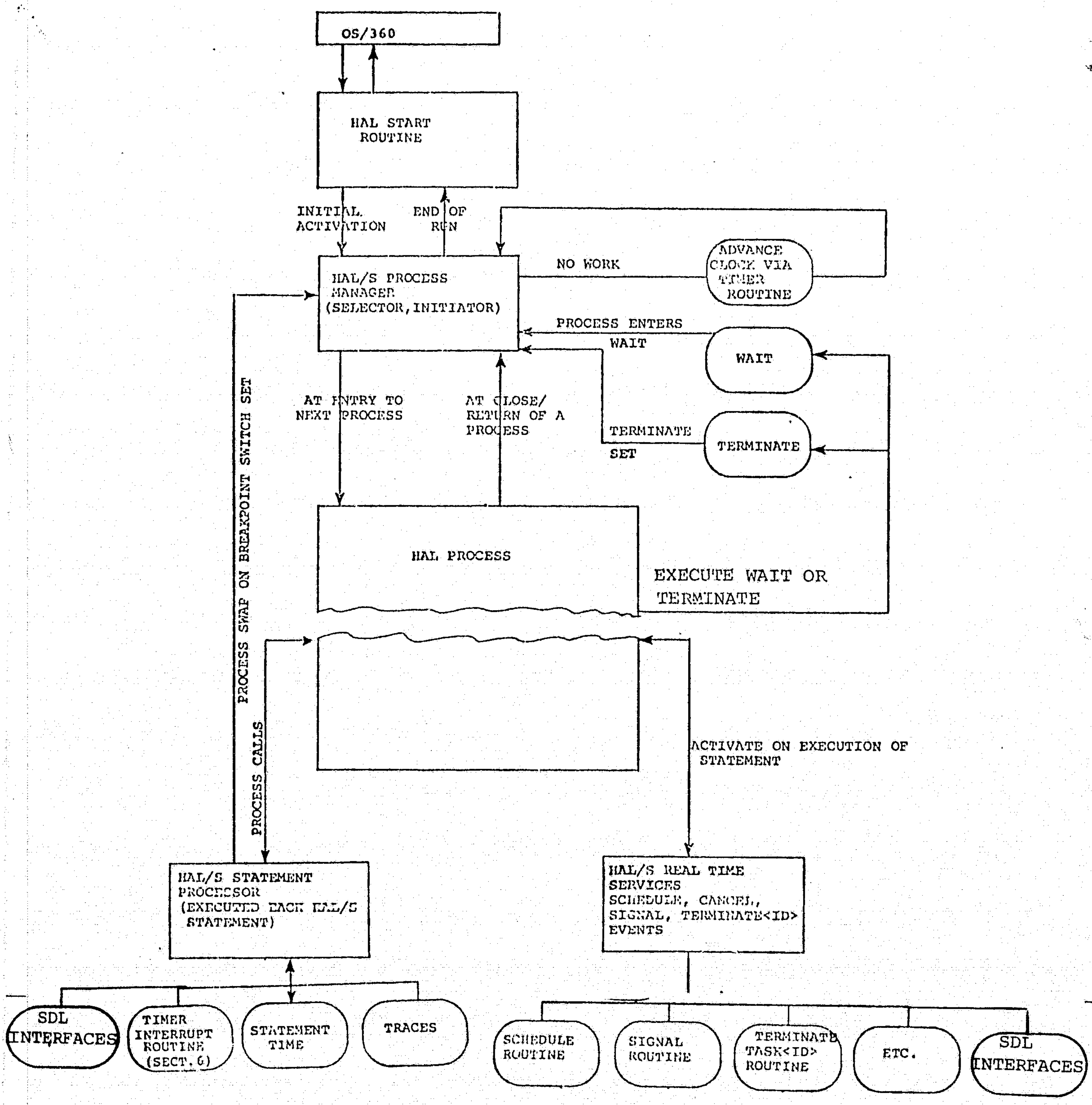
When a process executes a wait or terminate (self) statement it results in a process swap and the appropriate action is taken for updating the PCB entry.

A process continues to run until it either ends normally or executes a CLOSE or RETURN statement. At this point, the process manager selects the next ready process.

The process manager completes the run when all queues are empty. If an abnormal error condition occurs, it causes the run to be aborted.

Figure 10-4

OVERVIEW OF CONTROL AND DYNAMIC STRUCTURE
HAL/S-360 Real TIME



OS/360
HAL START ROUTINE
INITIAL ACTIVATION
END OF RUN
HAL/S PROCESS MANAGER (SELECTOR, INITIATOR)
NO WORK
ADVANCE CLOCK VIA TIMER ROUTINE
PROCESS ENTERS WAIT
WAIT
TERMINATE SET
TERMINATE
AT ENTRY TO NEXT PROCESS
AT CLOSE/ RETURN OF A PROCESS
HAL PROCESS
EXECUTE WAIT OR TERMINATE
PROCESS SWAP ON BREAKPOINT SWITCH SET
PROCESS CALLS
ACTIVATE ON EXECUTION OF STATEMENT
HAL/S STATEMENT PROCESSOR (EXECUTED EACH HAL/S STATEMENT)
HAL/S REAL TIME SERVICES SCHEDULE, CANCEL, SIGNAL, TERMINATE<ID> EVENTS
SDL INTERFACES
TIMER INTERRUPT ROUTINE (SECT. 6)
STATEMENT TIME
TRACES
SCHEDULE ROUTINE
SIGNAL ROUTINE
TERMINATE TASK<ID> ROUTINE
ETC.
SDL INTERFACES

10.2.1 HALSTART Routine

The HAL/S system load module is given control by the operating system with an ATTACH macro (it may be also CALLed). Once the "HAL load module" gets control from OS, the HALSTART routine performs various initialization functions. It prints out a HAL/S header, and sets up run time parameters input through JCL PARM field such as lines/page, channel # for system messages, # of errors before abort, debugging options, etc. It also issues SPIE and STAE macros to trap program interrupts and abnormal abort (ABEND) conditions.

The SPIE macro specifies an exit routine address which is used in the HAL system to signal the appropriate HAL error conditions for recoverable errors, performs fix up if required and continues execution. The STAE macro is used to specify an exit routine address which prints HAL unique diagnostic information before OS-360 terminates the run.

HALSTART must initiate the run. It does this by scheduling the "initial HAL process" to establish the first entry in the queues. HALSTART then calls the Process Manager.

10.2.2 HAL/S-360 Process Manager - DISPATCH

The Process Manager is the function which controls the state of execution of all processes in the priority queue. It consists of a process selector which chooses a process ready for execution, and a process initiator which controls the starting, cycling, and normal end of process execution. The scheduler and terminator which create and remove processes from the system are part of the application process control services.

10.2.2.1 The Process Selector (Dispatcher). The process selector chooses a process, then gives it control, so that it may proceed with execution. The choice is limited to those processes in the ready state. If there are no ready processes, the system would normally (in a flight computer environment) enter an idle state, and would remain idle until a process is brought to a ready state - normally through the occurrence of a time or event interrupt. In the HAL/S-360, however, the system is advanced through this time interval by decrementing the simulated clock to zero - forcing an interrupt. This should cause a process to enter a ready state and if not, the HAL/S-360 run is ended.

In general, there may be more than one ready process, so the choice is based on priority; i.e. the relative importance of the various ready processes, represented by the relative order of PCB's on the priority queue.

After the selector picks a process, it either uses the resume information (save area) in the PCB to restart the process at its suspended or swapped point, or it initiates the process at its beginning if it has not yet executed.

Figure 10-5 indicates that the selector starts at the top of the queue when looking for the first ready process. If the selector was entered because a process entered the wait state, search time is considerably reduced if the selector first checks the swap flag. If it is not set, the search may start with the next process on the queue instead of at the top. The swap flag is set whenever a process having a higher priority than the running process is readied.

10.2.2.2 Process Initiator (Figure 10-6). The process initiator is a routine which gets control from the process selector the first time a process starts executing. The program or task which was scheduled as a process is called as a subroutine of the process initiator. When the program or task executes a RETURN or CLOSE at its highest level, control comes back to the process initiator, which performs the following functions:

1) Causes the process to wait until all dependents have terminated.

2) If the process is not cyclic or is a cancelled cyclic process, it is terminated by calling the terminate subroutine, and control is passed to the process selector.

3) If another cycle of a cyclic process is indicated, the program or task is called again, after possibly placing the process into an inter-cycle wait state (EVERY or AFTER, options from SCHEDULE statement). If the cycle type is AFTER, the timer enqueue routine is called to start the AFTER interval. The EVERY interval is set up once when the process initiator is entered, and is automatically repeated by the timer interrupt routine.

Figure 10-5
PROCESS SELECTOR

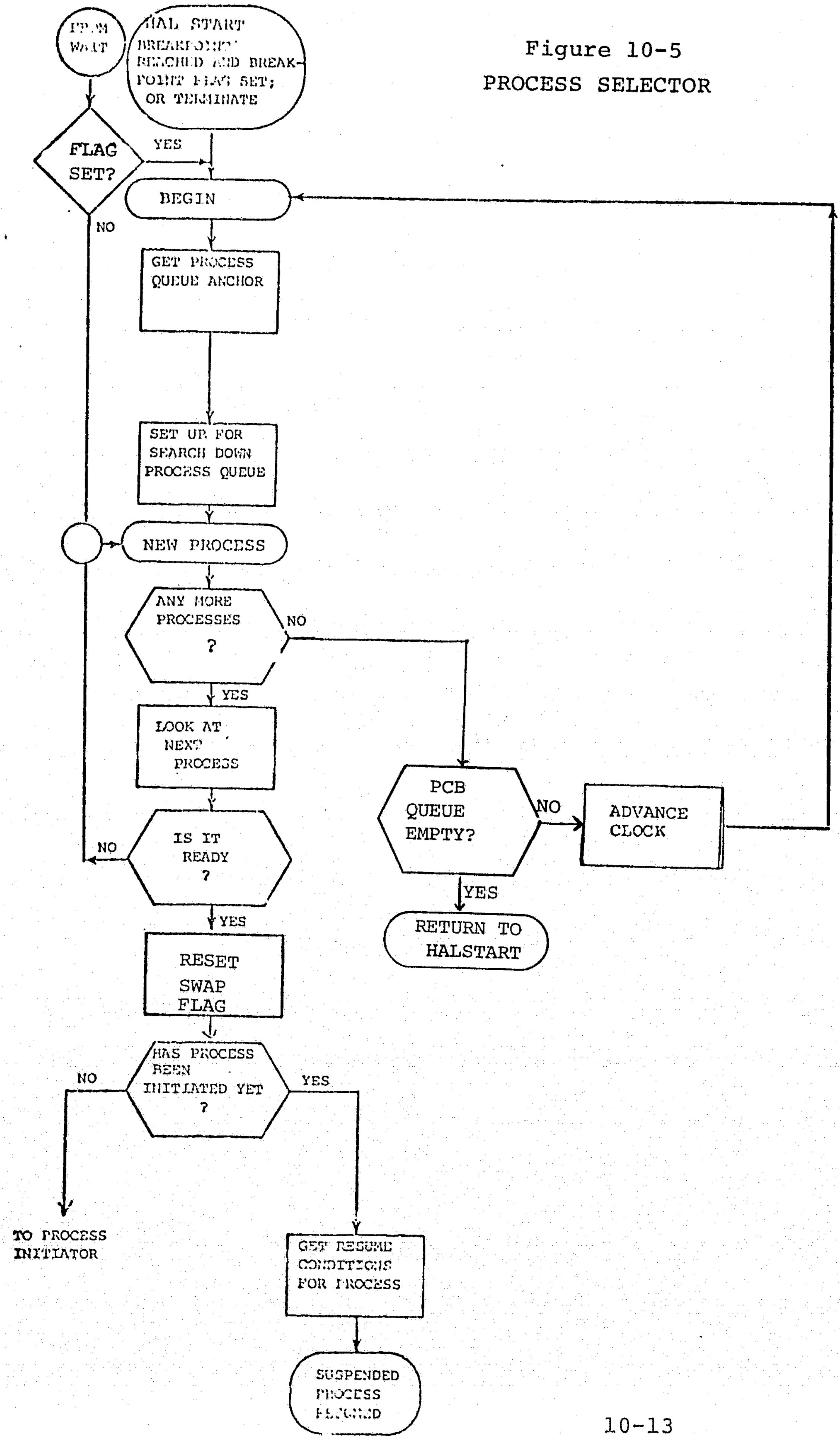
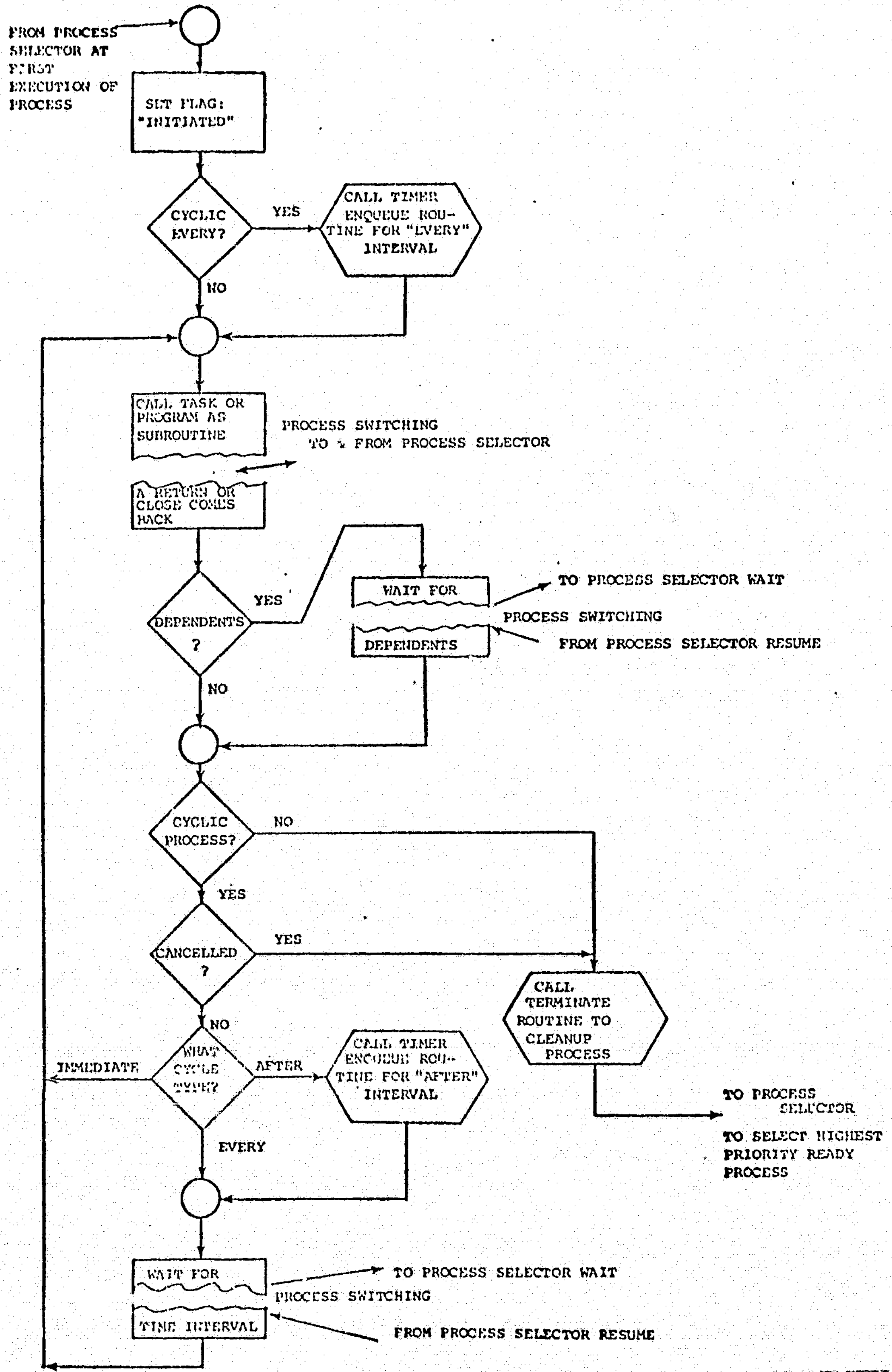


Figure 10-6

PROCESS INITIATOR


FROM PROCESS SELECTOR AT FIRST EXECUTION OF PROCESS
SET FLAG: "INITIATED"
CYCLIC EVERY?
YES
CALL TIMER ENQUEUE ROUTINE FOR "EVERY" INTERVAL
NO
CALL TASK OR PROGRAM AS SUBROUTINE
PROCESS SWITCHING TO & FROM PROCESS SELECTOR
A RETURN OR CLOSE COMES BACK
DEPENDENTS ?
YES
WAIT FOR
TO PROCESS SELECTOR WAIT
PROCESS SWITCHING
DEPENDENTS
FROM PROCESS SELECTOR RESUME
NO
CYCLIC PROCESS?
NO
YES
CANCELLED ?
YES
CALL TERMINATE ROUTINE TO CLEANUP PROCESS
NO
IMMEDIATE
WHAT CYCLE TYPE?
AFTER
CALL TIMER ENQUEUE ROUTINE FOR "AFTER" INTERVAL
TO PROCESS SELECTOR
TO SELECT HIGHEST PRIORITY READY PROCESS
EVERY
WAIT FOR
TO PROCESS SELECTOR WAIT
PROCESS SWITCHING
TIME INTERVAL
FROM PROCESS SELECTOR RESUME

10.2.3 The Process Scheduler - SCHEDULE

The Process Scheduler is the process service routine which gets control when a HAL SCHEDULE statement is executed. It creates a process by putting a new Process Control Block (PCB) containing the proper information on the priority queue. When the scheduler returns to its caller, the new process is either in the ready state or in the wait state (if the AT, IN, or ON option was specified). It is then the dispatcher's (i.e. process selector's) responsibility to give it control at a process swap point.

The options to the SCHEDULE statement are handled by separately testing for the occurrence of each one. If an option is specified, the appropriate processing is performed. Sometimes this is accomplished by a call to a system routine such as the event enqueue routine to set up an event expression, or to the timer enqueue routine to enter an interval in the timer queue. A parameter is passed to these routines specifying what action to perform (ready or cancel the process) when the requested condition (time interval expires or event expression becomes true) occurs. The Event Processor is called to process the event associated with the program or task.

Other SCHEDULE option processing is done local to the scheduler. A specified priority is assigned by setting the priority field in the PCB (used to determine the position on the priority queue). If the option DEPENDENT was specified, the scheduler places the new PCB on the dependence queue of the running process.

Parameters to the scheduler routine are listed below:

A) OPTIONS:

DEPENDENT

initial conditions (none, IN, AT, ON)

PRIORITY

REPEAT options (none, EVERY, AFTER, REPEAT with no delay)

cancel condition (none, UNTIL<event exp>, <UNTIL time>, WHILE<event exp>)

B) LABEL or RUN-TIME REFERENCE - program or task entry point address.

C) TASK/PROGRAM - is process a task or a program?

D) WAIT TIME - (optional) time specified in AT or IN phrase.

E) CANCEL TIME - (optional) time specified in EVERY or AFTER phrase.

F) PERIOD - (optional) time specified in EVERY or AFTER phrase.

G) WAIT EVENT EXPRESSION - (optional) pointer to event expression structure used in ON phrase.

H) CANCEL EVENT EXPRESSION - (optional) pointer to event expression structure used in UNTIL or WHILE phrase.

Functional flow of the scheduler is illustrated in Figure 10-7.

10.2.4 CANCEL Process Service Routine

The CANCEL statement provides a safe way to terminate a process, avoiding the danger of half-results. If the process has not yet begun execution or is in between cycles of execution, it can be safely terminated by immediately calling the terminate subroutine. In any other state, however, the process is allowed to run to the end of its current cycle. A non-cyclic process in this case would be unaffected. The cancel flag in the PCB is set by the CANCEL routine, and tested by the process initiator before starting another cycle. If it is set, the processor initiator calls the terminate subroutine. See Figure 10-8 for a flowchart of the CANCEL Serivce Routine.

10.2.5 TERMINATE

The TERMINATE statement allows for immediate and unconditional termination of a process and all its dependents. Termination involves cleanup of pending conditions (time, event) and allocated resources, and removal of the PCB from the priority queue. Since these actions must be taken for all kinds of termination (TERMINATE, CANCEL, RETURN, CLOSE), a terminate subroutine is used to carry out the cleanup work. The TERMINATE statement service routine merely locates the PCB address, checks if the active process is allowed to terminate the specified process, then calls the terminate subroutine. A flowchart of the TERMINATE Service Routine appears in Figure 10-9.

Figure 10-7

PROCESS SCHEDULER

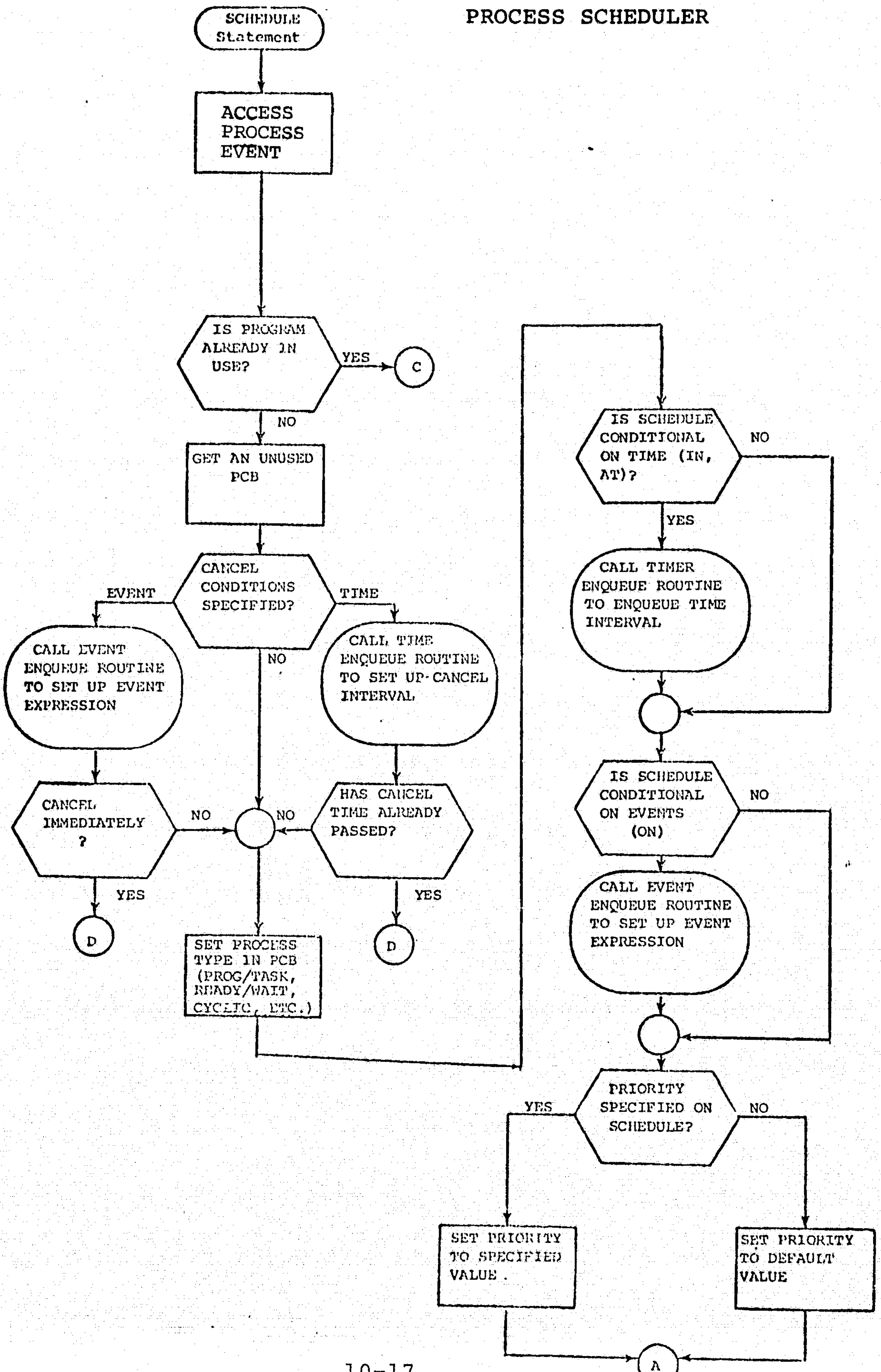

SCHEDULE Statement
ACCESS PROCESS EVENT
IS PROGRAM ALREADY IN USE?
YES
C
NO
GET AN UNUSED PCB
CANCEL CONDITIONS SPECIFIED?
EVENT
TIME
NO
CALL EVENT ENQUEUE ROUTINE TO SET UP EVENT EXPRESSION
CALL TIME ENQUEUE ROUTINE TO SET UP CANCEL INTERVAL
CANCEL IMMEDIATELY ?
NO
NO
HAS CANCEL TIME ALREADY PASSED?
YES
D
YES
D
SET PROCESS TYPE IN PCB (PROG/TASK, READY/WAIT, CYCLIC, ETC.)
IS SCHEDULE CONDITIONAL ON TIME (IN, AT)?
NO
YES
CALL TIMER ENQUEUE ROUTINE TO ENQUEUE TIME INTERVAL
IS SCHEDULE CONDITIONAL ON EVENTS (ON)
NO
CALL EVENT ENQUEUE ROUTINE TO SET UP EVENT EXPRESSION
PRIORITY SPECIFIED ON SCHEDULE?
YES
NO
SET PRIORITY TO SPECIFIED VALUE .
SET PRIORITY TO DEFAULT VALUE
A

Figure 10-7 (Cont'd.)

PROCESS SCHEDULER

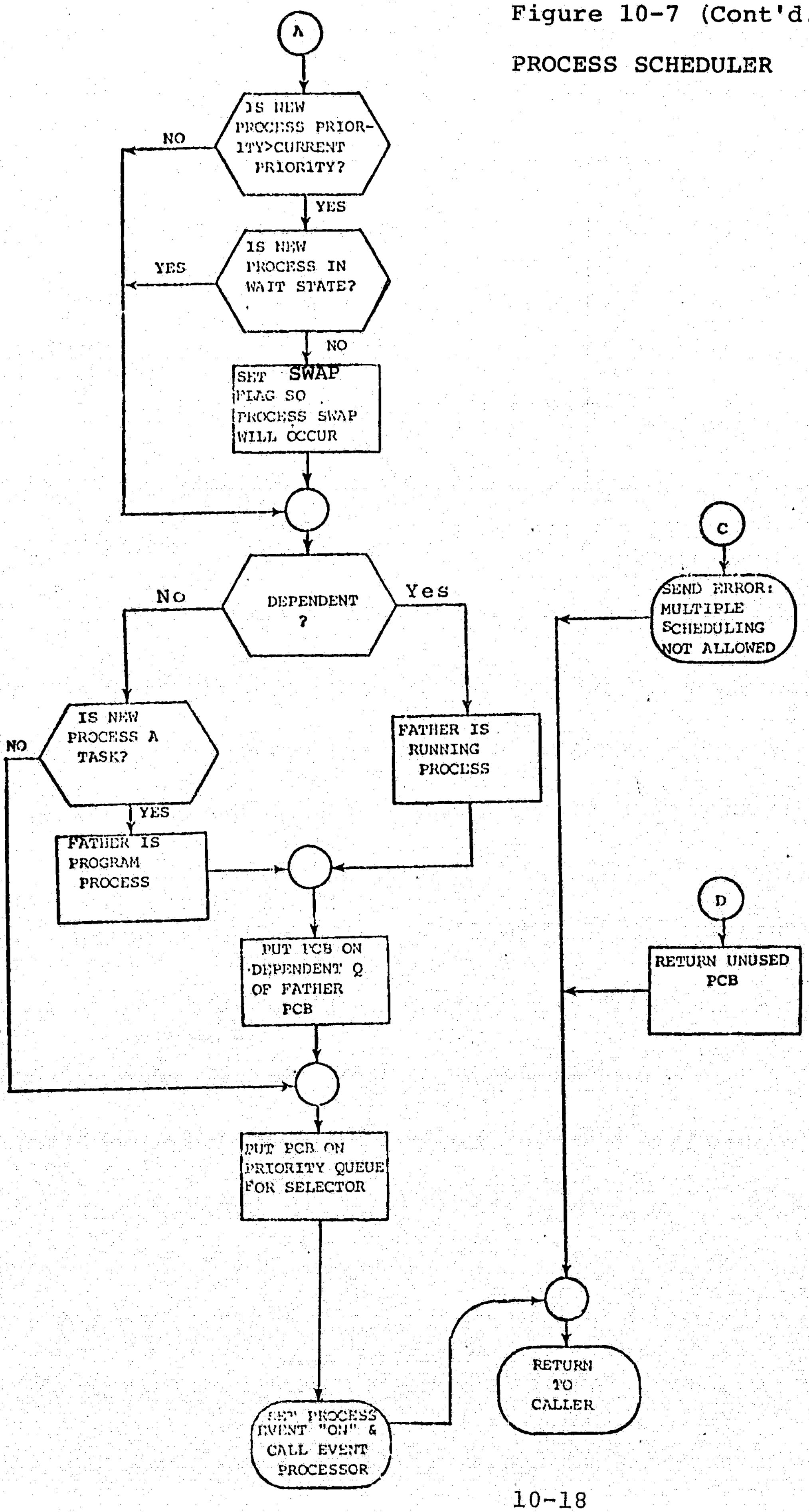

A
IS NEW PROCESS PRIORITY>CURRENT PRIORITY?
NO
YES
IS NEW PROCESS IN WAIT STATE?
YES
NO
SET SWAP FLAG SO PROCESS SWAP WILL OCCUR
DEPENDENT ?
No
Yes
C
SEND ERROR: MULTIPLE SCHEDULING NOT ALLOWED
IS NEW PROCESS A TASK?
NO
YES
FATHER IS RUNNING PROCESS
FATHER IS PROGRAM PROCESS
D
PUT PCB ON DEPENDENT Q OF FATHER PCB
RETURN UNUSED PCB
PUT PCB ON PRIORITY QUEUE FOR SELECTOR
SET PROCESS EVENT "ON" & CALL EVENT PROCESSOR
RETURN TO CALLER

Figure 10-8

CANCEL STATEMENT SERVICE ROUTINE

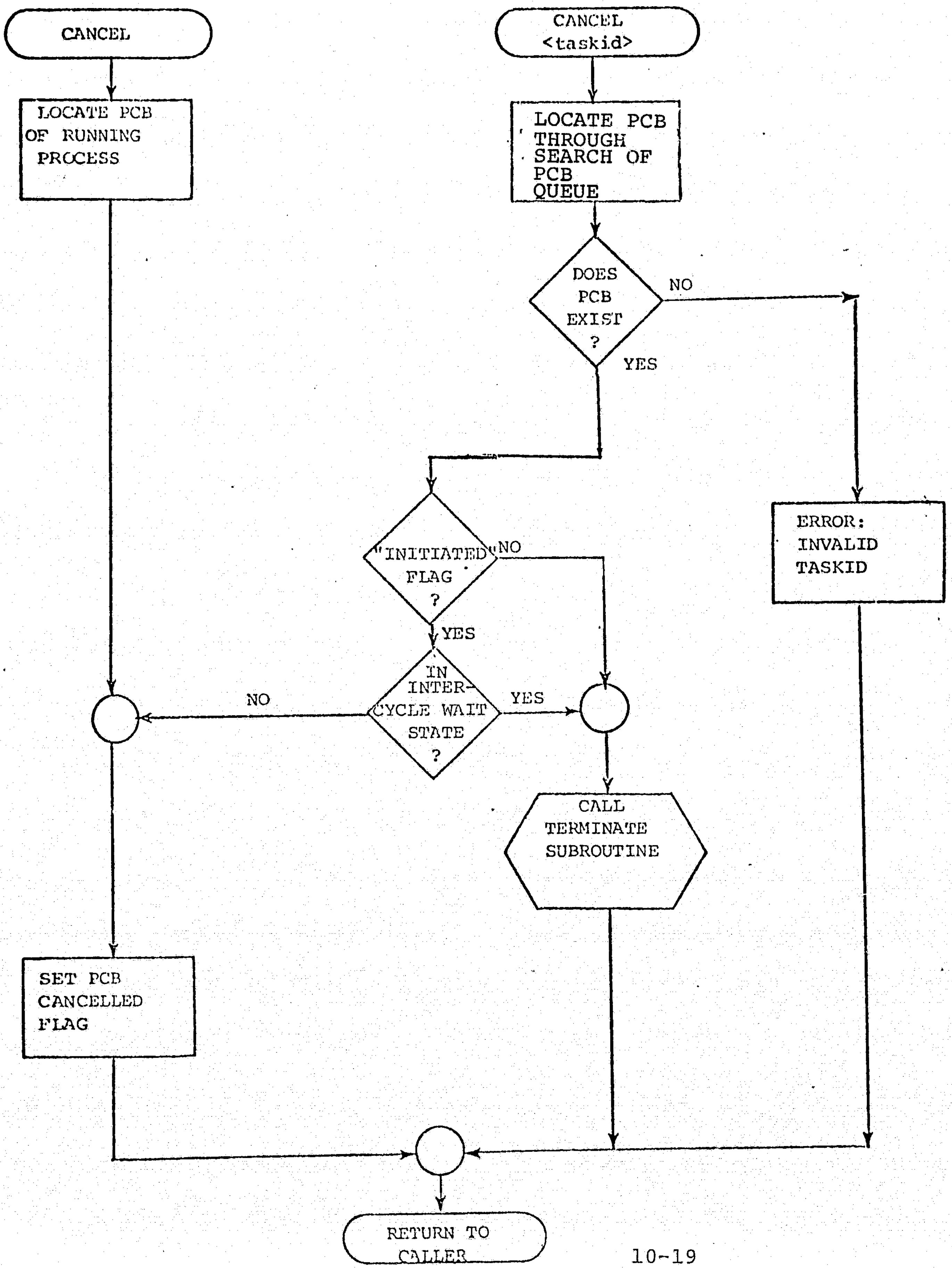
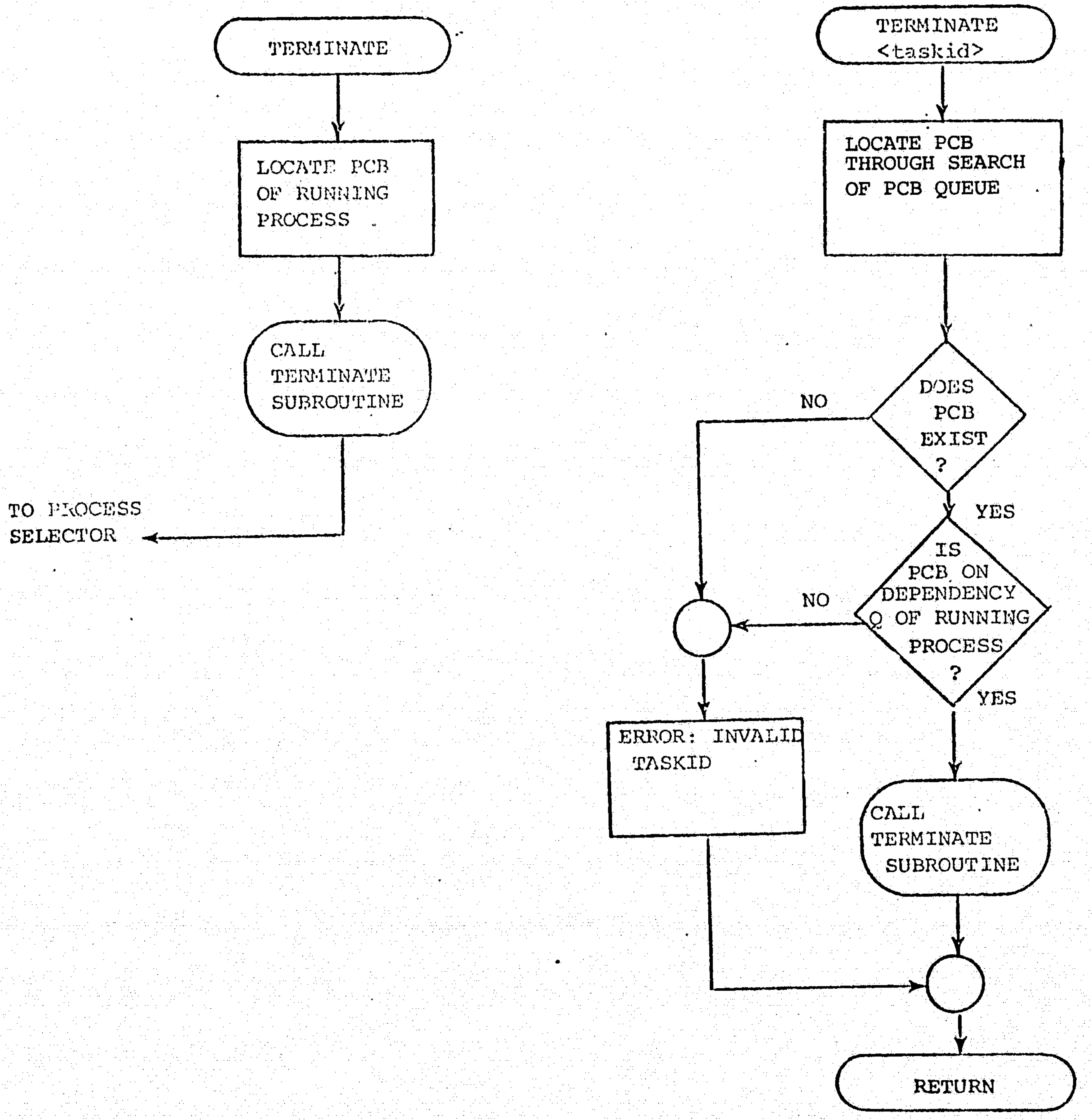


Figure 10-9

TERMINATE STATEMENT SERVICE ROUTINE


TERMINATE
LOCATE PCB OF RUNNING PROCESS
CALL TERMINATE SUBROUTINE
TO PROCESS SELECTOR
TERMINATE <taskid>
LOCATE PCB THROUGH SEARCH OF PCB QUEUE
DOES PCB EXIST ?
NO
YES
IS PCB ON DEPENDENCY Q OF RUNNING PROCESS ?
NO
YES
ERROR: INVALID TASKID
CALL TERMINATE SUBROUTINE
RETURN

10.2.5.1 Terminate Subroutine. It is called by the TERMINATE statement service routine, by the process initiator, and by the following routines when a cancel condition occurs and the process can be immediately terminated: CANCEL statement service routine, event processor, and timer interrupt routine. It performs the following functions on the process to be terminated.

a) Cancels its active event expressions (found by searching the event queue).

b) Cancels its active timer intervals (found by searching the timer queue).

c) Frees EXCLUSIVE code it may have entered.

d) Frees any lock groups it may have acquired by entering an UPDATE block.

e) Turns its associated process event off.

f) Removes and frees its PCB from the priority and dependency queues.

g) Terminates all its dependents in an identical manner.

h) Readies the father process if it is waiting for dependents and the terminating process is its last dependent.

i) Calls the event processor to process event expressions involving the process events reset in e).

The terminate subroutine may cause other processes to become ready because: 1) termination may satisfy the father's dependency wait; 2) turning the process event off may satisfy a WAIT FOR or SCHEDULE ON event expression; and 3) freeing a shared resource (e.g. UPDATE lock) may wake up a process PCB, and, if it has a higher priority than the running process, a process swap occurs when the service routine returns to the process or to the process selector.

In addition to causing a process to be made ready, the terminate subroutine, in turning off the process event, may cause another process to terminate if a cancelling event expression is satisfied. The terminate subroutine and event processor are coded to avoid recursive calls in such a situation.

10.2.6 Event Handling

The event handling system of process management carries out the signalling of events are performs specific actions when logical combinations of events, called event expressions, become true. Events are declared HAL language variables which have a boolean true/false or on/off state. These software events may be signalled (caused to change state) by a program statement. If a real time statement with an event expression is executed, the expression is immediately evaluated. If its value is not true, it becomes an "activated" event expression. An "activated" event expression is monitored until it becomes true or until the associated process is terminated. When an event change state, "activated" event expressions are re-evaluated to determine if they have become true. If they have, the requested action is taken (ready or cancel a process). Thus, event expressions have a life time beyond the execution of the containing statement.

The following statements can signal (change the state of) an event:

SET, RESET, SIGNAL - explicitly sets or pulses the state of the event (see Figure 10-10).

SCHEDULE - implicitly sets the process event state to true, if the program or task was declared with a process event.

RETURN, CLOSE, (at program or task level), CANCEL, TERMINATE - implicitly sets the process event state to false, if the program or task was declared with a process event.

The following statements may explicitly specify an event expression:

WAIT FOR - causes the executing process to wait until the event expression is true (see Figure 10-11).

SCHEDULE (with ON option) - causes the newly created process to wait until the event expression is true.

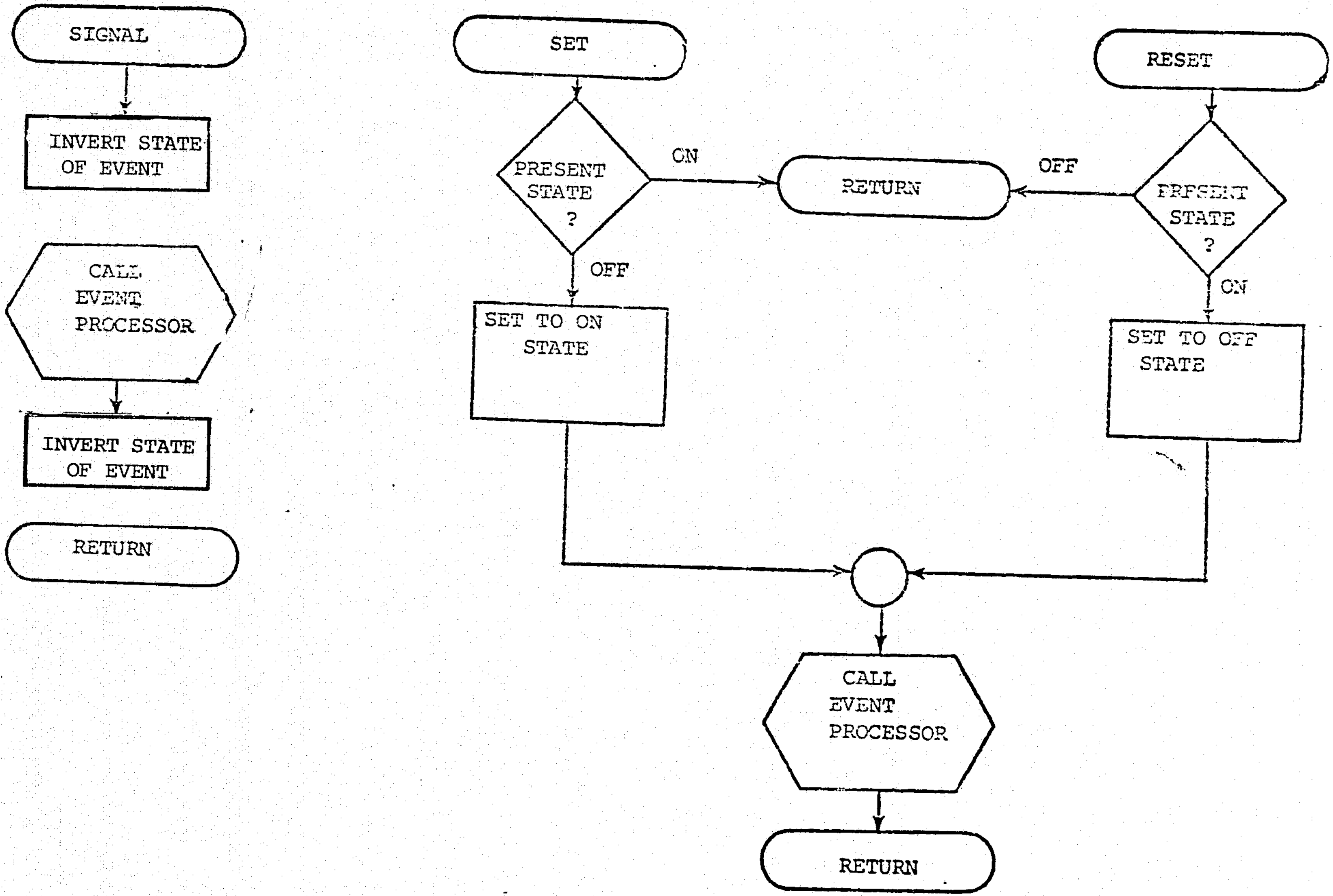


Figure 10-10

SET, RESET, SIGNAL PROCESSING

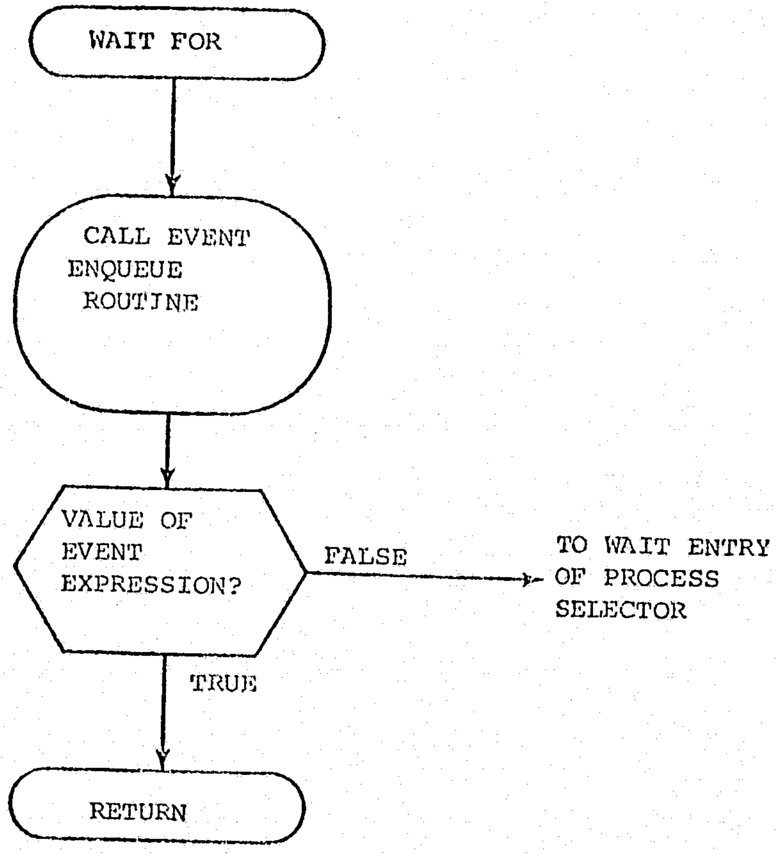

WAIT FOR
CALL EVENT ENQUEUE ROUTINE
VALUE OF EVENT EXPRESSION?
FALSE
TO WAIT ENTRY OF PROCESS SELECTOR
TRUE
RETURN

Figure 10-11

WAIT FOR ROUTINE

SCHEDULE (with the WHILE option) - causes cancellation of the newly created process if the event expression is false (an implicit "NOT" is applied to the event expression).

SCHEDULE (with the UNTIL option) - causes cancellation of the newly created process if the event expression is true, with the stipulation that at least one cycle will be allowed to execute.

Note: In addition, event expressions may be used in any context where a boolean or bit expression is allowed. However, in these contexts, HAL/S does not monitor the event expressions. They are evaluated only once at the time the containing statement is executed, and unlatched events always appear in the false state.

The routines associated with these HAL statements are called by the HAL compiled code and in turn call system event and event expression handling routines. There are four types of event expressions; two specify wait conditions (WAIT FOR, SCHEDULE ON), and two specify cancel conditions (SCHEDULE UNTIL, SCHEDULE WHILE). Since the UNTIL and WHILE phrases are mutually exclusive, the SCHEDULE statement can potentially specify two event expressions. Since event expressions can remain "activated" asynchronously with respect to execution of compiled code, an event expression must therefore be communicated to the routine through an event expression structure, created by the compiler and passed by a pointer in the parameter list of the WAIT or SCHEDULE routine. See Figure 10-12. The WAIT or SCHEDULE routine then calls the enqueue routine described below.

10.2.6.1 Event Expression Enqueue Routine. This routine is called by the WAIT FOR routine and by the scheduler to:

1) Test if the event expression is immediately true by calling the Event Expression Evaluator.

2) If it is not, copy the event expression information to an event block and enqueue the block on the event block queue, thereby activating the event expression condition. (Event blocks are diagrammed in Figure 10- .) If the expression is the wait type, the appropriate wait state is set in the PCB.

This routine has the following parameters:

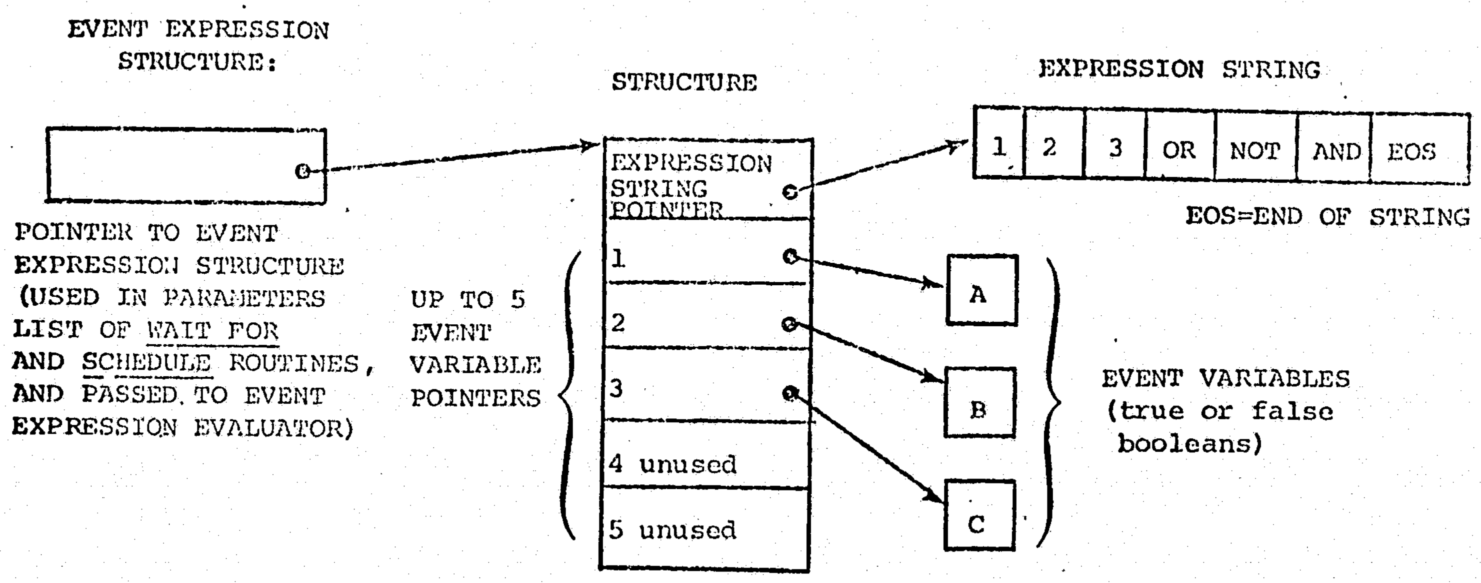
1) TYPE of event expression (SCHEDULE ON, UNTIL, or WHILE, or WAIT FOR).

2) PCB POINTER.

Figure 10-12

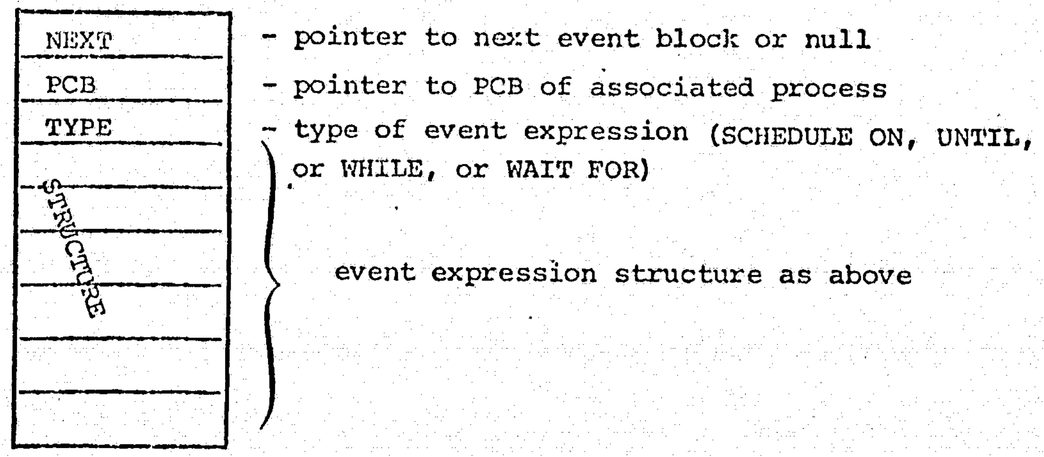
EVENT EXPRESSION STRUCTURE, EVENT BLOCK, EVENT BLOCK QUEUE

EVENT EXPRESSION: A AND NOT (B OR C)

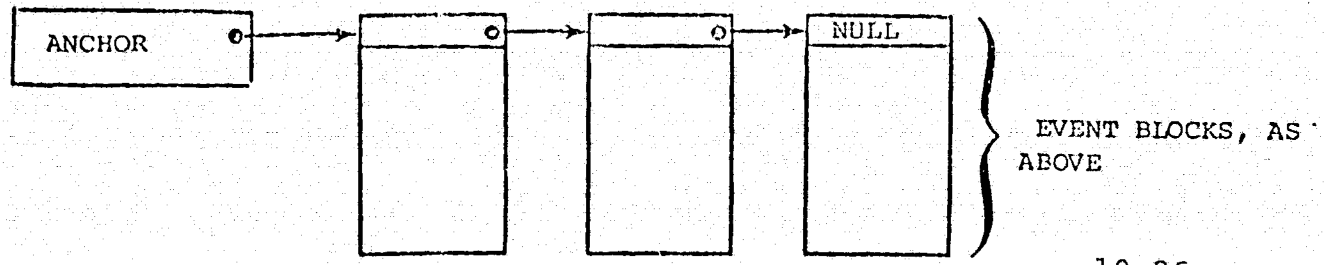

EVENT EXPRESSION STRUCTURE:
STRUCTURE
EXPRESSION STRING
EXPRESSION STRING POINTER
1
2
3
OR
NOT
AND
EOS
EOS=END OF STRING
POINTER TO EVENT EXPRESSION STRUCTURE (USED IN PARAMETERS LIST OF WAIT FOR AND SCHEDULE ROUTINES, AND PASSED TO EVENT EXPRESSION EVALUATOR)
UP TO 5 EVENT VARIABLE POINTERS
1
2
3
4 unused
5 unused
A
B
C
EVENT VARIABLES (true or false booleans)

The expression string is an encoded reverse Polish form of the event expression suitable for stack evaluation. Events A, B, and C are represented by 1, 2, and 3 respectively, indicating the relative positions in the event expression structure. The operators AND, OR, NOT, and EOS (End of string) are coded in a way which distinguishes them from event variable representations.

EVENT BLOCK:


NEXT
PCB
TYPE
STRUCTURE
- pointer to next event block or null
- pointer to PCB of associated process
- type of event expression (SCHEDULE ON, UNTIL, or WHILE, or WAIT FOR)
event expression structure as above

EVENT BLOCK QUEUE: representing 3 "activated" event expressions



3) EVENT EXPRESSION STRUCTURE POINTER.

If the expression is immediately true, an event block is not queued, and the routine returns with an indicator that the expression was not activated. In this case, the WAIT FOR routine does not pass control to the process selector, but returns control to the executing process.

The event expression structure must be copied to the event block because it is created by the compiled code in temporary storage, and does not remain beyond the execution of the statement. See Figure 10-13 for a flowchart of this routine.

10.2.6.2 Event Expression Evaluator Routine. This routine is called by 1) the enqueue routine described above, and 2) by the event processor (described next) when an event has changed state. It takes a pointer to an event expression structure as input and returns a boolean result which is the value of the represented event expression. Using the polish string form of the expression and a simple push-down stack, it actually carries out the logical operations on the event variables. Since the condition is satisfied when the expression value is false for the SCHEDULE WHILE type and true for the other types, the routine inverts (applies the NOT operation to) the result of a WHILE expression. Thus, the Evaluator always returns true if an event expression condition is satisfied. See Figure 10-14 for a flowchart of the Event Expression Evaluator.

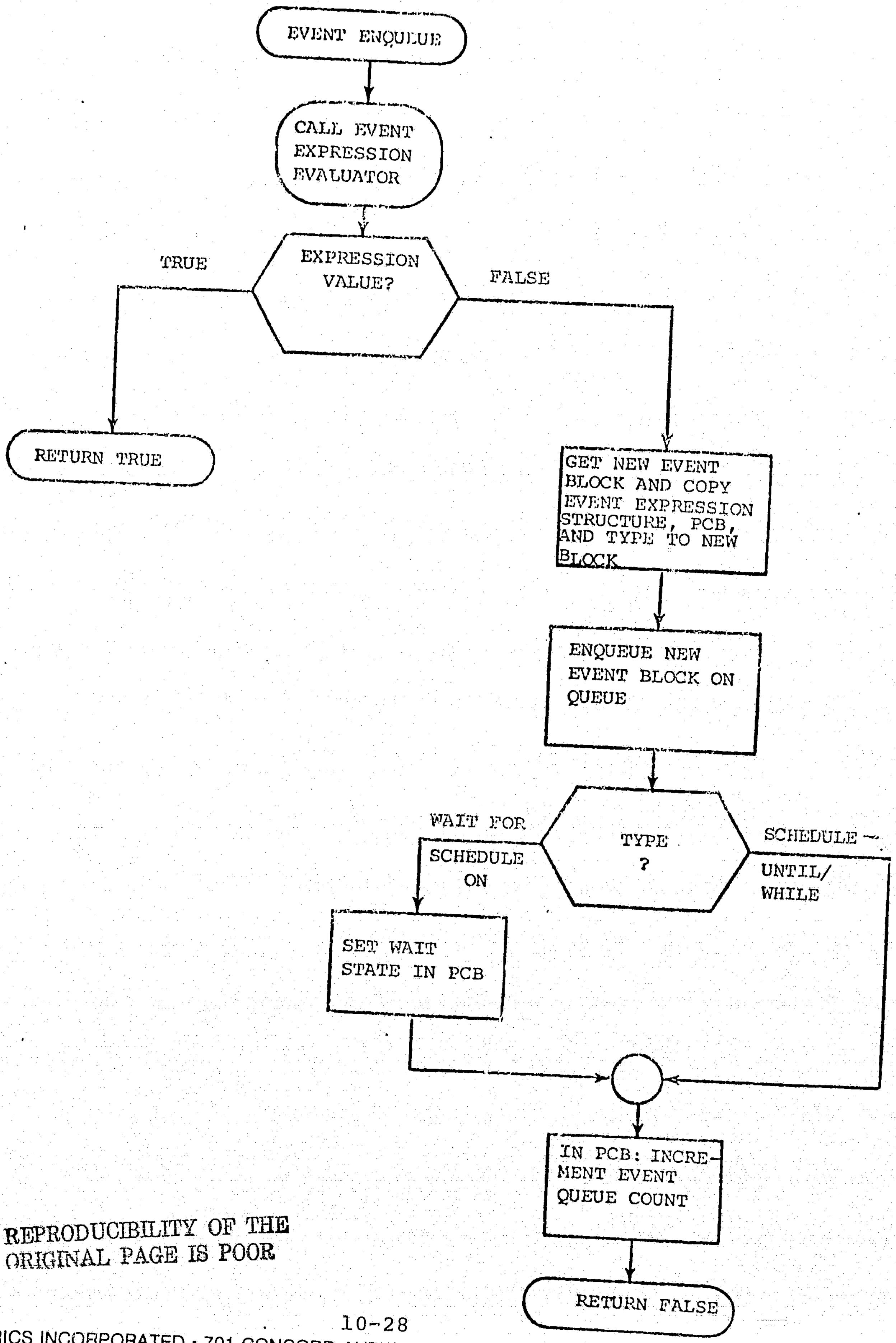
This routine has the following parameters:

1) TYPE

2) POINTER to event expression structure

10.2.6.3 Event Processor. This routine is called by the SET, RESET, and SIGNAL Service Routines for normal events and by the Scheduler and the Terminate Subroutine for process events. It re-evaluates activated event expressions by calling the Event Expression Evaluator for each event expression on the event block queue. If the Evaluator returns with a true expression, the Event Processor performs the appropriate action for that condition (readying or cancelling a process), and the event block is removed from the queue and freed. If an event block is encountered with the "terminated" flag set, it is removed and freed. The Terminate Subroutine need only set this flag to de-activate an event expression. See Figure 10-15 for a flowchart of the event processor.

Figure 10-13


EVENT ENQUEUE
CALL EVENT EXPRESSION EVALUATOR
EXPRESSION VALUE?
TRUE
FALSE
RETURN TRUE
GET NEW EVENT BLOCK AND COPY EVENT EXPRESSION STRUCTURE, PCB, AND TYPE TO NEW BLOCK
ENQUEUE NEW EVENT BLOCK ON QUEUE
TYPE ?
WAIT FOR
SCHEDULE ON
SCHEDULE — UNTIL/ WHILE
SET WAIT STATE IN PCB
IN PCB: INCREMENT EVENT QUEUE COUNT
RETURN FALSE

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

Figure 10-14

EVENT EXPRESSION EVALUATOR

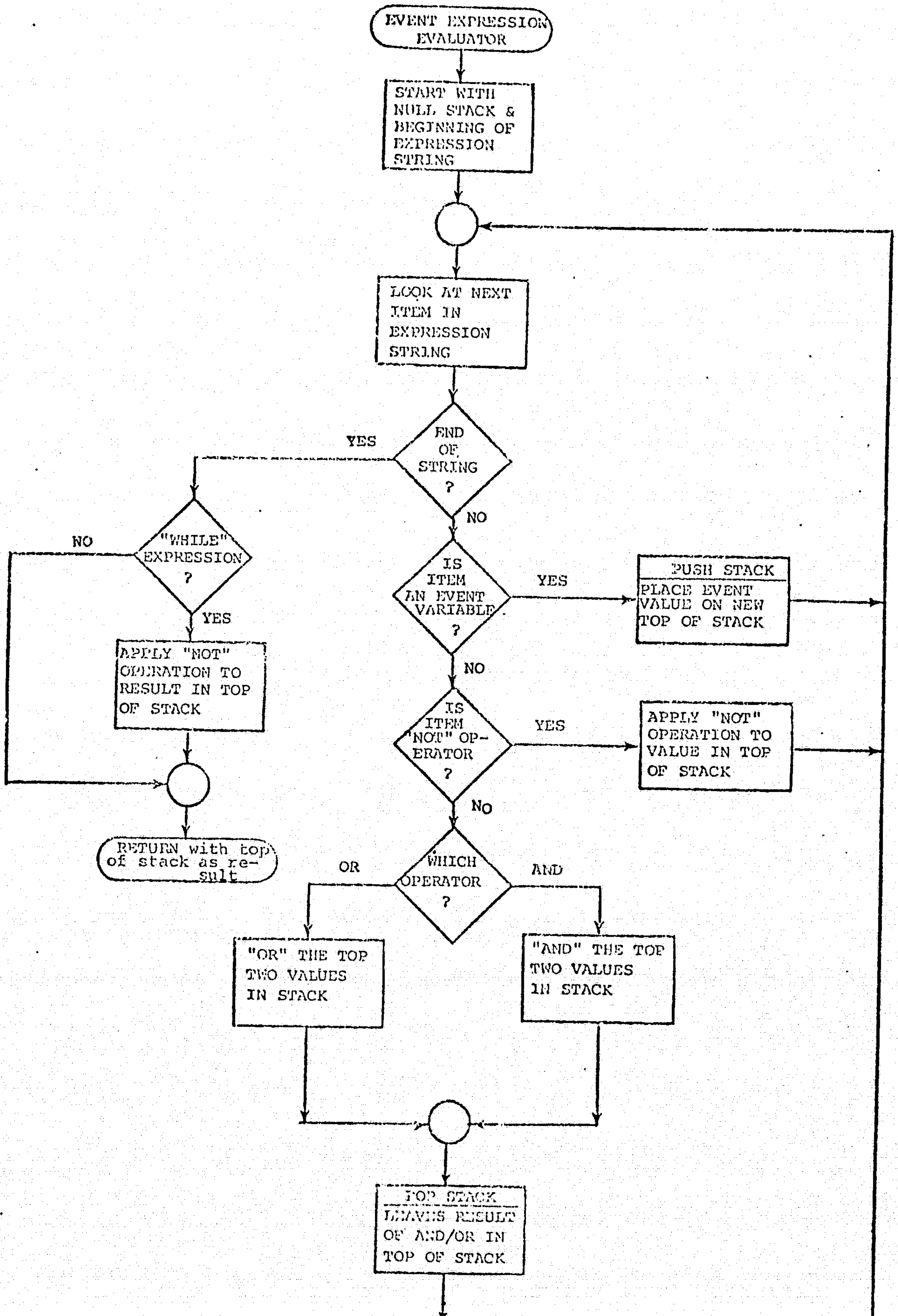
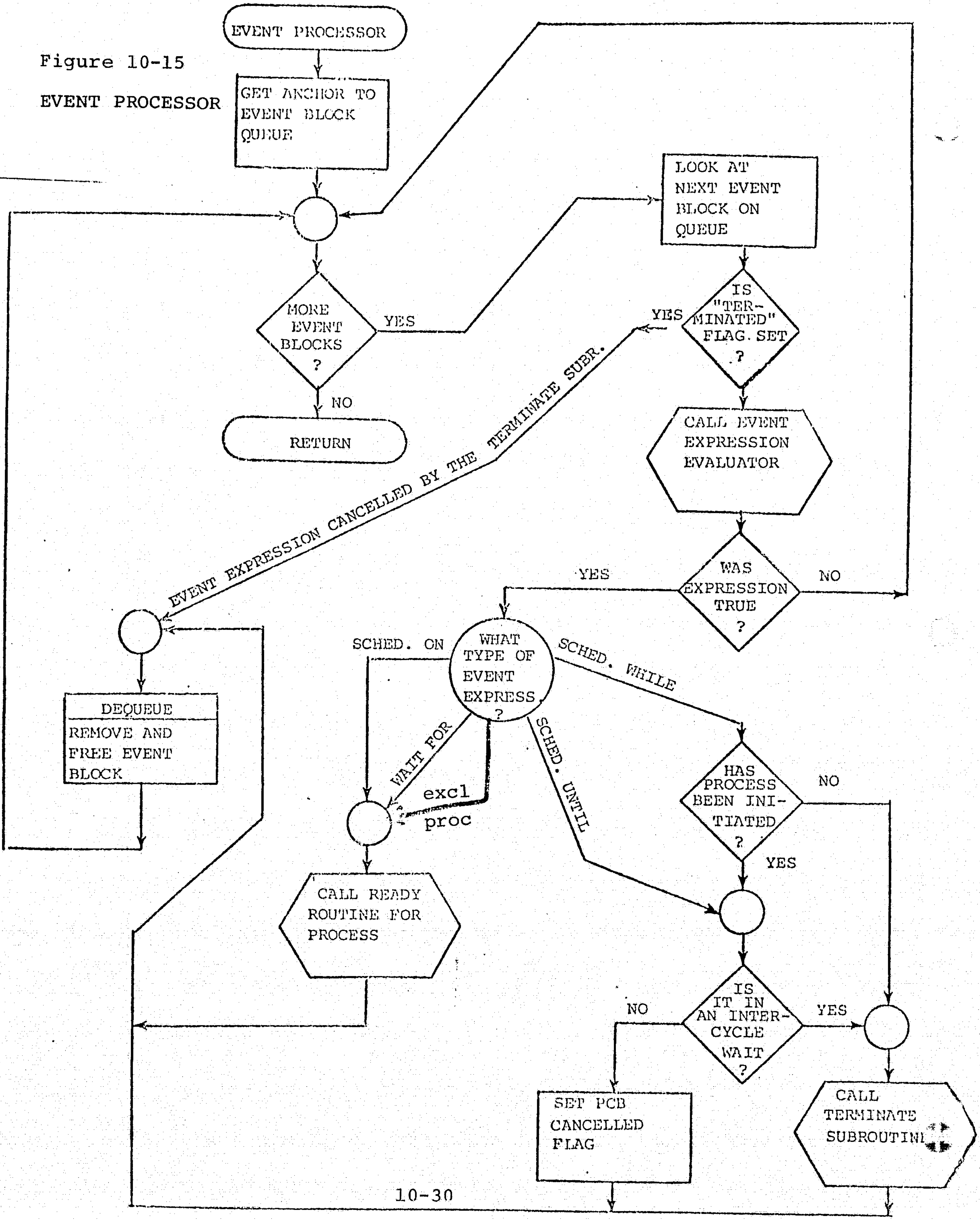

EVENT EXPRESSION EVALUATOR
START WITH NULL STACK & BEGINNING OF EXPRESSION STRING
LOOK AT NEXT ITEM IN EXPRESSION STRING
END OF STRING ?
YES
NO
"WHILE" EXPRESSION ?
NO
YES
APPLY "NOT" OPERATION TO RESULT IN TOP OF STACK
RETURN with top of stack as result
IS ITEM AN EVENT VARIABLE ?
YES
PUSH STACK
PLACE EVENT VALUE ON NEW TOP OF STACK
NO
IS ITEM "NOT" OPERATOR ?
YES
APPLY "NOT" OPERATION TO VALUE IN TOP OF STACK
NO
WHICH OPERATOR ?
OR
AND
"OR" THE TOP TWO VALUES IN STACK
"AND" THE TOP TWO VALUES IN STACK
POP STACK
LEAVES RESULT OF AND/OR IN TOP OF STACK

Figure 10-15

EVENT PROCESSOR


EVENT PROCESSOR
GET ANCHOR TO EVENT BLOCK QUEUE
MORE EVENT BLOCKS ?
YES
NO
RETURN
LOOK AT NEXT EVENT BLOCK ON QUEUE
IS "TERMINATED" FLAG SET ?
YES
EVENT EXPRESSION CANCELLED BY THE TERMINATE SUBR.
CALL EVENT EXPRESSION EVALUATOR
WAS EXPRESSION TRUE ?
YES
NO
DEQUEUE
REMOVE AND FREE EVENT BLOCK
WHAT TYPE OF EVENT EXPRESS ?
SCHED. ON
WAIT FOR
excl
proc
SCHED. UNTIL
SCHED. WHILE
CALL READY ROUTINE FOR PROCESS
HAS PROCESS BEEN INITIATED ?
NO
YES
IS IT IN AN INTER-CYCLE WAIT ?
NO
YES
SET PCB CANCELLED FLAG
CALL TERMINATE SUBROUTINE

10.2.7 Timer Management

The following real time statements make use of timer management routines:

WAIT - causes the active process to wait for a specified time interval or until a specified time.

SCHEDULE (IN or AT option) - causes the newly created process to wait before initial execution.

SCHEDULE (REPEAT EVERY or AFTER option) - causes the newly created process to execute cyclicly with a specified period between either the beginning (EVERY) or the end (AFTER) of one cycle to the beginning of the next.

SCHEDULE (UNTIL option) - causes the newly created process to be cancelled at a specified time.

These timing services are provided by two routines which control the use of the interval timer. The timer enqueue routine is called by any routine requesting a time interval. A type code indicates what action is to be performed when the specified time arrives. The timer interrupt routine is called by the statement processor when the software interval timer drops to zero. These two routines operate on a timer queue, each element of which represents a separate timer request. The queue is ordered by time of expiration, so that the first element on the queue is the next to expire. The value in the timer is such that it will cause an interrupt at the time specified in the top queue element.

10.2.7.1 Timer Enqueue. The timer enqueue routine takes the following actions:

1) If the time value (time of expiration) was supplied in relative form (as determined by the type), it is converted to absolute form.

2) If the time of expiration is already past, the routine returns with a "not enqueued" indication.

3) Otherwise, a new queue element is acquired, the input parameters are copied to it, and the element is placed on the queue by order of time of expiration.

4) If the new element was placed on top of the queue in 3), the value in the hardware timer is altered to reflect the new top element.

5) The routine returns with the "enqueued" indication.

A flowchart appears in Figure 10-16.

This routine has the following parameters:

1) PCB pointer

2) TIME VALUE (relative or absolute)

3) INTERVAL TIME

10.2.7.2 Timer Interrupt Routine. This routine gets control from the statement processor when the timer causes a pseudo interrupt. It takes the top element (the one representing the expired interval) off the queue, carries out the specified action, frees the old top queue elements, and loads the timer with the appropriate value for the new top element. The actions for the expired elements are to ready or cancel a process. A special test is made for an interval representing a SCHEDULE statement REPEAT EVERY option, since there is the possibility that the last cycle ran longer than the specified period between beginnings of cycles. If the process is not in an inter-cycle wait state, an error is indicated, and the process is not made ready. This causes the cyclic process to skip a cycle.

There is also a special element on the queue (called the clock element) which is used to keep the timer running in the absence of any timer requests. Both the clock element and any REPEAT EVERY elements are re-enqueued instead of freed, since they represent self-perpetuating intervals. The most appropriate value for the clock interval is the maximum value that can be placed in the timer. A flowchart appears in Figure 10-17.

Figure 10-16

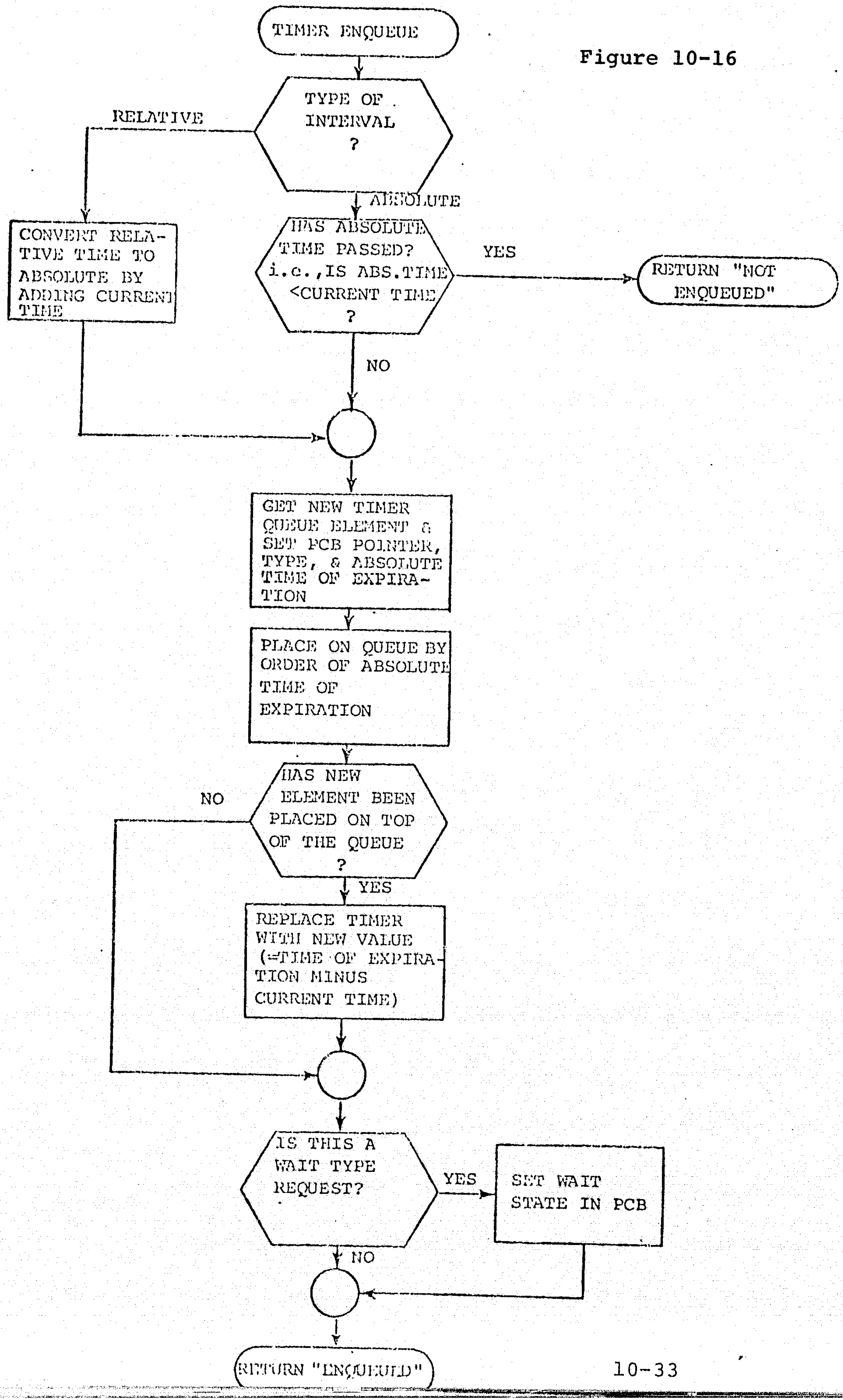
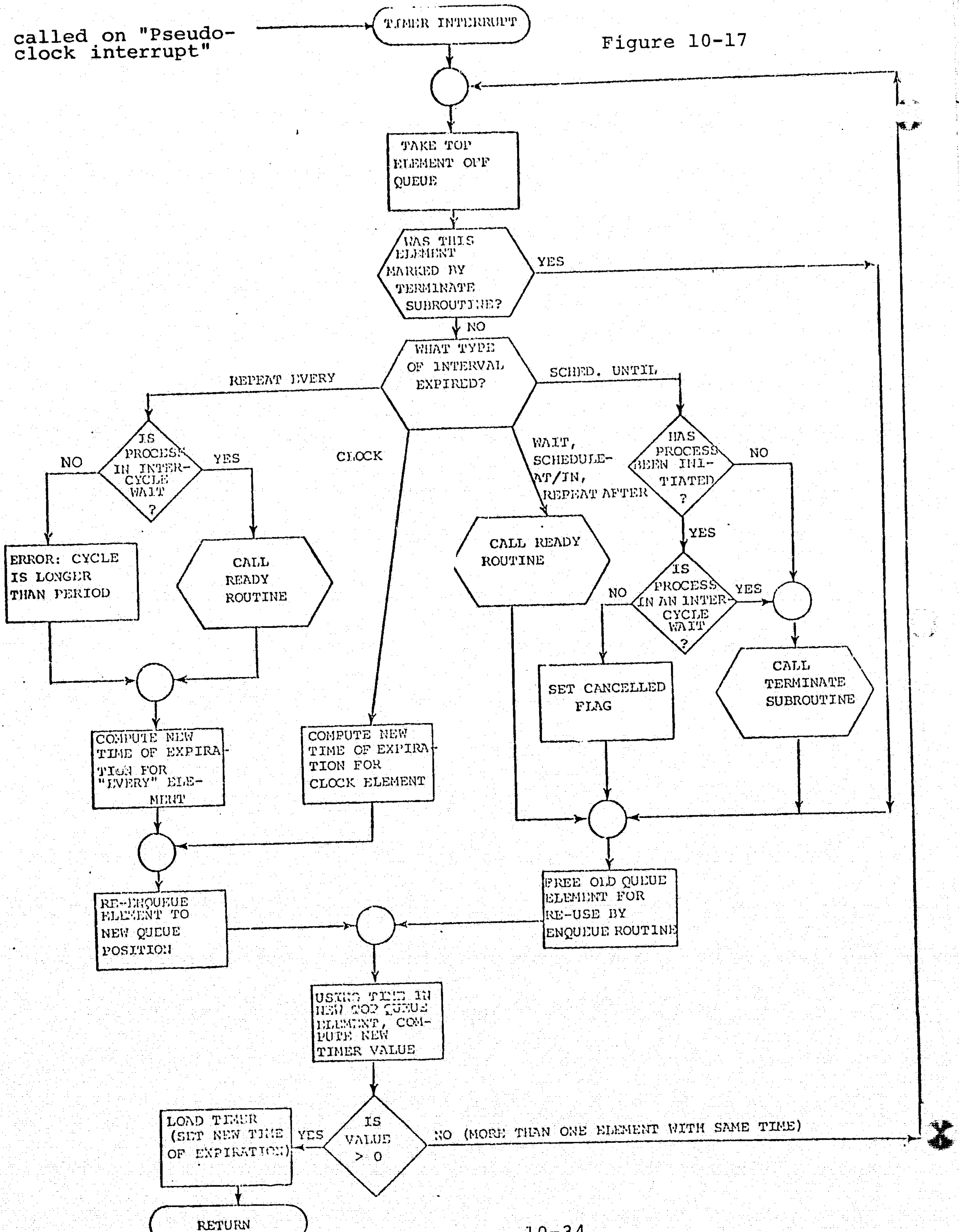

TIMER ENQUEUE
TYPE OF INTERVAL ?
RELATIVE
ABSOLUTE
CONVERT RELATIVE TIME TO ABSOLUTE BY ADDING CURRENT TIME
HAS ABSOLUTE TIME PASSED? i.e.,IS ABS.TIME <CURRENT TIME ?
YES
RETURN "NOT ENQUEUED"
NO
GET NEW TIMER QUEUE ELEMENT & SET PCB POINTER, TYPE, & ABSOLUTE TIME OF EXPIRATION
PLACE ON QUEUE BY ORDER OF ABSOLUTE TIME OF EXPIRATION
HAS NEW ELEMENT BEEN PLACED ON TOP OF THE QUEUE ?
NO
YES
REPLACE TIMER WITH NEW VALUE (=TIME OF EXPIRATION MINUS CURRENT TIME)
IS THIS A WAIT TYPE REQUEST?
YES
SET WAIT STATE IN PCB
NO
RETURN "ENQUEUED"

Figure 10-17


called on "Pseudo-clock interrupt"
TIMER INTERRUPT
TAKE TOP ELEMENT OFF QUEUE
WAS THIS ELEMENT MARKED BY TERMINATE SUBROUTINE?
YES
NO
WHAT TYPE OF INTERVAL EXPIRED?
REPEAT EVERY
CLOCK
WAIT, SCHEDULE-AT/IN, REPEAT AFTER
SCHED. UNTIL
IS PROCESS IN INTER-CYCLE WAIT ?
NO
YES
ERROR: CYCLE IS LONGER THAN PERIOD
CALL READY ROUTINE
CALL READY ROUTINE
HAS PROCESS BEEN INI-TIATED ?
NO
YES
IS PROCESS IN AN INTER-CYCLE WAIT ?
NO
YES
SET CANCELLED FLAG
CALL TERMINATE SUBROUTINE
COMPUTE NEW TIME OF EXPIRA-TION FOR "EVERY" ELE-MENT
COMPUTE NEW TIME OF EXPIRA-TION FOR CLOCK ELEMENT
RE-ENQUEUE ELEMENT TO NEW QUEUE POSITION
FREE OLD QUEUE ELEMENT FOR RE-USE BY ENQUEUE ROUTINE
USING TIME IN NEW TOP QUEUE ELEMENT, COM-PUTE NEW TIMER VALUE
IS VALUE > 0
YES
NO (MORE THAN ONE ELEMENT WITH SAME TIME)
LOAD TIMER (SET NEW TIME OF EXPIRATION)
RETURN

10.3 Statement Processor

Summary

The statement processor is a multi-purpose routine that gets control at the execution of every HAL/S-360 statement unless the NOTRACE option was specified at compile time. It functions as a clock to simulate flight computer time, as a recorder of diagnostic information, and as an interface to an external monitor controlling the simulation on a statement level. Because it is executed so frequently and because all of its functions may not be needed all of the time, a variable statement processor has been implemented which can be tailored dynamically, providing only those functions which are needed and thereby reducing CPU time. This also makes possible faster stand-alone operation, since the interface function is unneeded and has been eliminated from the default statement processor. This section outlines the technical method used, describes the optional features, and details the new interfaces controlling the variable statement processor.

Technical Method

The following method of dynamically swapping statement processors results in zero time and near-zero space overhead if it is not used, and a minimum of overhead if it is. All possible versions (256, given all combinations of 8 binary options) of the statement processor exist as separate load modules in a special run-time library. Selected versions are loaded into main memory only if and when requested by a service routine call. Actual overlaying of code is performed only at the start of the next call to the statement processor, allowing the swap request to be made from a statement processor exit routine. If n versions are selected, only n+1 OS LOADs are performed, no matter how many times the n versions are swapped. Each version is assembled with the minimum instructions needed to perform the selected options.

A statement processor re-configuration service routine may be called through the HALSIM simulation vector table. This routine is callable at any time after the HAL/S-360 load module is loaded. It performs the following actions:

1) (First call only) LOAD the Version Vector Table (VVT) and save its address in HALSYS.

2) (First call for given version only) LOAD the specified version and save its address in the VVT.

3) Save the address of the specified version in HALSYS.

4) Modify the first instruction of the statement processor to cause a branch to the swapper routine.

5) Return to the caller.

The next time the statement processor is called, the swapper routine gets control. Only four instructions long, it performs the following actions:

1) Locates the version already in main memory using the address saved in HALSYS by 3) above.

2) Overlays the existing statement processor using one MVC instruction. This also corrects the modification made in 4) above.

3) Branches to the new statement processor.

The details of the interfaces for the statement processor and its reconfiguration service routine are given in the HAL/S-SDL Interface Control Document.

11.0 THE MACRO LIBRARIES

The HAL/S compilers are written in XPL and execute on compatible IBM 360 computers. HAL/S-360 generates 360 machine code and HAL/S-FC generates AP-101 machine code. The object code produced by the compiler contains calls to library routines. The library routines have been written in the assembly language of the target computer. In order to facilitate the writing of assembly language routines which interface with object code of a HAL/S compiler, a collection of macros has been written for the 360 and a second collection for the AP-101. The AP-101 macros are described in Section 5.2.7 of the HAL/S-FC Compiler System Specification. The 360 macros are described in Sections 3.7 and 5.1.4 of the HAL/S-360 Compiler System Specification.

12.0 ACCESS ROUTINES FOR THE SDF TABLES

SDFPKG is an IBM-360 assembly language program comprised of five CSECTS: SDFPKG, LOCATE, PAGMOD, NDX2PTR, and SELECT. Its function is to provide a demand paging form of access to data contained within SDFs. SDFPKG can be separately link edited and employed as a loadable and deletable service module, or it may be linked directly with other software. The latter is the case with the HAL/S-360 stand-alone diagnostic system. It is important to realize that SDFPKG is not part of the HAL/S compiler but rather a collection of routines for accessing tables built by phase 3 of the compiler. The use of these tables is up to the individual user.

The HAL/S-360 Compiler System Specification (Section 5.9) describes the use of the Access routines. This section augments the description in the Compiler System Specification, providing details inappropriate in that forum.

12.1 Paging Area

Paging is done directly between core memory and the PDS (Partitioned Data Set) containing the Simulation Data Files generated by Phase 3. This is made possible by the list of TTRs contained within the last physical record of each SDF. A TTR is given for each record of the file. Reads can thus be accomplished via a FIND, POINT, READ sequence. Figure 12-1 shows the physical layout of an SDF with the TTR record (or page) at the end of the file. The TTR record contains pointers to all other file records and is itself in turn pointed to by a TTR in the User Data area of the PDS directory entry.

SDF records (or pages) are always 1680 bytes long. This is true even of the TTR page which may contain as little as 4 bytes of data. SDFPKG reads SDF pages from a PDS into a "paging area" which may consist of from 1 to 250 1680-byte areas. The upper limit can be increased by altering an assembly parameter in SDFPKG. This would, however, increase the size of SDFPKG by 16 bytes per added entry since the Paging Area Directory (PAD) would have to increase correspondingly. At the other extreme, SDFPKG will usually function properly with a 1 page paging area (if no Reserves are requested), but 2 pages is a recommended minimum.

The PDS containing SDFs to be read is normally identified by a HALSDF DD card. At the time of the initialization call, however, an alternate DDNAME can be specified. The PDS may have catenation levels as long as the user intends only to read data. If it is desired to "modify" an SDF (by requesting SDFPKG to operate in UPDAT mode) none of the pertinent SDFs may reside within a catenated level. This is an OS restriction.

At the time of the SDFPKG initialization (INITIALIZE) call the user program must specify the size of the "nucleus" paging area. This initially allocated area will then be available to, and exclusively controlled by SDFPKG until the time of the termination call (TERMINATE). SDFPKG makes provisions for dynamic expansion and contraction of the paging area size (within the 250 page limit) via one or more AUGMENT (increase paging area) calls and RESCIND (remove all augments) calls. The RESCIND call always reduces the paging area size to the initial (nucleus) area.

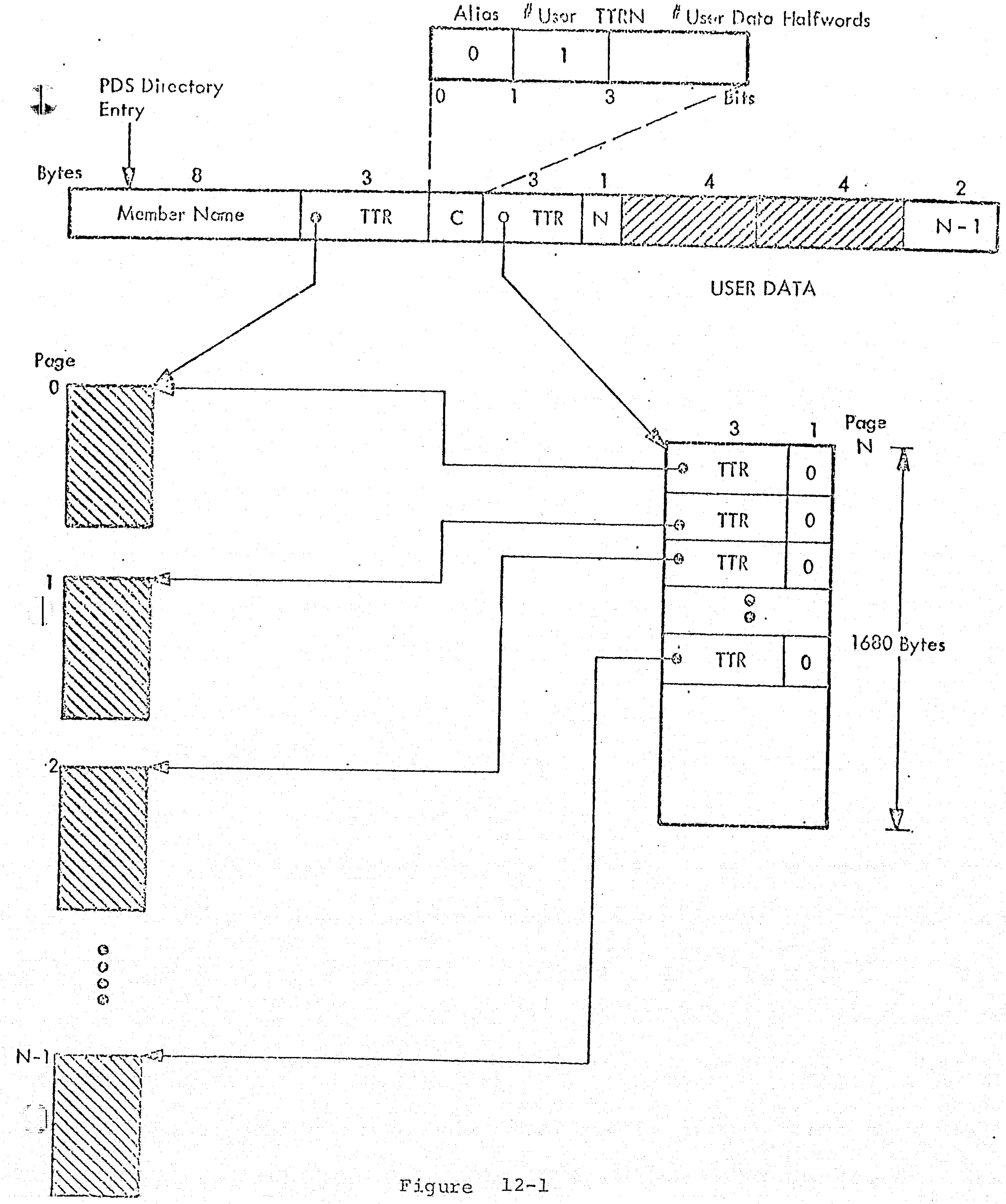


Figure 12-1

PDS-level Organization of the Simulation Tables

SDFPKG acquires the core memory necessary for the nucleus paging area either by executing a GETMAIN or by receiving it from the user program. The core memory necessary for AUGMENTs, however, must always be provided by the user. If SDFPKG is instructed to GETMAIN the nucleus paging area, it will free it via FREEMAIN at the TERMINATE call. This is true of any GETMAINs performed by SDFPKG.

12.2 Virtual Memory Considerations

SDFs are built by Phase 3 in a virtual memory environment and they are manipulated by SDFPKG in the same way. This implies that all SDF data items have "pointer" addresses (i.e. address in virtual memory space). In addition, if the item resides in core, it has a core memory address. As described in the HAL/SDL ICD, a pointer consists of a fullword whose high-order 16 bits contain an SDF page (record) number, and whose low order 16 bits contain an offset relative to that page (i.e. a displacement of from zero to 1679 bytes). SDF pages are numbered from zero so the pointer consisting of a fullword of zeros identifies the first byte of data in an SDF.

The fundamental form of data access provided by SDFPKG accepts a pointer as input and returns the core address of the corresponding data as output. The core address, of course, lies somewhere within the paging area. If the necessary SDF page was already in the paging area this is a fast operation. If not, paging is performed as necessary and is transparent to the user program. This process of "location" can be requested explicitly by the user software through a LOCATE call, but normally the user program will employ the higher-level SDFPKG mode calls which will then perform the necessary "locates" implicitly and totally internal to SDFPKG.

Whether locates are explicit or implicit, the important point is that almost all SDFPKG mode calls result in returning to the user the core location (and corresponding virtual memory pointer for reference purposes) of some data item. This data item may be an SDF Directory Root Cell, Block Data Cell, Symbol Data Cell, Statement Data Cell, Block Node, Symbol Node, Statement Node, or merely some arbitrary SDF location if an explicit LOCATE call was made. The page containing the item of interest is in core memory at that point and the user program may extract data (or insert data) using the core address provided.

It is normally the case, and especially true when a small paging area is used, that data "located" in this fashion may be overwritten as a result of a subsequent SDFPKG. If the user program wishes to guarantee the continued existence of the located data at the advertised core address, the RESV (Reserve) disposition parameter should be specified at the time of the initial mode call. SDFPKG then increments a reserve count maintained in the Paging Area Directory (PAD) for the page containing the located data and ensures that that page will not be overwritten until the reserve count has been decremented to zero. At some later time the user program can "free" the data by making any mode call that re-locates the data item and specifying RELS (Release). Since it is actually pages and not specific locations that are reserved it is only necessary to locate any part of the page in order to free it.

Users should be careful to limit the use of RESERVES if small paging areas are employed since each reserve makes one more paging area slot unavailable for further reads. Also, all pages that are reserved should be ultimately released. A RESCIND call will result in an abort (Abend 4011) if reserved pages are detected in the augmented portion of the paging area.

The third disposition parameter MODF (Modify) can only be used if UPDAT mode was specified at the time of the INITIALIZE call. MODF informs SDFPKG that the located item will be altered by the user. As a result, SDFPKG will rewrite the affected page back to the PDS (HALSDF or alternate DDNAME) prior to overlaying it with newly read pages. Again, SDFs to be altered cannot lie in a catenation level.

If the user program cannot determine until <u>after</u> the SDFPKG call that RESV, RELS, or MODF is desirable, then one or more of these disposition parameters can be specified by a DISP (mode 6) call which applies such parameters retroactively to the previously located item.

12.3 SDF Selection

SDFPKG allows simultaneous access to an unlimited number of SDFs. This means that the paging area can contain assorted pages from a number of different SDFs. In order for SDFPKG to know which SDF is to be referenced in support of the users call, it is necessary for the user to specify or "select" the proper SDF. This can be done in two different ways. The first method is to make an explicit SELECT call to SDFPKG with the 8 EBCDIC character SDF name (##CCCCCC) as input. Unless overriden all further SDFPKG data access requests will be directed to this SDF. The second method is called "Auto-Selection". By specifying the AUTO_SELECT disposition parameter and including the SDF name as an auxiliary input, SDFPKG calls will reference the specified SDF. Auto-selection is slightly slower than explicit selection but is useful if SDFs are to be randomly referenced.

When an SDF is selected for the first time following the INITIALIZE call, SDFPKG performs a BLDL for that PDS member, extracts the TTR list from the last SDF page, extracts certain data from the Directory Root Cell and incorporates all of this information into a File Control Block (FCB) for that SDF. The FCB is allocated from a block of memory called the FCB area which is discussed in the next section. The new FCB is then linked into a binary tree structure ordered by SDF name so that later selections can rapidly find the FCB needed to access data in the file. With one exception, once an FCB is created, it is maintained until a TERMINATE call resets all SDFPKG variables and data areas. This means that the FCB area may eventually become filled with FCBs and require extension.

If the user program knows beforehand that SDFs will be accessed in a serial fashion, or if core space is at a premium, then SDFPKG can be instructed at the time of the INITIALIZE call to operate in the ONEFCB mode, i.e. only one FB is kept so that a new SELECT will cause the new FCB to be built over the old one.

12.4 FCB Area

The FCB Area is similar to the Paging Area in that an initial amount must be allocated at the time of the INITIALIZE call. The user can specify what the allocation is to be or accept the default of 1024 bytes. Additionally, the user has to decide whether to provide SDFPKG with an FCB Area or to let SDFPKG obtain one via a GETMAIN. If the user supplies an FCB Area, then he must be prepared to supply additional areas (via the AUGMENT call) whenever the current FCB Area is exhausted. This condition is signalled by a return code of 12 meaning that a select failed due to insufficient space to construct an FCB. If the user does not wish this flexibility, then SDFPKG can be allowed to GETMAIN the initial FCB Area, or the MISC parameter can be set to 1 on the INITIALIZE call, which will allow automatic GETMAINs regardless of who allocated the initial area. In this mode of operation, subsequent GETMAINS for 512 bytes each will be performed as needed and this activity will be totally transparent to the user program. Again, all such GETMAIN'ed areas are freed when SDFPKG is called to TERMINATE. ONEFCB mode is available regardless of whether the user or SDFPKG is responsible for FCB Area allocation. It should also be noted that although the AUGMENT call can extend either the

Paging Area or FCB Area (or both simultaneously), the RESCIND call only applies to the Paging Area, i.e. the FCB Area can only grow.

Each FCB requires 60 bytes plus 8 bytes for each page of the associated SDF. FCBs are thus highly variable in length.

12.5 Paging Strategy

The Paging Area Directory (PAD) contains an entry for each core slot (up to 250) and each entry contains, among other data, a reserve count and a usage count for the page. As mentioned, the reserve count is used to lock the page in its core slot as long as the count is non-zero. The usage count, however, keeps track of how recently that page had been accessed relative to the other pages in core. A global count of "locates" is maintained within SDFPKG and is inserted into the usage count field of the PAD entry when the page is accessed. The effect is one of a pseudo-clock. When an SDF page must be read into a core slot from the PDS, the core slot that is both unreserved and least recently accessed is overlayed by the new data. If, however, the modification flag for that PAD entry indicates that the old page is in a modified state (UPDAT mode only) then the page is written out prior to being overlain. At the TERMINATE or RESCIND call all modified pages are written out to the PDS.

13.0 XPL -- INTERMETRICS VERSION

The standard XPL language provides insufficient support for a compiler as sophisticated as the HAL/S compilers. Intermetrics has added facilities in three ways:

1) Direct extensions to the language.

2) Additional implicitly declared procedures and variables.

3) An extensive set of MONITOR calls.

These added facilities are described in Sections 13.1, 13.2, and 13.3. In addition to the extensions mentioned above, facilities have been developed for dealing with large XPL programs:

4) Documentation aids and user options.

5) Perform updating functions on XPL source programs.

6) Make modifications in XPL load modules.

13.1 Direct Extension of the XPL Language

Declaration Statements

In addition to the DECLARE statement, the following declaration statements are supported:

a) ARRAY <var-name> (<dimension>) <data type>;

This statement behaves exactly like the DECLARE statement with one exception; the data is not allocated in the standard XPL data area, thus preventing the waste of a significant amount of the XPL base register addressing space. Instead, a data-area relative pointer is generated which is used to address the data. The purpose of ARRAY data is simply to extend the severely restricted addressing range for DECLARE data at the expense of a code penalty for each reference. Large but infrequently used tables are prime candidates for ARRAY-type declarations.

b) BASED <var-name> <data type>;

This statement reserves a word to contain a pointer to a block of data which exhibits the properties of the specified data type. No dimension information is required, and will be ignored if specified. It is the user's responsibility to guarantee that the pointer word is properly set prior to any references to the variable. Unless over-ridden using a special case of the ADDR built-in function, pointer de-referencing will always occur on any reference to the variable. The pointer may be set using the assigment:

```
COREWORD(ADDR(<var name>)) = address;
```

The dynamic address may either be the address of existing data (to allow equivalencing) or may be obtained from a MONITOR call (which performs an OS GETMAIN call) for true dynamic allocation.

c) COMMON <var name> [(<dimension>)] <data type>;

This statement also behaves exactly like the DECLARE statement except that the data is allocated in an area which will remain in core between program phases. This allows XPL programs to be separated into individual phases with a common data base for tables, etc.

d) COMMON ARRAY <var name> (<dimension>) <data type>;

This is the COMMON equivalent for ARRAY data, the purpose being to allow allocation of large arrays without using up the base register resources.

e) COMMON BASED <var name> <data type>;

This statement behaves exactly like the BASED statement except that the pointer is allocated in the common data area for shared use by subsequent phases.

The following restrictions apply to the above mentioned data types:

1) ARRAY, COMMON, and COMMON ARRAY statements may not be used to allocate data of type CHARACTER, and

2) BASED and COMMON data of any kind may not be initialized via the INITIAL feature.

It is also now possible to initialize variable with negative numbers using the form:

- <constant>

thus eliminating the necessity of using twos-complement hexidecimal constants for initializing with negative quantities.

The LITERALLY attribute is also somewhat changed from the original XPL. Originally, any variable declared LITERALLY went into a global table and remained in effect for the balance of the compilation, regardless of the nesting depth at the time of the declaration. Now, data declared LITERALLY is kept in the symbol table, and is removed from the table when the enclosing procedure is ended. As a side-effect, variables declared LITERALLY can now have cross-reference information collected on them.

13.2 Additional Implicitly Declared Procedures and Variables

A number of built-in functions have been added to the compiler to assist in program development or to allow for faster execution of frequently used functions.

1) The following functions have changed in meaning from the original description:

COREWORD(X)

According to "A COMPILER GENERATOR", X is a word index, or word-aligned address. However, in the Intermetrics version, X must be a byte address, and the user must himself guarantee that the lower-most two bits are 0's (fullword aligned).

ADDR(<var>)

This function is identical to the described specification except in the case where <var> is declared as BASED. In this case, ADDR(BASED_VAR) yields the address of the pointer word for BASED_VAR. If the address of the beginning of the data pointed to by BASED_VAR is desired, use the form ADDR(BASED_VAR(0)).

2) The following built-in functions have been added to the XPL system:

LINE_COUNT

This function returns the number of lines which have been printed on the SYSPRINT file since the last page eject.

SET_LINELIM(<number>)

This procedure establishes the number of lines which will be printed on the SYSPRINT file before an automatic page eject and header line will be printed.

LINK

This procedure performs the functions necessary to exit the current program phase and pass control to the next phase on the PROGRAM DD sequence, preserving COMMON data and any other dynamically allocated space which has not been deallocated.

PARM_FIELD

This function returns a character string which contains the entire parameter specification coded on the PARM= option on the EXEC card. If no PARM is specified, a null string will be returned.

STRING(X)

This function transforms the variable X (which should be FIXED for proper usage) into a CHARACTER descriptor. X should have the form:

```
   Length-1        Data Address
 ___________|_________________________

   8 bits           24 bits
```

The data pointed to by the data address should be a series of EBCDIC bytes to be treated as a CHARACTER string.

STRING_GT(S1,S2)

This function returns a TRUE value if the contents of string S1 is greater than the contents of string S2, based on the collating sequence of the characters, irrespective of the lengths of S1 and S2. Otherwise, the value is FALSE. This is functionally equivalent to padding the shorter of S1 or S2 with blanks and then comparing the strings.

ABS(X)

This function returns the absolute value of X (Note: "80000000", the maximum negative number has no representable absolute value, and returns "7FFFFFFF", the maximum positive number simply to guarantee that the result of ABS is always positive).

13.3 MONITOR Calls

CALL MONITOR(0,n);

Closed output file n and performs a FREEPOOL on the DCB.

F=MONITOR(1,n,name);

Writes any data remaining in the buffer for PDS output file n. Issues STOW macro using member name indicated by 'name' (must be 8 characters padded with blanks). Then close and FREEPOOL's DCB. Returns 0 if member is new. Returns 1 if member was replaced.

F=MONITOR(2,n,name);

Performs FIND macro in PDS input file n using member name specified by 'name' (must be 8 characters). If n=4 or n=7, first FIND attempt uses DDNAME INCLUDE and then tries DDNAME OUTPUT6. Returns 0 if member found. Returns 1 if member not found.

CALL MONITOR(3,n);

Closes input file n and performs FREEPOOL on DCB.

CALL MONITOR(4,n,b);

Changes LRECL and BLKSIZE of FILE(n) to "b" instead of default of 7200. Must preceed first use of FILE(n).

CALL MONITOR(5,ADDR(DW));

In forms monitor of location of double word aligned work area (DW) to be used as communication area for later use by monitor calls 9 and 10. Monitor calls 9 and 10 will abend if MONITOR(5) is not performed first.

F=MONITOR(6,ADDR(based_var),n);

Performs conditional GETMAIN of n bytes of storage (SUBPOOL=22) and places address of storage into based_var pointer. Storage is set to zero. Return code is 0 if storage was obtained and 1 if not enough storage was available.

F=MONITOR(7,ADDR(based_var),n);

Performs FREEMAIN of n bytes of storage at address obtained from based_var pointer. The based_var pointer is not modified.

CALL MONITOR(8);

Not in use. If called, produces ABEND 3000.

F=MONITOR(9,op);

Performs floating point evaluation as specified by value of 'op'. Operands are obtained from work area whose address was passed via a MONITOR(5) call. The first operand is taken from the first double word of the work area and the second operand from the second double word. The result is placed in the first double word of the work area. A SPIE exit is used to detect underflow and overflow conditions. Return code is 0 if the operation succeeds, 1 if the operation fails (under or overflow).

The values of op are:

OP	Function
1	add
2	subtract
3	multiply
4	divide
5	exponential (arg1**arg2)
6	SIN(arg1)
7	COS(arg1)
8	TAN(arg1)
9	EXP(arg1)
10	LOG(arg1)
11	SQRT(arg1)

F=MONITOR(10,string);

Performs character to floating point conversion upon characters in 'string'. Return code is 0 if result is valid, 1 if conversion was not possible. The result is placed in the first double word of the work area provided by the MONITOR(5) call.

CALL MONITOR(11);

Not used - a no-op.

string=MONITOR(12,p);

Converts floating point number in first double word of work area to standard HAL character form. Value of 'p' indicates whether operand is single precision (p=0) or double precision (p=8).

point=MONITOR(13,name);

Performs DELETE of current option processor and then LOADs an option processor specified by 'name'. The option processor loaded is called and passed a pointer to the PARM field in effect at the time of compiler invocation. The option processor passes the PARM field and establishes an options table (see Chapter 9) whose address is passed back as a return value. If 'name' is a null string, the pointer to the existing options table is returned.

F=MONITOR(14,n,a);

Interface to routines which create Simulation Data Files. Value of 'n' selects a function; value of 'a' supplies supplementary data:

n	Function	a
0	Open	option flags
1	Write	area address
2	Stow & Close	member name

I=MONITOR(15);

Returns Revision Level and Catenation Number from last MONITOR(2) call. Catenation number is obtained from PDS directory data and Revision Level from user data field as specified in the HAL/SDL ICD. The values are returned in the left and right halfwords of the result.

CALL MONITOR(16,n);

Sets flags in byte to be returned as high order byte of return code at end of compilation. Flags are passed as right most byte of fullword 'n'. If high order bit of 'n' is zero, flags are OR'ed into existing flags. If high order bit of 'n' is one, flags replace existing flags.

CALL MONITOR(17,name);

Causes 'name' to be copied to third parm field (if any) passed to MONITOR by the program that invoked the compiler. See HAL/SDL ICD.

T=MONITOR(18);

Returns elapsed CPU time since beginning of run in units of .01 seconds.

F=MONITOR(19,addr_list,size_list);

Performs a list form conditional GETMAIN. Returns 0 if GETMAIN succeeds, 1 if GETMAIN fails. Storage obtained is not cleared. Subpool 22 is used.

CALL MONITOR(20,addr_list,size_list);

Performs a list form FREEMAIN using same type operands as MONITOR(19).

I=MONITOR(21);

Performs a variable conditional GETMAIN which acquires the largest remaining contiguous area of main storage. The memory is immediately FREEMAINed and the amount obtained is returned as the value of the call.

F=MONITOR(22,n,a);

Cause LOADing, calling and DELETEing of Simulation Data File Access Package (SDFPKG). Used only by HALSTAT.

string=MONITOR(23);

Returns the 10 character string obtained from the ID field of the File Control Block of the first phase of the compiler. The ID field is maintained by the XPLZAP program and contains the identifying string printed on the header of each page of the HAL listing.

13.4 Documentation Aids and User Options

The XPL compiler, unless specifically requested otherwise, will give a complete source listing of the XPL program plus a symbol table listing including variable cross-reference information based on statement numbers (negative numbers indicating statements where assignments are performed). The compiler has the additional capability to provide, upon specific request, a summary at the end of each procedure, indicating which global variables have been referenced and/or assigned, and which global procedures have been called. In addition, the compiler has another option which expands the symbol cross-reference data to include the list of procedure names which either referenced a global data item or called a global procedure, thus providing a two-way cross-reference set.

Control toggles can now be set in four ways inside of XPL comments:

$<char> :	invert the current sense of toggle <char>
$<char>+:	turn on toggle <char>
$<char>-:	turn off toggle <char>
$<char>@:	set toggle on or off depending upon the setting at the start of the compilation

In addition, the appearance of '$<char>' in the PARM field will turn on the corresponding toggle for the entire compilation. The following toggles are useful:

Toggle	Action
L	List Program source, annotated with statement number, current relative program counter, and current procedure name. (Default = On).
D	Dump symbol table and other useful statistics at the end of the compilation. (Default = On).
R	Collect cross-reference data for each symbol (based on statement numbers) and print with symbol table. (Default = On).

S — Dump symbol table at the end of each Procedure, if any local data is declared.
(Default = Off).

I — Print Impact summary, indicating variables outside the scope of any procedure which were referenced, plus procedures called.
(Default = Off).

V — Expand variable cross reference to include names of procedures referencing data and names of procedures calling other procedures.
(Default = Off).

Z — Allow execution of XPL program even if severe errors were detected by compiler.

The following PARM field options are recognized by the compiler:

LISTING2

- list only lines containing errors and associated errors messages on the LISTING2 dataset.

SYTSIZE = nnn

- expand the default symbol table size from the default size (200) to nnn, which is the predicted high-water mark of the symbol table.

REFSIZE = nnn

- expand the default cross-reference table from the default size (500) to nnn, which is the predicted number of cross-reference entries.

MACROSIZE = nnn

- expand the number of LITERALLY declarations from the default size (100) to nnn.

PROCSIZE = nnn (needed only in conjunction with $V toggle)

- expand the number of allowable procedure definitions from the default SIZE(SYTSIZE/4) to nnn. Note that REFSIZE must also be increased by about 30% when $V is On.

NLIST

- change the default settings for toggles L, D, and R to Off.

Two additional output files can be generated upon request from the XPL compiler. The following description shows the form of the output, the file name, and the toggle switch which turns on the output.

Output File	Toggle	Description

OUTPUT8	U	For each XPL procedure, create a PDS member containing a template of the form:
		P: PROCEDURE(ARG);
		DECLARE ARG BIT(16);
		DECLARE LOCAL_VAR BIT(16);
		END P;
		which describes the procedure definition and all of the locally declared variables. If the I toggle is on, a copy of the impact summary is also included in the PDS.

OUTPUT6	W	For each XPL procedure which is called from other XPL procedures, create a PDS member which duplicates the listing generated via the V toggle.

For both PDS files, the member name is derived from the procedure name by:

1) eliminating all underline characters,

2) truncating the name to 8 characters, if necessary,

3) if duplicate of previously generated name, truncate name to 7 characters, if necessary, and catenate on uniqueness number.

The members on OUPUT6 may be later merged with the corresponding members on OUTPUT8 to either create a new PDS or sequential file which is a complete data description for each procedure in an XPL compilation.

13.5 Updater

UPDATER performs one (and only one) of three operations in each of its runs. Operation for a given job step is determined by the first card in the input stream, which is called the DIRECTOR card. This card and other control cards are characterized by having '$$' in columsn 1-2. The first word on the directory card must be NEW, NUMBER, or UPDATE. Any of these may be followed by one of the words LIST or NOLIST; the default is LIST for NEW and NOLIST otherwise. When the LIST option is in effect, a complete listing of the output file is written into the data set named on the SYSPUNCH DD card; usually 'SYSOUT=A, DCB=(RECFM=FBA,LRECL=133,BLKSIZE=7182)' is used. The heading for this listing is taken from the DIRECTOR card if any non-blanks are found after the control information (non-blanks here and between control information means characters other than blanks, commas, or equal signs).

NEW

The NEW operation takes card-images from the SYSIN input stream, adds file numbers, and stores the numbered file (into the OUTPUT3 data set).

NUMBER

The NUMBER operation is similar, except that it takes records from the source specified on the INPUT3 DD card (80 bytes or longer), truncates to 80 bytes if necessary, appends file numbers, and stores the modified file.

UPDATE

The UPDATE operation requires a NUMBERed file as input (INPUT DD card), and produces a modified file as output. The DIRECTOR card may additionally specify RENUMBER, in which case the output file is written with equally spaced numbers. (The order of RENUMBER and LIST/NOLIST, if both are specified, is not significant.)

After the word NUMBER on the NUMBER card, or after the word RENUMBER on the UPDATE card, the form INCR N, where N is a number, may be specified. This will cause the number N to override the default value of 100 for renumbering the file. The first record on the output file will have the value N, the second record will be 2*N, etc.

Following the UPDATE DIRECTOR card, UPDATE control cards and detail cards are supplied. If none are present, the input file is simply copied to the output file. This form of the UPDATE operation may be used to duplicate a file with or without renumbering, or if LIST is specified and OUTPUT3 is DD DUMMY, a listing of the INPUT3 file is obtained.

Detail cards may be specified in two ways. UPDATER was designed to handle card images which have no space allocated for card numbers. However, in many cases, the card image actually does have space for a number, and UPDATER makes use of this: the first form of detail card is simply a card with ordinary text in columns 1 through 72, and a card number somewhere in 73-80. Any reasonable form for the number is valid, so long as it has no imbedded blanks and has a nonzero value. UPDATER replaces columns 73-80 with blanks when it moves the card to the output file.

The second form of detail card is required when some of the columns 73-80 are needed for text. In this case, the detail card is made up by a control card containing the number, followed by the text-card. For example,

```
$$ 34625
. . . THIS CARD MAY CONTAIN TEXT BEYOND COLUMN 72. . .
```

In both cases, the detail card is added to the file. If its number matches that of a record already present, that record is replaced; otherwise, the detail card is inserted.

The DELETE control card is of the form '$$ DELETE M' or '$$ DELETE M THRU N', where M and N are numbers. In the second form, N must be >= M. The effect of this request is to cause any records in the range M through N, inclusive, to be deleted. M and N need not be numbers of actual records in the file, but a warning is issued if no records at all are found in the M-N range.

The INSERT control card is of the form '$$ INSERT AFTER M' or '$$ INSERT AFTER M INCR N', and causes all the following cards up to the next control card to be inserted after the last record whose number is not greater than M. If renumbering is in effect, the number-increment used is the standard renumbering increment; if not, either the specified increment N, or a default value if INCR N is not specified, will be used so long as the resulting number does not equal or exceed the number of the next sequential record. If it does, renumbering is automatically activated from that point on.

The REPLACE control card is of the form '$$ REPLACE M', '$$ REPLACE M THRU N', '$$ REPLACE M INCR J', or '$$ REPLACE M THRU N INCR J', where M, N, and J are numbers. When the THRU form is used, N must be >= M. The effect of this command is to delete records in the range of M thru N inclusive, replacing them with all cards following up to the next control card, with numbering beginning at M. If INCR is not specified, the default increment of 10 is used. The same effect as in INSERT takes place if the numbering of the inserted cards exceeds that of the next sequential reocrd. It is not necessary that numbers M or N be in the input file, but a warning will be issued if there are no cards within the specified line number range.

The EXTRACT command is used to remove a section of a program from a larger file. It may be used as often as necessary to isolate various segments from a program. The allowable forms are '$$ EXTRACT M' or '$$ EXTRACT M THRU N', where M and N are numbers. The effect of the command is to skip from the current input record to line M, and then to copy lines M thru N inclusive to the output file.

The END command is used in conjunction with the EXTRACT command. The form is simply '$$ END'. If this card is at the end of a series of EXTRACT commands, the last specified record on the previous control card (or insertion if any were made) will be the last record on the output file. If the END card is not present, the rest of the input file from the current record to the end of the file will be copied to the output file.

Updater requires that detail-card numbers, the FROM values on DELETE cards, and AFTER numbers on INSERT cards form a strictly monotonic sequence. In the event that an invalid number sequence or other serious error is detected, updater causes the job-step to abend. This allows the use of 'DISP=(NEW,KEEP,DELETE)' on the OUTPUT3 DD card to avoid using up a data set name in case of a bad update.

The value of the renumbering increment is 100, and of the default insert-increment is 10.

When the listing option is in effect, it is necessary to specify "PARM='FREE=44000'" on the EXEC card of the job step; otherwise "PARM='ALTER'" may be used. It is suggested that a SYSPUNCH DD card always be used; if a listing is not wanted, use '//SYSPUNCH DD DUMMY'.

13.6 XPL ZAP

XPLZAP is a program designed to allow inspection and modification of XPL object files. It can be used on either single programs or concatenated compiler complexes. All modifications are logged in a free area in the File Control Block, up to a limit of 440 changes per module.

Each XPL file consists of four sets of data, each with its own mode of addressing. The program area addresses correspond to the addresses which appear to the right of the statement in the compiler listing. Local branch addressing is computed relative to the first instruction in the procedure. The data area addresses correspond to the sum of the displacements shown in the symbol table dump and the contents of the corresponding base register, which appear in the summary information at the end of the listing. The descriptor area has only one dedicated base register, and thus the displacement as shown in the symbol table may be used directly. The file control block may also be examined, but changes to this area are not recommended, as program failures may result.

The program is designed to operate either interactively or in the batch mode. In the batch mode, the control card images are printed on the output listing. In either mode, control card errors will inhibit subsequent modifications (until the next file command is given).

All control cards consist of a command character followed by a set of operands. All addresses and replacement operands are hexadecimal digits. The end of a control card or the character ';' stops the control card scan. In the following description, the character α is used to indicate the addressing mode. The allowable forms are:

Character	Addressing Mode
	Program area
D	Data area
C	Descriptor area
F	File Control Block area
I	Compiler Identification area

Any other characters for α will cause the program area to be used.

All addresses are truncated to the nearest halfword address. All replacement or verification data must be specified in halfword multiples, separated by blanks or commas. For the commands which accept string operands, do not attempt to specify the character quote (') within the string. This must be done in hexadecimal.

The compiler identification area is a 10 character field which is used to describe the particular compiler version. There is only one per XPL program complex, and it must be modified in its entirety. The standard format is:

'XXXX-RR.VL'

where:

XXXX indicates the compiler name,

RR indicates the release number,

V indicates the version number, and

L indicates a ZAP sub-level (blank or 0 being equivalent to unzapped complex).

The enclosed prototype JCL illustrates all of the necessary DD statements to run XPLZAP. The sequence "YOUR XPL FILE" is to be replaced by the appropriate data set name and any other specifications required by the installation to access the data set. A second example shows an actual XPLZAP run.

XPLZAP Command Summary

Items enclosed in brackets ([ ]) are optional operands.

Command	Description
Lα address* [length]	List "length" bytes in hexadecimal beginning at the specified address. If length is omitted, it defaults to 32 (20_{16}).
Dα address* [length]	Display "length" bytes in EBCDIC beginning at the specified address. If length is omitted, it defaults to 32 (20_{16}).
Rα address*r1,r2,...rn	Replace n halfwords starting at the specified address by halfwords r1, r2, etc. Previous errors will inhibit changes.
Vα address*v1,v2,...vn	Verify n halfwords starting at the specified address comparing with halfwords v1, v2, ..., vn.
X	Signify end of run. Any clean up will be performed at this time. An END-OF-FILE on SYSIN is equivalent.
G	Print the date and time of generation of the XPL module.
H	Print the log of previous replacements (includes addresses and date and time replacements were made).
M n	Specifies the maximum number of XPL object modules in a complex. This card must be the first card in an XPLZAP run if more than one XPL module is in a file, even if only one is being altered.
F n	Specifies that XPL module n is to be examined and/or altered. This command also allows replacements to take place if previous commands were in error.
C x+y x-y	Calculate the result of the expression involving hexadecimal operands, and print result. The expression is evaluated left to right with no precedence. Only + and - are supported.

* If α = 'I', the address field is omitted.

Command	Description
Rα address*'string'	Replace n/2 halfwords starting at the specified address by the n characters enclosed in quotes (n must be even).
Vα address*'string'	Verify n/2 halfwords starting at the specified address with the n characters enclosed in quotes (n must be even).
Sα address* s1,s2,...,sn	Search for occurrences of the pattern s1,s2,...,sn beginning at the specified address through the end of the area specified by α.
Sα address* 'string'	Search for occurrences of n/2 halfwords containing the n characters enclosed in quotes beginning at the specified address through the end of the area specified by α. (n must be even.)
; anything	No action (for commenting).

* If α = 'I', the address field is omitted.

```
//JOBNAME  JOB ACCT,PROGRAMMER.ID,REGION=60K,TIME=1
//XPLZAP   EXEC PGM=XPLSM,PARM='BATCH'
//STEPLIB  DD DISP=SHR,DSN=HALS.MONITOR
//PROGRAM  DD DISP=SHR,DSN=HALS.XPLZAP
//FILE1    DD DISP=OLD,DSN="YOUR XPL FILE"
//SYSPRINT DD SYSOUT=A,DCB=BLKSIZE=1330
//SYSIN    DD *
   <XPLZAP CONTROL CARDS>
      .
      .
      .
/*
```

FIGURE 1.   PROTOTYPE JCL

```
//JOBNAME  JOB ACCT,PROGRAMMER.ID,REGION=60K,TIME=1
//XPLZAP   EXEC PGM=XPLSM,PARM='BATCH'
//STEPLIB  DD DISP=SHR,DSN=HALS.MONITOR
//PROGRAM  DD DISP=SHR,DSN=HALS.XPLZAP
//FILE1    DD DISP=OLD,DSN=HALS.COMPILER
//SYSPRINT DD SYSOUT=A,DCB=BLKSIZE=1330
//SYSIN    DD *
M 4 ;  THIS EXAMPLE ZAPS BOTH THE FIRST AND SECOND FILES IN A 4 FILE COMPL
F 1 ;  IT DEFINES A .1 RELEASE LEVEL IN PHASE 1
VI  ' 360-13.0 '
RI  ' 360-13.01'
F 2 ;  NOW MAKE ACTUAL CHANGES IN SECOND FILE
V 51D6 4760
R 51D6 47F0
V 5254 9101 78A9 4780 F3BC
R 5254 1B11 4910 78B0 4780 F32E 47F0 F3BC
V 51C0 F31C ;  FIX RECOGNIZING CSE'S ACROSS CONDITIONALS
R 51C0 F32E
/*
```

FIGURE 2.   TYPICAL XPLZAP RUN

13.7 JCL and DD Names

Sample JCL for documenting XPL run:

```
//XPL  EXEC  PGM=MONITOR,
//           PARM='SYTSIZE=1800,REFSIZE=20000,LISTING2,$I,$V,$U,$W'
//STEPLIB DD DISP=SHR,DSN=HALS.MONITOR
//PROGRAM DD DISP=SHR,DSN=HALS.XCOMLINK
//INPUT2  DD DISP=SHR,DSN=HALS.LINKLIB
//SYSIN   DD DISP=SHR,DSN=your  XPL source program
//SYSPRINT DD SYSOUT=A
//LISTING2 DD DISP=MOD,DSN=your error log dataset
//OUTPUT8 DD DISP=OLD,DSN=your procedure template PDS
//OUTPUT6 DD DISP=OLD,DSN=your procedure reference PDS
//FILE1   DD DISP=OLD,DSN=your XPL object file
//FILE2   DD UNIT=SYSDA,SPACE=(CYL,3)
//FILE3   DD UNIT=SYSDA,SPACE=(CYL,3)
//FILE4   DD UNIT=SYSDA,SPACE=(CYL,3)
```

XPL Reference	DD NAME
=INPUT(0)	SYSIN
=INPUT(1)	SYSIN
=INPUT(2)	INPUT2
=INPUT(4)	INCLUDE (PDS)
=INPUT(5)	ERROR (PDS)
=INPUT(6)	ACCESS (PDS)
=INPUT(7)	INCLUDE or OUTPUT6
OUTPUT(0)=	SYSPRINT
OUTPUT(1)=	SYSPRINT (including carriage control)
OUTPUT(2)=	LISTING2
OUTPUT(3)-	OUTPUT3
OUTPUT(4)=	OUTPUT4
OUTPUT(5)=	OUTPUT5 (PDS)
OUTPUT(6)=	OUTPUT6 (PDS)
OUTPUT(7)=	OUTPUT7
OUTPUT(8)=	OUTPUT8 (PDS)
FILE(1,n)	FILE1
FILE(2,n)	FILE2
FILE(3,n)	FILE3
FILE(4,n)	FILE4
FILE(5,n)	FILE5
FILE(6,n)	FILE6