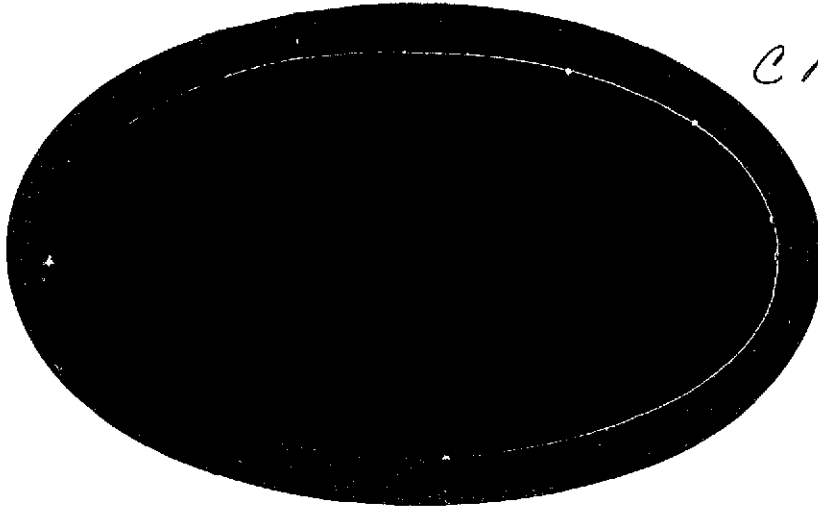


CR 134295

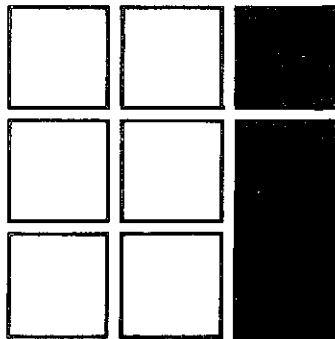


(NASA-CR-134295) HAL/S-FC COMPILER SYSTEM
FUNCTIONAL SPECIFICATION (Intermetrics,
Inc.) ~~83~~ P CSCI 09B

77 HC #725

N74-25733

G3/08 40529
Unclas



INTERMETRICS

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

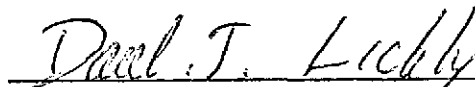
PRICES SUBJECT TO CHANGE

HAL/S-FC* COMPILER SYSTEM
FUNCTIONAL SPECIFICATION

29 April 1974

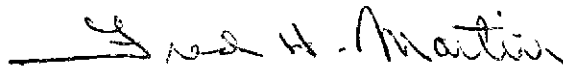
IR #59-3

Approved:



Daniel J. Lickly
HAL Language/Compiler Dept. Head

Approved:



Dr. F. H. Martin
Shuttle Program Manager

* For the IBM AP-101 Computer.

FOREWORD

This document was prepared for the Johnson Space Center, Houston, Texas, under contract NAS 9-13864.

Table of Contents

	<u>Page</u>
1. INTRODUCTION	1
1.1 Scope of Document	1
1.2 Language Definition	1
1.3 Scope of Implementation	2
1.3.1 Stand-Alone Environment	2
1.3.2 Software Development Laboratory Environ.	3
2. COMPILER ORGANIZATION	5
2.1 Overall Compiler Structure	5
2.1.1 The Monitor	5
2.1.2 Phase 1 - The Syntax Analysis Phase	7
2.1.3 Phase 2 - The Code Generation Phase	11
2.2 Internal Data Transfer	15
2.2.1 Monitor/Phase Data Relationships	15
2.2.2 Phase 1/Phase 2 Data Relationships	15
2.2.3 Data Passed to the Table Generation Phase	16
2.3 Compiler Development	17
3. COMPILER/ENVIRONMENT DATA INTERFACE	19
3.1 User Interface	19
3.1.1 Compiler Inputs	19
3.1.2 Compiler Outputs	21
3.2 System Interface	25
3.2.1 Job Control Language	25
3.2.2 Compiler Inputs	25
3.2.3 Compiler Outputs	26
3.2.4 Link Edit Support	28
3.3 SDL Interface	28
3.3.1 User Interface	28
3.3.2 System Interface	30



4.	COMPILER SYSTEM REQUIREMENTS	31
4.1	General	31
4.2	Hardware Requirements	31
4.2.1	Processor	31
4.2.2	Memory	31
4.2.3	I/O Support	31
4.3	Software Requirements	32
5.	RUNTIME SOFTWARE SUPPORT PACKAGE	33
5.1	Vector-Matrix Routines	34
5.2	Character Routines	36
5.3	Conversion Routines	36
5.4	Input/Output	37
5.5	Mathematical Functions	39
5.6	Miscellaneous Functions	41
5.7	Arithmetic Functions	41
5.8	Real Time Interfaces	43
6.	RESTRICTIONS AND DEPENDENCIES	45
6.1	Introduction	45
6.2	Compile-Time Characteristics	45
6.2.1	Character Set	45
6.2.2	Compilation Dependent Language Features	45
6.3	Runtime Characteristics	49
6.3.1	Character Set	49
6.3.2	Computer Dependent Language Features	49
Appendix A:	Compiler Directives	53
Appendix B:	Compiler Error Messages	57
Appendix C:	Compiler Options	65
Appendix D:	Runtime Errors	67

1. INTRODUCTION

1.1 Scope of Document

This document defines the functional requirements to be met by the HAL/S-FC compiler. It also defines the hardware and software compatibilities between the compiler system and the environment in which the compiler system operates.

The specification is for the HAL/S-FC compiler and the associated runtime facilities which implement the full HAL/S language. The HAL/S-FC compiler system will interface with the Software Development Laboratory (SDL). The functional specification of this HAL/SDL interface is also a part of this document.

This document describes both the construction of the HAL/S-FC system as functionally separate units and the interfaces between those units. The remainder of Section 1 presents an overview of the system's capabilities. Section 2 describes the internal structure of the compiler. Section 3 shows the data interfaces which will exist between the compiler and its environment. Section 4 specifies the hardware/operating system requirements of HAL/S-FC compiler. The general runtime support packages are found in Section 5. Finally, Section 6 specifies the computer-dependent aspects of the HAL/S-FC implementation. Several Appendices follow containing further details of specific facilities.

1.2 Language and Interface Definition

The HAL/S-FC compiler system will implement the HAL/S language as defined in the HAL/S Language Specification, dated January 22, 1974, including approved changes as of March 6, 1974. All references to the HAL/S language imply this document.

In addition, interfaces to other software systems which will be met by the HAL/S-FC compiler system are specified in the HAL/FCOS Interface Control Document (ICD) and the HAL/SDL ICD. In the event of any conflicting specification among any of these documents, the following document precedence will prevail:

Highest precedence	- HAL/S Language Specification
	- HAL/FCOS ICD
	- HAL/SDL ICD
Lowest precedence	- HAL/S-FC Compiler System Functional Specification

1.3 Scope of Implementation

The HAL/S-FC compiler will be implemented to compile HAL/S programs on the IBM 360/75 computer and to produce code for the IBM AP-101 computer. (The compiler system will be compatible, with minor modifications, with the IBM 360/370 computer series.)

The HAL/S-FC compiler will provide a complete system for the organized production of HAL/S programs. The computer system will facilitate a modular approach to the generation of large program units by allowing separate compilations of subsections of HAL/S code. In allowing such modular compilation, however, the static verification possible under a single large compilation scheme will not be lost. The compiler will automatically maintain such static verification information during separate compilations.

The compilation system will provide a generalized facility for the management of HAL/S program resources. Such resources as individual global variables or subroutines may be restricted to use by designated programmers. Additionally, management decisions governing use of particular HAL/S language features will be enforced automatically by the compiler. Since such individual management restrictions will not be known until well into the Space Shuttle program, the HAL/S-FC compiler will implement a system of Compiler Directives which will describe administrative action to be taken by the compiler. Directives will be added as the need for more controls arises.

Following compilation, the compiler system will aid in the checkout portion of program development. For each compilation unit, a Simulation Data File (SDF) will be generated for use by the SDL. Each SDF will contain symbol, statement, and cross-referencing information which can be accessed during simulation. A runtime package will be supplied to implement HAL/S language features not supported via in-line code generation. The compiler-generated executive linkages and the runtime package will be compatible with the Flight Computer Operating System (FCOS).

1.3.1 Stand-Alone Environment

The HAL/S-FC compiler system will be capable of executing within the standard batch processing facilities of OS/360. Operation in this environment is known as Stand-Alone

execution. The HAL/S-FC system will provide the facilities necessary to proceed from initial compilation through execution and debugging on an appropriate Interpretive Computer Simulator or an AP-101 in the Stand-Alone environment. These facilities will include:

- a. Complete compilation of all HAL/S language constructs. The design goal for compilation speed is 1000 cards per minute.
- b. Separate compilation of PROGRAM's, COMPOOL's, and external PROCEDURES (COMSUBS) in one invocation of the compiler.
- c. Production of compilation diagnostics.
- d. Generation of object code. The object code produced will not exceed that produced via hand-coded assembler language methods by more than 15% in either memory requirements or execution time.
- e. Link-editing of object code to resolve all external references.
- f. A complete runtime package implementing all built-in functions and operations defined in HAL/S. This package will include (or be compatible with) a full realtime support system (FCOS).

1.3.2 Software Development Laboratory Environment

The HAL/S-FC system will be capable of operating within the Software Development Laboratory (SDL). When operating in the SDL environment, the HAL/S-FC system will provide facilities which include:

- a. Complete compilation of all HAL/S language constructs. The design goal for compilation speed is 1000 cards per minute.
- b. Separate compilation of PROGRAMs, COMPOOLs, and external PROCEDURES (COMSUBS) in one invocation of the compiler.
- c. Production of compilation diagnostics.

- d. Generation of object code compatible with SDL link edit facilities. The object code produced will not exceed that produced via hand-coded assembler language methods by more than 15% in either memory requirements or execution time.
- e. A complete runtime package implementing all built-in functions and operations defined in HAL/S. This package will be compatible with a full realtime support system (FCOS).
- f. Provision for runtime package diagnostic capability with appropriate interfaces to the FCOS.
- g. Interfaces to the SDL necessary to allow implementation of the SDL. This includes static data interfaces in the form of tables and system control.

2. COMPILER ORGANIZATION

This section describes the functional operations that will be performed by the various parts of the HAL/S-FC compiler. The information is organized into three categories:

- 2.1 Internal operation of various elements in the compiler. Data interfaces between elements is mentioned only briefly.
- 2.2 Details of interfaces between various elements including descriptions of data transmitted.
- 2.3 Details of the XPL Translator Writer System¹ which will be used to develop the compiler.

2.1 Overall Compiler Structure

The HAL/S-FC compiler will be 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 6. The four modules of the compiler (Monitor, Phase 1, Phase 2, Table Generation) are described in more detail in the following sections.

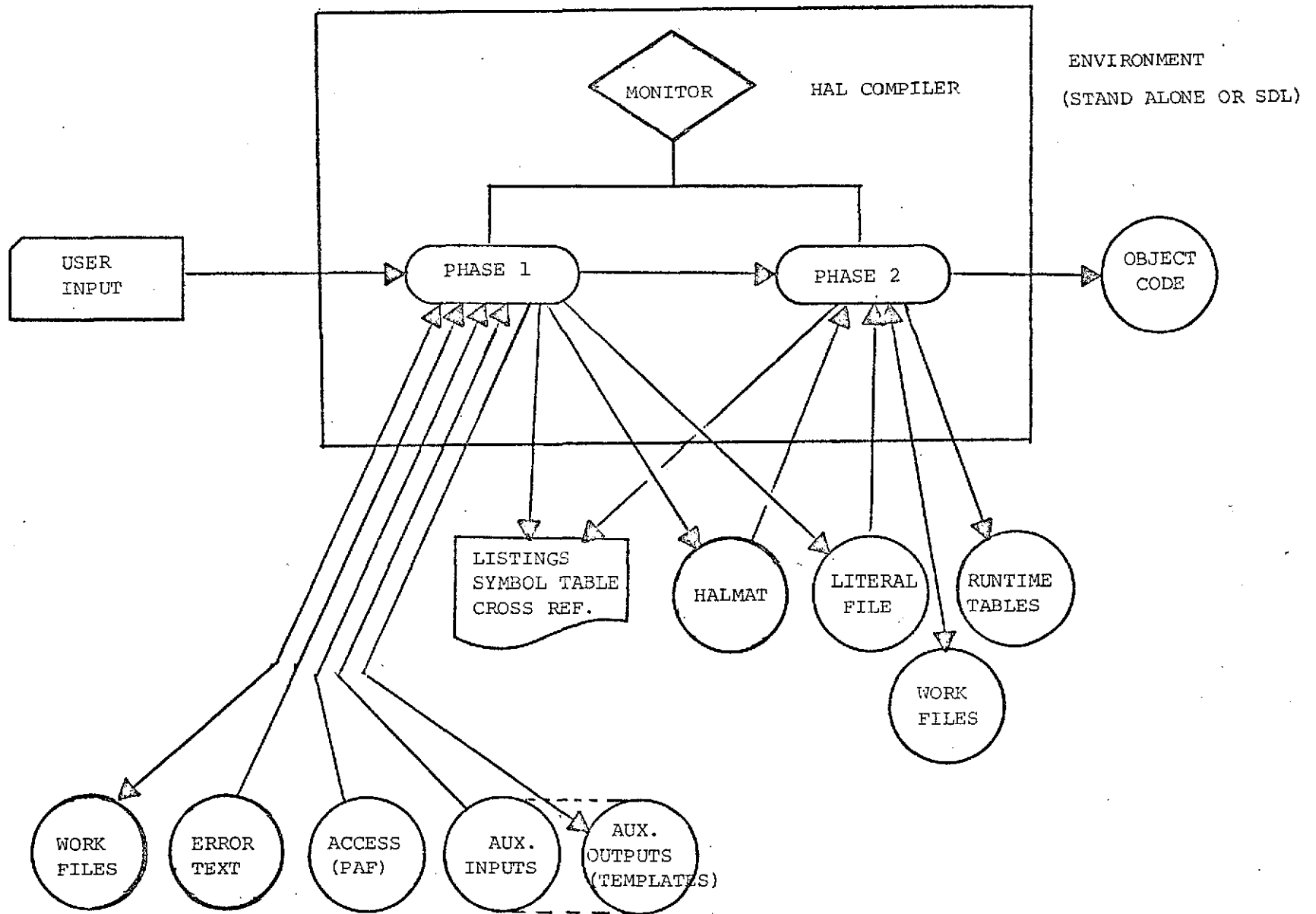
2.1.1 The Monitor

The Monitor will be the controlling module in a compilation. It will perform all sequencing and control operations.

The sequencing function of the Monitor will direct the compilation by deciding which of the other modules are in the computer memory. The Monitor will make use of overlay techniques to make maximum utility of available memory. The Monitor will supervise loading and execution of the other modules and pass any required information to the modules.

The control function of the Monitor will handle all interfaces between secondary modules in memory and the operating system under which the entire compiler runs. These

1. Mc Keeman, W.M., Horning, J.J., Wortman, D.V., A Compiler Generator, Prentice-Hall, Inc., 1970.



Overall Structure of the HAL/S-FC Compiler

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 Monitor will be written in OS/360 Basic Assembler Language.

2.1.2 Phase 1 - The Syntax Analysis Phase

The Syntax Analysis Phase will perform all syntactic and semantic analysis of the user's HAL/S source statements. This analysis will be driven by a parsing system which will generate a complete parse of the input. The parsing algorithm will detect and identify all syntax errors in the source statements and will make information generated as a result of the parse available to other sections of Phase 1.

Phase 1 will be 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 will be implemented by the compiler to allow separate compilation of COMPOOLS and external PROCEDURES and FUNCTIONS (COMSUBS). This system will allow 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 will be 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 will automatically generate these templates whenever a Unit of Compilation of type 1 or 2 is compiled. The templates will be compatible with standard INCLUDE library formats. See Section 3.2.2.

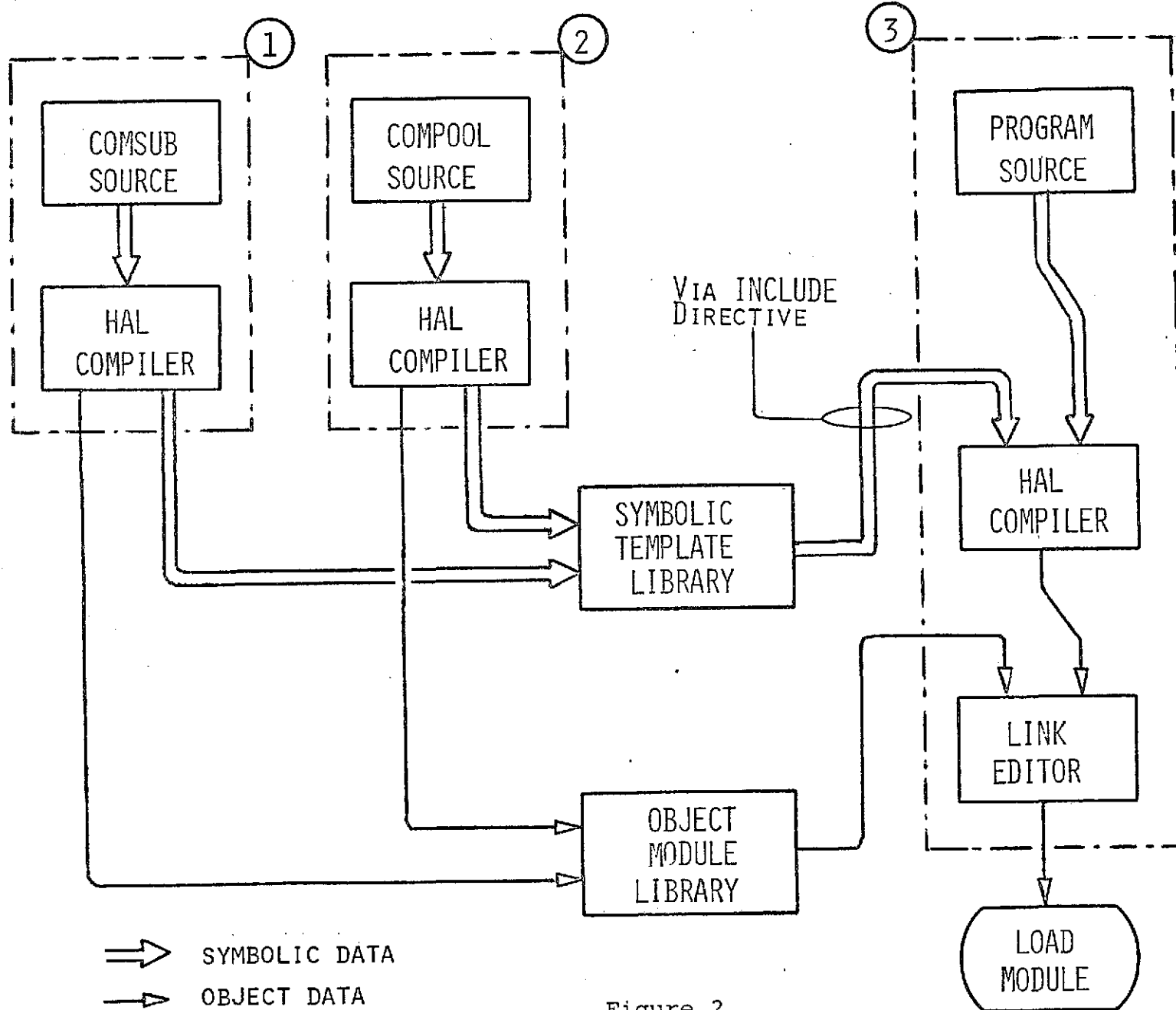


Figure 2.

HAL COMPILATION SYSTEM

Phase 1 is also responsible for production of the source listing and the symbol table/cross reference table listing. Details of the listings and other Phase 1 data interfaces may be found in Section 3. Phase 1, written in the XPL language, will consist 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.

2.1.2.1 The Scanner. The Scanner is sometimes called the Lexical Analyzer. It will scan the sequence of characters that comprise the source input (letters, digits, punctuation, spaces) and will generate a stream of tokens which are meaningful symbols of the Syntax Analyzer, (e.g. reserved words, identifiers, literals, and other so-called terminals). It will discard the semantically irrelevant text and handle imbedded comments. The proper interpretation of multi-line input will be done in the Scanner.

Each symbol will be converted to an internal "token" in a simplified format so that the analyzer is presented with a stream of uniform symbols. This will permit 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 will be called upon by the Syntax Analyzer as needed to deliver the next token from the input stream.

2.1.2.2 The Syntax Analyzer. The Syntax Analyzer will decompose the input stream tokens (parse the lexical format) to determine if it is legal according to the grammar of the language. Once the parse verifies the syntactical correctness of the input, control will be passed to appropriate semantic analysis routines.

The parse will be conducted using the table-driven algorithm of the XPL Translator Writer System. A description of this approach is given in Section 2.3.

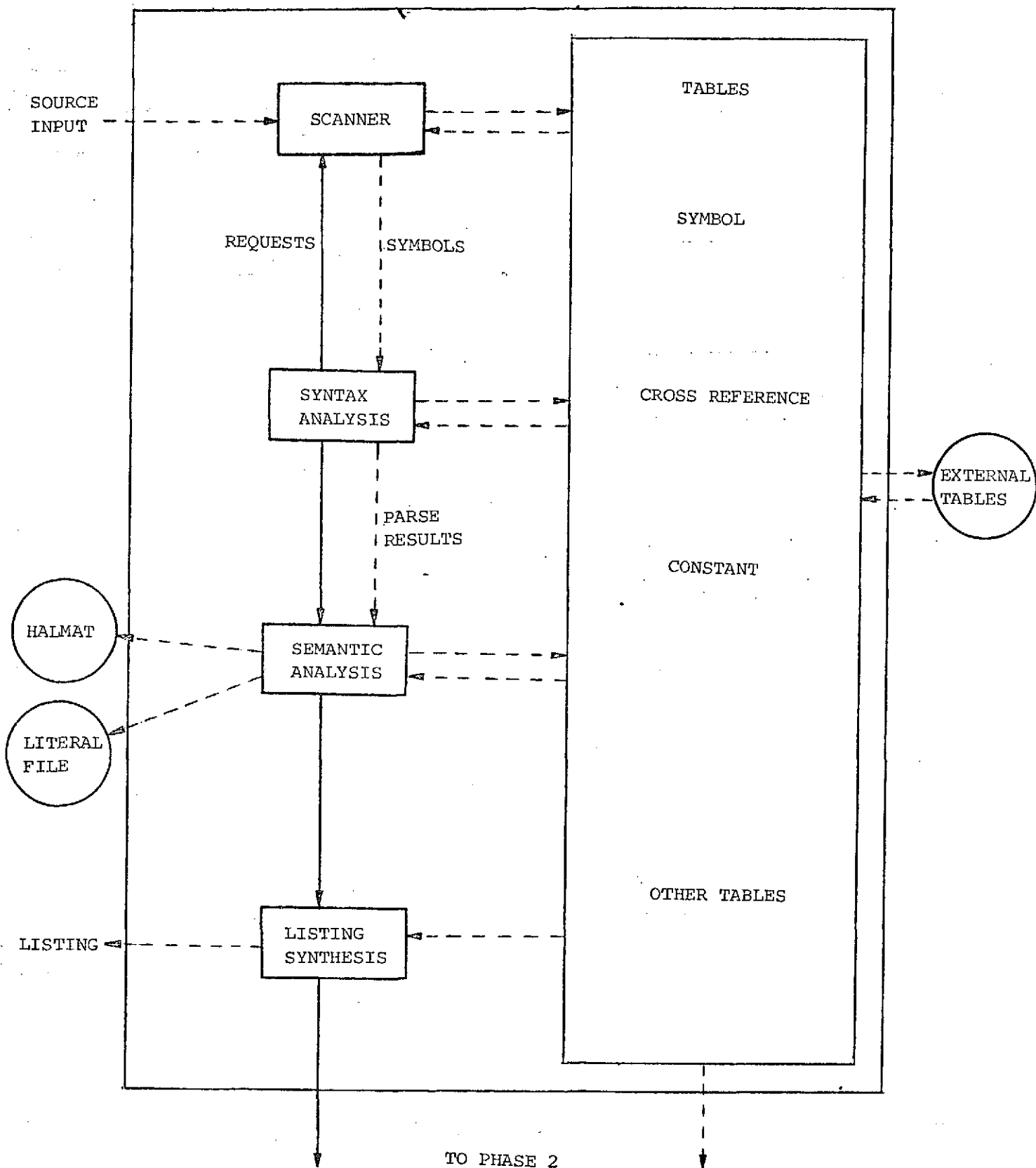


Figure 3.
Phase 1 Organization

2.1.2.3 The Semantic Analysis Routines. Once a complete syntactic check has been performed and the format identified, a semantic routine will be invoked. Given the particular construct and access to the compiler tables, the analysis routine will check for semantic correctness and then interpret 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 Section 2.2.2.

2.1.2.4 The Listing Synthesizer. At appropriate points in the analysis, the Listing Synthesizer will be given control. This routine will generate the fully annotated primary source listing by synthesizing the source statements. Further discussion of the primary source listing may be found in Section 3.1.2. The synthesis will be driven by the tables and other data generated during syntactic and semantic analysis.

2.1.3 Phase 2 - The Code Generation Phase

The Code Generation Phase will accept any necessary data from previous phases and use that data to direct the generation of object code for the target computer.

Phase 2 will produce, 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 will contain four (4) distinct sections:

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

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

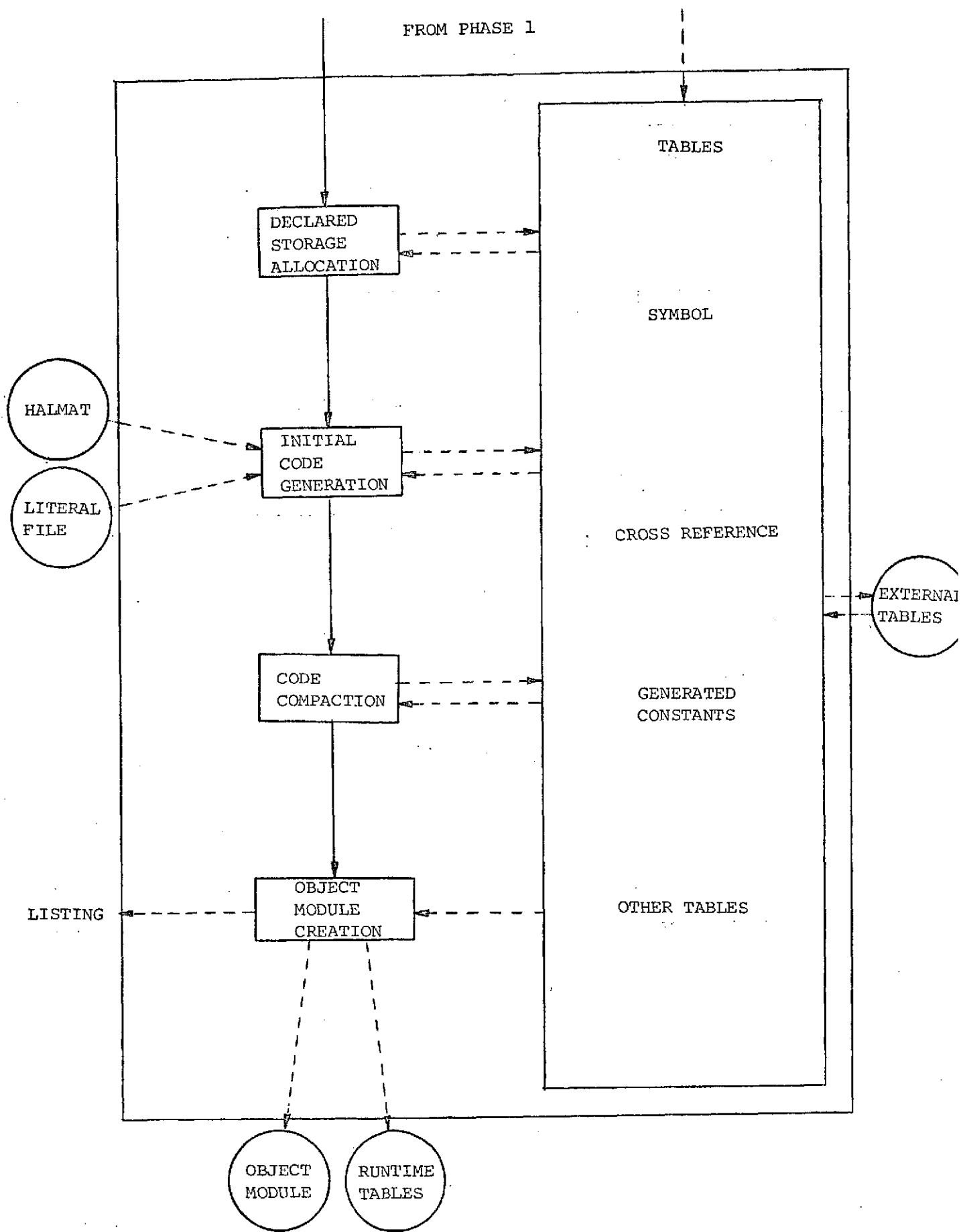


Figure 4.

Phase 2 Organization

2.1.3.1 Declared Storage Allocation. Using symbol table information generated by Phase 1, this routine will allocate the necessary memory for data explicitly declared by the user. The assignment of storage will be done in a manner to best take advantage of word alignment and frequency of use. Base registers are assigned to data at this time. With the AP-101 the various address classes (e.g. displacement <56, displacement <2K, bank 0, bank 0 displacement <2K, PC relative) will be considered to improve code generation.

2.1.3.2 Initial Code Generation. Using operations specified in the HALMAT intermediate code language, this module will generate actual target machine instructions. These instructions will be maintained in an internal form to allow for modification of instructions by later sections of Phase 2.

Included in the Code Generation section will be the building of the list of generated constants. This data will originally be obtained from the Literal File, which contains the constants in a generalized internal form. The generated constants will be specific to the context in which they are to be used; (e.g. generate an integer constant rather than a floating point constant).

During this phase, local 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 commutative 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.

2.1.3.3 Code Compaction. This section will operate 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 to reach any possible destination. The compaction process reduces this to a short instruction when possible.

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

2.1.3.4 Object Module Creation. This section will transform the internally coded instructions and data into standard 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.

In addition, this section will produce the formatted object code listing, if requested by the user, as well as the runtime tables needed during execution of the compiled code.

2.2 Internal Data Transfer

Communication between Phases of the HAL/S compiler will occur 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 will exist in the compiler. The relationships to be discussed in this section are those involved in inter-Phase communication. In general, data transfer will be in one direction only; i.e. since phases will operate in sequence and not concurrently, data will flow from earlier to later phases.

2.2.1 Monitor/Phase Data Relationships

The Monitor will not participate in the actual generation or retrieval of any inter-Phase data. It will act 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.

2.2.2 Phase 1/Phase 2 Data Relationships

The interface between Phase 1 and Phase 2 will be generated in the most target-machine-independent manner possible. The degree to which this machine-independence is achieved will determine the ease with which the code generator (Phase 2) can be modularly replaced. Such a replacement scheme will allow efficient implementation of a complete HAL/S compiler for additional target machines.

Phase 1 will pass information to Phase 2 via both in-memory tables and external files. The data passed via a common memory area will include all symbol table and cross reference table information. These tables will 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 will include status information, special request information, error condition data detected in Phase 1, and some literal data information.

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

The second I/O file will contain a description of the compiled HAL/S program in an intermediate language form known as HALMAT. The HALMAT for a given compilation will describe the HAL/S source program in an elemental, operation-by-operation form. All HAL/S statements will be represented as groups of operations. The operations will 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 HALMAT language itself will describe only HAL/S constructs and refer only to the tables generated by Phase 1. It will therefore be independent of the target machine's object code format. The form and organization of the HALMAT will, however, permit an orderly, operation-by-operation generation of target code by Phase 2.

2.2.3 Data Passed to the Table Generation Phase

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

The external file passed to the table generator will contain information concerning the individual HAL source statements as scanned by Phase 1 and translated into AP-101 code in Phase II. The file contains information to identify and locate 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.

2.3 Compiler Development

The HAL/S-FC compiler will be 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 will have 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-FC. 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.

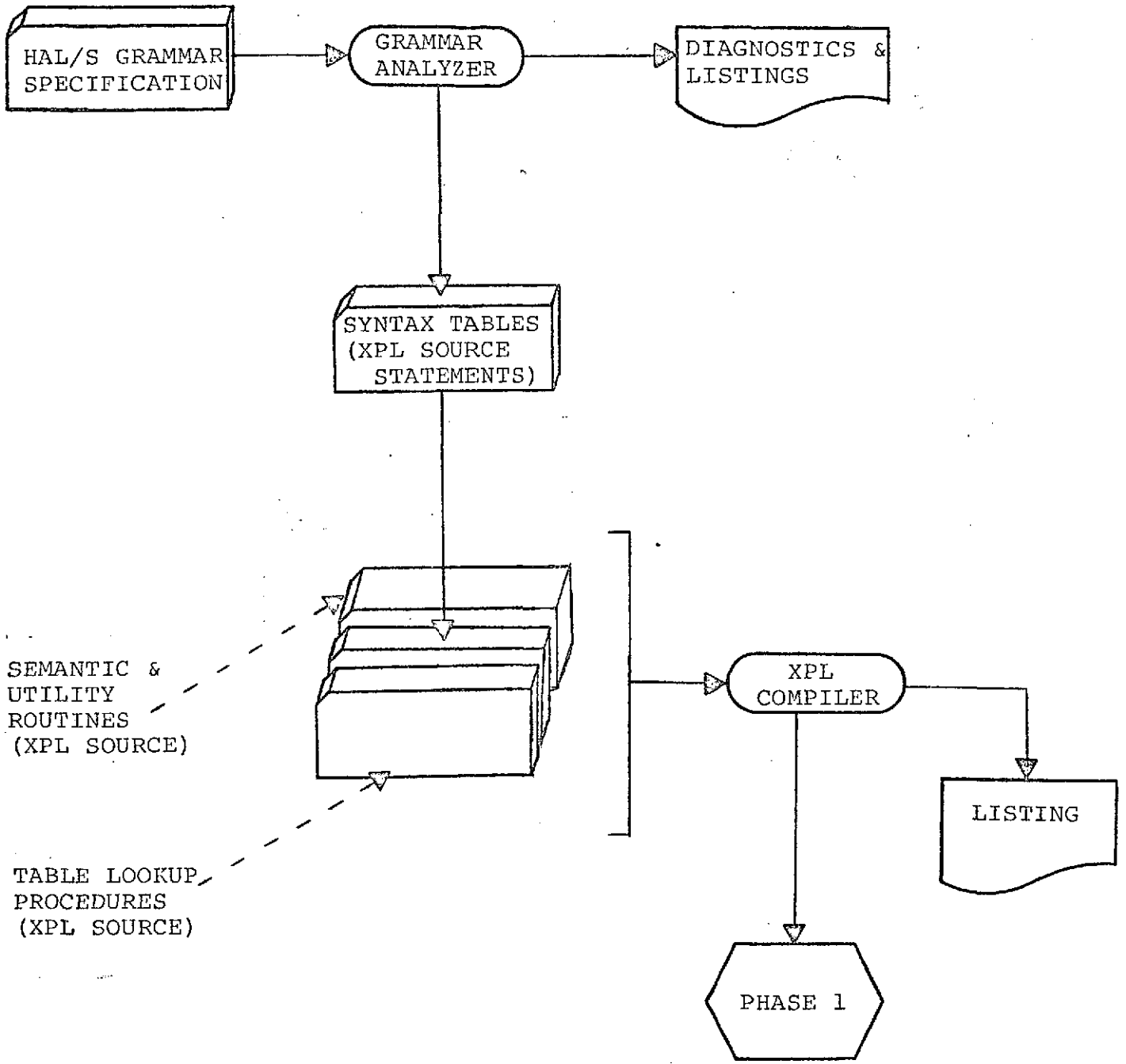


Figure 5.
Using the XPL TWS to Implement Phase 1

3. COMPILER/ENVIRONMENT DATA INTERFACE

This section describes the data interfaces which will be maintained by the HAL/S-FC compiler to provide communication with the compiler's environment. The interfaces described here are those which will receive data from or send data to areas outside the compiler. Compiler inter-phase communication has been described in Section 2.2 and is not a part of this Section.

The compiler/environment interfaces exist primarily in two areas. These areas are covered in the following sections:

- 3.1 User Interface - data "handled" by the user such as source inputs and listings.
- 3.2 System Interface - data read by or generated by the compiler and manipulated by other software.

Any special considerations in these two interface areas due to the SDL are detailed under an additional heading in Section 3.3.

3.1 User Interface

The user interface includes all data which is directly user-oriented. This data includes all user source inputs and compiler controls and all listings produced by the compiler.

3.1.1 Compiler Inputs

User-supplied inputs to the compiler will include:

- a. HAL/S Source Statements. The HAL/S source statements will originally be entered into the HAL/S-FC compiler in the form of 80 column card images. The text of the statements will occupy columns 2-80.

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

- 'C' in column 1 indicates a comment card. The remainder of the card may contain any user information. The contents of the card will be ignored by the compiler.
- 'D' in column 1 indicates a compiler directive card. Compiler directives will inform the compiler of user requests for specific compilation features. Refer to Appendix A. for descriptions of some individual directives.
- '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.
- 'Ø' blank in column 1 will be treated by the compiler as if it were an 'M'

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

- b. Controls and Options. User-defined options which affect the action taken by the compiler will be accepted either from specific Directive cards(e.g. INCLUDE) in the input stream as in (a) above, or via the JCL PARM field which is made available to individual phases by the Monitor. Appendix C. describes some specific options.

3.1.2 Compiler Outputs

User-oriented information produced by the compiler in the form of listings and printed tables will include:

- a. The Primary Source Listing. The HAL/S-360 compiler will provide standard, automatic annotation of its output listing to enhance the readability of the HAL/S source code. The HAL/S system will allow each programmer to enter his programs in free-form input consistent with his own coding preferences. The compiler will edit the input during compilation into a standard listing form so that all program listings will observe the same coding rules.

Although original HAL/S source input will be in the form of 80 column card images, the compiler will treat the input as a continuous stream of information. Elements of the source listing will be generated statement-by-statement, regardless of the original input form.

The editing performed by the compiler will include 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 will be in the form of marks supplied to indicate the type or organization of individual symbols. The marks will be generated as follows:

Overpunches - Variables of type vector, matrix, character, bit, or structure will appear in the listing with a characteristic mark above the variable name as in 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 organizations will be enclosed in brackets:

[A] for arrays,

{S} for structures.

Bracketing will occur in addition to overpunching.

Underlining - All REPLACE variables will be underlined when they appear in the listing,

e.g. REPLACE A BY "B";

C = A + D;

Statement indentation will be 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 will perform alignment of associated statements (e.g. END and CLOSE statements will be indented identically as their respective DO or PROCEDURE statements).

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

- b. Block Summaries. At the close of each PROCEDURE, TASK, PROGRAM, FUNCTION, or UPDATE block, the compiler will provide a summary of interactions between the block being closed and the outer scope in which the block is nested. The information will include both variable and block references, (e.g. a block summary for a PROCEDURE will list all variables used in that PROCEDURE and any code blocks referenced by that PROCEDURE).
- c. Program Layout. At the close of any PROGRAM, the compiler will provide a summary of all blocks contained within the PROGRAM. This summary will

list the name and type of each block and will indicate by indentation, the nesting relationships which exist between the blocks.

- d. Symbol Table Listing. The compiler will, at the end of compilation of a compilable unit, display the complete symbol table generated during the compilation. The table will be sorted alphabetically and will identify each user-defined symbol by name. The table will identify all attributes of the symbols, such as type, array/structure size, matrix/vector size, character string length, precision, etc.
- e. Cross Reference Table Listing. The compiler will, in the Symbol Table Listing, display a complete cross reference map for each symbol defined in the symbol table. This table will indicate, by number, the statements in which individual symbols appeared in the compilation. In addition, the listing will indicate the type of reference made to the symbol by distinguishing between assignment, simple reference, and use as a subscript. Also, the cross reference listing will summarize 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.
- f. Replace Text Listing. For each variable defined to be a REPLACE variable, the compiler will list the text that was substituted for the variable.
- g. Error Messages and Error Message Summary. When compilation errors are detected, the compiler will insert an error message in the primary source listing at the point of detection. The error message will be identified by an error number, an error text indicating the cause of the error, and an error severity, (see Section 3.2.2 and Appendix B.) At the end of the primary source listing, a summary of errors will be printed indicating which statements in the compilation received error messages.

- h. The Secondary Source Listing. Since the primary source listing will be completely reformatted by the compiler, an optional secondary source listing may be requested which will list the original card images as they were input to the system.

- i. The AP-101 Code Listing. On request, the compiler will produce a formatted, mnemonic listing of AP-101 code produced by the code generation Phase. This listing will identify basic machine instructions by their standard assembler language mnemonics. References to data and to program addresses will be identified by symbolic reference. Corresponding HAL/S data names will be indicated in the listing. The Assembler Code Listing will show 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 CSECT in a nested manner). The individual lines in the Assembler code listing will be compatible in format with the absolute listing function of the AP-101 link editor.

3.2 System Interface

The system interface includes all data accessed by the compiler or produced by the compiler which is not directly user oriented. These data include any auxiliary inputs used during compilation and any tables produced for use at runtime.

3.2.1 Job Control Language

The HAL/S-FC compiler system will be invoked through standard OS/360 Job Control Language (JCL). The JCL will be used to define the particular parts of the HAL/S-FC system to be invoked and to define all data upon which the compiler will operate. Additionally, the JCL will provide a means for user specification of compiler system options.

3.2.2 Compiler Inputs

The HAL/S-FC compiler will require data in addition to user-supplied source code and directives. These data will include:

- a. An INCLUDE Library. This library will contain all 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 will take the form of a partitioned data set. An individual member of the data set will be the minimum data which can be INCLUDE'd.

- b. A Program Access File. This file will contain information used by the compiler to assign ACCESS rights to individual users. The structure of the data set will be a partitioned organization with each member specifying the ACCESS rights for one Program Identification Name (PIN). See Appendix A. for definition of the PIN.
- c. An Error Text File. This file will contain the text of all error messages which may be generated by the compiler. The data set will have partitioned organization and each member will contain the text for one error message (see Appendix B. for details of the error message classification scheme). The Error Text File will also contain the severity of the individual errors. This severity information will be under administrative control and may be modified to change the effect of individual errors on a compilation.

The error messages themselves will have provision for the inclusion of imbedded text in the message. The imbedded text feature will allow the compiler to more accurately diagnose errors by specifying compilation-dependent data in the message text.

3.2.3 Compiler Outputs

As part of the normal compilation process, the compiler will issue the following sets of data:

- a. Templates for COMPOOLS and external procedure COMSUBs. Whenever a COMPOOL or COMSUB is compiled, the HAL/S-FC compiler will produce 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 will be generated to be compatible with the INCLUDE library (see Section 3.2.2).

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

- b. Simulation Tables. A Simulation Table will be produced (optionally) by the HAL/S-FC compiler for each unit of compilation, including COMPOOL. These tables will provide the information about HAL symbols and statements that are necessary to conduct the FC and ICS simulation processes and to reduce simulation output into a convenient and readable form. Each table will be stored as a member of a PDS, separate from the associated object code, and thus can be retrieved as needed by the simulation processors.

A Simulation Table contains several distinct collections of data that are organized into a complex hierarchical list structure. For this reason the term Simulation Data File has been sometimes applied.

A Simulation Table is logically divided into the following three parts:

- ⊙ Director Data - which contains global information about the file, identifies the locations of the various component parts, and provides information that can be used to reduce the time needed to access specific data items.
- ⊙ Symbol Data - which provides all required information about the symbols defined within the compilation unit. This information includes the normal attributes (i.e., name, data type, dimensionality) and in addition supplies relative core locations of symbols, structure template linkages for structure elements, and lists of statements in which the symbols are modified (assigned).
- ⊙ Statement Data - which provides attribute information about the statements in a compilation. It also supplies information on relative core locations of the first and last machine instructions in a statement, identifies labels that are attached to the statement, and lists the variables that are assigned values by the statement.

- c. Object Modules For All Compiled Code. The HAL/S-FC compiler will produce complete object modules which are compatible with the AP-101 link editor. The object modules will contain all interfaces to runtime facilities which are necessary to implement the full HAL/S language and associated error handling facilities.

3.2.4 Link Edit Support

The final step in preparing HAL compiler output for simulation or execution is performed by the AP-101 linkage editor program. Until the final version of the linkage editor is available to support all features of the HAL/S-FC compiler, an interim program, known as HALLINK-FC (see Figure 6), will be supplied to perform the link edit functions.

Inputs to HALLINK-FC are object modules on cards, tape, or disk, control cards, and runtime libraries. Their formats are identical to those of the AP-101 Link Editor.

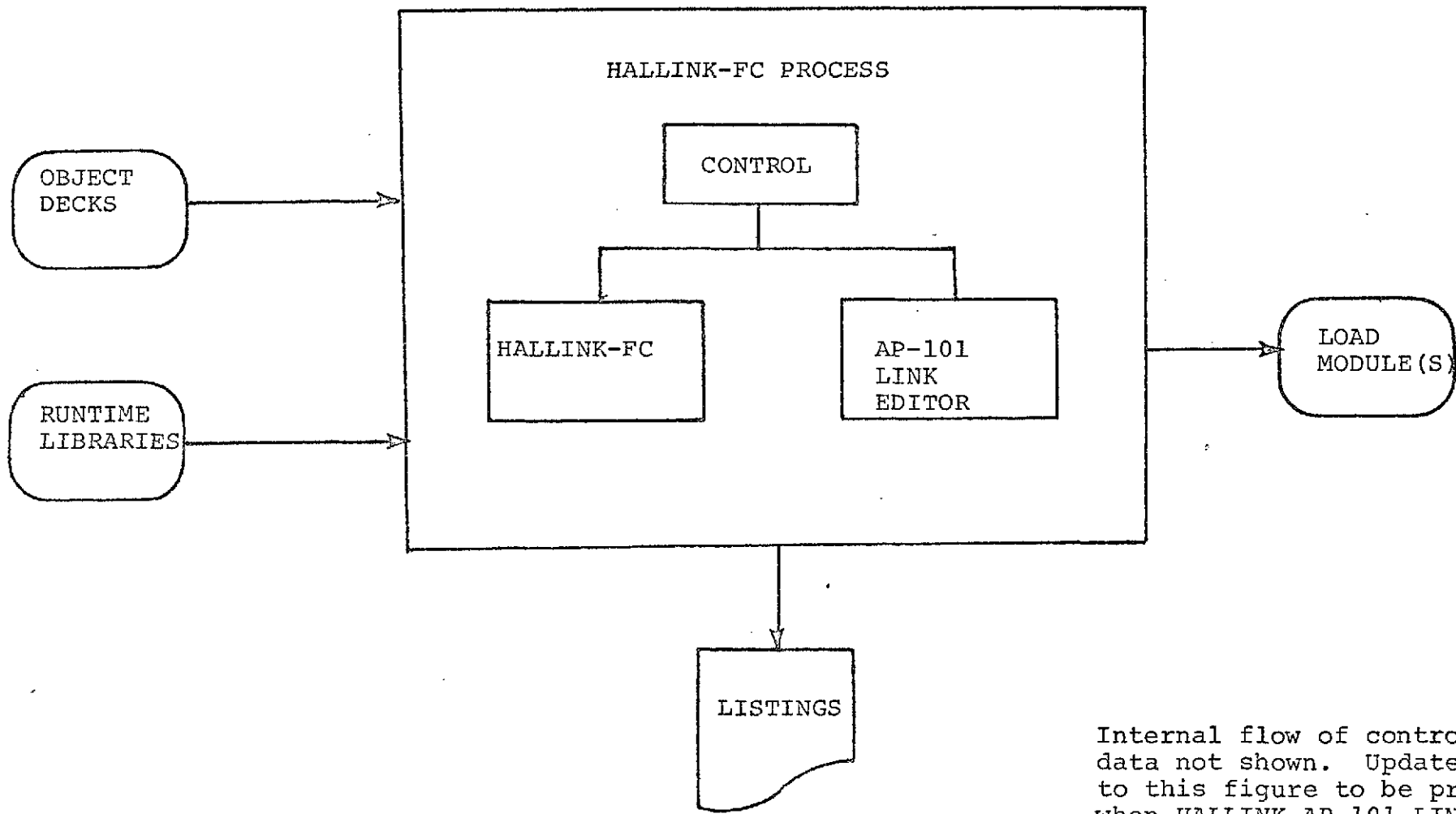
Outputs at least include, but are not restricted to, those provided by the AP-101 Link Editor; e.g., listings of control cards, map of storage and load modules.

3.3 SDL Interfaces

The HAL/S-FC compiler will supply interfaces necessary to allow implementation of the SDL. These interfaces will be in the form of compilation control and data management information supplied by the SDL. The compiler will include an option to indicate operation within the SDL environment. When running in this mode, HAL/S-FC will provide the following extensions (and/or restrictions) to the interfaces detailed above in Sections 3.1 and 3.2.

3.3.1 User Interface

3.3.1.1 Compiler Inputs. The HAL/S-FC compiler will accept as source input logical records of fixed lengths (LRECL) where $80 < \text{LRECL} < 132$. Certain fixed fields of these records will be recognized for data management purposes and provision made for printing this information in the listing, if desired.



Internal flow of control and data not shown. Update to this figure to be provided when HALLINK AP-101 LINK EDITOR interface complete.

Figure 6.

3.3.1.2 Compiler Outputs. The compiler will print data management information on the listing related to specific fields on the input records.

3.3.2 System Interface

The HAL/S-FC compiler will optionally write a duplicate object deck for each compilable unit. This deck is to be used by the SDL for data management purposes.

As a result of a compilation, the compiler will issue return codes to the SDL indicating the state of the compilation (e.g. template status, simulation table status, etc.).

4. COMPILER SYSTEM REQUIREMENTS

4.1 General

This section specifies in detail the system requirements which are necessary to compile programs written in the HAL/S language.

4.2. Hardware Requirements

4.2.1 Processor

The HAL/S-FC compiler may only be run on a System/360 Model 75 or equivalent which supports the full 360 instruction set, including all floating point instructions. The development machine is a 360/75, which meets all existing operational requirements.

4.2.2 Memory

The compiler will require a minimum region of 300K bytes to perform efficiently. This will accommodate 600 user-defined symbols, and 2K bytes of user-defined source macro code. An approximate formula for the determination of required memory size as a function of user-defined options is TBD.

4.2.3 I/O Support

4.2.3.1 Unit Record. Any standard IBM compatible card reader (2540 or equivalent) may be used to read HAL programs. For printing, any standard IBM compatible printer with a carriage width of 132 characters will be necessary. The printer must supply all printable characters legal in the HAL/S language.

4.2.3.2 Mass Storage. The compiler will require that the on-line file system support physical record lengths of 7200 bytes. Thus, for compiler work files, the IBM 2314 direct access storage system (or functional equivalent) will be a minimum requirement. A minimum of 300 tracks of 2314 storage (or equivalent) will be required.

4.3 Software Requirements

The host operating system under which the HAL/S-FC system runs must be OS/360 MVT, release 21.6, or an upwards compatible equivalent. The standard sequential and direct access methods must be supported. The standard OS storage management facility (GETMAIN, FREEMAIN) must also be supported as described in the IBM publication, "OS/360 Supervisor Services". The HAL/S-FC compiler system will use standard OS/360 data sets supported under the BSAM, QSAM, BPAM, and BDAM access methods.

5. RUNTIME SOFTWARE SUPPORT PACKAGE

An essential ingredient in the successful execution of any compiler system is a powerful, efficient, and well-integrated collection of runtime routines. These routines become especially important in light of the comprehensive set of operational primitives. In the HAL/S Language, it is these routines that supply computational power to support many of the higher level concepts of HAL.

However, strictly from a theoretical point of view, there is no absolute need for a runtime library. Everything in the library could be submerged in the compiler and issued as needed as part of the code generation process. In fact, if a particular routine were only used once, it would undoubtedly be more efficient to expand it in place with the rest of the generated code rather than collecting it into a library. But practical considerations dictate that often-used operations be centralized in a subroutine library to reduce the amount of redundant code sequences and thus save space. There are two factors to be taken into consideration in determining the suitability of routines for centralization:

1. Frequency of use. Will it be used often enough to justify subroutinization?
2. Size of code particles. Is enough code removed from in-line sequences to cover the overhead of the linking mechanisms and still produce a size savings?

Thus, the library routines can be thought of as extensions to the code generation algorithms that are lengthy enough to warrant their definition as separate routines. While scalar addition is an in-line operation, matrix addition is a called routine; however, the same rules apply to both.

The following sections will identify the runtime routines required. Although there are differences in how the routines are invoked - some are explicitly named (SIN or UNIT), others are associated with HAL operations (Vector Add or Character Concatenate), and others appear implicitly (Integer to Scalar Conversion) - they all fundamentally serve the same purpose.

5.1 Vector-Matrix Routines

A set of generalized routines to handle vectors of both single and double precision of a minimum of 64 components and matrices of both single and double precision of at least 64 by 64 elements, will be provided. In addition, particular routines or entry points for vectors of size 3 and 3 by 3 matrices will be supplied for some of the more common cases. Unless otherwise specified, the following routines will be supplied in both single and double precision.

1A Vector Assignment

SP to SP

SP to DP

DP to SP

DP to DP

1B Matrix Assignment

SP to SP

SP to DP

DP to SP

DP to DP

2A Vector Add

2B Matrix Add

3A Vector Subtract

3B Matrix Subtract

4A Vector times Scalar

4B Matrix times Scalar

6A Vector divided by Scalar

6B Matrix divided by Scalar

- 7A Vector times Matrix
- 7B Matrix times Vector
- 8A Vector Dot Product
- 8B Matrix-Matrix Product
- 8C Vector Outer Product
- 8D Vector Cross Product
- 9A Vector Negate
- 9B Matrix Negate
- 10A Vector Magnitude
- 10B Unit Vector
- 11A Matrix Transpose
- 11B Matrix Inverse
- 11C Matrix Trace
- 11D Matrix Determinant
- 11E Identify Matrix
- 12A Vector Comparison
- 12B Matrix Comparison
- 13A Vector Precision Conversion
 - SP to DP
 - DP to SP
- 13B Matrix Precision Conversion
 - SP to DP
 - DP to SP
- 14A Vector Assign All
- 14B Matrix Assign All

5.2 Character Routines

A set of generalized routines to handle character strings that vary in length from 0 (the null string) to 255 characters will be provided.

- 1 Character Assignment
- 2 Character Concatenate
- 3 Character Comparison
- 4 LJUST
- 5 RJUST
- 6 INDEX
- 7 TRIM

5.3 Conversion Routines

A set of routines to convert from one to another of the HAL data types will be provided.

- 1 Integer to Character
- 2 Scalar to Character
 - SP
 - DP
- 3 Vector to Character
- 4 Matrix to Character
- 5 Bit to Character
 - Binary
 - Octal
 - Hex
 - Decimal

- 6 Character to Integer
- 7 Character to Scalar
- 8 Character to Vector
- 9 Character to Matrix
- 10 Character to Bit

Binary

Octal

Hex

Decimal

5.4 Input/Output

The HAL/S-FC compiler system will supply the routines necessary to implement the HAL/S I/O statements: READ, READALL, and WRITE. Device numbers 0-9 will be reserved for 360 compatibility.

The following I/O routines are planned:

1A Vector Input

SP

DP

1B Vector Output

SP

DP

2A Matrix Input

SP

DP

2B Matrix Output
SP
DP

3A Integer Input

3B Integer Output

4A Scalar Input
SP
DP

4B Scalar Output
SP
DP

5A Bit Input

5B Bit Output

6A Character Input

6B Character Output

6C Character Input (READALL)

7 Carriage Control Functions (Input & Output)
SKIP
LINE
PAGE
TAB
COLUMN

5.5 Mathematical Functions

These functions return a scalar data type. Arguments may be Integer or Scalar. (Integers are converted to Scalar unless otherwise specified.) Array arguments yield array results.

1. ARCCOS
Inverse trigonometric cosine; argument in closed interval $[-1, 1]$; results in closed interval $[0, \pi]$.
2. ARCCOSH
Inverse hyperbolic cosine; arg not less than 1.
3. ARCSIN
Inverse trigonometric sine; arg in closed interval $[-1, 1]$; result in closed interval $[-\pi/2, \pi/2]$.
4. ARCSINH
Inverse hyperbolic arc sine; arg any value.
5. ARCTAN
Inverse trigonometric tangent; arg any value; results in open interval $[-\pi/2, \pi/2]$.
6. ARCTANH
Inverse hyperbolic tangent; $|\text{arg}| < 1$.
7. COS
Trigonometric cosine; arg in radians; $|\text{arg}| < \begin{cases} 2^{18}\pi & \text{(SINGLE)} \\ 2^{50}\pi & \text{(DOUBLE)} \end{cases}$
8. COSH
Hyperbolic cosine; $|\text{arg}| < 175.366$
9. EXP
Exponential, (e^{arg}) ; $|\text{arg}| < 174.673$.

10. LOG
Natural logarithm; arg positive and non-zero.
11. SIN
Trigonometric sine; arg in radians; $|\text{arg}| < \begin{cases} 2^{18}\pi & \text{(SINGLE)} \\ 2^{50}\pi & \text{(DOUBLE)} \end{cases}$
12. SINH
Hyperbolic sine; $|\text{arg}| < 175.366$
13. TAN
Trigonometric tangent; arg in radians; arg not multiple of $\pi/2$; $|\text{arg}| < \begin{cases} 2^{18}\pi & \text{(SINGLE)} \\ 2^{50}\pi & \text{(DOUBLE)} \end{cases}$
14. TANH
Hyperbolic tangent; arg any value.
15. SQRT
Square root; arg positive.
16. RANDOM
Returns random number selected from a rectangular distribution over the range 0-1.
17. RANDOMG
Selects a random number from a Gaussian distribution with a mean of zero and a variance of one.
18. EXPONENTIATION ROUTINES (a^B)
- a. Integer Base - Integer Exponent - Integer Result
 - b. Integer Base - Scalar Exponent - Scalar Result
 - c. Scalar Base - Integer Exponent - Scalar Result
 - d. Scalar Base - Scalar Exponent - Scalar Result

5.6 Miscellaneous Functions

1. DATE
2. RUNTIME
3. CLOCKTIME
4. PRIO

5.7 Arithmetic Functions

Unless otherwise specified, these functions yield the same data type of result as their arguments. (These functions are mentioned because they appear as a built-in function to the user. Some may be generated in-line, according to the criterion presented at the beginning of this section, and so will not appear in the library.)

1. ABS (a)
Finds the absolute value of the argument.
2. CEILING (a)
Determines the least integer value that is greater than or equal to the argument
3. FLOOR (a)
Determines the greatest integer value that does not exceed the argument.
4. ROUND (a)
Rounds the argument to nearest integer value.
5. SIGNUM (a)
Returns +1, 0, -1 as argument is positive, zero, and negative, respectively.
6. SIGN (a)
Returns +1, -1 as argument is positive or zero, and negative, respectively.

7. TRUNCATE (a)

Rounds the argument towards zero to the nearest integer value; equivalent of FLOOR for positive arguments and CEILING for negative ones.

8. MOD (a,b)

MOD extracts the remainder c such that $(a-c)/b = N$ where N is an integer number. c is the smallest positive number that must be subtracted from a in order to make N an integer number.

9. DIV (a,b)

DIV performs an integer divide of a by b ; (arguments are forced to integer before divide).

10. REMAINDER (a,b)

Return the integer remainder after performing the integer div function, DIV.

11. SIZE (any)

Result: Finds the unknown amount of arrayness as indicated by starred (*) dimensionality and returns the value as an integer data type.

Argument: Any HAL data type with starred arrayness including a structure.

12. LENGTH (string)

Finds the string length and returns it as an integer data type.

13. MAX (a)

Finds maximum value of all elements of argument.

14. MIN (a)

Same as MAX, except finds minimum.

15. ODD (a)

Argument is scalar or integer. Result is Boolean which is true if a is ODD, otherwise false. Scalar arguments are rounded to nearest integer value before evaluation of ODD.

16. SUM (a)

Finds sum of all elements of argument.

17. PROD (a)

Finds product of all elements of argument.

5.8 Real Time Interfaces

1. SCHEDULE

Input: Parameters describing the required and optional operands of the SCHEDULE statement.

Result: Creates a process which can be given the CPU under appropriate conditions.

2. WAIT

WAIT UNTIL

Input: An absolute or relative scalar time in units of seconds.

Result: The running process is made to wait the appropriate time.

3. WAIT FOR

Input: An event expression.

Result: The running process is made to wait until the event expression is true.

4. WAIT FOR DEPENDENT

Input: None.

Result: The running process is made to wait until all dependent processes have completed or terminated.

5. SIGNAL, SET, RESET

Input: An event variable.

Result: The state of the event variable is altered, and if any event expressions become true, the appropriate action is taken (a process is made ready, or a cyclic process is cancelled).

6. TERMINATE

Input: 1) none or 2) a program or task name.

Result: 1) the running process is terminated or
2) the specified program or task process is terminated.

Note: Termination is unconditional, immediate, terminating the cyclic nature, if present, and all dependent processes.

7. CANCEL

Input: 1) none or 2) a program or task name.

Result: The actions specified for the CANCEL statement occur.

8. UPDATE PRIORITY

UPDATE PRIORITY...TO

Input: 1) none or 2) program or task name, and priority.

Result: The priority of 1) the running process or 2) the specified process is changed to the specified priority.

6. RESTRICTIONS AND DEPENDENCIES

6.1 Introduction

The implementation of the HAL/S-FC compiler on the IBM/360 will contain certain restrictions which are due to either the 360 hardware and available facilities, or to the hardware and organization of the AP-101. These restrictions, which include specific language implementation dependencies as described in the HAL/S Language Specification, are explained in the following sections.

6.2 Compile-Time Characteristics

The use of the IBM/360 computer as the host machine for generation of AP-101 code will determine certain rules governing use of the HAL/S language and compiler. These rules are explained in this section.

6.2.1 Character Set

The character set, as described in the HAL/S Language Specification document, which may be used as compiler input, will be available in its entirety on the 360. The HAL/S-FC compiler will recognize this full character set. The coding scheme for compiler input will be EBCDIC. No other coding scheme will be recognized.

6.2.2 Compilation Dependent Language Features

The HAL/S-FC compiler implementation will have the following language dependencies.

6.2.2.1 Data Type and Size Restrictions.

- o Up to 74 decimal digits may appear in an arithmetic literal preceding the exponent field.

- ⊙ There will be no limit to the number of exponent modifiers which may be appended to an arithmetic literal, so long as the total length of the literal does not exceed 256 characters. However, application of each exponent to the mantissa to which it belongs may not cause generation of a number outside the range 10^{-78} to 10^{75} .
- ⊙ Bit literals may not indicate values requiring more than 32 bits.
- ⊙ Character literals may not contain more than 255 characters.
- ⊙ Comment brackets (`/*` and `*/`) may not enclose more than 255 characters in any one comment.
- ⊙ There will be no compiler-imposed limit on the number of COMPOOL blocks existing in a program complex except the limit implied by the number of user-defined symbols allowed.
- ⊙ The implementation of the ACCESS attribute will be as described in the HAL/SDL Interface Control Document.
- ⊙ Each declared dimension in an array will be limited to the range 2 to 32767.
- ⊙ Arrays will be limited to 3 dimensions.
- ⊙ Implementation of the NONHAL attribute is TBD.
- ⊙ The maximum declarable row or column dimension of a matrix will be 64.
- ⊙ The maximum declarable dimension of a vector will be 64.
- ⊙ Character variables will be limited to a declared length of 255 characters.
- ⊙ Bit variables may have declared lengths of from 1 to 32 bits.
- ⊙ The number of declared multiple copies allowed for a structure variable will be limited to the range 2 to 32767.

6.2.2.2 Program Organization Limits.

- ⊙ The number of cases in a DO CASE statement will be limited to 256.
- ⊙ The number of internal code blocks within any one compilation unit will be limited to 256.
- ⊙ The depth to which code blocks may be nested within one another in any compilation unit will be limited to 16 levels.
- ⊙ The maximum nesting level of DO ... END groups will be 16 levels.
- ⊙ The maximum depth to which function invocations may be nested will be 20.
- ⊙ Up to 32767 literals may be defined in any one compilation unit.
- ⊙ The maximum number of characters which may be defined in all character string literals in one compilation unit will be under user-control.
- ⊙ The maximum number of user-defined symbols allowed in one compilation will be under user control.
- ⊙ The maximum size of REPLACE macro text will be under user control.
- ⊙ The total number of elements in initial/constant lists in any one compilation will be limited to 32767.
- ⊙ The maximum number of parameters allowed in the definition of any REPLACE macro will be 6.
- ⊙ The maximum size of any argument used in the expansion of a REPLACE macro will be limited to 250 characters.
- ⊙ The depth to which macro expansions may invoke other macros may not exceed 4 levels of nesting.
- ⊙ A compilation unit will be limited in the size of its code segment to 32K half words.

- ⊙ The data defined in a compilation unit will be limited to 32K half words.
- ⊙ The sum total size of all COMPOOLS allowed in a single mission load shall be at least 32K half words.
- ⊙ The maximum number of external names (ESD's) allowed in one compilation will be limited to 500, of which no more than 256 may be CSECTs defined by the compilation unit.
- ⊙ The maximum number of ON ERROR statements which may be potentially active at any time in any compilation unit will be limited to 100.
- ⊙ The number of arguments indicated in a subroutine invocation may not exceed 100. This limit includes arguments of nested FUNCTION invocations; i.e. FUNCTION invocations which are themselves part of an argument expression.

6.2.2.3 Input/Output Statements.

- ⊙ Sequential I/O device numbers 0-9 will be reserved for test I/O functions.
- ⊙ The use and interpretation of other devices and numbers is TBD.

6.3 Runtime Characteristics

Designation of the IBM AP-101 as the target machine will determine certain rules governing operation of HAL/S programs run on that machine. These rules are explained in this section.

6.3.1 Character Set

The HAL/S-FC compiler will generate object code in which the representation of character data is TBD. (Until the character representation is known, generated code will use EBCDIC).

6.3.2 Computer Dependent Language Features

The hardware available on the AP-101 and design of runtime support software determine the behavior of certain language features described here. The behavior being described is of language features whose implementations have been subjected to the compile-time rules of Section 6.2.2.

6.3.2.1 Data Types and Size Dependencies.

- ⊙ The dynamic size of character variables may not exceed 255.
- ⊙ Bit variables may have lengths of from 1 to 32 bits.
- ⊙ Integer variables will be represented as either 16 or 32 bit signed quantities for SINGLE and DOUBLE declaration specifications respectively.
- ⊙ Single precision scalar values will be represented in the AP-101 in the 32 bit floating point format (1 sign, 7 exponent, 24 mantissa).
- ⊙ Double precision scalar values will be represented in the AP-101 in the 64 bit floating point format (1 sign, 7 exponent, 56 mantissa).

- ④ Runtime conversions of double precision scalar to single precision scalar will be performed by truncating the right-most 32 bits from the double precision mantissa.
- ④ Runtime conversion of double precision integer values to single precision integer values will be performed by eliminating the left-most 16 bits of the double precision value.
- ④ Runtime conversion of integer values to scalar values will be performed by converting the integer value to a double precision scalar value retaining all significant digits. If the final result of the conversion is to be single precision, the standard double-to-single precision scalar conversion will be applied to the intermediate scalar value.
- ④ Runtime conversion from scalar to integer value will result in an error condition if the scalar value cannot be represented in the integer form.
- ④ Runtime conversion of single precision integer to double precision integer will be performed by propagating the sign bit of the single precision value through the 16 high order bit positions of the double precision value.
- ④ Runtime conversion of single precision scalar to double precision scalar will be performed by padding 32 zero bits to the right of the single precision mantissa.

6.3.2.2 Runtime Conversions To and From Character Type.

- ⊙ When any HAL/S initial data is to be converted from its internal representation to a character representation, the general rules specified in the HAL/S Language Specification governing formats will apply. The specific AP-101 representations of data cause the following specific conversion formats to apply:

Integers - an 11 character field will be generated with the number right-justified. A floating minus sign is added if the number is negative.

Single Precision Scalars - a 14 character field will be generated as follows:

`sx.xxxxxxxEtxx`

where s is a blank or minus sign,

x is a digit 0 to 9,

t is a plus or minus sign.

Double Precision Scalars - a 23 character field will be generated as follows:

`sx.xxxxxxxxxxxxxxxxxxEtxx`

- ⊙ Conversions of character data to other data types will have the same restrictions governing the format of the character data as those restrictions placed upon compiler literal data (Section 6.2.2.1).

6.3.2.3 Input/Output. The runtime functions of HAL/S input/output statements are TBD.

APPENDIX A.

Compiler Directives

Any card images input by the user may contain commands to the compiler in the form of Compiler Directives. A card image with a "D" in column 1 is considered to be such a Directive. Some Directives whose use and form are presently known are listed below. Other uses for Directives are also identified.

1. PROGRAM - The general use of COMPOOL data or library programs may be restricted by using the access right attribute when defining the resource to be controlled.

Each Compilation Unit (PROGRAM, external PROCEDURE or FUNCTION) that is to participate in the controlled access of COMPOOL variables must identify itself to the compiler by establishing a Program Identification Name (PIN). The PIN is specified in a PROGRAM Compiler Directive of the form:

```
D PROGRAM ID = <id>
```

where <id> is a 1 to 8 character name defining the PIN.

The PIN will be used by the compiler to access a data file called the Program Access File (PAF). The PAF will contain an entry for each possible PIN. The PAF is intended to be a centrally managed access control list which, for each PIN, identifies the HAL/S resources to which the program in question may have write access. Any attempt to use an ACCESS controlled resource which has not been found in a program's PAF entry will be flagged as an error.

2. DEVICE - The Device Directive has the following form:

```
col 1
```

```
D DEVICE CHANNEL=n XXXX
```

PRECEDING PAGE BLANK NOT FILMED

"CHANNEL=n" must be present. This keyword defines the channel number n (with the range 0-9 reserved for 360 compatibility) to which the directive is applied. These channel numbers are the same as those used in the HAL I/O statements READ(n), READALL(n) and WRITE(n).

The field XXXX can indicate PAGED or UNPAGED. If no field is specified, UNPAGED is assumed.

The occurrence of PAGED identifies the specified channel as printer compatible; i.e., carriage control characters are supplied and the output device is assumed to be page oriented.

The occurrence of UNPAGED or no field identifies the specified channel as input and output compatible. No carriage control is assumed and the device is thought of as pageless.

The DEVICE directive may appear at the same location in a source program as a comment card. It need not appear before I/O statements for the specified channel. If no DEVICE directives are encountered, default attributes are assigned to channels used in I/O statements as follows:

- a. A channel number for which no directive is found appearing only in a write statement will be assigned the PAGED attribute by default.
- b. A channel number for which no directive exists that appears in both input and output statements will be assigned the UNPAGED attribute.

Conflicts arising between DEVICE directives and I/O usage will be flagged as errors. For example:

```
D   DEVICE      CHANNEL=2    PAGED
   :
M   READ(2) A;
```

This is an error since the DEVICE directive indicates an output-only, printer compatible channel, and the READ statement attempts to input from that channel.

Since the defaults are applied if no directive is found, the creation of an UNPAGED, output-only data set requires a DEVICE directive with the UNPAGED or no option. Omission of the DEVICE directive would cause the channel number (which in this case occurs only in a write statement) to be assigned the PAGED attribute.

3. INCLUDE - The form of the INCLUDE Directive is:

```
D      INCLUDE    <name>    [NOLIST]
```

where name defines the entry in the auxiliary symbolic library to be included in the compilation. For HAL/S-FC, the name must be a 1 to 8 character string beginning with a letter.

The source included as a result of the Directive will be listed in the primary source listing unless the NOLIST option is specified.

4. HEADING - A Directive to allow the user to define the page heading for the primary source listing will be provided. The exact form of this directive is not known at this time.
5. Future Directives may be defined to accommodate management rules governing use of particular HAL/S language features. For example, a Directive may be defined which prohibits use of any realtime control statements in the compilation containing the Directive.

APPENDIX B.
Compiler Error Messages

Any compile time diagnostic messages printed by the HAL/S-FC compiler will originate on a direct access file. The file will be organized to allow the compiler's error printing routine to gain rapid access to an individual error message.

The compiler will internally identify an error condition by an error class and an error number within the class. These two pieces of information will form the index information needed to access the error message file.

The error message file entry for an individual error will contain the severity of the error, the text of the error message, and an indication of a point in the message where specific imbedded text may be inserted by the compiler.

The following table lists the tentative error classification system. Within each class, subclasses have been identified to further qualify the error type. Following the classification table is a partial listing of some tentative error message text. The full listing can be produced automatically with an existing Intermetrics utility program. This system will allow easy monitoring and updating of error messages.

PRECEDING PAGE BLANK NOT FILMED

ERROR CLASSIFICATIONS

Note: "∅" denotes a blank.

CLASS A: ASSIGNMENT STATEMENTS

A ARRAY ASSIGNMENT
V COMPLEX VARIABLE ASSIGNMENT
∅ MISCELLANEOUS ASSIGNMENT

CLASS B: COMPILER TERMINATION

B HALMAT BLOCK SIZE
N NAME SCOPE NESTING
S STACK SIZE LIMITATIONS
T TABLE SIZE LIMITATIONS
X COMPILER ERRORS
∅ MISCELLANEOUS

CLASS C: COMPARISONS

∅ GENERAL COMPARISONS

CLASS D: DECLARATION ERRORS

A ATTRIBUTE LIST
C STORAGE CLASS ATTRIBUTE
D DIMENSION
F FUNCTION DECLARATION
I INITIALIZATION
L LOCKING ATTRIBUTE
Q STRUCTURE TEMPLATE TREE ORGANIZATION

S FACTORED/UNFACTORED SPECIFICATION
T TYPE SPECIFICATION
U UNDECLARED DATA
∅ MISCELLANEOUS

CLASS E: EXPRESSIONS

A ARRAYNESS
B BIT STRING EXPRESSIONS
C CROSS PRODUCT
D DOT PRODUCT
L LIST EXPRESSIONS
M MATRIX EXPRESSIONS
O OUTER PRODUCT
V VECTOR EXPRESSIONS
∅ MISCELLANEOUS

CLASS F: FORMAL PARAMETERS & ARGUMENTS

D DIMENSION AGREEMENTS
N NUMBER OF ARGUMENTS
S SUBBIT ARGUMENTS
T TYPE AGREEMENT

CLASS G: STATEMENT GROUPINGS (DO GROUPS)

B BIT TYPE CONTROL EXPRESSIONS
C CONTROL EXPRESSIONS
E EXIT/REPEAT STATEMENTS
L END LABEL
V CONTROL VARIABLE

CLASS I: IDENTIFIERS

L LENGTH
R REPLACED IDENTIFIERS
S QUALIFIED STRUCTURE NAMES

CLASS L: LITERALS

B BIT STRING
C CONVERSION TO INTERNAL FORMS
F FORMAT OF ARITHMETIC LITERALS
S CHARACTER STRING

CLASS M: MULTILINE FORMAT

C OVERPUNCH CONTEXT
E E-LINE
O OVERPUNCH USE
S S-LINE
∅ COMMENTS

CLASS P: PROGRAM CONTROL & INTERNAL CONSISTENCE

A ACCESS CONTROL
C COMPOOL BLOCKS
D DATA DEFINITION
E EXTERNAL TEMPLATES
F FUNCTION RETURN EXPRESSIONS
L LABELS
M MULTIPLE DEFINITIONS
P BLOCK DEFINITION
S PROCEDURE/FUNCTION TEMPLATES
T TASK DEFINITIONS
U CALLS FROM UPDATE BLOCKS
∅ MISCELLANEOUS

CLASS Q: SHAPING FUNCTIONS

A ARRAYNESS
D DIMENSION INFORMATION

S SUBSCRIPTS
X ARGUMENT TYPE
∅ MISCELLANEOUS

CLASS R: REAL TIME STATEMENTS

E ON/SEND ERROR STATEMENTS
T TIMING EXPRESSIONS
U UPDATE BLOCKS

CLASS S: SUBSCRIPT USAGE

C SUBSCRIPT COUNT
P PUNCTUATION
Q PRECISION QUALIFIER
R RANGE OF SUBSCRIPT VALUES
S USAGE OF ASTERISKS
T SUBSCRIPT TYPE
V VALIDITY OF USAGE

CLASS T: I/O STATEMENTS

C CONTROL
D DEVICE NUMBER
∅ MISCELLANEOUS

CLASS U: UPDATE BLOCKS

I IDENTIFIER USAGE
P PROGRAM BLOCKS
T I/O

CLASS V: COMPILE-TIME EVALUATIONS

- A ARITHMETIC OPERATIONS
- C CATENATION OPERATIONS
- E UNCOMPUTABLE EXPRESSIONS
- F FUNCTION EVALUATION

CLASS X: IMPLEMENTATION DEPENDENT FEATURES

- A PROGRAM ID DIRECTIVE
- D DEVICE DIRECTIVE
- I INCLUDE DIRECTIVE
- U UNKNOWN OR INVALID DIRECTIVE

ERROR MESSAGES FOR MAJOR CLASSIFICATION A
CLASSIFICATION "A" ERRORS ARE RELATED TO ASSIGNMENT STATEMENTS

- AA1 -SEVERITY 1
ARRAYNESS OF LEFT HAND SIDE OF ASSIGNMENT DOES NOT MATCH THAT OF RIGHT HAND SIDE
- AA2 -SEVERITY 1
ARRAYNESS OF ?? IS INCONSISTENT WITH THAT OF OTHER LEFT HAND SIDE VARIABLES
- AA3 -SEVERITY 1
ARRAYNESS OF ?? DISAGREES WITH ARRAYNESS OF ITS SUBSCRIPTING
- AV0 -SEVERITY 1
ARGUMENTS ON EITHER SIDE OF NAME ASSIGNMENT ARE INCOMPATIBLE.
- AV1 -SEVERITY 1
TYPE OF ?? IS ILLEGAL FOR ASSIGNMENT FROM GIVEN LEFT-HAND SIDE.
- AV2 -SEVERITY 1
MATRIX DIMENSIONS DISAGREE ACROSS ASSIGNMENT
- AV3 -SEVERITY 1
VECTOR LENGTHS DISAGREE ACROSS ASSIGNMENT
- AV4 -SEVERITY 1
TREE ORGANIZATIONS DO NOT MATCH ACROSS ASSIGNMENT
- AV5 -SEVERITY 1
ONLY ONE OPERAND IN ASSIGNMENT IS A NAME PSEUDO-FUNCTION
OR NULL.
- A1 -SEVERITY 1
ILLEGAL ASSIGNMENT TO CONSTANT OR PARAMETER ??
- A2 -SEVERITY 1
?? POSSESSES SUBSCRIPTS ILLEGAL FOR THE ARGUMENT OF A NAME
PSEUDO-FUNCTION IN ASSIGNMENT CONTEXT.
- A3 -SEVERITY 1
?? DOES NOT POSSESS THE NAME ATTRIBUTE - IT IS THEREFORE
ILLEGAL AS ARGUMENT OF A NAME PSEUDO-FUNCTION IN
ASSIGNMENT CONTEXT.

APPENDIX C.

Compiler Options

The following is a list of anticipated options to direct the execution of the HAL/S-FC compiler. The keywords describing the option will be placed in the PARM field of the JCL invoking the HAL/S-FC compiler.

1. Generate an unformatted listing of original source input card images.
2. Generate a memory dump if certain interval compiler errors occur.
3. Generate a formatted code generation listing showing all generated object code.
4. Indicate compiler operation within the SDL.
5. Produce duplicate object deck.
6. Supply header information.
7. Other options available as the need is recognized.

PRECEDING PAGE BLANK NOT FILMED

APPENDIX D.

Runtime Errors

The following tables list some anticipated runtime error conditions which may occur during execution of a HAL/S-FC program. The tables list any standard fix-ups performed by the runtime system.

PRECEDING PAGE BLANK NOT FILMED

TABLE 1. Error Behavior Table for 'SYSTEM' Action

Message	Explanation	Standard Fixup
EXPONENT OVERFLOW	the exponent of a scalar or of a vector/matrix element has overflowed	the result is set to the maximum value representable on the machine
EXPONENT UNDERFLOW	the exponent of a scalar or of a vector/matrix element has underflowed	the result is set to zero
SCALAR DIVISION BY ZERO	a scalar division by zero has occurred	the result is set to the maximum value representable on the machine
EXPONENTIATION OF ZERO TO POWER ≤ 0	a negative or zero power was specified	the result is set to zero
SQUARE ROOT HAS ARG < 0		the result is the square root of the absolute value of the argument

Message	Explanation	Standard Fixup
EXP FUNCTION HAS ARG > 174.673		the result is set to the maximum value representable on the machine
LOG FUNCTION HAS ARG < = 0		if the argument was zero then the result is set to the maximum representable negative value, else it is set to the log of the absolute value of the arg.
SIN OR COS FUNCTION HAS $ \text{ARG} \begin{cases} 2.621\text{E}5 \\ 1.126\text{E}15 \end{cases} \text{ PI}$	the two figures are for single and double precision arguments respectively	the result is set to $\frac{\sqrt{2}}{2}$
SINH OR COSH FUNCTION HAS ARG > 175.366		the result is set to the maximum value representable
ARCSIN OR ARCCOS FUNCTION HAS $ \text{ARG} > 1$		the result is set to zero

Message	Explanation	Standard Fixup
TAN FUNCTION HAS $ \text{ARG} > \left\{ \begin{array}{l} 2.621\text{E}5 \\ 1.126\text{E}15 \end{array} \right\} \text{PI}$	the two figures are for single and double precision respectively	the result is set to one
TAN FUNCTION TOO CLOSE TO SINGULARITY	the argument is too close to an odd multiple of $\pi/2$	the result is set to the maximum representable value
CASE VARIABLE OUT OF RANGE	the value of the case variable is either <1 or greater than the number of cases and there was no ELSE clause	the do case statement is ignored
CLOSE REACHED ON FUNCTION	no return statement was encountered prior to reaching the close of the function	none: IGNORE not allowed
SCALAR TOO LARGE FOR INTEGER CONVERSION		the result is set to the maximum representable value

Message	Explanation	Standard Fixup
INTEGER DIVISION BY ZERO	DIV OPERATOR HAS ZERO DIVISOR	The result is set to the maximum representable value
ILLEGAL CHARACTER SUBSCRIPT	Character component subscripting out-of-bounds	The out-of-bounds subscript(s) set to first or last character
BAD LENGTH IN LJUST OR RJUST	The length is less than the string length	Truncation to the specified length occurs on the left (RJUST) or right (LJUST)
MOD DOMAIN ERROR	In $A \text{ mod } B$ $B=0$ and $A < 0$	Returns A (negative)
CHARACTER TO SCALAR CONVERSION	The string was not in standard internal format for integers or scalars.	The result is zero.

Message	Explanation	Standard Fixup
CHARACTER TO SCALAR CONVERSION DURING INPUT	Same as above.	The variable is left unchanged.
CHARACTER TO INTEGER CONVERSION	The string was not in standard internal format for integers	The result is zero.
CHARACTER TO INTEGER CONVERSION DURING INPUT	Same as above.	The variable is left unchanged
NEGATIVE BASE IN EXPONENTIATION	$A^{**}B$ where $A < 0$	The result is $ A ^{**}B$
VECTOR/MATRIX DIVISION BY ZERO		The result is the original vector/matrix

MESSAGE	EXPLANATION	STANDARD FIXUP
ILLEGAL BIT STRING DURING INPUT	Character other than blank, zeros or ones	Variables left unchanged
ARG OF INVERSE IS SINGULAR		The result is the identity matrix
ARG OF UNIT FUNCTION IS. NULL VECTOR	Every component of the vector was zero in value.	The result is a vector all of whose components is zero.
ILLEGAL BIT STRING		Returns zero
FIXED POINT OVERFLOW		The uncorrected residual is returned

Message	Explanation	Standard Fixup
PRINT ON INPUT CHANNEL N or INPUT ON PRINT CHANNEL N	I/O was attempted on specified channel in print mode as well as read/readall mode.	The channel remains in the original mode, I/O in the new mode is ignored.
ILLEGAL SKIP COUNT ON CHANNEL n	The number of skips is negative.	The SKIP function is ignored.
MARGIN VIOLATION ON CHANNEL n	A TAB or COLUMN I/O control function was specified which forced the device mechanism off the left or right margin.	The horizontal position is reset to either column one (left margin error) or just off the right-hand margin. On an I/O transfer this latter causes an immediate skip to the next line.

Message	Explanation	Standard Fixup
ILLEGAL PAGE COUNT ON CHANNEL n	A PAGE I/O control function with negative argument was specified	The PAGE command is ignored.
ILLEGAL LINE COUNT ON CHANNEL n	In a LINE I/O control function on an "unpaged" channel an argument less than the current line number was specified, or in the case of print mode a value greater than the number of lines per page was specified.	In the first case the LINE function is ignored. In the second, the effect is PAGE (1).
ILLEGAL NUMERIC FIELD ON CHANNEL n	An invalid character was found while reading a numeric field (Valid characters: 0-9, -, +, ., E, B, H)	The field with the invalid character(s) is skipped. The variable remains unchanged as if a null field were encountered.
ILLEGAL BIT OR CHARACTER STRING	In READ mode, character/bit strings must be delimited by apostrophes, with included apostrophes doubled.	The field is treated as a numeric field with regard to separators, and is skipped. The character/bit variable remains unchanged as if a null field were encountered.

Message	Explanation	Standard Fixup
END OF FILE ON CHANNEL n	Error 40+n is signalled if the end of file is reached while reading on channel n.	The remainder of the I/O statement is ignored. A further read on that channel will close and reopen the file at line 1 again.
ERROR IN HAL SOURCE		Continue