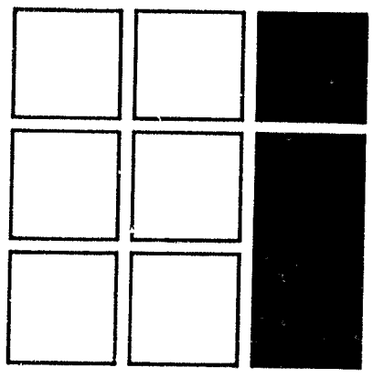


INTERMETRICS

CR-115126

MSC-01848

Final Report

Development of an MSC Language and Compiler

Volume I.

June 1971

Prepared under Contract NAS 9-10542 by:

INTERMETRICS, INC.
380 Green Street
Cambridge, Mass. 02139

Foreword

This document is the final report of a programing language development contract for advanced manned spacecraft. This effort was sponsored by the National Aeronautics and Space Administration's Manned Spacecraft Center, Houston, Texas under contract NAS 9-10542. It was performed by Intermetrics, Inc., Cambridge, Massachusetts under the technical direction of Mr. Daniel J. Lickly. The Technical Monitor for the Manned Spacecraft Center was initially Mr. John E. Williams and later was Mr. Jack R. Garman, FS/5.

Acknowledgements

Technical personnel of Intermetrics contributing to the development of the HAL language and compiler were:

Mr. Richard Arratia
Mr. Brent C. Hall
Mr. Thomas A. James
Mr. Alex L. Kosmala
Mr. Ronald E. Kole
Mr. Daniel J. Lickly
Dr. Fred H. Martin
Dr. James S. Miller
Dr. Philip M. Newbold
Mr. Joseph A. Saponaro
Mr. Woodrow H. Vandever

Table of Contents

CHAPTER 1

REPORT ORGANIZATION

This chapter is intended to delineate the intent and scope of the final report and to direct the reader's attention to the technical authoritative documents which have been produced by Intermetrics during the period of performance of the contract.

The purpose of the final report is to summarize briefly the technical material, review the objectives, progress, and results of each contract task, and evaluate the accomplishments of this effort within the overall status of projected MSC activities.

This final report is not the main source of technical material describing the results of the contractual effort. Several other documents have been written as end-item deliverables and are the references in their respective areas. These reports are:

1. Requirements Analysis for a Manned Spaceflight Programming Language and Computer, MSC-01845, August, 1970, Intermetrics, Inc., Cambridge, Massachusetts.

2. The Programming Language HAL - A Specification, MSC-01846, June 1971, Intermetrics, Inc.

3. HALMAT, An Intermediate Language of the First HAL Compiler, MSC-01847, June 1971, Intermetrics, Inc.

Also complete in itself is, A Guide to the HAL Programming Language, which has been incorporated as Vol. II of this final report.

The rest of Vol. I is organized as follows: Chapter 2 presents an overview of the contract and the objectives. Chapter 3 is a technical summary of the HAL language. Chapter 4 discusses each contract task and contract related topics and attempts to summarize significant information about each.

CHAPTER 2

OVERVIEW

2.1 Introduction

Designs for the next generation of manned space vehicles are currently being formulated. They involve advanced computer systems performing comprehensive tasks of guidance, control, navigation, monitoring, data reduction, and communication. The typical aerospace computer that was employed for each of these roles in the past was of modest performance capability and possessed a limited storage facility which was considered to be just sufficient for each function. However, it is expected that the new generation of on-board computing systems will demand the on-line availability of vastly increased computational power and system resources. The chief characteristics of future space missions that will contribute to the need for extensive on-board computer facilities are:

a) the variety and complexity of the systems and the activities which the computing system is expected to service. Examples of these are vehicle control, life support, inventory management, scientific experiments, communications with the crew, with other spacecraft and the ground, etc.

b) the vast amounts of data that are expected to be generated on-board must be reduced and stored. Estimates of the bulk of data generated by the systems aboard an orbiting space base have been as high as 10^{11} bits per day [9].

c) the long duration of the planned missions of the future will require the computing system to be capable of performing a leading role in a large variety of missions, and of providing support to numerous and varied programs of scientific exploration. These tasks will involve a massive amount of supporting software which must be available on-board.

Manned spaceflight computers, thus far, have been special purpose machines performing tasks principally in guidance, navigation, control and pilot displays. The computer has been provided with a restricted instruction set, small working memories with no secondary storage capability, and established interfaces

to a limited number of output devices or special-purpose displays. For the most part, programming has been accomplished in basic machine language. Although a number of higher-order programming languages has been developed, none has been applied to manned spaceflight computer software development.

Future programs such as the shuttle and space station, will require more complex software within the flight computer. The functional processing requirements for the on-board system will increase in scope. In addition to performing guidance, navigation, and control, the computer will handle centralized data management functions while responding to requests from a number of general-purpose display and control units. Other functions will include in-flight monitoring, flight planning and management, the control and collection of data from a number of experiment sensors, and in the case of the space base, provision for a modest amount of on-board software development. The advanced spaceborne computer will perform functions common to a large ground-based data processing facility, providing many diverse computational services, and will involve extensive man-machine interfacing.

Evolving flight computer system hardware will also impact software design. Distributed multicomputers as well as large centralized multiprocessing systems are being proposed as candidate flight computer systems. These systems attempt to provide high reliability and flexibility as well as increased computational power. They portend a more complex environment for the software involving problems of resource conflicts, data protection, error detection and recovery, and parallel processing.

Past experience has shown that developing software for manned space projects such as Apollo is a task of major proportions. Heavy penalties in cost and time have been paid for underestimating the manpower and time necessary to produce effective, qualified, and documented software. The problems of design, control, and management of software have not been easy to determine; techniques and procedures to cope with them have been slow to evolve. The development and application of a higher order programming language is an essential step toward achieving a more orderly and controlled software production effort.

2.2 Objectives of the Contract

The principal objectives of the effort were threefold:

1. develop a set of requirements that a language must satisfy if it is to be useful to the future needs of MSC;

2. analyze existing computer languages that appear likely candidates and synthesize a resultant language that is best suited for NASA purposes;

3. design and implement a compiler that will accept statements in the developed language and produce code that will operate on the IBM 360/75 at the RTCC at MSC.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

CHAPTER 3

A SYLLABUS OF HAL

3.1 Shuttle Language Requirements

As a result of an extensive language requirements analysis, Intermetrics designed HAL in order to satisfy the following requirements:

1. The principal application of HAL is for the development of manned spaceflight computer software for the 1972-1980 period and this includes Shuttle and Space Station applications. (Initial orientation will be toward the Shuttle system.)

2. Software applications should include: (a) navigation, guidance, targeting and general mission programming; (b) vehicle control and stabilization; (c) operating systems; (d) on-board checkout and system monitoring; (e) data management; (f) communications and displays; (g) compiler and support software.

3. The language and compiler should be designed for a wide range of flight computer systems and should be capable of supporting simplex configurations as well as advanced multicomputer and multiprocessor computer systems.

4. The language should be machine-independent with a minimum of exceptions.

5. The language and compiler must contain specific features to aid in achieving high software reliability. The design shall:

   a) strive toward clarity and readability in the language;

   b) enforce programming standards and conventions;

   c) perform extensive automatic checking;

6. The output format of the language should endeavor to present data types, attributes and operations in an unambiguous way. An equation should look like an equation.

Character strings (or text) should be easily differentiated from vectors, or arrays. The compiler will annotate the output listing, accordingly.

7. The language should be oriented toward a general class of technical personnel involved in manned spaceflight projects, not solely highly trained programmers.

3.2 Chronology of Software Development (Program Documentation)

The format of a space programming language; i.e., its source input and printed appearance, should be designed to achieve maximum readability, ease in transfer of knowledge and understanding, and it should provide a basis for program documentation. Most existing higher order programming languages strive towards these goals as secondary objectives, with program composition as the first priority. Certainly any new programming language must be easy to use but composition is only the "front-end" of a long process to develop reliable space software. In perspective, stronger emphasis must be placed on software control techniques and accompanying documentation.

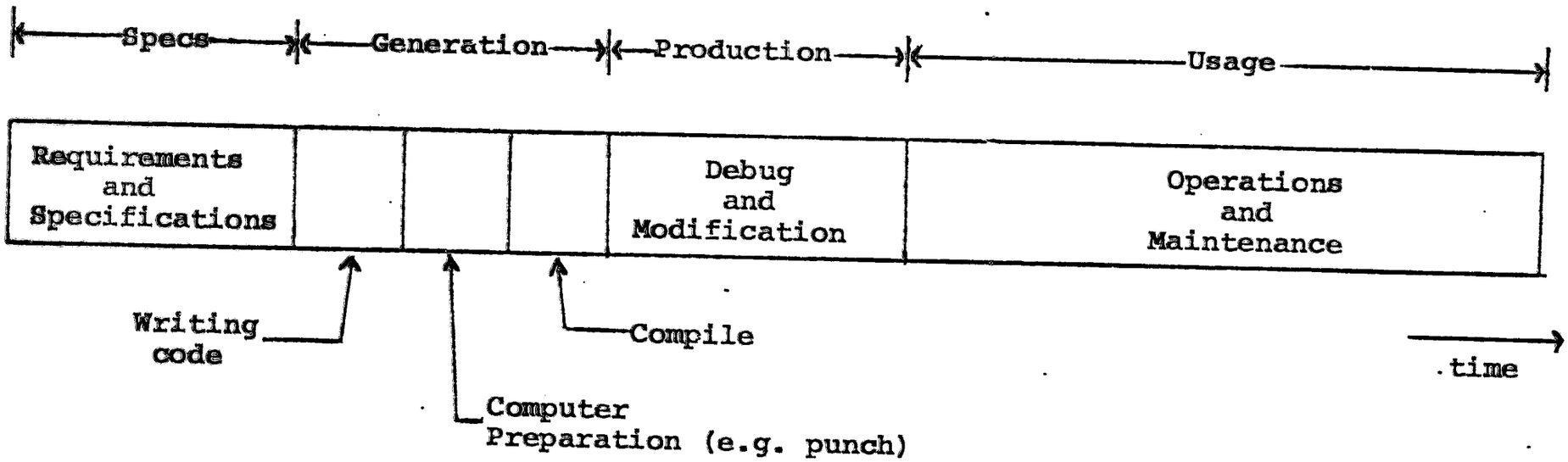
The important point illustrated in Fig. 3.1 is that the anticipated operational life-span of software will far exceed in time and effort that taken to generate the programs. Consequently, the language should emphasize readability and clarity for maintaining the software rather than emphasizing ease in program preparation.

The printed appearance of a langauge greatly influences its usefulness as a communications medium. Most of today's higher order languages, being fundamentally compatible with card punches or data terminals, are written in single line format. A multi-line, or two dimensional, format can bring a programming language closer to being natural in expression and mathematical form. Equations, whether scalar or vector-matrix, look like equations and promote understanding among programmers and technical managers. Subscript and superscript lines, in which exponents and subscripts appear in standard mathematical notation, provide an opportunity to make data forms self-evident. For example:

$$\overline{VDOT} = -\overline{G} + K_3^2 \overset{*}{M}\, \overline{C}$$

is obviously a differential equation involving the vector data types $\overline{VDOT}$, $\overline{G}$, $\overline{C}$; the matrix data type $\overset{*}{M}$; and the square of the subscripted scalar K_3.

Fig. 3.1 CHRONOLOGY OF SOFTWARE DEVELOPMENT



Some Observations

1. The writing of code is closely tied to the specifications.
2. The time required for computer preparation is small compared to the program life.
3. A lengthy period of debug and modification must be provided.
4. Period of program usage extends many times that of program generation.
5. Many more people will use a program than generated it.

Conclusion

The computer language should promote understanding of the software. The listing should tend toward self-documentation.

The appeal of the two-dimensional format is not simply esthetic; the conventional mathematical notation for input and output will be decisive factors in promoting software reliability. Coded operations will become visible to managers and supervisors who have ultimate project responsibility. The language can provide the basis of communication for a broad spectrum of engineers, scientists and technicians contributing to the shuttle program.

3.3 Salient Features of HAL

In order to meet the stated language requirements HAL includes the following features:

1. Two-dimensional input-output and annotation of variables

   An equation which involves exponents and subscripts may be written, for example, as

$$C_I = (X\ A_J^2 + Y\ B_K^2)^{3/2}$$

   (HAL also provides for an optional one-line format.)

   In addition, in an effort to increase program reliability and promote HAL as a more direct communcations medium between specifications and code, the HAL program listing is annotated with special marks. Vectors, matrices, and arrays of data are instantly recognized by bars, stars and brackets. Thus, a vector becomes $\overline{V}$, a matrix M, and an array [A]. Further, bit strings appear with a dot, i.e., $\dot{B}$ and character strings with a comma, $\acute{C}$. With these special marks as aids, the source listing is more easily understood and serves as an important step toward self-documentation.

2. Complete vector-matrix arithmetic

   HAL can be used directly as a "vector-matrix" language in implementing large portions of both on-board and support software. For example, simplified equations of motion might appear as:

$$\overline{A} = \overset{*}{B}\ \overline{ACC};\quad \overline{G} = -MU\ UNIT(\overline{R})/\overline{R}.\overline{R};\quad \overline{VDOT} = \overline{A} + \overline{G};\ \overline{RDOT} = \overline{V};$$

3. Bit and character manipulations

For handling I/O devices, communications (up/down telemetry and text messages), and support programs (executive, compiler, etc.) HAL provides for the necessary "bit-pushing" and character manipulations. Individual bits may be treated as Boolean quantities or grouped together in strings. Strings of bits and/or characters can be concatenated, deconcatenated, and converted to other data types.

4. Data arrays and structures

In anticipation of the need to process large volumes of measurement and experimental data (on advanced Shuttle or Space Station missions) and to facilitate general file handling or parallel computations, HAL provides for organizations of data. Multi-dimensional arrays of any single type can be formulated, partitioned, and used in expressions. A hierarchical organization called a structure (similar to the title, chapter, section, paragraph organization of a book) can be declared, in which related data of different types may be stored and retrieved as a unit or by individual reference.

5. Real-time control

HAL is a real-time control language; that is, certain defined blocks of code called programs and tasks can be scheduled based on time or the occurrence of anticipated events. These events may include external interrupts, specific data conditions, and programmer-defined software signals. Undesirable or unexpected events, such as abnormal conditions, may be handled by instructions which enable the programmer to specify appropriate action.

HAL's real-time control features permit the initiation and scheduling of a number of active tasks. This will be a necessity in order to cope with the computational tasks required of the Shuttle's integrated avionics system.

6. Controlled data sharing

Program reliability is enhanced when a software system can create effective isolation for various subsections of code as well as maintain and control commonly used data. HAL is a block-oriented language in that a block of code can be established with locally defined variables that cannot be altered by sections of program located outside the block.

Independent programs can be compiled and run together with communication among the programs permitted through a centrally managed and highly visible data pool. For a real-time environment, HAL couples these precautions with a locking mechanism which can protect, by programmer directive, a block from being entered, a task from being initiated, and even an individual variable from being written into, until the lock is removed.

These measures cannot in themselves ensure total software reliability but HAL does offer the tools by which many anticipated problems, especially those prevalent in real-time control, can be isolated and solved.

3.4 HAL Program Organization

In order to accommodate all Shuttle programs in a single computer, or substantial portions in distributed computers, it is imperative that programs be isolated from one another except at controlled and visible interfaces. This isolation should prevent the unrestricted access of common data and the arbitrary transfer of control to any location in the instruction logic.

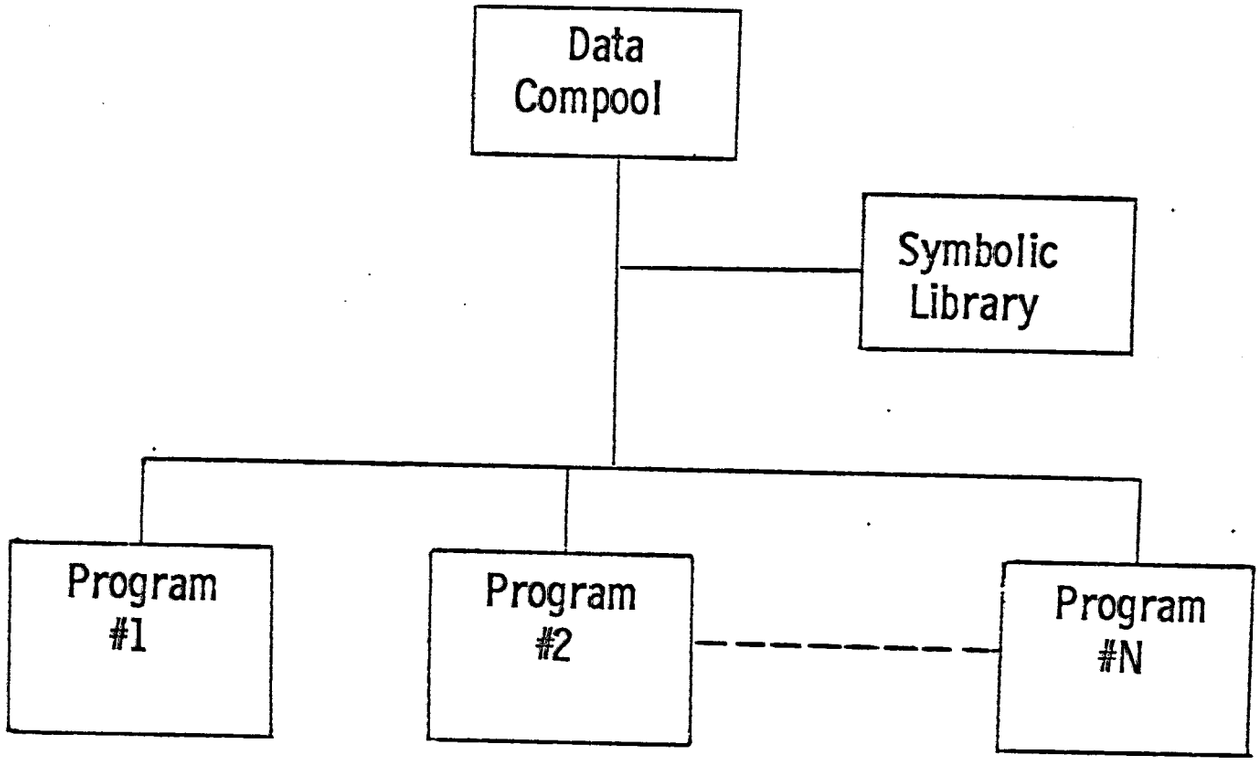
Software techniques now exist which allow many programs, designed to do various related and unrelated functions, to be written and incorporated in a single computer without conflict. The apprehension that the Shuttle DMS might be a bigger and more complicated Apollo-type effort with even more erasable conflicts and control interferences is blunted by the introduction of effective software modularity through language and compiler.

Fig. 3.2 illustrates the HAL program organization. The individual numbered programs represent independently compilable units. Thus, for example, Program #1 might be rendezvous navigation, Program #2 - autopilots, Program #3 - environmental system monitoring. Independent compilation permits divergent groups of people to contribute to the whole and yet progress at varied paces with measures of local management control.

The communication between programs is provided through a common data pool (COMPOOL). The COMPOOL is a centrally defined and centrally maintained group of definitions. Variable names and location labels in the COMPOOL are potentially known to all programs and, in fact, provide the only means of communication between programs.

Thus, for the Shuttle, the many tasks can be apportioned into programs which are managerially or functionally convenient. Information interfaces among programs then become visible at

Fig. 3.2 HAL PROGRAM ORGANIZATION



the COMPOOL level and can be monitored with respect to definition and usage by a central authority.

3.5 Blocks of Code (Name Scope)

For the purposes here, tasks, procedures, and functions may be considered as subroutines (or blocks). As stated, names defined in the COMPOOL are potentially known in every program. Names defined at the program-level are potentially known within all included (or nested) subroutines, and so on. The region in which a name is known is referred to as its scope. Names are only potentially known because any particular name can be declared again in an inner block and then its scope would become all the nested blocks within this block. The example in Fig. 3.3 may help to illustrate these principles:

Two desirable effects of the scope of rules are:

1. Common data must be declared at the highest level and only once. This contributes to more direct management control and better visibility.

2. Local variables may be defined within inner blocks and remain unaffected by outside definitions. For example, a programmer declaring X in procedure CHARLIE need not fear that any other program will over-write his quantity. That is, this particular X is not addressable from outside this block. In fact, the X in GRAB must refer to different memory cells.

For the Shuttle application, a name scope, or block-oriented, language means that many programs and subsections of programs (i.e., subroutines) can "live" in the same computer, isolated, and unaware of each other, incapable of writing-over or otherwise interfering with variables or locations that are not mutually defined.

3.6 Control of Shared Data

In order to illustrate the problems in a general way, that can arise in sharing data, consider the examples shown in Fig. 3.4.

In both examples TASK B interrupts TASK A during the execution of a statement. The interruption may be caused by a hardware or software interrupt or by a "job swap" based on priority. In Example 1, presume that the interruption occurred while the matrix X was being read. When TASK A resumes, the

Fig. 3.3 BLOCKS OF CODE (NAME SCOPE)

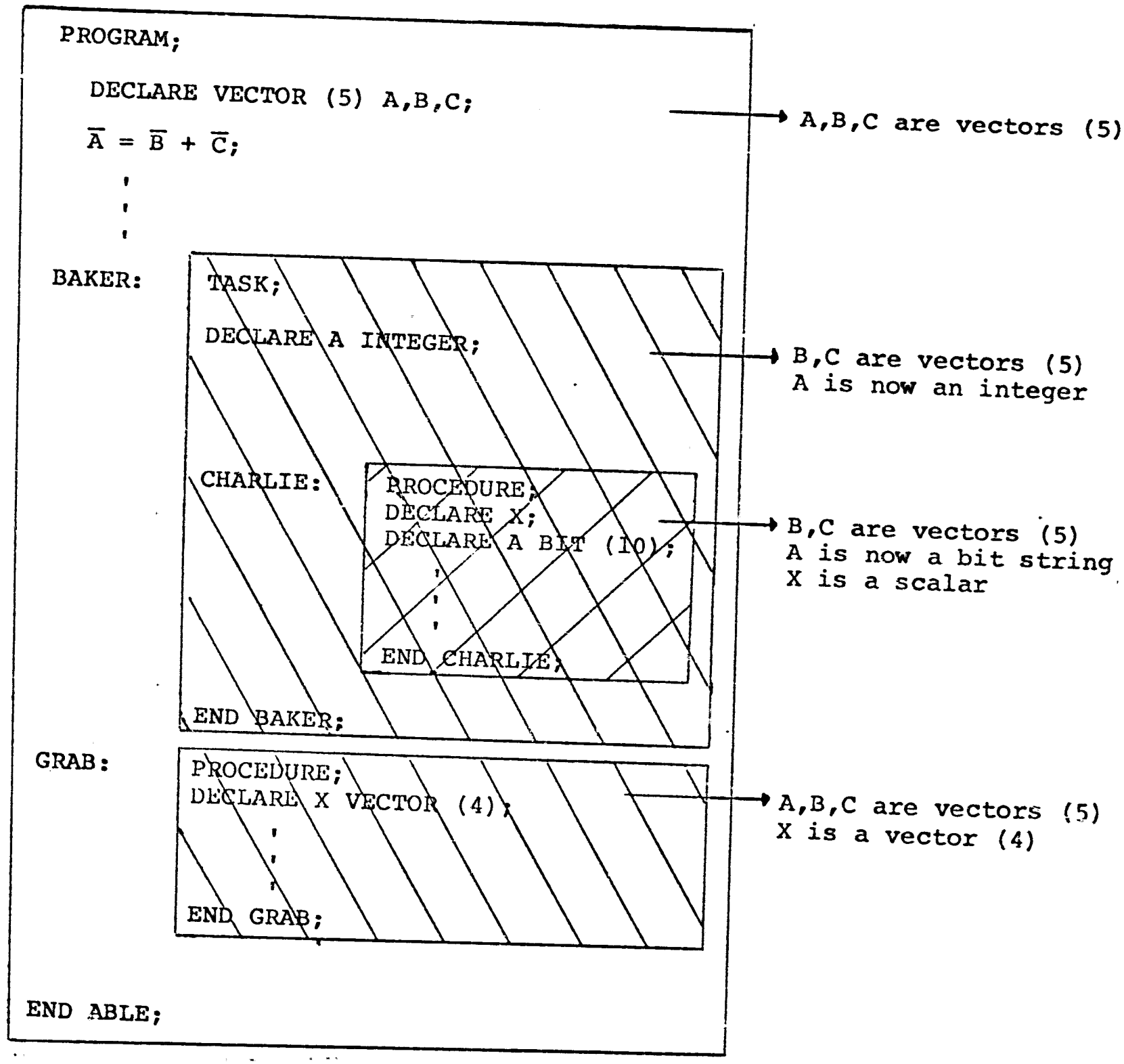
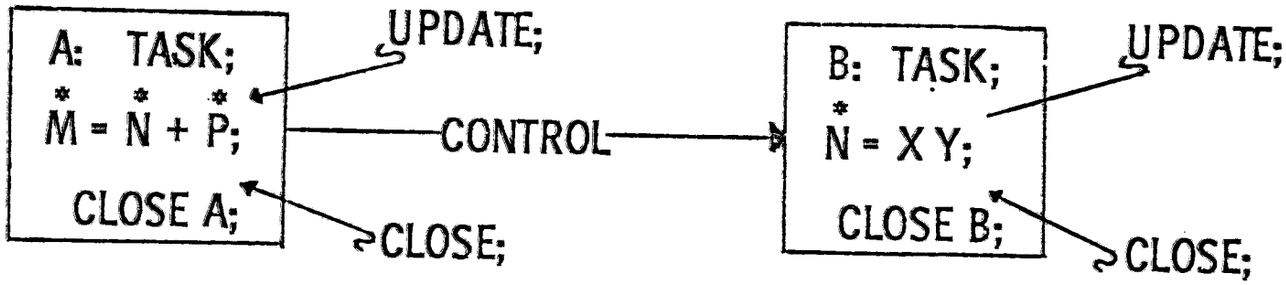
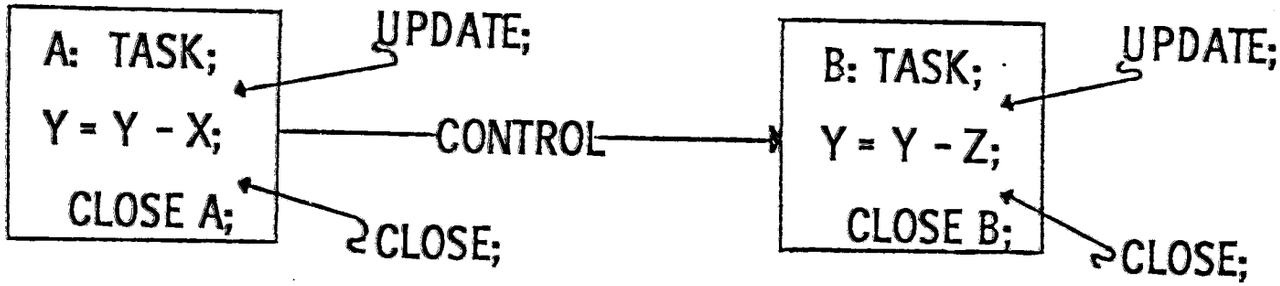


Fig. 3.4 CONTROL OF SHARED DATA

EXAMPLE 1: READ AND WRITE CONFLICTS



EXAMPLE 2: UPDATE CONFLICTS



NOTES:

1. B "INTERRUPTS" A IN BOTH CASES
2. #1 TASK A RESUMES USING OLD AND NEW VALUES FOR N*
3. #2 TASK A RESUMES "CLOBBERING" THE VALUE FOR Y SET BY TASK B

computation of $\overset{*}{M}$ will continue using some "old" $\overset{*}{N}$ data and the "new" $\overset{*}{N}$ data assigned in TASK B. In order to prevent this conflict, initiation of TASK B would have to be stalled until the reading of $\overset{*}{N}$ in TASK A is completed.

In Example 2, presume that the interruption occurs just after the current value of Y is loaded into the accumulator. When TASK A resumes, the "old" value of Y (i.e., not reflecting the update of Y in TASK B) is restored into the accumulator, X is subtracted and the result assigned to Y. In order to prevent this conflict, the initiation of TASK B would have to be stalled until the value of Y is updated in TASK A.

The approach taken, in HAL, towards solving the problems represented above, is to confine the read and write accesses of shared variables to identified UPDATE blocks. Consider Example 1 and suppose that the statements in question (in TASKS A and B) were enclosed within UPDATE blocks.

In TASK A a read-lock is established for $\overset{*}{N}$, because it will be read only. After the interruption, a write-lock is established for $\overset{*}{N}$ and TASK B proceeds toward completion using copy-data for $\overset{*}{N}$ rather than active data. At the end of the update-block in TASK B, the process stalls because of the read-lock imposed in TASK A. As a result, TASK A is allowed to continue with consistent "old" $\overset{*}{N}$ data. After completion of TASK A, a copy-cycle is effected in TASK B and $\overset{*}{N}$ is updated. All conflicts are eliminated.

The use of an UPDATE block is not a simple solution to the data sharing problem and presumes a sophisticated compiler; and yet the goal is worth the effort.

For the Shuttle, data sharing will be a necessity. A unified approach, through a compiler, as outlined above, will permit safe operation in multi-program and even multi-processor environments.

3.7 HAL Code Can Look Like The Specification

HAL exhibits full scalar-vector-matrix capabilities; vectors and matrices of arbitrary size may be declared. As part of the HAL syntax, vector-matrix operations include: inner and outer products, cross products, transpose and inverse, matrix-vector products, etc. By raising exponents onto an exponent line and not requiring a special symbol for multiplication (e.g., the FORTRAN*),HAL arithmetic code can look remarkably like a written specification. Consider Fig. 3.5;

Fig. 3.5 EXAMPLES OF ARITHMETIC OPERATIONS

(From Apollo Navigation Equations)

HAL	GSOP Specification
$\overline{Z} = \overset{*}{W}^T \overline{B}$;	$\underline{Z} = W'^T \underline{b}$
$\overline{OMEGA} = \overline{Z}\ \overset{*}{W}^T/(ZMAG^2 + ALPHA^2)$;	$\underline{\omega}^T = \frac{1}{Z^2 + \overline{\alpha}^2}\ \underline{Z}^T W'^T$
$\overline{DELX} = \overline{OMEGA}\ DELQ$;	$\delta\underline{X} = \underline{\omega}\ \delta Q$
$\overline{X} = \overline{X} + \overline{DELX}$;	$\underline{X} = \underline{X}' + \delta\underline{X}$
$F = 1 + (ALPHA^2/(ZMAG^2 + APLHA^2))^{1/2}$;	$W = W' - \frac{\underline{\omega}\ \underline{Z}^T}{1 + \sqrt{\frac{\overline{\alpha}^2}{Z^2 + \overline{\alpha}^2}}}$
$\overset{*}{W} = \overset{*}{W} - \overline{OMEGA}\ \overline{Z}/F$;	

where $\underline{b}$ ≡ geometry vector

W ≡ square root of covariance

α^2 ≡ measurement variance

X ≡ state vector

a set of rendezvous navigation equations has been reproduced from the Apollo GSOP Specification document (MIT). This set incorporates the measurement data, δQ, and updates state and error covariance information. HAL code parallels the specification almost line-for-line with few formalisms. Note that the vectors, matrices and scalars are apparent and their marks aid in understanding the programmer's intent. The adjacent vectors OMEGA and $\overline{Z}$, in the last line, imply an outer product.

Some further examples of arithmetic expressions are:

	MATHEMATICAL NOTATION	HAL EXPRESSION
1.	ab	A B
2.	a(-b)	A(-B) or -A B
3.	-(a + b)	-(A + B)
4.	a^{x+2}	A^{X+2}
5.	$a^{x+2}{}_c$	A^{X+2} C
6.	ab/cd	A B/C D
7.	$(\frac{a+b}{c})^{2.5}$	$((A+B)/C)^{2.5}$
8.	$\frac{a}{1 + \frac{b}{(2.7 + c)}}$	A/(1 + B/(2.7 + C))
9.	$(\underline{v}^T\underline{y})M^{-1}(\underline{v}+\underline{y})$	$(\overline{V}.\overline{Y})\overset{*}{M}^{-1}(\overline{V}+\overline{Y})$
10.	$a(\underline{y}\ \underline{v}^T)^T\ (\underline{v} \times \underline{w})$	$A(\overline{Y}\ \overline{V})^T(\overline{V} * \overline{W})$

3.8 Control, Logic, and Computation

Fig. 3.6 illustrates the applicability of HAL in implementing a time critical routine. The example selected is that of cross-product steering of the Apollo Command and Service Module. When XSTEER is entered, TGO the time-to-go to engine cut-off, is compared with 4 seconds. If TGO satisfies the inequality then all the statements between DO and END are executed. That is, the vehicle command rate in navigation-base coordinates is zeroed, and an indicator switch SW is turned off. (Note that the "over-dot" indicates a bit string; in this case, a Boolean,

Fig. 3.6 CONTROL, LOGIC AND COMPUTATION

(Cross product steering of Apollo vehicle)

Involves scalars, 3-d vectors, 3x3 matrices, "Booleans"

```
XSTEER:  IF TGO < 4 THEN DO;
                    OMEGA_CNB = 0;
                    SW = OFF;
                    SCHEDULE ENGINE_OFF AT (TIME+TGO),
                             PRIORITY (20), E_OFF_ID;
                    GO TO START;
                       END;
DELM = C B DELT - DELV;
OMEGA_C - K(VG*DELM)/(ABVAL (VG) ABVAL (DELM));
OMEGA_CNB = SMNB REFSMMAT OMEGA_C;
GO TO START;
```

where TGO ≡ "time-to-go"

VG ≡ "velocity-to-be-gained"

OMEGA_ ≡ rate command

or flag.) A routine ENGINE_OFF is then scheduled (via a HAL real-time control statement) to be executed at a proper time hence, with priority 20. E_OFF_ID identifies this particular scheduled job, uniquely, so that it might be referenced at a future time, perhaps to terminate it. If TGO $\geq$ 4 seconds then none of the above instructions would have been executed. Instead, the steering command rate OMEGA_C would be computed based on the cross-product of the "velocity-to-go" ($\bar{V}G$) and the acceleration related term ($\bar{D}ELM$). Finally this rate is transformed from reference to stable member coordinates (REFSMMAT) and then from stable member to navigation base coordinates ($\bar{S}MNB$) for application to the autopilot within another routine.

Although the mnemonics may be unfamiliar to the reader, it is easy to see that HAL's expressiveness makes an important contribution toward self-documentation.

3.9 Subscripts and Partitions

The elements of vectors, matrices, bit and character strings, arrays and structures may be referenced by appropriate subscripting.

The first component of a vector or a one-dimensional array, is given the subscript 1, the second 2, etc. up to the total number of elements. Thus, for a 9-element vector, i.e.,

```
DECLARE V VECTOR(9);
```

the components may be written as,

$$V_1\ V_2\ V_3\ \cdots\ V_9.$$

A matrix or two-dimensional array may be thought of as being composed of horizontal rows and vertical columns. The first of the two subscripts refers to the row number, the second to the column number. For instance, a matrix of two rows and three columns would require the declaration

```
DECLARE B MATRIX(2,3);
```

and the elements could be referred to by writing:

$$B_{1,1}\ B_{1,2}\ B_{1,3}\ B_{2,1}\ B_{2,2}\ B_{2,3}$$

A range of subscripts may also be selected and used for partitioning; e.g., $\bar{V}_{1\ TO\ 4}$ partitions a larger vector $\bar{V}$ and selects the first four components to form a vector. The same basic approach is used to subscript bit and character strings; i.e., B_{16} selects the 16^{th} bit of a string, and $C_{5\ TO\ 10}$ selects characters 5,6,7,8,9,10 from the original character string.

Fig. 3.7 illustrates the partitioning of a covariance matrix in order to computer rms errors after a landmark measurement and initialization prior to the next measurement.

3.10 Bit and Character Manipulations

It is anticipated that for the Shuttle, bit and character strings will be required for: logical decision sets, interfaces with hardware, up and down telemetry formulation and decoding, processing of text as status information for ground and on-board display (or recording), complex logical decisions for checkout, monitoring and equipment reconfiguration, interface with machine code (if necessary), and the writing of special support software like an executive, assembler, compiler, simulator, etc.

The manipulation of bit strings, in HAL, is accomplished using the following four operators:

OPERATOR	DEFINITION
NOT ($\bar{}$,^)	complement
CAT (||)	concatenation
AND (&)	logical AND
OR (| or ¦)	logical OR

and certain built-in functions and conversions to other data types.

NOT complements each bit in the string; CAT joins two strings; AND and OR perform bit-by-bit logical operations on the corresponding bits of two bit operands. For example,

1. NOT $\dot{B}$

   Each bit in the string is complemented

2. $\dot{A} = \dot{B}_{4\ TO\ 8}$ AND BIN'10110';

   A logical AND is performed on a bit-by-bit basis.

Fig. 3.7 EXAMPLES OF MATRIX PARTITIONING

Given: 9x9 covariance matrix E of errors in position, velocity and landmark location. That is,

$$\overset{*}{E} = \begin{bmatrix} \overset{*}{E}_{p-p} & \overset{*}{E}_{p-v} & \overset{*}{E}_{p-\ell} \\ \overset{*}{E}_{v-p} & \overset{*}{E}_{v-v} & \overset{*}{E}_{v-\ell} \\ \overset{*}{E}_{\ell-p} & \overset{*}{E}_{\ell-v} & \overset{*}{E}_{\ell-\ell} \end{bmatrix}$$

1. RMS Errors

$$\text{RMS\_POS} = \text{SQRT}(\text{TRACE}(\overset{*}{E}_{1 \text{ TO } 3,\ 1 \text{ TO } 3}));$$

$$\text{RMS\_VEL} = \text{SQRT}(\text{TRACE}(\overset{*}{E}_{4 \text{ TO } 6,\ 4 \text{ TO } 6}));$$

2. Initialize $\overset{*}{E}$ for new landmark

$$\overset{*}{E}_{1 \text{ TO } 6,\ 7 \text{ TO } 9} = 0;$$

$$\overset{*}{E}_{7 \text{ TO } 9,\ 1 \text{ TO } 6} = 0;$$

$$\overset{*}{E}_{7 \text{ TO } 9,\ 7 \text{ TO } 9} = \text{MATRIX}_{3,3}\ (A^2,\ 0,\ 0$$

$$0,\ B^2,\ 0$$

$$0,\ 0,\ C^2);$$

The manipulation of character strings is accomplished using the concatenation operator, CAT or (||). For example:

1. $\overset{'}{T} = \overset{'}{C}_{4 \text{ TO } 9} || \overset{'}{A}_{1 \text{ TO } 5};$

   The joining together of two character "sub"-strings.

2. WRITE(DISPLAY)'BATTERY VOLTAGE EQUALS'||VOLTS;

   The value of VOLTS is joined to the standard message.

Fig. 3.8 illustrates the use of bit and character strings in decoding a systems status variable and displaying an appropriate status message. On entering DECODE the first three bits of SYSTEM_STATUS are examined and the proper CASE (i.e., system) selected depending on the integer value 1,2,3,4, etc. A partial message is created. The last three bits are then examined to determine status. Thus, the bit string 011Ø001 would cause the message

IMU SYSTEM STATUS: O.K.

3.11 Real Time Control and Error Recovery

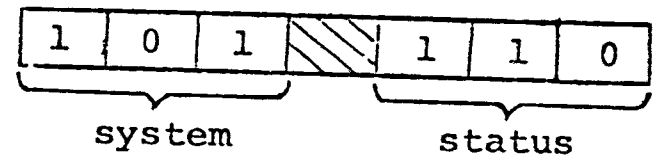
The real-time control of HAL programs consists of the interrelated scheduling of PROGRAMS and TASK blocks, the reliable sharing of common data, and the recovery from abnormal error conditions.

The concepts and language features have been designed for general applicability to real-time control programming. It is recognized that depending upon specific hardware environments and operating system designs, certain features may not find utility.

HAL real-time control commands permit the scheduling of independent or dependent tasks (i.e., dependent on the existence of another task), based on time and events, establishing absolute and relative priorities, and assigning unique I.D. words for future reference. Before or after the initiation of a task, its priority may be changed or the task may be terminated using the appropriate I.D. word. Once operating, a task may be stalled and/or reactivated based on time or events. Events may be defined to be interrupts or programmer-initiated occurrences. Tasks may then be scheduled or stalled based on single events or combinations of events. The HAL real-time commands include:

Fig. 3.8 BIT AND CHARACTER MANIPULATIONS

Suppose the system-status word is made up as follows:



Example:

```
          A = 'SYSTEM STATUS:';
DECODE:   DO CASE SYSTEM_STATUS 1 TO 3;
          MESSAGE = 'ENGINE'||A;          CASE 1
          MESSAGE = 'POWER'||A;           CASE 2
          MESSAGE = 'IMU'||A;             CASE 3
          MESSAGE = 'LIFE_SUPPORT'||A;    CASE 4
             ***                             ***
           END;
          DO CASE SYSTEM_STATUS 4 TO 6;
          MESSAGE = MESSAGE||'O.K.';              CASE 1
          MESSAGE = MESSAGE||'RECONFIGURED';      CASE 2
          MESSAGE = MESSAGE||'IN SELF-CHECK';     CASE 3
             ***                                     ***
           END;
END_DECODE: WRITE(DISPLAY)MESSAGE;
```

SCHEDULE	unconditionally	WAIT FOR...	an event
SCHEDULE...ON	event (interrupt)	SIGNAL	enables occurrence of programmer-defined event
SCHEDULE...AT	time		
SCHEDULE...IN	time increment	PRIO CHANGE	changes priority of task
WAIT	time increment		
WAIT UNTIL...	a time	TERMINATE	terminates task

Fig. 3.9 illustrates an example of HAL real-time control.

During execution of HAL programs, an error condition may be detected by the system. Examples of errors might be:

overflow/underflow

divide by zero, or subscript out of range

Depending upon implementation such errors may be hardware of software detected. In any case, execution cannot continue and the system must offer generally applicable alternatives (e.g., aborting the current task, etc.). In order to provide the programmer with some control after the occurrence of an error, perhaps to reset flags or previously initiated I/O commands (e.g., engine jets), HAL permits programmer-defined error conditions and alternatives. An example might be

$$\text{ON ERROR}_{1 \text{ TO } 5} \text{ GO TO RECOVERY;}$$

which sets up a remedial action on the occurrence of 5 errors, and

$$\text{SEND ERROR}_{5};$$

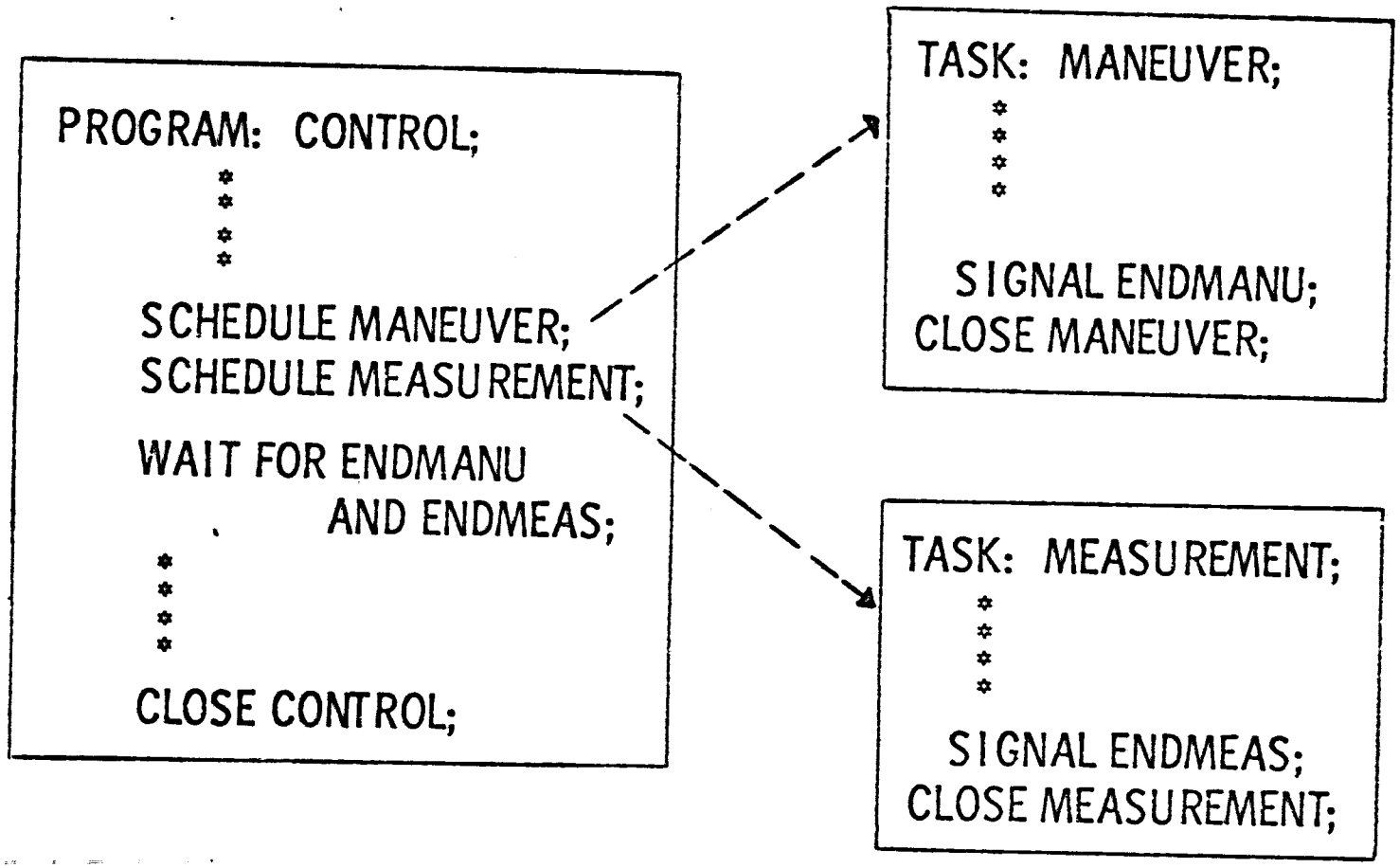
which signals the actual occurrence of the particular ERROR.

3.12 Summary

HAL has been designed for applicability to advanced manned space missions. As such, reliability in terms of readability and data protection has been emphasized. By incorporating a full spectrum of data types, including bits and characters, HAL finds utility for on-board software, in fixed- or floating-point machines as well as for ground support, simulation, analysis, test and checkout, compilers and assemblers. HAL provides an excellent programming language with which to build special purpose 'user languages' for display, checkout, etc. It is anticipated that development of HAL will continue on a schedule necessary to support Shuttle software design and implementation.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Fig. 3.9 EXAMPLE OF HAL REAL-TIME CONTROL



ENDMANU and ENDMEAS are programmer-defined events

CHAPTER 4

HAL LANGUAGE AND COMPILER STATUS

4.1 Language Requirements Analysis

The first phase of the contract that was undertaken was the definition of the requirements to be placed on a programming language for advanced space missions. Some of the guidelines that evolved to focus the direction that the language would take are given below.

4.1.1 Scope of the Language. A large manned spacecraft project is comprised of many software activities, including a) software for the onboard computer system, b) the analysis, simulation and test programs necessary to develop this onboard software, c) software for ground-based systems for in-flight mission control and support, d) mission planning and analysis software, e) simulation and flight training software, f) post-flight mission data reduction and analysis software. The language requirements for all of these cover a broad spectrum. The emphasis in this contract was to formulate a language principally oriented at the development and maintenance of reliable software for onboard applications, and ground-based real time control and support. Although designed for space application, the language should also be well suited for general aerospace engineering problems, and directly applicable to mission planning and analysis.

It should be noted that the size and complexity of potential flight computer systems which are targets for the language could vary considerably from a simplified candidate for the space shuttle to more sophisticated and complex systems for the space base. The language should be designed with sufficient features to enable it to be "scaled" up or down according to the complexity of the system without having to develop a completely new language for each level of complexity. Therefore, although the initial language design encompassed a broad spectrum of objectives, applicability to the possibly limited requirements of the near future always remained a paramount consideration.

Experience has shown that approximately 5 or more years are required to design, develop, and verify software for a project

of the magnitude of the Space Shuttle or Space Base systems. It is evident that the management of such an effort will require considerable insight, visibility and control to assure timely production of reliable software. System planners must be careful to employ the preventive measures necessary to avoid the pitfalls of poor design and implementation.

Accordingly, the programming language and compiler will serve as tools to assist in the program preparation. The following sections present a description of some of the key problems in developing manned spaceflight software, which have influence on requirements of the programming language and compiler.

4.1.2 Software Reliability and Verification. A unique characteristic in the development of flight computer software is the required degree of reliability and the amount of validation necessary to achieve that reliability. The language and compiler should contain specific features to aid in this development.

Apollo software demanded the highest pre-flight confidence, and because neither the computer, the language, nor the programming techniques were designed with checkout in mind, verification was accomplished only through the philosophy of laboriously defining and carrying out tests for every logical path, every logical state. For any complicated program this task is essentially impossible and the approach reduces to "the more tests, the more confidence". All too often, success criteria are subjective and inconclusive. What is needed is the development of a technique that ensures more deterministic behavior of the software in its operational environment within the flight computer. The technique should attack specifically the problem of minimizing the test effort involved in the integration of individual program modules. Checkout of individual modules is not as significant a problem as groups of modules in a system or mission sequence. The objective should be to specify a finite number of tests for each function, and to achieve a definable level of confidence once all tests are successful. The technique should be aimed at limiting the possible number of "states" of each software element by ensuring that its interaction with the environment is predetermined, well defined and bounded in some sense. To a large extent this can be met by the incorporation of features into the compiler or language which ensures "reliability" of the software and lessens the amount of testing.

4.1.3 Communications and Software Documentation. In a large programming effort many individuals, some representing quite different activities, are required to communicate with each other,

and each does so in a way that has meaning to himself. The manager attempts to specify his requirement; the engineer attempts to describe his design, qualitatively to the manager and quantitatively to the programmer; the programmer is plagued with documenting the code to a level that will be useful for descriptive material, debugging activities, and a quick reference, as the authority for 'what is really going on in the machine'. Clearly, all contributors want and have a need to know at least some of the intricacies of the implementation. The problem here is language. Each group has only a limited comprehension of the others' mode of expression. The engineer designs and expresses his algorithms using conventional mathematics, or perhaps FORTRAN-like statements. The programmer must take this specification and translate it into his language: traditionally a basic assembly language appropriate to the particular computer. The programmer must then explain his efforts to the user in the field, by using other media. He might have charts drawn to describe detailed functional flow. He might use word statements or user-guides, or other apparently helpful devices.

In many projects the coding language isolates the programmers from everyone else associated with the effort. The programmer becomes too busy to learn the physics and objectives of the mission and is too busy to explain to others how the code works. He, therefore, is forced to assume an increasing share of the total responsibility. Small indispensable groups of experts direct and shape the code and become the overworked "authorities".

A properly designed programming language will prove to be a useful analytical tool for the designer, a convenient and useful program tool for the programmers and will provide a medium for communication and documentation for technical management. It will assist in bridging the communications gap. The language will be oriented toward a general class of technical personnel in the manned spaceflight project concerned with software. Users are presumed to have technical backgrounds and some familiarity with aerospace problems. The language will not be oriented solely at the highly experienced flight computer programmer, solely at the novice or non-scientific programmer, nor solely at management personnel.

It was a specific objective of HAL to promote the ability of the user quickly and easily to learn to read, write, understand, and "think in" the language, and to document his results in a clear and unambiguous manner.

4.1.4 Summary of Requirements. The following is a summary list of language and compiler requirements developed as result of an analysis of manned spaceflight programming requirements.

4.1.4.1 General Requirements

4.1.4.1.1 The principal application of the language is for the development of manned spaceflight computer software for the 1972-1980 period and this includes shuttle and space station applications. (Initial orientation will be towards the shuttle system.)

4.1.4.1.2 Software applications should include:

a. Navigation, guidance, targeting and general mission programming.

b. Vehicle control and stabilization

c. Operating systems.

d. Data management

e. Communications and displays

f. Compiler and support software.

4.1.4.1.3 The language and compiler should be designed for a wide range of flight computer systems and should be capable of supporting simplex configurations as well as advanced multi-computer and multi-processor computer systems.

4.1.4.1.4 The language should be machine-independent with a minimum of exceptions restricted to clearly identified areas.

4.1.4.1.5 The language and compiler must contain specific features to aid in achieving high software reliability. The design shall:

a. Strive toward clarity and readability in the language.

b. Endorce programming standards and conventions.

c. Perform extensive automatic checking.

4.1.4.1.6 As general goals, the language should, in descending order of importance:

a. Enable the programming of software for a wide variety of manned spaceflight applications.

b. Be easy to read and understand.

c. Be easy to debug.

d. Be easy to modify.

e. Be easy to use.

f. Be easy to learn.

g. Be easy to transfer to another computer.

h. Enable the enforcement of standards and conventions.

4.1.4.1.7 The output format of the language should strive toward presenting data types, attributes and operations in an unambiguous way. An equation will look like an equation. A character string (or text) will be easily differentiated from a vector, or array. The compiler will annotate output listing.

4.1.4.1.8 Language should be oriented toward a general class of technical personnel involve in manned spaceflight projects, not solely highly trained programmers.

4.1.4.2 Specific Requirements Language

4.1.4.2.1 To enhance readability, the language should possess distinct names and labels.

4.1.4.2.2 The language should possess the following data types: integers, fixed/floating point scalars, vectors, matrices, booleans, bit and character strings, and labels.

4.1.4.2.3 The language should possess the following data organizations: arrays of similar data types, "collections" of different data types (e.g., structures).

4.1.4.2.4 The language should possess at least the following data attributes: precision, dimension, initialization, global variable lock and unlock, and static and automatic storage.

4.1.4.2.5 The language should possess a complete set of scalar and matrix-vector arithmetic operations.

4.1.4.2.6 Boolean operations in the language should be the logical AND, OR and NOT operators and should include a convenient method of setting, resetting and inverting booleans. The language should possess, as a minimum, the following set of relational operators:

a. equal

b. not equal

c. less than, and less than-or equal

d. greater than, and greater than-or equal

4.1.4.2.7 The language will possess a flexible set of conditional and unconditional program transfer instructions, and data calls.

4.1.4.2.8 The language syntax must be accomplishable with a defined common character set. This common set is:

```
A - Z
0 - 9
+ - =
" ' . , ; : ! ?
< > ( ) / _
* # $ % & @
```

plus blank or space. The compiler will not reject an expanded character set; e.g. ^, ~, [ ], etc., where the expanded set provides convenient alternate forms when available.

4.1.4.2.9 The language will possess the capability of dealing with I/O operations, conditional error procedures, and real-time tasking. These may not be an integral part of the language syntax.

4.1.4.2.10 The language will include iteration statements nested to any level.

4.1.4.2.11 Basic machine language coding will not be permitted everywhere but will be restricted to clearly identified areas such as special subroutines.

4.1.4.2.12 Simple replacement type macros will be provided by the compiler.

4.1.4.2.13 The language should allow definition of global and local data which can be applied to independent program sub-sections.

4.1.4.2.14 The language should provide for data sharing indicators and control of global information for real-time use among program sub-sections.

4.1.4.2.15 The language will be designed for a 2-D input stream. (An optional 1-dimension input will also be provided.)

4.1.4.2.16 For character strings, the language will possess the string operator CONCATENATE and a form of deconcatenation. For bit strings the language will possess the logical AND, OR, NOT, and a method of shifting in addition to the string operator CONCATENATE and a form of deconcatenation.

4.1.4.2.17 The language will not provide for complex number arithmetic or data declarations.

4.1.4.2.18 Language will <u>not</u> include any specific code optimization directives.

4.1.4.3 Specific Requirements-Compiler

4.1.4.3.1 The compiler should allow independent compilation of sub-sections of the total program.

4.1.4.3.2 The compiler will possess a full library of mathematical functions.

4.1.4.3.3 A system to handle a collection of shared data in an orderly fashion is required.

4.1.4.3.4 The language will not provide for extensive or ambiguous mixed data-type operations.

4.1.4.3.5 The compiler should provide execution checking with reference to indexed data organizations; e.g. arrays.

4.1.4.3.6 The compiler will annotate the output listing to increase readability.

4.1.4.3.7 The output listing of the language will be in 2-D format.

4.1.4.3.8 The compiler will produce reentrant code when needed, but recursive code is not required.

4.1.4.3.9 The output character set will not be restricted. In order to achieve a maximum of self-expression the compiler will be capable of utilizing the full character set of the output device.

4.2 Language Analysis and Synthesis

4.2.1 Language Comparisons. The next step in the chronological progress of the contractual effort was the analysis of the potential of existing languages. Eight prospective candidates were examined for their suitability. Although absolutes are difficult to prove or even demonstrate conclusively - in fact, space programming could be accomplished with unlikely languages such as COBOL or LISP, or even none at all (witness most previous space programming efforts) - nevertheless, essential properties of certain languages made them relatively better or poorer prospects as measured by the requirements criteria. The candidates that were surveyed and a pithy comment on each are given as follows:

1. CLASP: tuned for small flight computers. (See 4.2.2)

2. SPL Mark IV: CLASP's big brother, general purpose replete with tables.

3. JOVIAL: The father of both CLASP and SPL, an old-timer and direct decendent of ALGOL 58.

4. ALGOL: ALGOL 60 was the trail blazer in its day and still the academic yardstick.

5. FORTRAN: The first and foremost, and still number one, and probably will be for the forseeable future despite its myriad shortcomings and venerable age.

6. PL/1: IBM's answer to the maiden's prayer - indeed, all maiden's prayers. The union of FORTRAN and COBOL that turned out to look like ALGOL.

7. MAC: The maverick from MIT that pioneered vector-matrix algebra and readability.

8. APL: The mathematicians' nirvana with its idyllic elegance. Concise notation, powerful operators, total generality, and indecipherability.

4.2.2 CLASP Evaluation. The most serious drawback to CLASP is one it shares with almost all the other languages - that it was not designed to enhance readability and self-documentation. This is of prime importance since it not only is a major contributor to the achievement of program reliability, but promotes all the other "EZ/2's" of the requirements section, easy to maintain, easy to learn, etc.

CLASP is oriented towards a small-scale fixed-point airborne computer. While this species will still be around for some time - especially in the military market, although there too sounds are being made that foretell of floating-point - the mandatory requirements clearly stressed the need for a more general approach. The operational environment of a future space mission langauge can best be described as a large-scale, cooperative programming effort. Large-scale is self-evident. Cooperative means that many programmers will be working together on a joint venture and sharing the tools, routines, and data on a real-time system. Contrast this with a commercial time-sharing environment where each user is trying to solve his own problem. As a consequence, CLASP has over-emphasized fixed-point arithmetic capabilities and short-cuts to code

optimization. An example is the arbitrary scaling of variables to minimize shifting in the computer; this is potentially an insidious hazard.

Other CLASP deficiencies include:

1. CLASP reflects its heritage as it still shows signs of the JOVIAL syndrome, such as overly brief abbreviations (A,B,I,E,L) and unusual syntax (periods '.' at one side or the other or certain labels) which detract from the legibility.

2. CLASP should encompass more comprehensive vector-matrix operations. The inclusion of a special matrix multiply operator /*/ (an unsightly choice, at that) does not do justice to the vector-matrix algebra needs. It has been estimated that 80% of the scientific computations for space flight programs involve matrix calculations [7]. Most languages ignore the needs of linear algebra; exceptions being MAC and APL.

3. No mention has been made of the ability to separately compile program sections and integrate them with an orderly sharing of commonly used data (COMPOOL concept), and subroutines. The control, interlocking, and accessing of data and procedures is vital.

4. Better bit handling capabilities are needed. The logical formulae are a strange sort of orphans since there is no equivalent data type for them to be tied to. And PACK and UNPACK are awkward functions, and not even true substitution operations at that.

5. It has an unclear scope of names of variables and labels.

6. A parameterization capability is required. A form of compile-time variable (the integer constant in CLASP) should be permitted anywhere that a literal constant is valid.

7. Real-time control capabilities are minimal.

8. The rigid formulation permitted for subscripts is far too restrictive and inflexible. It is an apparent anachronism left over from FORTRAN.

9. It lacks a subroutine library and I/O routines.

10. It needs the facility to create and minipulate complex data structures.

11. It should be more machine independent: word-length and hardware dependent functions should be modified; and the easy lapse into machine code via the DIRECT directive should not be encouraged.

12. The documentation should be less vague, ambiguous, and conflicting.

13. It has no character or bit concatenation operations.

14. Minor syntactical forms often lacked appeal.

4.2.3 The Lineage of HAL. As a result of the analysis of the various languages, it was decided that selecting one of the existing languages or a subset of them, or even grating onto them was not justified. It would have been nice to adopt an "off-the-shelf" language, but the demand and need for program integrity for future space missions dictate that all should be done that is possible at an early date to design into the language and software the special mechanisms required to achieve these objectives. And, furthermore, modern compiler building techniques permit the flexibility of language constructs that allow the language to grow and be shaped for maximum utility in the intended usage. On the other hand, a language that is too new and different has both learning and compatibility problems.

What were the desirable features of other languages that HAL tried to incorporate?

1. MAC: The inherent readability of the three-line format made it most attractive. In HAL it is generalized to multiple lines. Another characteristic that was brought over intact was the vector-matrix operations and symbolisms. They were integrated with the arrayness or repeatability concepts of most languages.

2. PL/1: The basic syntactical structure of HAL is straight from PL/1, which in turn owes much of its form to ALGOL. The conditionals, DO groups, DECLARE statements, and STRUCTURES are almost pure PL/1 with word changes here and there. On the other hand, PL/1 is a very rich and full language and has many, many features that are not needed nor applicable. Moreover, there have been efficient implementation of PL/1 subsets, which is what HAL superficially resembles.

3. JOVIAL: The JOVIAL (and full SPL) COMPOOL concept and objectives, if not the detailed apparatus, have been added to HAL.

4. APL: The generality of APL, especially in arrayness served as a model for HAL.

5. Several particular features were borrowed:

   a. The CASE statement of EULER

   b. The text replacement macro of XPL

6. HAL owes a debt (as do many languages) to FORTRAN for pioneering the general form for algebraic languages.

The result of this eclectic synthesis is HAL which is described further in Chapter 3 and Vol. II of this report and in Ref. 2.

4.2.4 The Suitability of FORTRAN. In meeting the language requirements, FORTRAN would rank at the bottom of the list. It lacks many of the features that are wanted. It really is only useful for scientific type calculations and has an awkward form that is hard to read at that. However, it has one big advantage going for it, it is well-known and widely used. And although differences pop-up in various implementations, they are usually easy to reconcile. This universality is such an asset that a strong case could be made that if a special language for the job were prohibited then it would be better to take FORTRAN and augment it with large set of special purpose subroutines than to accept a lesser known language that does only a fraction of the job and is not directly under control of NASA. However, the better choice still appears to shape the language to the NASA tasks. This seems the only way that the education and deepening insight which invariably accompanies the system development can effectively feedback into the subsequent work.

4.3 The HAL Compiler

Development efforts began early in the contract on the production of the compiler and were continued in parallel with the language development tasks. The first few months were spent in the development of tools and techniques that would be used in producing the HAL compiler. The objective was a compiler that would run on the IBM 360/75 at MSC and produce code for the 360/75 executed under RTOS. It was also a goal to separate the code generation process so that a code generator for another

computer could be substituted with a minimum of disturbance to the rest of the compiler.

The HAL compiler has been designed to take advantage of automatic compiler generation techniques. As a key ingredient, the total grammar of HAL is expressed in a meta-language and changes are easily incorporated and thoroughly checked for ambiguities and inconsistencies by a grammar analysis program that automatically produces the syntax tables that are used by the compiler. The system used to accomplish this is the XPL compiler generator system. It is described in the book, A Compiler Generator, by McKeeman, Horning, and Wortman [8].

4.3.1 The XPL Compiler Writing System. Three basic components comprise the XPL compiler writing system: ANALYZER, XCOM, and SKELETON. XPL is the name of a language (a PL/1 subset) in which these programs are written. Fig. 4.1 illustrates the interaction of these programs and shows the stages that are gone through in order to compile and run a HAL program. The main steps are:

1. The grammar of the language is described in a meta-language, BNF. Analyzer accepts the grammar and punches out decision tables used by the compiler.

2. The syntax tables are combined with SKELETON, a proto-compiler that contains the basic parsing and stacking mechanisms, and with the code generator that must be written to do semantic interpretation and output execution for each syntactical element.

3. The resultant source language statements are fed into XCOM which is a compiler that translates XPL statements into executable machine code for the IBM/360. This object code is the HAL compiler.

4. HAL source statements can then be compiled and executed.

A description of major elements is given below:

1. ANALYZER - The grammar analyzer is provided the description of the langauge in BNF, Backus-Naur Form. BNF is the meta-language most widely used to define programming languages. It offers a readable method to express any context-free grammar in an unambiguous way. In order to satisfy ANALYZER the grammar must meet certain requirements:

   a. The grammar must be unambiguous; each sentence must have a unique phrase structure.

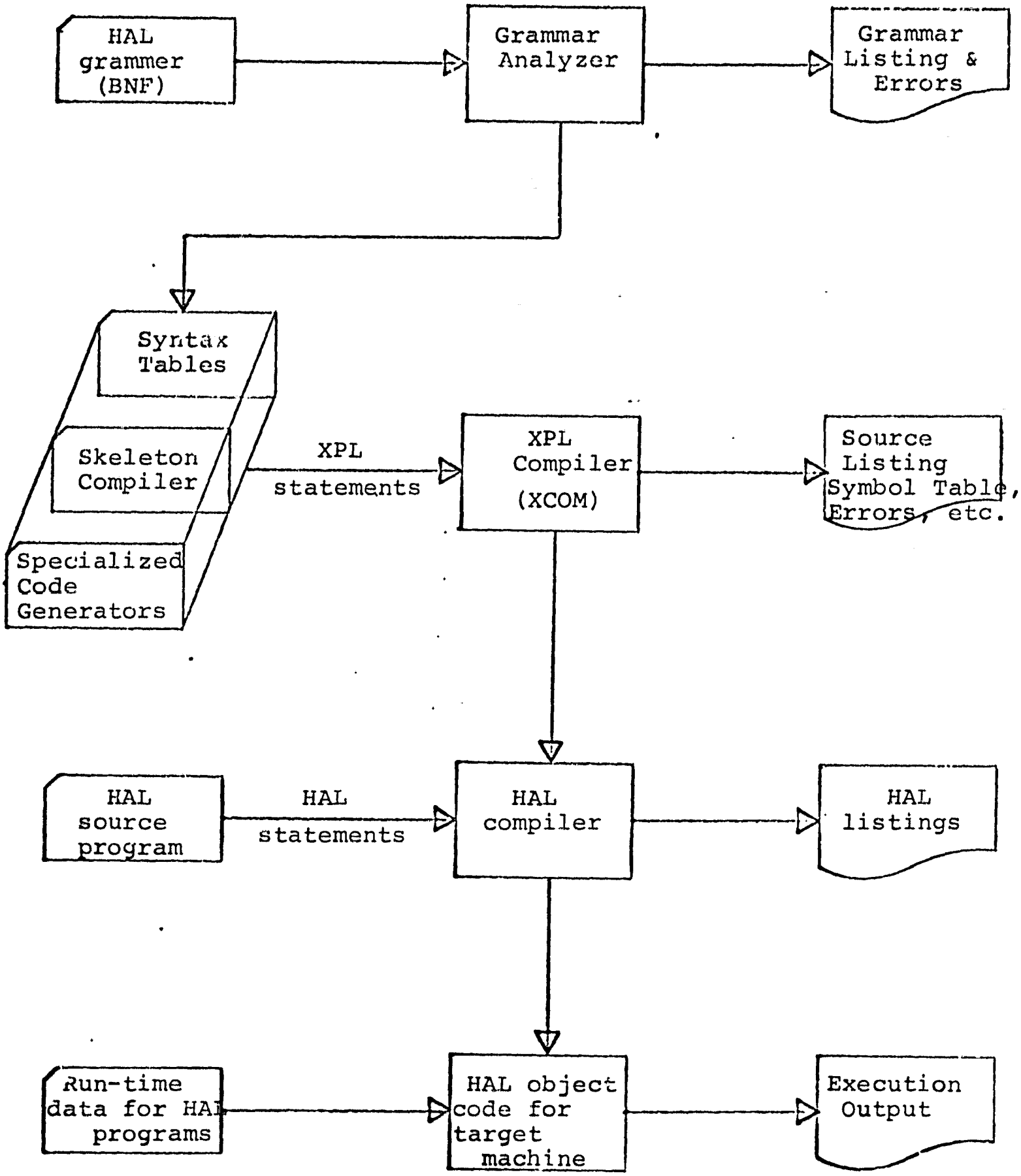


Fig. 4.1 XPL COMPILER DEVELOPMENT SYSTEM

b. The grammar must be compatible with the parsing algorithm used by the translator.

If a grammar does not measure up, error diagnostics are issued to aid corrective action. The method used is to "run the parser backwards". Instead of reducing a string of text to an end-item or goal symbol, it expands the goal symbol into all the possible forms that are described by the grammar and tabulates the decisions needed to parse the produced forms. Expansion continues until all possible cases have been examined. This exhaustive analysis of all permutations points out the loopholes that may be present in possibly obscure syntactic combinations, and ensures that the resulting parser will systematically handle all input forms. It cannot, of course, guarantee that every syntactical form will be given its intended interpretation or that correct code can and/or will be generated.

2. XCOM - XCOM is a one-pass compiler which translates XPL source statements into IBM/360 machine instructions. It was designed and constructed using the XPL system and thus is self-compiling. XPL is a dialect of PL/1 with a number of deletions and restrictions and a few extensions. Although upon superficial examination, XPL does not appear to differ significantly from PL/1, it is simplified enough so that it produces fairly efficient machine code and consequently will compile at about 7000 words per minutes, on the 360/75.

3. SKELETON - This proto-compiler is the framework to which the decision tables from ANALYZER can be added to form a syntax checker. The basic parsing algorithm, which is a bottom-up technique and a version of extended precedence, is built into SKELETON. To this can be added other XPL source code to accomplish semantic interpretation, code generation, and symbol table manipulation in order to round-out a complete compiler.

All three of these programs are written in XPL; ANALYZER and XCOM are compiled and the object versions executed, but SKELETON is only needed in source form and must be combined with other source checks before a useful group is formed and can be compiled (See Fig. 4.1).

ANALYZER is really a pre-processor and is only run when a change in the language or grammar is made. This should be pinned-down early and the syntax tables remain fixed through many revisions of the compiler. Because of its independence, ANALYZER could be written in almost any language; we have seen versions of ANALYZER in both PL/1 and FORTRAN IV.

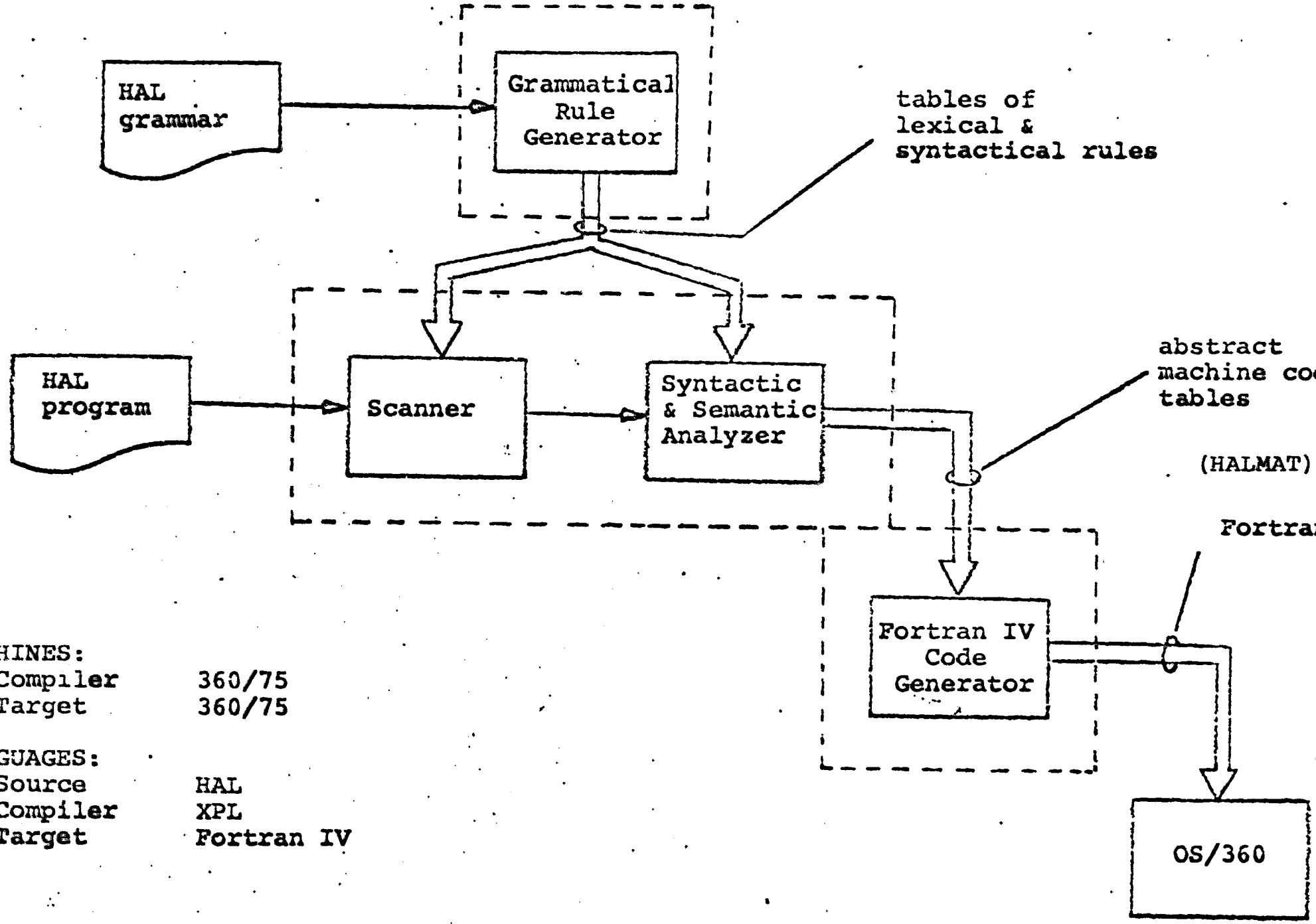
4.3.2 HAL Compiler Structure. The actual structure of HAL is a two-pass compiler. It has two phases of execution. PASS I carries out all the syntactic and virtually all the semantic analysis of the source text. At the same time symbol and literal tables are generated, together with text in the intermediate language HALMAT described in Reference 3. PASS II takes this intermediate text, and carries out the remainder of the semantic analysis, performs a small amount of code optimization, and generates executable object text in Fortran IV. Fig. 4.2 illustrates the process. The XPL system has been designed as a one-pass system (at least, core resident) and extensive changes had to be made to support an overlayed second pass.

Producing FORTRAN source statements as output may appear highly unusual if not eccentric, and was not decided upon lightly. First, the FORTRAN route is contemplated only for the 360 or other general purpose computer facilities. For a flight computer, the intermediate FORTRAN step would be omitted and assembly language statements would be produced. (See Fig. 4.3). Second, the only reasonably generalized alternative would be to produce input for the 360 Assembler which is itself no speed demon. Thus, the extra step of the FORTRAN compiler is not as bad as it might seem since it replaces the Assembler in the process. (Both FORTRAN G&H output machine code directly, and provides an optional optimizer when wanted.) However, the reasons for going through FORTRAN were two-fold:

1. It was extremely desirable to achieve compatibility with the great body of FORTRAN programs now in existence at MSC and elsewhere. If HAL is going to be widely used, then there must be a way to work new HAL programs into large FORTRAN simulations. To require massive conversion is unreasonable. Thus, FORTRAN compability was very important.

2. Fortran is universally implemented and offered a means to promote machine transferability. With ASA Standard FORTRAN IV it should be possible to take the FORTRAN source images and compile them on many different computers. Of course, not everything in HAL can be done in standard FORTRAN - there must be some machine language subroutines - but the great bulk of the job can be done in Fortran which promises an easy road to machine portability.

At this point an obvious question arises, "Why not use FORTRAN if HAL generates FORTRAN?" The answer lies in the way



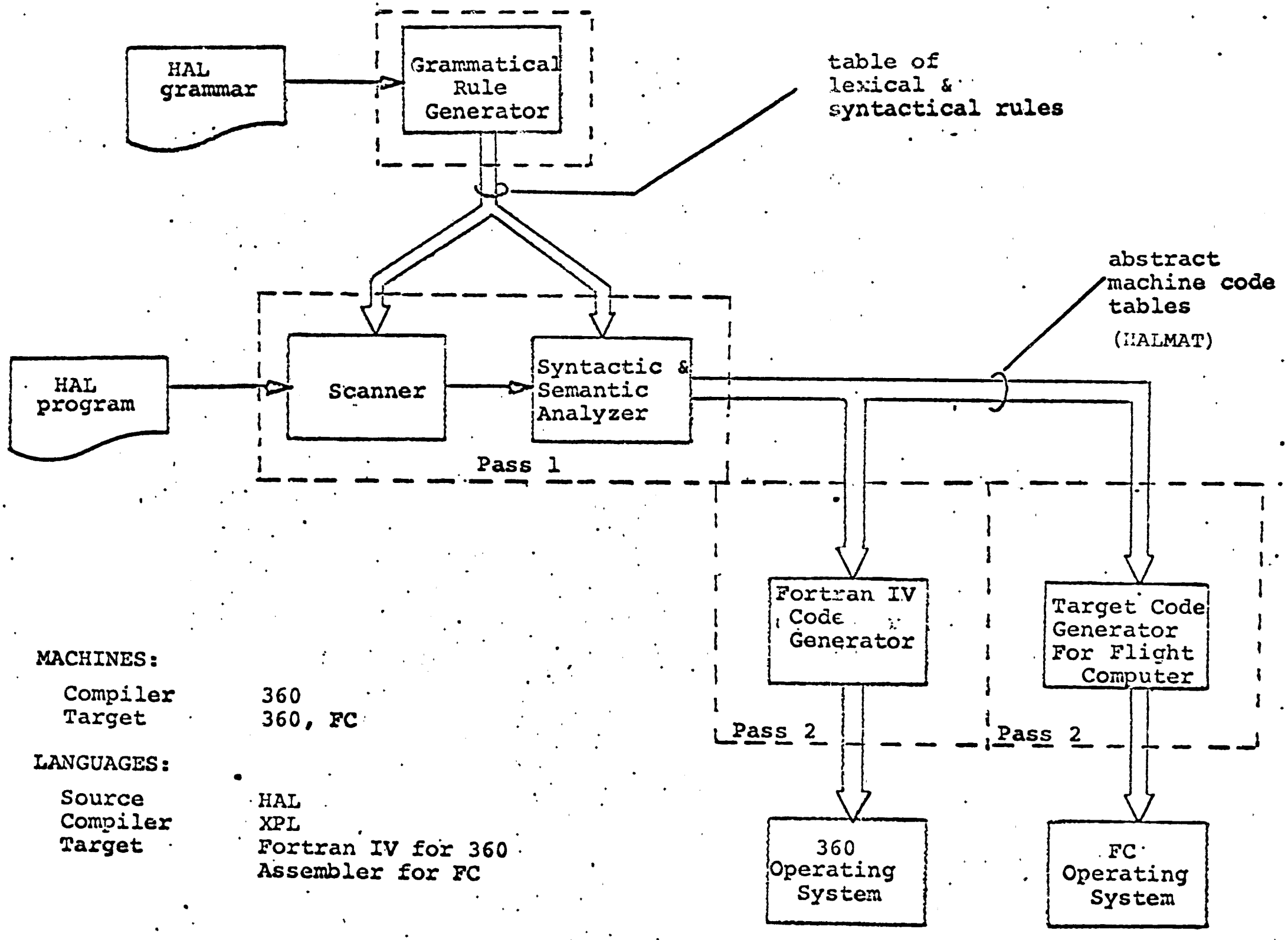
MACHINES:

Compiler	360/75
Target	360/75

LANGUAGES:

Source	HAL
Compiler	XPL
Target	Fortran IV

Fig. 4.2. CONSTRUCTION OF THE HAL COMPILER SYSTEM



MACHINES:

Compiler	360
Target	360, FC

LANGUAGES:

Source	HAL
Compiler	XPL
Target	Fortran IV for 360 Assembler for FC

Fig. 4.3. PROPOSED CONSTRUCTION OF HAL COMPILER FOR FLIGHT COMPUTERS

that FORTRAN is being used. Because it is possible to devise techniques to implement HAL in FORTRAN does not mean that they are closely related. HAL and FORTRAN are no more similar than FORTRAN and PL/1. FORTRAN is merely a tool or vehicle that is used to produce the final machine instructions. There is no more reason a priori to suspect that FORTRAN is qualified as a replacement language because it is used in the translation process than to suggest that assembly language could replace FORTRAN because many implementations convert FORTRAN to assembly language.

FORTRAN was rejected as the space programming language early in our work. FORTRAN just does not offer the desired features: clarity, readability, etc. HAL was designed, specifically, to meet the objectives of advanced manned spacecraft missions. To this end HAL includes:

- specific features to enhance readability
- complete vector-matrix mathematics
- ability to organize and manipulate large arrays and structures of data
- full bit and character capability in order to handle streams of binary data and facilitate communications
- sophisticated real time control statements
- advanced features to increase reliability; i.e. COMPOOL, name scope, control of shared data

FORTRAN could be extended to incorporate some of these features, but the result then would hardly be FORTRAN - especially if the readability issue were tackled.

At this point, an evaluation of the use of FORTRAN as an intermediate language would rate it a satisfactory decision. It has turned out more difficult than expected - both in creating a parenthetical language from the HALMAT code and in designing a FORTRAN system that will execute the HAL operations. Assembly language would have offered a more direct path. But on balance, it is felt that it was worth it. Future developments will hopefully bear this out.

REFERENCES

1. Requirements Analysis for a Manned Spaceflight Programming Language and Computer, MSC-01845, August, 1970, Intermetrics, Inc., Cambridge, Mass.

2. The Programming Language HAL - A Specification, MSC-01846, June 1971, Intermetrics, Inc.

3. HALMAT, An Intermediate Language of the First HAL Compiler, MSC-01847, June 1971, Intermetrics, Inc.

4. J.S. Miller, et al, Multiprocessor Computer System Study, Final Report, NAS 9-9763, Intermetrics, Inc., Cambridge, Mass., March 1970.

5. A.L. Kosmala, et al, Engineering Study for a Mass Memory System for Advanced Spacecrafts, NASA 9-9763, Intermetrics, Inc., August 1970.

6. W.M. McKeeman, "Language Directed Computer Design", FJCC, 1967.

7. C.V. Ramamoorthy and M. Tsuchiya, "A Study of User-Microprogrammable Computers", SJCC, 1970.

8. W.M. McKeeman, J.J. Horning, and D.B. Wortman, A Compiler Generator, Prentice-Hall, 1970.

9. Progress Review, Space Station Program: Phase B Definition, Document PDS70-1212, North American Rockwell, February 19, 1970.

10. J. Feldman and D. Gries, "Translator Writing Systems", CACM (11, 2), February 1968.

11. B. Randall and L. Russell, ALGOL 60 Implementation, Academic Press, Inc., 1964.

12. R.J. Rubey, W.C. Nielsen, and L. Bentley, Flight Computer and Language Processor Study, NAS 12-2005, LOGICON, INC., San Pedro, California, July 1969.

13. L. Carey, A. Kroger, and R. Nimensky, Space Programming Language (SPL/J6) Programmer's Manual, AF-FO-4701-68-C-0135, System Development Corporation, SAMSO-TR-68-383, Santa Monica, California, November 1968.

14. IBM System/360 PL/I Reference Manual, Form C28-8201-1, March 1968, IBM United Kingdom Laboratories Ltd., Hursley Park, Winchester, Hampshire, England.

15. J. Sammet, Programming Languages: History and Fundamentals, Prentice-Hall, 1969.

16. S. Rosen, Programming Systems and Languages, McGraw-Hill, 1967.

17. N. Wirth and H. Weber, "EULER: A Generalization of ALGOL and its Formal Definition", Communications of the ACM, January and February, 1966.

18. T. Cheatham, et al, "On the Basis for ELF-An Extensible Language Facility", NAS 12-563, Mass. Computer Associates, Wakefield, Mass., FJCC, 1968.

19. F. Corbato, "PL/I as a Tool for System Programming", DATAMATION, May 1969.

20. M. Richards, "BCPL: A Tool for Compiler Writing and System Programming", SJCC, 1969.

21. F. Corbato and J. Saltzer, "Some Considerations of Supervisor Design for Multiplexed Computer Systems", MIT Project MAC, Memo MAC-M-372, May 1968.

22. F. Corbato, "Sensitive Issues in the Design of Multi-Use Systems", MIT Project MAC, Memo MAC-M-383, December 12, 1968.

23. R. Freiburghouse, "The Multics PL/I Compiler", General Electric, SJCC, 1969.

24. C. Lang, "SAL-Systems Assembly Languages", SJCC, 1969.

25. N. Wirth, "PL 360", Journal of ACM, Vol. 15, No. 1, January 1968.

26. J. Donovan, "Digital Computer Programming Systems", MIT, 1970.

27. MIT Instrumentation Laboratory, Users Guide to MAC-360, September 1967.

28. D. E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley, 1968.

29. D. W. Barron, Recursive Techniques in Programming, American Elsevier, 1968.

30. E.W. Dijkstra, "The Structure of the 'THE' Multiprogramming System", CACM 11, 5., May 1968, pp. 341-346.

31. B.W. Lampson, "Dynamic Protection Structures", Proceedings, Fall Joint Computer Conference, Vol. 35, 1969, pp. 27-38.

32. B. Higman, A Comparative Study of Programming Languages, American Elseview, 1967.

33. E.W. Dijkstra, A Primer of ALGOL 60 Programming, Academic Press, 1962.

34. A.D. Falkoff and K.E. Iverson, APL/360: User's Manual, IBM 1968.

35. P. Wegner, Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, 1968.

36. M. Elson and S. Rake, "Code-generation Technique for Large-Language Compilers", IBM System Journal, No. 3, 1970.

37. D. Knuth, "An Empirical Study of Fortran Programs", Computer Science Department Report No. CS-186, Stanford University, 1970.

38. H. Lawson, Jr., "Programming-Language-Oriented Instruction Streams", IEEE Transactions Computers, Vol. C-17, No. 5, May 1968.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840