

E-2728
Revision 1
TOP-DOWN, BOTTOM-UP
STRUCTURED PROGRAMMING AND
PROGRAM STRUCTURING
by
M. Hamilton, S. Zeldin
December 1972

(NASA-CR-128971) TOP DOWN, BOTTOM UP N73-25211
STRUCTURED PROGRAMMING AND PROGRAM
STRUCTURING (Massachusetts Inst. of Tech.)
CSSL 09B Unclas
G3/08 06858

PRICES SUBJECT TO CHANGE

**CHARLES STARK DRAPER
LABORATORY**

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE, MASSACHUSETTS, 02139

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

N O T I C E

THIS DOCUMENT HAS BEEN REPRODUCED FROM THE BEST COPY FURNISHED US BY THE SPONSORING AGENCY. ALTHOUGH IT IS RECOGNIZED THAT CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED IN THE INTEREST OF MAKING AVAILABLE AS MUCH INFORMATION AS POSSIBLE.

E-2728

Revision 1

TOP-DOWN, BOTTOM-UP
STRUCTURED PROGRAMMING AND
PROGRAM STRUCTURING

by

M. Hamilton, S. Zeldin

December 1972

Approved: R. H. Battin Date: 12/21/72
R. H. BATTIN, DIRECTOR, MISSION DEVELOPMENT
APOLLO GUIDANCE AND NAVIGATION PROGRAM

Approved: Ralph A. Ragan for Date: 21 Dec 72
D. G. HOAG, DIRECTOR
APOLLO GUIDANCE AND NAVIGATION PROGRAM

Approved: Ralph A. Ragan Date: 21 Dec 72
R. R. RAGAN, DEPUTY DIRECTOR
CHARLES STARK DRAPER LABORATORY

The Charles Stark Draper Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ACKNOWLEDGMENT

This report was prepared under DSR Project 55-23890, sponsored by the Manned Spacecraft Center of the National Aeronautics and Space Administration through Contract NAS 9-4065. This study is authorized by the NASA/MSC Task Review Integration Panel (TRIP) on Task 26-S.

The authors would like to express appreciation to P. Rye and P. Adler for the helpful criticisms given for this revision of the report.

The publication of this report does not constitute approval by the National Aeronautics and Space Administration of the findings or the conclusions contained herein. It is published only for the exchange and stimulation of ideas.

TOP-DOWN, BOTTOM-UP, STRUCTURED PROGRAMMING AND PROGRAM STRUCTURING

Abstract

The purpose of this report is to inform engineers, programmers and managers of new design and programming techniques for Shuttle software. Based on previous APOLLO experience, recommendations are made to apply top-down structured programming techniques to Shuttle software. New software verification techniques for large software systems are recommended. HAL, the higher order language selected for the Shuttle flight code, is discussed and found to be adequate for implementing these techniques. Recommendations are made to apply the workable combination of top-down, bottom-up methods in the management of Shuttle software. Program structuring is discussed relevant to both programming and management techniques.

by: M. Hamilton
S. Zeldin

December 1972

CONTENTS

<u>Section</u>		<u>Page</u>
1.0	Introduction	1
2.0	Software Development Techniques for APOLLO	2
3.0	New Software Programming Techniques for Shuttle	3
3.1	The Top-down Concept	4
3.2	A Case for Structured Programming	8
3.3	Using HAL Effectively	18
3.4	Structured Flow Charts	21
4.0	Program Structuring	28
5.0	Software Management Techniques	31
5.1	Shuttle Software Parallel Efforts	33
5.1.1	Tool Development	36
5.1.2	Bottom-up Off-line Module Building	36
5.1.3	Official Top-down Building Process	37
5.2	Pseudo Modules	39
6.0	Preliminary Structured Programming Rules for Designing Algorithmic Shuttle Software	41
7.0	Conclusion	42
	REFERENCES	45
APPENDIX 1	Application of Top-Down Structured Programming to Shuttle Algorithms—A First Attempt at Structured Flow Charting	47
APPENDIX 2		81

ILLUSTRATIONS

<u>Figure</u>		<u>Page</u>
1	A Representation of a Top-down Moon Program	6
2	The Data Module	9
3	Structured Program Consisting of Sequential Blocks of Code	11
4	An Example of an Efficient Program Written Before the Concepts of Structured Programming Were Developed	13
5	IF ? THEN X1 ELSE X2	14
6	DO CASE L: case L of (X1; X2; ...Xn)	14
7	Repetitive HOL Statements	15
8	The Structured Nature of the AGC Executive	17
9	The Basic Unit of a Structured Flow Chart	22
10	Decision Statements	24
11	The CALL and its Accompanying Data Module	25
12	Basic Data Module Symbols	26
13	Data Flow for CALL or LABEL	27
14	The Structured GO TO Symbol	26
15	The Structured RETURN Symbol	28
16	Ideal Top-down Philosophy	34
17	Shuttle Software Parallel Efforts	35
APPENDIX 1	Application of Top-down Structured Programming to Shuttle Algorithms—A First Attempt at Structured Flow Charting	47
1-1	ENTRY NAVIGATION FUNCTIONAL FLOW DIAGRAM	49
1-2	ENTRY_NAVIGATION DATA MODULE	51
1-3	ENTRY_NAVIGATION PROGRAM	55
1-4	INITIALIZE PROCEDURE DATA MODULE	57
1-5	INITIALIZE PROCEDURE	59
1-6	STATE_ESTIMATE_INIT DATA MODULE	62
1-7	STATE_ESTIMATE_INIT PROCEDURE	63
1-8	W_ESTIMATE_INIT DATA MODULE	65
1-9	W_ESTIMATE_INIT (LABEL)	66
1-10	USE_NAV_AIDS_TO_UPDATE_X_AND_W DATA MODULE	68

ILLUSTRATIONS (Cont.)

<u>Figure</u>		<u>Page</u>
1-11	USE NAV_AIDS_TO UPDATE_X_AND_W PROCEDURE	69
1-12	USE_DRAG_MEAS DATA MODULE	71
1-13	USE_DRAG_MEAS PROCEDURE	72
1-14	ELEV_MEAS DATA MODULE	74
1-15	ELEV_MEAS PROCEDURE	75
1-16	RANGE_MEAS DATA MODULE	76
1-17	RANGE_MEAS PROCEDURE	76
1-18	AZIMUTH_MEAS DATA MODULE	77
1-19	AZIMUTH_MEAS PROCEDURE (K_{AZ} , \bar{i}_D , \bar{i}_C)	77
1-20	UPDATE_X_AND_W_DATA MODULE	78
1-21	UPDATE_X_AND_W PROCEDURE	79
APPENDIX 2		81
2-1	Entry Navigation Functional Flow Diagram	82
2-2a-h	Entry, Approach and Landing Navigation, Detailed Flow Diagram	83-90

1.0 Introduction

The concepts "top-down", "bottom-up", "structured programming", and "program structuring" require some clarification. An attempt is made to define these terms and then, using these concepts, to define an approach to flight software development for the future.

Since 1968, these terms have appeared in the literature as an attempt to communicate the need for an organized approach to creating software^{1,2,3}.

We will use the same terms, but will show how they apply in the context of building flight software. We are considering the two main areas in the development of Shuttle flight software: (1) the design of the contents of the software itself, i.e., programming style and (2) the management techniques which include the layout of the software system and the process of building the software system.

Structured programming - the process of enforcing organization and discipline in the programming process. Modularity is a main concern. Modularity includes the organization of sets of instructions and, in addition, the organization of data. Program blocks are arranged sequentially so that the flow of a program is visible. (An ideal approach is to eliminate the concept of GOTO s.)

Top-down - an organizational process whereby steps are taken in the following order: (1) the total concept is formulated, (2) the functional specification is designed, (3) the functional specification is refined at each intermediate step where the intermediate steps include code or processes required by the previous step and (4) the final refinements are made to completely define the problem.

Bottom-up - the reverse of top-down whereby: (1) subroutines of lower level modules are created first, (2) the intermediate steps integrate the lower level steps, and (3) the final step links all the previous steps together. The entire problem is not defined until the final step is completed.

Program structuring - the process of defining and enforcing organization and discipline in the total software system design and implementation. Whereas structured programming applies to a programming style, program structuring applies to the process of defining the modules and their interfaces.

2.0 Software Development Techniques for APOLLO

On APOLLO the techniques mentioned above evolved out of necessity. The difference was either that we didn't use these terms when applying these techniques, since they were not known by the new terminology, or that we didn't always formalize or enforce these techniques in as many areas as we now propose. Examples of how we approached these techniques are as follows:

- 1) Many rules were enforced on programmers by the assembler, the systems software, the digital simulator and the assembly control supervisor⁴. Such rules included standard coding techniques, standard interfaces, common subroutines, etc.
- 2) Programming responsibility was allocated so that systems experts were responsible for system program modules and applications experts were responsible for mission modules.
- 3) AGC programs were designed to be modular, e.g., P40 is the program to provide the SPS engine guidance logic; V82 is the extended verb routine to calculate and display orbital parameters. There were mission modules and functional modules. Some modules were control modules, some modules were subroutine modules, some were data management modules and some were data modules.
- 4) The asynchronous AGC executive allows for a flexible and modular input of mission programs and routines. Rules for program interfaces were defined by the executive. Some software was synchronous and some was asynchronous. Some

was in the form of TASKs (time oriented), and some in the form of JOBS (priority oriented). All of these divisions blocked software into sections.

- 5) The error recovery software was divided into specific areas. These areas influenced the entire software structure both from a memory layout and from a multiprogramming point of view.
- 6) "Higher level" core software such as the display interface routines forced a standardization of techniques and prevented the programmer from using complicated or error-prone code.
- 7) Erasable memory was divided into sections. This division was dictated partly by the hardware and partly by the software.

Obviously, we were approaching what we now think of as modularity and enforcement of rules (structured programming). However, depending on programmer/supervisor preference, some programmers programmed top-down; some bottom-up; some a combination of the two. In addition, we were very much concerned with program structuring. The management approached top-down methods of developing an official assembly in later stages of software development.

3.0 New Software Programming Techniques for Shuttle

We have been talking for several months about applying "structured programming" and "top-down" techniques to software for the Shuttle. The application of more reliable techniques to the software development process will eventually result in a more efficient cost-effective process.

The way programmers/engineers attempt to solve a problem turns out to be very much dependent on the programming tools available. The tendency to use an inadequate higher order language (HOL) is reflected in popularized treatments of programming problems⁵. This conventional approach separates the engineering from the programming, that is, the engineering designs from the software designs. Since these techniques do

not automatically enforce programming rules and do not have all the appropriate language constructs to produce an efficient structured program, they invariably introduce errors and produce an inefficient and expensive verification process.

There are three basic principles to acquiring a top-down-structured program. First, the program should be designed and implemented by top-down methods. The second principle is to plan the software in a structured manner. This requires rules and enforcement of these rules on the part of everyone involved. The third principle is to program in a HOL which (1) enforces structured programming rules, (2) contains static and dynamic debug features, and (3) automates designs in the software development process. The HAL programming language selected for the shuttle flight software has features that are necessary to construct a structured program.

The formal process of structured programming and the enforcement of it is a relatively new thing for all of us. We are now attempting to use HAL in a structured way to implement the algorithmic software for the shuttle. (See Appendix 1.) Preliminary and tentative rules for Shuttle software implementation are suggested in Section 6.0.

3.1 The Top-Down Concept

The concept of top-down can be thought of as planning each level of the program and each level of the accompanying data modules from top to bottom, completely.

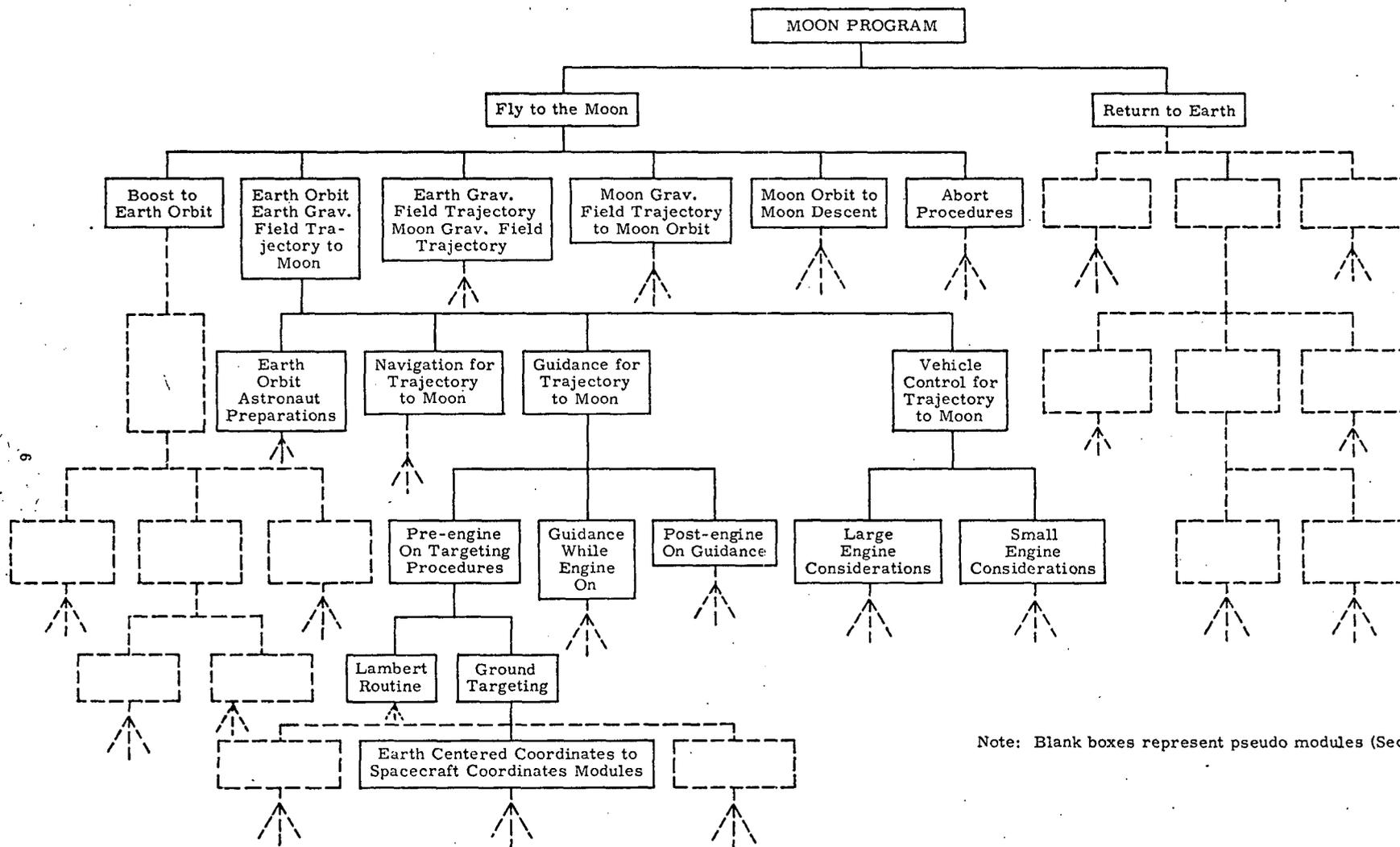
When writing a paper or preparing a talk, one first jots down notes. Then an outline is developed. After the outline is expanded by way of a few iterations, the paper is rewritten. Many revisions are usually necessary if the paper or speech is of any significance. A software program shouldn't be much different in the way it is created. Better organized papers and

speeches are, of course, much easier to follow and understand than a paper or speech that rambles back and forth. The same holds true for an individual program; even more so for a whole software system. However, an iterative process is necessary, just as when writing a paper. Each iteration in the design of the software system will bring the definition of each level closer to the best modular, top-to-bottom concept for the particular system in mind. During the design process, it will become apparent that some modules on one level will be applicable for use on another level; some data modules must be accessible to more than one program module; and, in some instances, it will be important that some data modules be inaccessible to particular program modules.

Top-down design is analagous to a tree where each level of planning is another branch. Each branch in turn can be a node with branches of its own. But, if one chooses any point at an outermost branch, it is always possible to retrace the growth of the tree and follow each mode back to the original trunk. Likewise, one never loses sight of the original problem by designing a software system top-down. Each level of planning should be broken into parts which take no more than one page of description. For example, if a program were Apollo, the top-level instructions would be, (Fig. 1),

- (1) "Fly to the moon."
- (2) "Return to earth."

The next level would be 2 pages. One would break down the trip to the moon and would include instructions, such as CALL BOOST, CALL EARTH ORBIT, etc. The other page would describe the return to earth. The next level would show a page for BOOST, a page for EARTH ORBIT, etc. The page for EARTH ORBIT would include instructions such as CALL NAVIGATION, CALL GUIDANCE, etc. A much lower level would break down instructions for routines such as Lambert, matrix coordinate transformations, etc.



Note: Blank boxes represent pseudo modules (Section 5.2)

Fig. 1. A representation of a top-down moon program.

At each level of planning the data modules are planned just as carefully as the program modules. Top-down data module design will indicate the data on the outer levels that must be available to the inner, or lower, levels; and at the same time will indicate those data modules that require no interfaces. The "scope" of the data is therefore an important concept. One of the niceties included in the Shuttle language, HAL, is the concept of scope. All data declared on outer levels are known to each inner level automatically in HAL, while each inner level variable is unknown to any outer levels.

Top-down design will also indicate those data modules where conflicts will arise when reading or writing into these modules. For example, a targetting routine may wish to READ the present state vector. If the integration routine has just updated the velocity of the spacecraft but has not updated the position of the spacecraft, precautionary measures must be taken in the design of this state vector data module so that the targetting routine will wait until the complete data module is updated. Again this automated design feature is available in the shuttle language, HAL.

Top-down design must include complete test specifications for each level of the functional and data modules of the program. Each engineer responsible for a particular shuttle algorithm should complete the set of test specifications before coding is begun. Once the engineer/programmer has proved the correctness of the program module by structured programming techniques, the code is tested completely to verify that the specifications have been coded correctly. A test specification will include:

- 1) various sets of initial conditions
- 2) figures of merit
- 3) timing and priority requirements
- 4) interface requirements, and
- 5) indicators as to which specification is being tested.

The test specification should be designed top-down. This will avoid repetitive testing of the same specification. Each test case must be meaningful; the complete set of test specifications should adequately test both the constraints imposed on each functional specification and each data specification, while at the same time the test specifications should not produce unnecessary test cases.

Top-down techniques can only work effectively if combined with structured programming techniques. It is important to realize that HAL must be used in a structured way if we are to make intelligent use of a powerful software tool. HAL is characterized by its ability to link the broadly based engineering solutions, which view the problem as a whole, to an effective programming approach, which places a high premium on correctness and creates real ease of modification. Proper use of the HAL concepts of modularity, scope, effective subroutine constructs, and error recovery techniques can only occur if the entire problem and all of its ramifications are understood by the programmer. Thus, the engineers who develop the guidance, navigation and control algorithms must take an active role in the development of Shuttle software.

3.2 A Case for Structured Programming

In the past, a programmer's objective was to generate code as efficiently as possible; there was not enough concern for those people who had to understand, modify, and many times debug a program long after the original programmer had disappeared from the programming effort. At present, it has not been possible to be 100% sure that there are no errors in actively used software. The task of proving any program correct by conventional means is expensive and is not guaranteed to be reliable. This is mainly due to the older techniques of generating and testing a program. The older method invariably shows the presence of errors in the program, but in fact, there is no way that testing can detect the absence of errors¹.

New concepts in programming style make it possible to attempt to prove a program is correct. The technique involves two basic steps. The first is to prove certain HOL constructs to be correct. The second is to allow the programmer the luxury of using these constructs in the same manner that one uses a mathematical theorem as a building block. By concatenating these building blocks, a simple sequence for a structured program develops. Not only can the entire sequence be proved correct, but the modularity of each building block anticipates future program modification.

The modular building process encompasses the two basic modular types: data modules and program modules.

Data modules are structured by 1) the individual programmer when defining local variables, 2) system design programmers when defining non-local variables, or 3) by the HOL automatically.

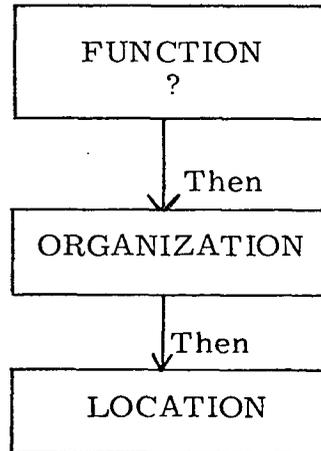


Fig. 2 The data module

The function of a data module is determined by program constraints, timing constraints, data interface constraints, error recovery constraints or I/O constraints. The organization of a data module is derived from

its function and specifies the data type (bit, integer, vector, matrix, character, scalar) and the array and structural qualities. The size, specified in the organization, and the function make it possible to define a block for the data module. The location for each block can be a location such as the COMPOOL, can be specified as outer level, or can be specified as local to a particular program. In addition, this location can be an absolute location such as a sensor known to the program via a particular I/O device.

Those data modules that are a part of the HOL can be assumed correct. In HAL, these built-in data-modules include matrices and vectors to provide programming of matrix-vector arithmetic and the array to provide modular arithmetic operations.

When constructing a structured program, the local data modules must be clearly distinguished from the non-local data modules. To aid the programmer in proving non-local data modules correct, the Shuttle language, HAL, provides a special program block, the UPDATE block. Reading or writing of shared data modules can be done inside an UPDATE block and the user at any program level is assured that the total data module at any point is correct and that the contents of that data module will not be the cause of any dynamic error.

A program module can be an open block, which is in-line (the IF construct) or a closed block (the PROCEDURE). A module is characterized by the particular function it performs. It has a single entrance, i.e., the single entry point of a procedure or the first statement of an in-line block. It also has a single exit in the sense of returning to the same place from which it was invoked, i.e., procedures return normally to the place from which they were called and all in-line blocks exit only to the statement immediately following the block⁶.

The one entrance, one exit structure of module linkage assures the programmer that the state of the program is always defined. That is, at any point of execution a simple dump reveals the current state of the program in terms of the set of active modules and their calling relationships. In the event a dynamic error does occur, the simple sequenced structure makes it possible to identify the error using the building blocks as coordinates of the program.

It has been shown that HOL statements involving concatenation, selection or repetition can be proven correct by the methods of enumeration, mathematical induction, and abstraction¹. Dynamic errors could still occur in a program proven "correct" because a structured programming theorem incorrectly assumes that (1) the numerical analysis for each problem is correct, (there are no scaling or accuracy problems to solve), (2) the specifications are coded correctly, (3) the HOL is correct, (4) the verification tools are correct, (5) the hardware is correct, (6) the systems software is written in the HOL and (7) non-local data modules have no local effects.

The simplest construct to prove correct is a group of sequential statements or blocks of code. These are nothing more than an ordered list in which all possibilities are clearly visible by simply following the code.

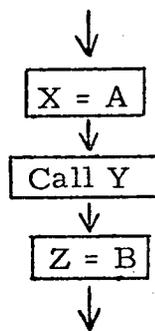


Fig. 3. Structured Program Consisting of Sequential Blocks of Code.

A programming style which advocates the use of branching (GOTO) for the main purpose of producing efficient code would produce a program difficult to understand, modify, debug, or prove correct. One could not simply follow the code. In this context, Fig.4 is presented as a representative example of a FORTRAN program written before the concepts of structured programming were made available.* This program is so "tricky" that it branches into the list of GOTO statements by fooling a particular FORTRAN compiler which assigns the address of the succeeding statement to TRUE. TRUE in this compiler is a variable, not a constant. Indexing into the list of GOTOs is accomplished in a BAL program not shown here, by using this bit of TRUE information. The code is extremely efficient, but obviously difficult to debug or modify.

We already know that 5% of official anomaly errors in Apollo flight software arose from simple branching errors⁷. Analysis of complicated branching errors is not yet complete. The available data does not account for the many wasted hours spent by the original programmers in detecting branching errors during Apollo development. It has been shown by others that statements with branching operations are 5 times more error-prone than statements without branching⁸.

If a HOL contains the proper control statements, it is theoretically possible to construct an efficient program without GOTOs⁹. These control statements are:

```
IFTHENELSE
DOFOR
DOWHILE
DOCASE
```

* This example was suggested by Saul Serben.

```

31010 FRTR001  FILLIST,REF,DECK,M94,XK7,NODICT
SUBROUTINE FRTRCH ( XERROR )
COMMON /STATES/ S(13),SA(13),SB(13),SC(13),SD(13),SE(13),SF(13),
1 SG(13),SH(13),SJ(13),SK(13),SL(13),SM(13),SN(13),SO(13),SP(13),
2 SQ(13),SR(13),SS(13)
DOUBLE PRECISION S,SA,SB,SC,SD,SE,SF,SG,SH,SJ,SK,SL,SM,SN,SO,
1 SP,SQ,SR,SS
COMMON /MUSRMU/ GM(4)
DOUBLE PRECISION GM
COMMON /XMMH/ B(20),BS(20),BWT(20),Q(20),QMIN(20),QMAX(20),
1 QW(20),QWCUT(20),NAMES(4,20),PART(20),PARTAL
DOUBLE PRECISION B,Q,QMIN,QMAX,QW,QWCUT
COMMON /XINPUT/ F(500)
DOUBLE PRECISION F
COMMON /INTCON/ M1(6),M2(6),M3(6),M4(6),M5(6),M6(6),M7(6),M8(6),
1 M9(6),M10(6),M11(6),F1(4),F2(4),F3(4),F4(4),F5(4),F6(4),F7(4),
2 F8(4),F9(4),F10(4),F11(4)
COMMON /VARBLE/ X(40),Y(40),XSCALE(40),YSCALE(40),E(40)
DOUBLE PRECISION X,Y,E,CONST
COMMON /XSETUP/Z(10),MX(40),LGO(20),MSTART
DOUBLE PRECISION Z
DOUBLE PRECISION TRX(40),TRY(40),BF(8),DTEMP,DP(19),DGAMMA
EQUIVALENCE (DTEMP,E),(DP,E(3))
EQUIVALENCE (F(141),TRY),(F(341),TRX)
DIMENSION BA(8),BB(8),BC(8),BD(8),BE(8)
DOUBLE PRECISION BA,BB,BC,BD,BE
EQUIVALENCE (F(60),BA),(F(68),BB),(F(76),BC),(F(84),BD),(F(92),BE)
EQUIVALENCE (F(49),SITE)
LOGICAL MGO, LGO
COMMON /THRUST/ TH(18)
DOUBLE PRECISION TH
DIMENSION THR(36)
EQUIVALENCE (TH,THR)
DATA BF / -6*0.000, 1.000, 0.000 /
LGO(1) = .TRUE.
GO TO 25
GO TO 1
GO TO 101
GO TO 108
GO TO 112
GO TO 126
GO TO 130
GO TO 145
GO TO 160
GO TO 175
GO TO 200
GO TO 220
GO TO 226
GO TO 228
GO TO 240
GO TO 250
GO TO 259
GO TO 260
GO TO 275
GO TO 280
GO TO 290
GO TO 300
GO TO 305
GO TO 315
GO TO 320
FRTR0010
FRTR0020
FRTR0030
FRTR0040
FRTR0050
FRTR0060
FRTR0070
FRTR0080
FRTR0090
FRTR0100
FRTR0110
FRTR0120
FRTR0130
FRTR0140
FRTR0150
FRTR0160
FRTR0170
FRTR0180
FRTR0190
FRTR0200
FRTR0210
FRTR0220
FRTR0230
FRTR0240
FRTR0250
FRTR0260
FRTR0270
FRTR0280
FRTR0290
FRTR0300
FRTR0310
FRTR0320
FRTR0330
FRTR0340
FRTR0350
FRTR0360
FRTR0370
FRTR0380
FRTR0390
FRTR0400
FRTR0410
FRTR0420
FRTR0430
FRTR0440
FRTR0450
FRTR0460
FRTR0470
FRTR0480
FRTR0490
FRTR0500
FRTR0510
FRTR0520
FRTR0530
FRTR0540
FRTR0550
FRTR0560
FRTR0570
FRTR0580
FRTR0590

```

13

Fig. 4 An Example of an Efficient Program Written Before the Concepts of Structured Programming Were Developed.

It is also true that these control statements can be proved correct from a logical point of view¹.

The IFTHENELSE provides a simple choice between two possibilities. Since this statement is always entered at the beginning and has a single EXIT, the entire construct can be thought of as a single module whose internal structure is not relevant to the context in which it is used.

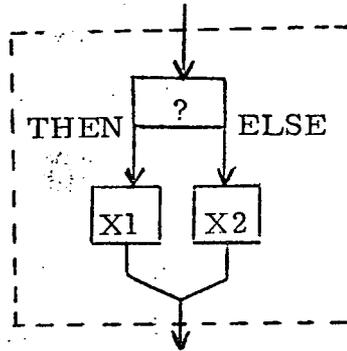


Fig. 5. IF? THEN X1 ELSE X2

The DO CASE construct is just a selection process easily proven correct by enumerative reasoning and, again, has the characteristic of a single entry and exit.

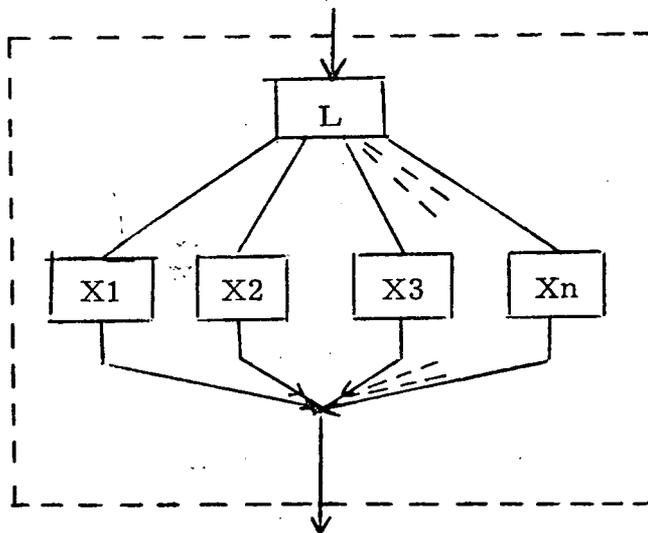
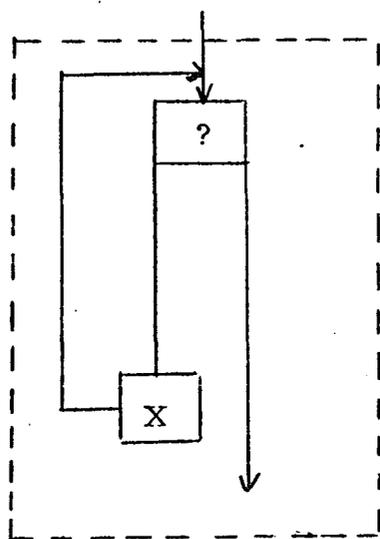
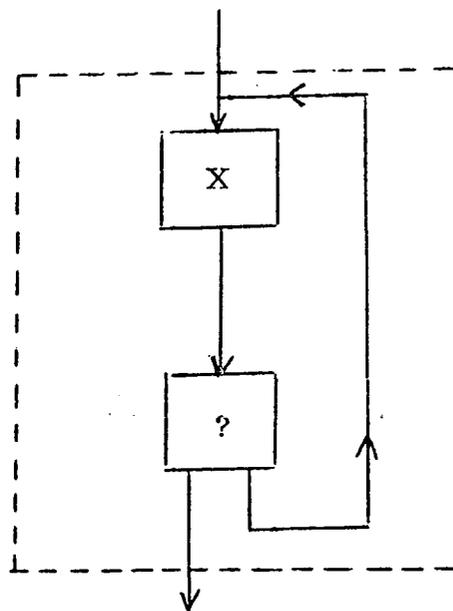


Fig. 6. DO CASE L: case L of (X1; X2;... Xn)

Repetition via DOWHILE or DOFOR statements (Fig. 7a) and REPEAT (Fig. 7b) can be proved correct by mathematical induction. Again, we have a control statement with a single entry and exit. As long as we enforce the rule that the loop variables of a DOWHILE or DOFOR must be a local variable (changed only within the realm of the DOWHILE or the DOFOR) the loop will not depend on an outside variable and the proof for correctness will not be difficult. Imagine how difficult it would be to prove programming logic correct where a GOTO depends on a non-local variable. In this case, branching via a GOTO could lead to an infinite repetition of a particular set of statements.



(a) DO WHILE or DO FOR:
while ? do X



(b) REPEAT: repeat X until ?

Fig. 7. Repetitive HOL Statements

At present it is not clear how much coding could be saved if HAL included the REPEAT statement. When a specification requires the REPEAT concept, methods of setting extra flags, creating extra procedures, creating

duplicate copies of the same block of code, and resetting loop variables are available.*

It is not clear that programming languages should completely eliminate the GOTO. Some languages lack the control statements mentioned above as necessary to produce an efficient structured program. In those cases, the GOTO must be accepted as a basic building block¹¹. Although HAL contains the necessary control statements, the GOTO cannot be eliminated from the language. Specifically, this syntax is incorporated into the error recovery statement "ON ERROR GOTO X". For this particular application of GOTO, there is no alternative since this HAL statement will be extremely useful in considering software restart protection and for flagging out expected abnormal termination. It is also not clear that using the GOTO for error recovery techniques violates any rules of structured programming. The ON statement may be thought of as user defined systems programming or special instructions to the compiler for dynamic monitoring of particular events. Another example of systems programming is the flight program executive. It is apparent that the concept of the AGC asynchronous executive actually falls into the structured programming framework as seen in Fig. 8.**

In fact, one of the few examples of a reasonably sized program that used the principles of structured programming is an experimental multi-programming operating system which now supports 5 or 6 concurrent users on a small computer¹². It purports brief development time and structure that was actually a tool that provided insight into the problem.

* The "Unified Filter" shuttle navigation algorithm contains more than one specification in the form of a REPEAT¹⁰. Further study into "work-around" procedures due to the lack of a REPEAT is required.

** Suggested by Phyllis Rye in an attempt to reconcile real-time flight systems operation with the structured concept.

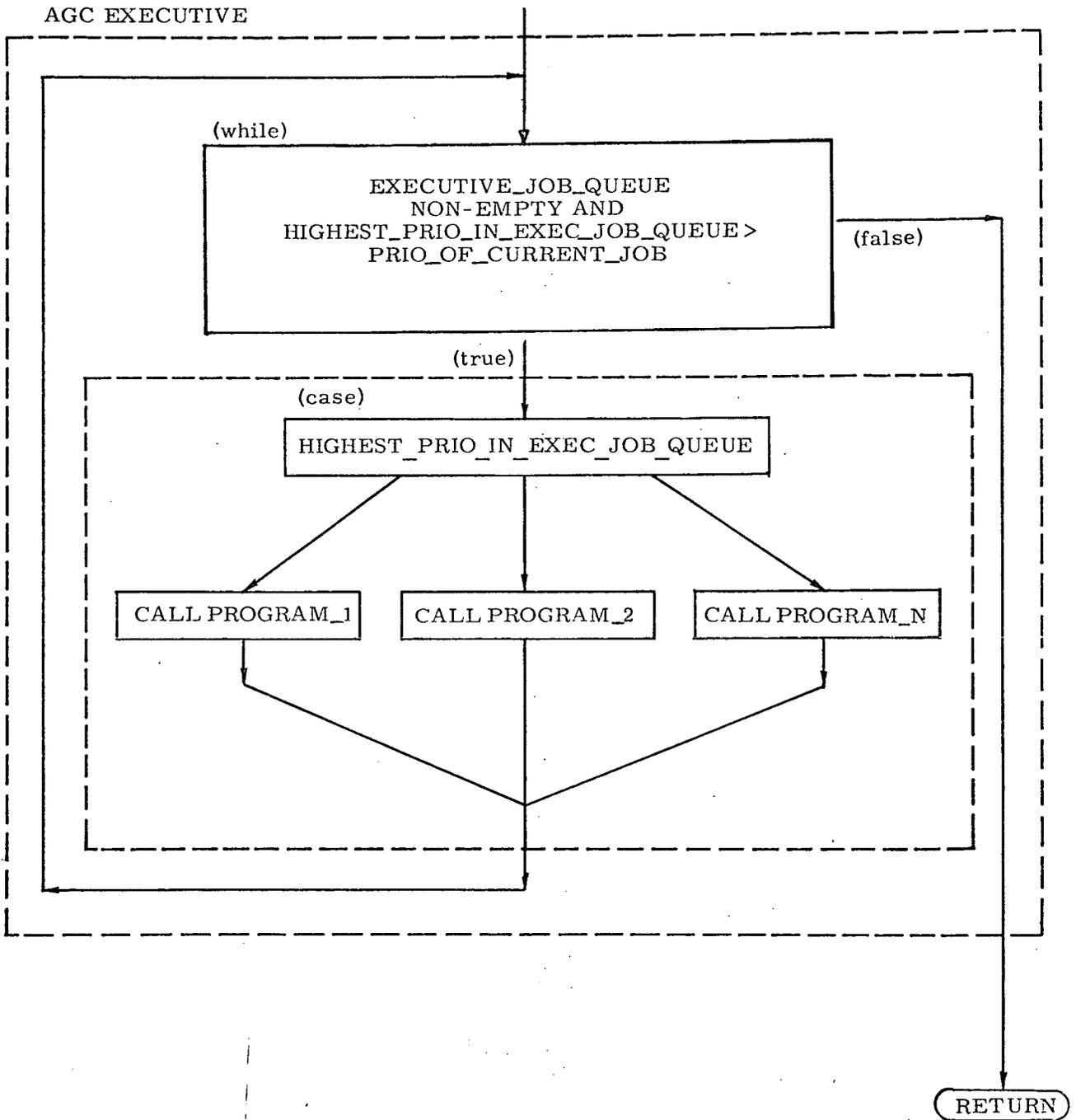


Fig. 8. The structured nature of the AGC executive.

Using structured programming techniques means that more time will be spent on software design at the initial stages of development. Ultimately, this should save time during the verification process. Many times, the method used to define a problem is to make a few examples (perhaps even doing a few hand calculations) until it is possible to form an abstraction of the problem. This abstraction becomes the basis for defining the total functional specification of the problem. After this mental process has been completed, the programmer is ready to design software using top-down structured programming techniques. This is an ideal situation. The real world of Shuttle software will have software development schedules that might tempt a programmer to use a GOTO as a quick and easy solution to a problem. When modifying a program, it may be difficult to avoid a GOTO without recoding a lot of logic. Potential interface problems exist for Shuttle systems programming since systems programs cannot be coded in the HAL language in its present form.

3.3 Using HAL Effectively

Software problems can be dealt with by either hardware tools or software tools¹³. For example, computational problems could have been reduced by a software floating point mechanism in HAL but a more efficient and less costly way of doing this is the provision for floating point mechanism in the hardware. Twenty two percent of pre-flight anomalies for Apollo occurred due to computational difficulties⁷. The analysis is yet incomplete in that (1) the proportion of these anomalies due to scaling difficulties is unknown and (2) the proportion of programmer time spent in debugging scaling programs is unknown. But this data indicates that the floating point hardware that will be part of the chosen flight computer will certainly save programmer debug time and save much code generation.*

* In a similar manner, the flight hardware design should not cause large overhead on subroutine mechanisms. For subroutines are a very important part of new programming styles. Likewise, computers for the future should have desirable test-cooperative features for dynamic verification.

Many software problems can be eliminated by appropriate HOL features. For example, the most time consuming aspect of Apollo verification was the verification of data modules. In fact, 24 per cent of pre-flight anomalies for Apollo occurred due to shared data modules⁷. For Shuttle, all data modules shared by many programs must be placed in a common area (COMPOOL) where access rights of various programs to data modules are specified and where special blocks are designated in HAL to prevent reading or writing of shared data during data module updates. Data modules shared within a program have scope in that all outer level variables are known to inner levels. The fact that inner level variables are not known to outer level variables makes it possible to define local variables with ease and automatically assures the programmer that local variables will not destroy other data modules.

Many features in HAL support structured programming concepts and actually enforce certain standards of programming style. Some features are optional. It is these optional features that a programmer must be aware of and use properly so that HAL can be used for the purpose for which it was intended — generating reliable shuttle software¹⁴.

Listed below are a set of HAL language features that automatically produce modularity and reliability in flight software.

- . Single entry point for programs, procedures and functions to comply with structured programming techniques.
- . Each subroutine has a single exit to comply with structured programming techniques.
- . Automatic checking for compatible dimensions and data types of parameters is performed at each subroutine interface to aid in program structuring verification.
- . HAL is a block structured language. PROCEDURES, FUNCTIONS, TASKS, PROGRAMS are blocks that can be designed to be used in constructing a structured program with structured data modules. Outer level variables are automatically known to inner levels unless otherwise specified. Local level variables are not known to outer levels. For

example, there are no equivalence or "common" problems as in FORTRAN.

- . Vectors and matrices are representative modular data types.
- . Arrays and structures represent modular data organization.
- . Implicit data type conversions avoid inconsistent data assignments.
- . Automatic storage allocation helps attain data modularity.
- . Automatic error recovery features exist for software restart techniques.

The following HAL features can produce modularity and reliability if used properly.

- . Control statements for structured programming

IFTHENELSE
DOWHILE
DOFOR
DOCASE

- . GOTOs restricted to local block to aid the programmer in producing a structured program.
- . The REPLACE statement may be useful for expanding functional specifications until the pseudo-module simulator is made available.
- . Array arithmetic for modularity.
- . Matrix-vector arithmetic for modularity.
- . Multi-dimensional arrays for visibility.
- . Matrix-vector notation in source listing for visibility.
- . Extensive explicit data type conversions to aid in data module manipulations.
- . Real-time syntax to schedule, sequence and terminate events and tasks for use in a realistic asynchronous multi-programmed environment.

- Provisions for special blocks to avoid real-time data sharing conflicts (UPDATE blocks).
- Re-entrant subroutines with the option to lock out more than one user to provide flexibility for a multi-programmed environment (EXCLUSIVE).

3.4 Structured Flow Charts

Shuttle software is to be characterized by a combination of two basic programming styles: structured programming and top-down techniques. The structured programming concept is characterized by an ordered set of program instructions. Accompanying structured data modules directly convey the flow of data. The use of top-down techniques results in program flow which can be compared to the organization of a book: the "table of contents" specifies the entire program flow on page one; each "chapter" is the expansion of a particular block. Conventional flow chart techniques can not adequately convey these organizations. The block structure, the scope, and the data flow inherent in any structured top-down program must be represented by a structured flow chart.

Structured flow charting is based on the premise that the functional flow of a program includes 1) decision statements based on program data 2) CALLs to sub-modules or other programs which manipulate data and 3) in-line equations, which can be thought of as language supplied CALLs or "degenerate" CALLs which manipulate data.

The functional representation of a program is the first page of the structured flow chart. The complete data module is represented on the second page. The functional program is the third page. The succeeding pages expand the modules found on page three. A complete data module should accompany each PROCEDURE, TASK, or functional block.

A complete data module, associated with a functional block, is defined as the set of data referenced and assigned within a module. This set includes data referenced or assigned within each sub-module and each CALL to an outside program. The complete data module is an inherent structure to any program module and should, therefore, accompany the functional flow of the module.

The intersection data module, associated with the CALL to a functional block, is defined as the intersection of the complete data module of the caller with the set described as the union of all complete data modules associated with the called sub-modules and programs. The intersection data module must accompany each CALL. For example, a PROGRAM - PROGRAM intersection includes all COMPOOL variables and sensors referenced in one program and assigned in the other program. Those data elements superfluous to a PROGRAM-PROGRAM intersection include constants and local variables. A PROGRAM - PROCEDURE intersection includes 1) all COMPOOL variables, outer level program variables, and sensors referenced in the PROGRAM and assigned in the PROCEDURE, 2) all COMPOOL variables, outer level program variables, and sensors assigned in the PROGRAM and referenced in the PROCEDURE.

In the case of in-line equations, the data module is explicitly expressed and need not be represented as a sub-module.

The basic unit of a structured flow chart is the "block". A "block" is a module which has a single entrance and a single exit. It will be represented as

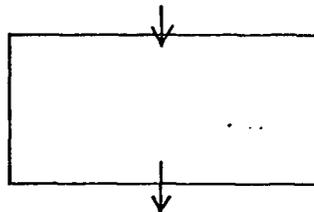


Fig. 9 The basic unit of a structured flowchart

The functional flow of the program depends on the decision structure where the object of that decision can be thought of as a sub-module to the decision statement. The statements used to make decisions are the basic structured statements shown in Fig. 10.

This structured flow chart representation of decision statements has been used at CSDL⁶. Inherent in this representation is the knowledge that any decision statement performs the required sub-module resulting from the decision and immediately returns to the next statement in the main program flow.

There is a similarity between the recommended structured notation for a decision statement and the conventional representation of a CALL statement in that both assume the reader is familiar with the language syntax and that the reader recognizes the implicit return.

In addition, the structured flow chart notation presents a more adequate representation of a CALL in that it recognizes that the main purpose of any CALL is to manipulate data flow. Associated with each CALL, therefore, is the data module associated with the CALL (Fig. 11).

The definition of the data module assumes that the overall program structuring has been completed and defined elsewhere. For structured flow chart notation, only the location (e.g., COMPOOL, local, etc.) and organization (e.g., matrix, array, etc.) must be specified for each data element of the data module. Just as the derivation for a particular equation is not specified in a flow chart, the function of the data module is not specified in the flow chart.

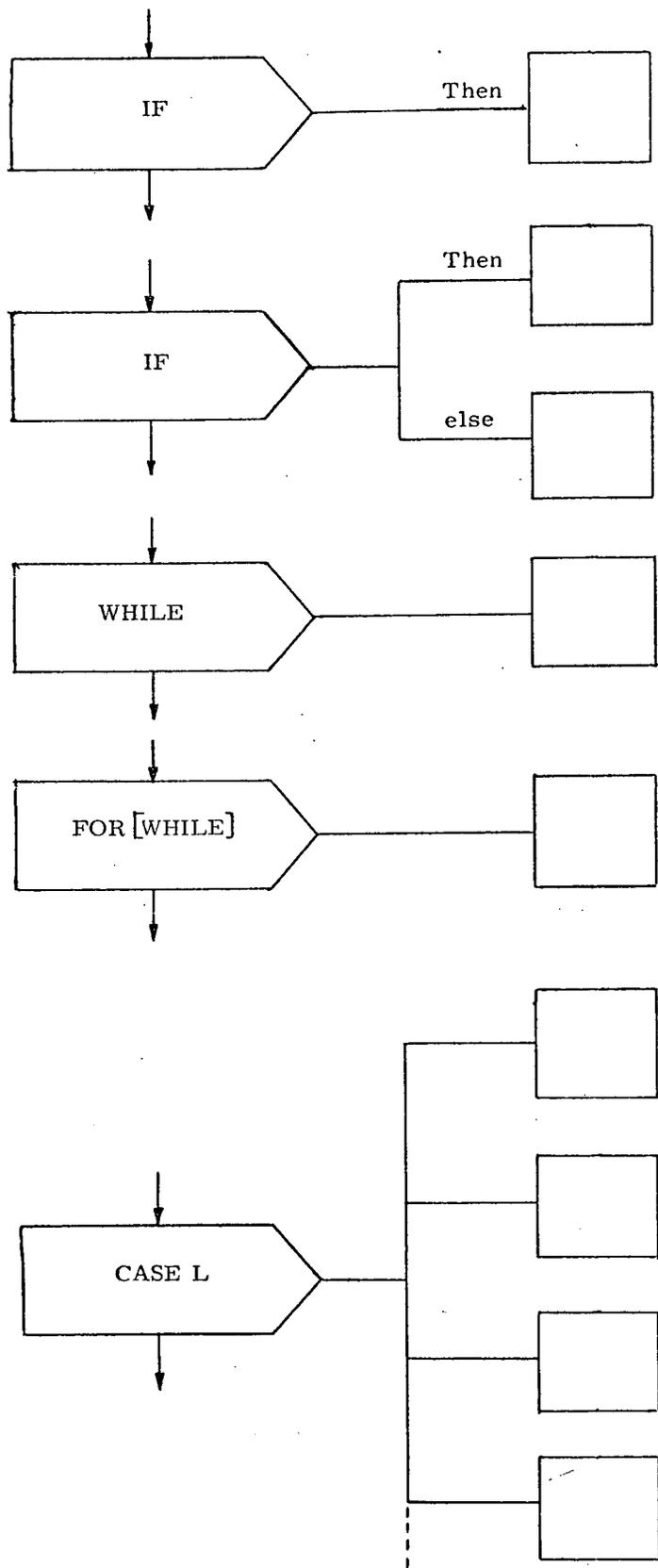


Fig. 10 Decision Statements

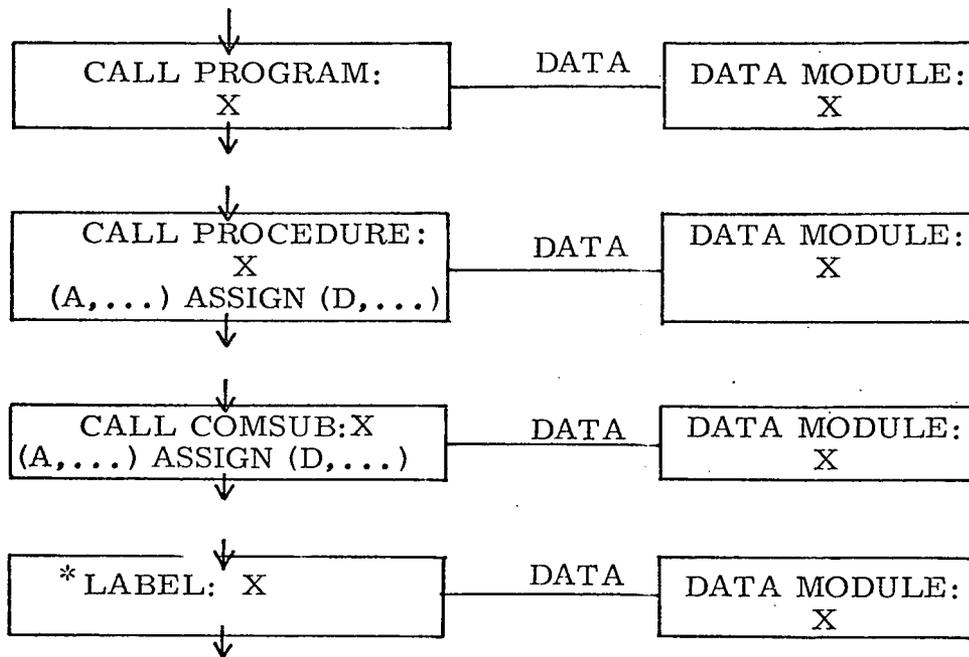


Fig. 11 The CALL and its accompanying data module.**

*This is a "degenerate" CALL in that it represents a simple DO group blocked for the purposes of modularity and top-down one-page representation.

**Further study is required before real-time "CALLS" such as SCHEDULE are incorporated into this structured flowchart notation.

The basic symbols required to understanding a data module are

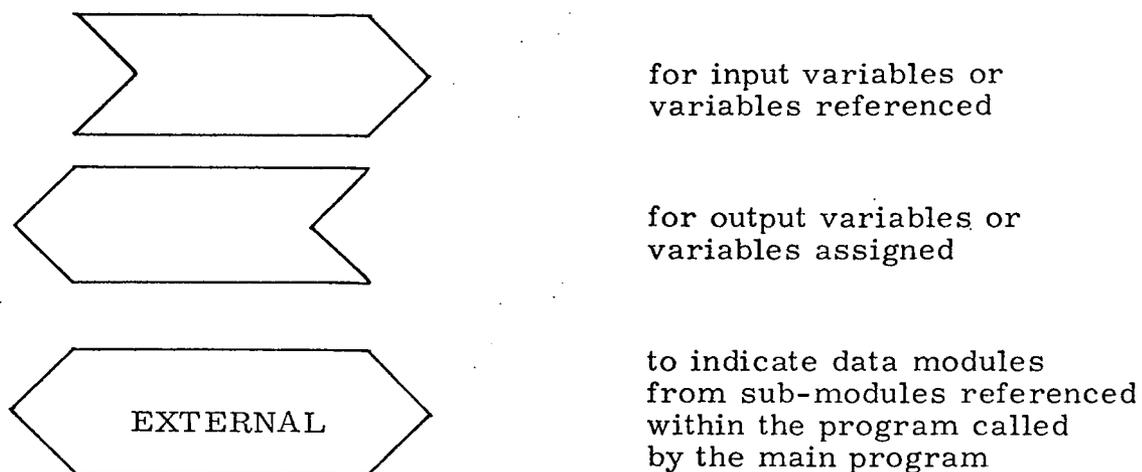


Fig. 12 Basic data module symbols

The notation shown in Fig. 13 is to be used to specify location and organization for data module elements. The location of a data sub-module is indicated within the basic symbol described in Fig. 12. The data elements, with organization indicated, are listed within the block accompanying the location notation.

In the event that a GOTO is required it can be represented as:

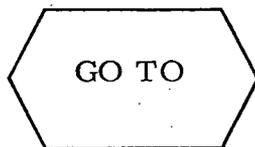
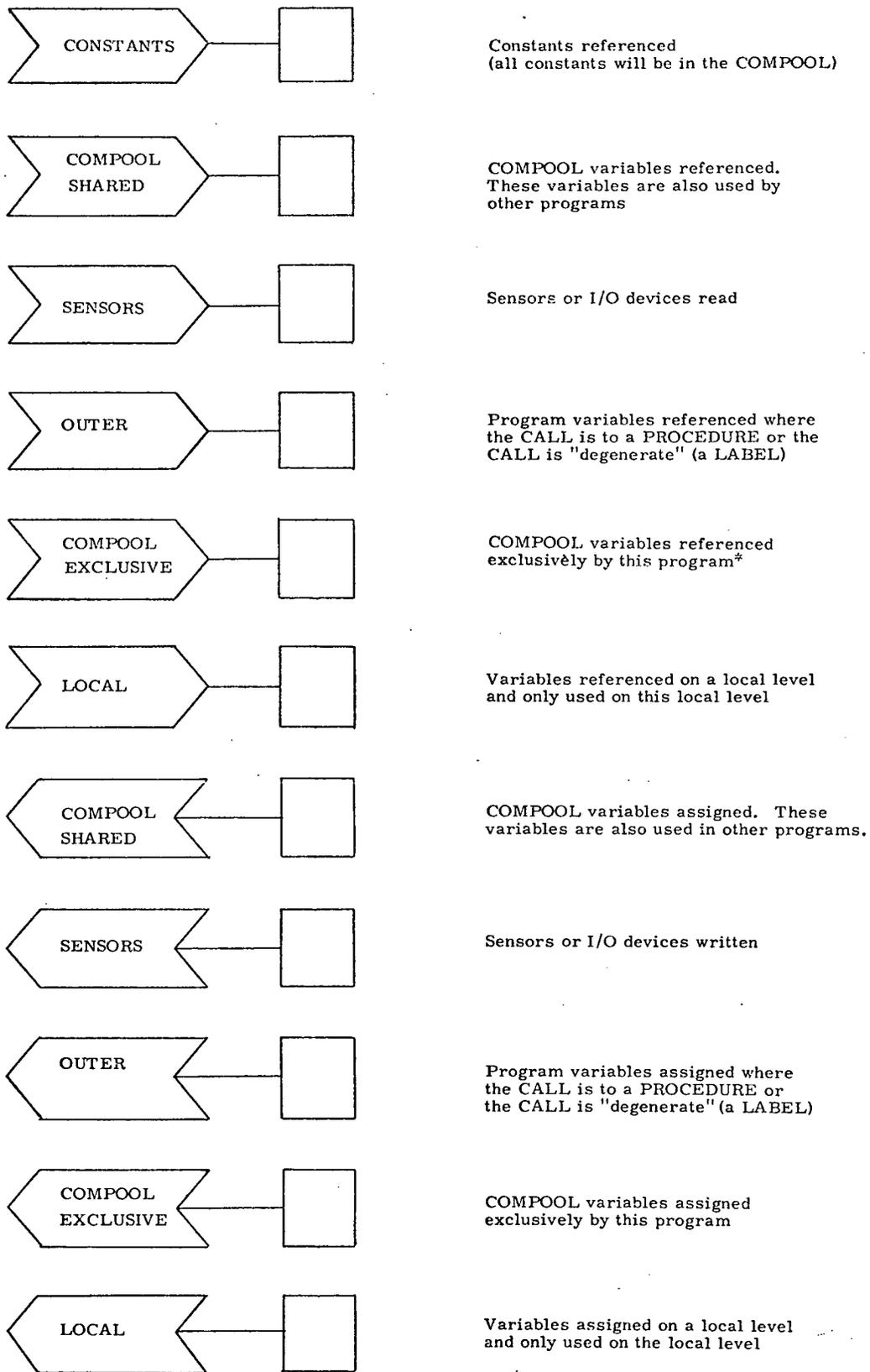


Fig. 14 The structured GOTO symbol

The symbol was chosen as a "cautionary" measure.

To read and write sensors, use the notation READ and WRITE. For example, IMU=READ (CDU) assigns the value of the sensor, CDU, to the local variable, IMU.



* If program variables could be static from one program call to another, this set of variables would be static variables at the program level

Fig. 13 The data flow performed by the object of the CALL or by the object of the LABEL

The RETURN statement shall be represented as

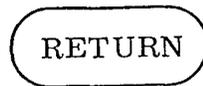


Fig. 15 The structured RETURN symbol

to distinguish it from the decision statements.

The entire set of notation described above may be concatenated in many ways to show complete structured flow. An example of a structured flow chart is presented in Appendix 1.

4.0 Program Structuring

The task of program structuring is of major importance; decisions made for this task determine and dictate the total structure of the software both statically and dynamically. Program structuring defines the building blocks, the control mechanisms and the interfaces of the software. It is in this effort that the inter-relationship of systems software, applications software and data is entirely mapped out.

In the same way that structured programs require rules and enforcement of these rules, the software system itself must be structured according to rules and provision must be made for techniques to enforce these rules. Software tools, including the HOL and the digital simulator, act as automatic aids for correct software structure. Likewise, the executive, including the systems software, can be an effective tool in the dynamic enforcement of these rules.

Program structuring encompasses the definition of 1) the software executive structure (systems software), 2) modularity (including program and data modules), 3) structure within each block, 4) the interface points

between levels and within levels, 5) the structure of the total program including such considerations as timing, memory, priority, error recovery, etc., 6) the requirements and interfaces of data modules common to more than one program (COMPOOL), and 7) automatic sequencing and the constraints imposed on the system due to non-automatic sequences.

There are many difficult problems in laying out software. One of the main problems is that of defining modularity. One definition of a module is a unit which performs a specific function. It has an internal structure and local (private) variables which are unknown to the outside world. We all talk about modularity, but are we talking about the same thing? Both program and data modules might be divided according to:

- . Mission phases
- . Mission functions
- . Blocks of memory
- . Divisions of software error recovery
- . Critical vs. non-critical mission phases
- . Subroutines
- . Control vs. calculations routines
- . Data divisions
- . Components of the assembly, e.g., systems vs. mission modules
- . Synchronous vs. asynchronous logic
- . Instruction sets, e.g., DOCASE
- . Real time vs. non-real time logic
- . Relocatable vs. fixed programs

The system designers must determine the scope of the modules as well as determine which modules are assigned to programs, procedures, tasks, functions, etc. In addition some provision will have to be made in HAL so that common modules will be easily accessible and yet controlled.

Adequate program structuring requires that each of the following topics be considered in great detail:

- 1) General requirements for modules must be determined as well as more specific requirements for different types of modules.
- 2) The extent to which the executive controls dynamic interfaces vs. the extent to which the control modules themselves control dynamic interfaces must be decided.
- 3) A mission module must be self-contained as much as possible. But we must consider that the resultant redundant routines within modules mean more memory and independent development and verification.
- 4) Putting "sacred" variables and constants in a COMPOOL results in more interfaces outside a module, yet guarantees uniformity and correctness of constants and limits to variables. These tradeoffs must be made wisely.
- 5) Additional static and dynamic verification features may be required in the HAL language/compiler to guarantee the reliability we are looking for.
- 6) Rules should be set up immediately for the programming of the algorithmic modules with emphasis on the prevention of unnecessary re-writing of the software later.
- 7) Compiler-simulator interfaces must be defined to provide automatic dynamic verification not included in flight software.
- 8) Dynamic diagnostics and recovery must be defined for the flight code as opposed to those features found only in preflight simulations.
- 9) Requirements must be determined for pseudo modules (Section 5.2).
- 10) Software must be laid out in order that errors or changes in one area will not affect any other area.
- 11) Software must be laid out in order that the high priority events are always executed on time.

- 12) Automatic software error recovery must be considered.
- 13) Software should be designed to prevent human errors from propagating throughout the system.
- 14) Automatic sequences with manual override vs. manually commanded sequences must be considered. Less automated sequencing would require more extensive error recovery logic.

The result of program structuring should be reliable software with cost-effective development and verification. Attaining these goals requires an intimate working knowledge of both the software and the software building process.

5.0 Software Management Techniques

Apollo management techniques have evolved into a well organized and successful method of building software⁴. We are now looking at the problems of managing software for the Shuttle by combining our knowledge from the past and the present with the more formalized approaches of top-down and program structuring techniques.

When a small software program is being designed, coded and debugged by one programmer, organization is not as vital as when several programmers or hundreds of programmers are involved in a single software system. In addition, there are hundreds of people indirectly involved with systems that must interface with the final program. It is understandable that if every programmer is isolated, designing, coding and debugging one small piece of a program, there will be infinite problems when integration of those pieces into the assembly is attempted. Not only must there be a grand plan dictating the interface requirements for each piece, but also how, where, and when a piece fits into a system.

For Apollo, we have solved many of the problems of building software in a formal way. One of the most important concepts was that of the assembly

control supervisor who, with the aid of systems programmers and application programmers, approved every change going into the official assembly. Rules and regulations for interface and module requirements were set up and enforced by the systems programming group, who were responsible for the design, implementation and verification of the systems software. Strict schedules were adhered to for assembly milestones. In addition, every change was tracked to an official specification.

We now propose to continue to adhere to all of the above basic management concepts. But for the Shuttle, we recommend an even further refinement of assembly milestones with a special emphasis on the order in which these milestones are completed. Specifically, we will discuss the official assembly building process by applying top-down methods with greater emphasis on the design of the program structure.

At first glance, the point of view taken here is conservative in that a few "bottom-up" techniques are considered necessary to make the "top-down" concept work.

If we were to look at the software building process from an absolute top-down point of view, we would develop software in the following order:

- 1) Define problem
- 2) Build software
- 3) Build software tools
- 4) Build hardware tools

Traditionally, we have designed things in a more bottom-up manner:

- 1) Define problem
- 2) Build hardware tools
- 3) Build software tools
- 4) Build software

If we were to follow a total top-down philosophy, we would approach the Shuttle software problem by designing the software logic (flow) from the top-down point of view. This design could conceivably result in an actual program listing coded top-down in a meta language.

From Figure 16 we can see that the HOL executive and software tools would be decided upon after the software was designed. Then the computer would be designed by incorporating HOL instructions and features required by the executive and other tools. By these methods, the total design would not be restricted by presently available hardware and software tools.

We, of course, are not proceeding in this way on the Shuttle. In fact, the HOL has been selected and the executive and computer selection are not far behind.

Certainly a subconscious (or unconscious) top-down philosophy has been applied by examining past software efforts. From Apollo experience, we have a fairly accurate picture of the software requirements. Thus, in a very crude sense, we have proceeded in a top-down manner. Of course, if the HOL and the computer do not really answer our software problems, we'll know that indeed the picture we had in our minds was not sufficient to initially jump over the actual design process and "GOTO" the HOL executive and computer decisions.

5.1 Shuttle Software Parallel Efforts

The software development process can be divided into three separate parallel efforts (Fig.17): (1) tool development, (2) bottom-up off-line module building and (3) official top-down building process.

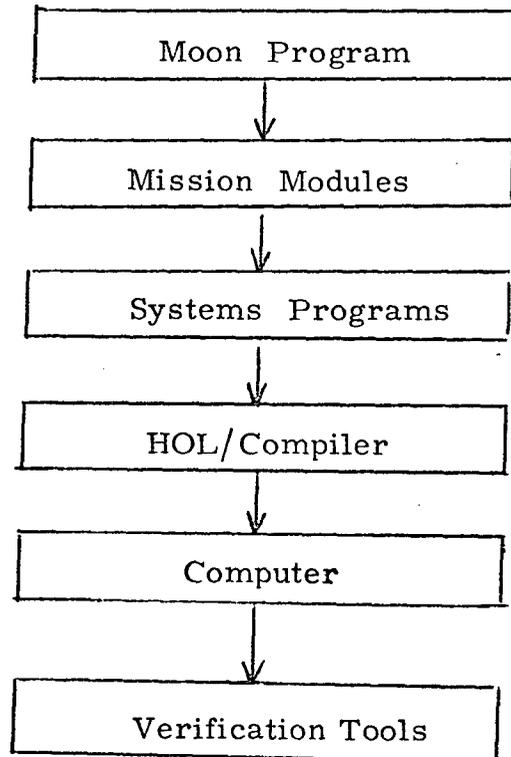


Fig. 16. Ideal Top-down Philosophy

5.1.1 Tool Development

The development of the software tools includes the building of 1) the HOL/compiler for both the host computer and the flight computer 2) the psuedo-module simulator for the top-down flight code (the necessity for this new tool is described below) 3) the all-digital simulator, including a functional simulator for algorithmic module verification and a simulator for flight code verification, 4) the systems test laboratory facility 5) the hybrid test facility and 6) the data management system.

5.1.2 Bottom-up Off-line Module Building

The term, "bottom-up", here is used to describe the module building process with respect to the official assembly building process. In the module building effort, engineers/programmers will first design system programming modules, algorithms and algorithmic modules with top-down, structured programming methods, then use HAL to program these modules for the host computer and then verify these modules by the functional simulator. The design and coding of the host computer programs should conform as much as possible to design requirements for the flight assembly.

The modules themselves can be designed and built applying top-down methods. However, development of smaller sub-modules for which the requirements are completely known can be worked on in parallel, just as the modules are worked on in parallel with the official assembly building process.

For example, if it is known that a Lambert targetting routine is needed, and what its requirements are, it could be developed early in the development effort.

After modules are verified with the functional simulator on the host computer, work proceeds towards preparing the modules for the flight

computer in the same HOL (HAL). These modules would be verified by the flight code simulator, as well as directly in the flight computer. This effort will hopefully require only minor changes in the design and implementation of the algorithmic flight code, since algorithmic modules, in general, will contain code that is relatively independent of the computer. Ideally, HAL code should be able to be carried directly over from the host computer to the flight computer. The only changes to algorithmic code should be those that are forced upon the modules by word length dependency, the system software requirements (e.g., error recovery techniques) or the program structuring requirements (such as dividing a module further into control and subroutine modules, data modules, etc.). The computer should not influence the algorithmic software, since HAL and the system software should serve as a buffer between the computer and the algorithmic code. Ultimately, the HAL compiler would be the only buffer if the core software were all written in HAL.

Flight code modules should be verified in a version of the official assembly which is a snapshot of a recent official assembly revision. Once a module is verified by automatic methods and "eyeball" methods, and has successfully completed all the officially defined "Level 3"⁴ tests, it is ready to be submitted to the assembly control supervisor of the official assembly for approval. After it has been approved, the module is incorporated into the official assembly.

Bottom-up off-line module building in parallel with the top-down assembly process is the combination of top-down and bottom-up methods necessary in building a large flight software system.

5.1.3 Official Top-down Building Process

The top-down building of the official assembly actually begins after the systems software tools are completed. The total system from a larger point of view is being constructed bottom-up, i.e., computer, systems

software, mission software. But the official assembly building process of the mission software itself is shown to be constructed top-down after the systems software is completed. The software verification tools, the flight computer, the HOL/compiler and the executive are all tools with which to build the flight software. The tools directly affecting the building of the mission software are the HOL/compiler and the systems software. The HOL/compiler is assumed to be completed at the start of the assembly building process.

The systems software is developed by creating separate building blocks for the official assembly. The first block or level should be completely designed, coded, verified and documented before components of the next block are allowed to enter the assembly. Each block should be "frozen" before components of the next block are incorporated into the assembly. Once a block is "frozen" each change to that block must be officially approved and treated as a major decision. An example of a traditionally "frozen" piece of software is the HOL/compiler. Without a completed language, the first block of software can not be entered into the official assembly. Likewise, each remaining level of software is dependent on the completion of the preceding level. The first two building blocks of the official assembly define the systems programming, while the third level of software interfaces the systems software with the mission modules.

Inner Core

The first level of software begins with what has recently been coined as "inner core" software. The inner core software consists of the basic executive structure. This includes job scheduling, interrupts and possibly a basic handling of error detection and recovery. The inner core software (as does the HOL) dictates the requirements for the "middle core" software.

Middle Core

The second level of software, known as "middle core" software, completes the system software structure, i.e., detailed interface logic covering error handling, I/O handling, downlink, uplink, hardware/software interface, man-machine interface, etc. The design, coding, debugging, and documentation of this software should be completed before the third level of software is entered into the official assembly.

Outer Core

The next building block of the assembly performs the two basic functions of (1) interfacing mission modules to the system software and (2) defining the mission program structure for a top-down and structured organization of mission modules. The outer core software contains what some of us referred to on Apollo as "glue". Essentially, this third building block consists of "CALLS" to mission modules. This building block is crucial in that it determines the interfaces, layout, timing, priorities, etc. of the entire mission program.

Mission Modules

The final building block of the official software assembly is the incorporation of the mission modules themselves, with all the necessary interfaces.

At any stage in the assembly building process, a checked out module can be incorporated into the official assembly as long as the pseudo simulator is available.

5.2 Pseudo Modules

CALL statements will not, of course, have any anything to CALL until there is something there. A dummy tag with a simple RETURN is

one way to complete the loop. A more sophisticated pseudo module, however, is recommended in order to (1) test the behavior of a module in the larger assembly environment before it is completed (2) test the behavior of a completed mission module with other modules before they are completed (3) save at least 50% of the computer time later on in the verification process. It would be desirable to have an option to run a "real" module with all other modules in a "pseudo" mode, since in the past, at least, most errors during Level 4 through Level 6 were interface errors⁵. These are the most time consuming verification simulations.

The pseudo modules would be designed to act as simple functional simulations of the real module by including such things as receiving inputs and returning outputs at given times. They might have given priorities and given timing, etc.

A real module, once it is ready to take its place in the assembly, might have pseudo sub-modules. Fig. 1 shows examples of real modules with pseudo sub-modules in a top-down environment.

The exact requirements for the pseudo-module simulator must be still further defined. For example, how far down in the top-down tree structure are the pseudo sub-modules included in the pseudo-module simulation option?

The PEARL system¹⁵, still in a partial development phase, as well as others, is an attempt to provide for these facilities. But these systems are far too limited to be applied to shuttle software.

The pseudo module concept, if developed correctly, could become the most significant new tool for the shuttle software effort. Not only can it be used in the development of the official top-down assembly program, but it also can be used as a powerful tool for building each of the off-line modules. Ideally, it should be the first verification tool to be completed in order that it can be used for building the other verification tools.

6.0 Preliminary Structured Programming Rules for Designing Algorithmic Shuttle Software

- . Define the functional specification first
- . Do not begin coding until all stepwise refinements are completed or pseudo modules specified.
- . Use HAL properly to make each program as modular as possible. For example, use PROCEDURES, TASKs, FUNCTIONs, DO groups, etc. for stepwise refinements.
- . Use HAL properly when considering data interfaces. For example, make proper use of UPDATE blocks.
- . Maintain data modularity. For example: use matrix-vector arithmetic whenever applicable; use array arithmetic for visibility and to avoid DO loops.
- . Use structured flow charts (Section 3.4).
- . Denote data-types appropriately in flow charts. For example: \bar{V} , $\overset{*}{M}$, $\overset{\cdot}{B}$, $\overset{\cdot}{C}$, $[A]$, $\{S\}$. If not annotated, a scalar is to be assumed.
- . Each engineer responsible for an algorithm should provide a test specification as well as the program functional and program data specifications.
- . Avoid use of GOTO—justify each GOTO used. If a GOTO is justified, be sure to jump forward to a higher statement number. Never jump back to a lower statement number. A GOTO may be justified if the GOTO exits to a statement immediately following an enclosing functional block.

- Specify all program constants in COMPOOL; include all documented facts related to each constant.
- Do not declare derived constants - be inefficient for the time being and specify these in terms of the original, more meaningful constant. For example:

If $PI = 3.1416$ is defined in the COMPOOL:

Then code $y = 2 PI x$
 Do not code $y = 6.28 x$
 Do not code $c = 2 PI$
 \vdots
 $y = c x$

- Minimize interface points outside of a module and within each module. For example, declare temporary variables on a local level.
- Use caution when defining DOFOR and DOWHILE loops that depend on non-local variables and non-local events.
- Labels and variable names should be meaningful.

7.0 Conclusion

From recent studies of APOLLO systems problems, we have come to realize even more the importance of sound basic principles for software design, development, and verification. Over 90% of the system problems would have been prevented by a better philosophy¹⁶. Correct software philosophy applies to programming techniques, program structuring and the management of the software development and verification process.

With new tools and new techniques, we must continuously remind ourselves not to waste time worrying about problems that no longer exist.

For example, the HOL (HAL) will now replace many of the tasks previously performed either automatically by the digital simulator or manually.

We do not want to take for granted the functions performed by APOLLO tools. In fact, we have taken a step backwards already in some cases. For example, the reliability of the computers presently being considered for Shuttle is inferior to that of the AGC (APOLLO Guidance Computer). Thus error detection and recovery becomes more important for the Shuttle.

We should not discard a technique or a tool merely because there were problems with it on APOLLO. In some cases, an alternative technique would have created greater problems. For example, the concept of an asynchronous executive has been met with some reservations. However, the APOLLO asynchronous executive provided flexible, and thus cost effective, developmental capabilities, as well as prevented more than one actual flight disaster.^{17,18,19}

We must expect new problems to occur for the Shuttle and we must prepare for them now; new tools and techniques always have problems to work out. For example, if core swapping becomes a technique on the shuttle flight computer, we should pay special attention to this area.

The approaches we have been proposing will lead to organization and enforcement of rules in the development of a large software system. Reliable cost-effective software is the ultimate aim.

REFERENCES

1. Dijkstra, E. W., Notes on Structural Programming, June, 1971.
2. Mills, Harlen, "Top Down Programming in Large Systems", Courant Computer Science Symposium, June 29-July 1, 1970.
3. Dijkstra, E. W., "The Humble Programmer", CACM, Vol. 15, October, 1972.
4. Hamilton, M., "Management of Apollo Programming and its Application to the Shuttle", CSDL Software Shuttle Memo No. 29, May 27, 1971.
5. McCracken, Daniel D., and Gerald M. Weinberg, "How to Write a Readable Fortran Program", Datamation, Oct., 1972.
6. Private communication with Bernard Goodman, CSDL.
7. Hamilton, M., "Anomaly Research and its Impact on Software Management", CSDL Shuttle Management Note No. 3, Feb. 10, 1972.
8. Youngs, E. A., "Error-Proneness in Programming" (University of N. Carolina doctoral thesis) Ann Arbor, University Microfilms, 1970.
9. Wulf, William A., "A Case Against the GOTO", Proceedings of the ACM, Vol. II, August, 1972.
10. Private communication with Donald Garvett, CSDL.
11. Hopkins, Martin, "A Case for the GOTO", Proceedings of the ACM Vol. II, August, 1972.
12. Liskov, Barbara H., "The Design of the Venus Operating System", CACM, Vol. 15, No. 3, March, 1972.
13. Drane, Lance W., Bruce J. McCoy, Leonard W. Silver, "Design of the Software Development and Verification System (SWDVS) for Shuttle NASA Study Task 35-S", CSDL R-721, August, 1972.
14. Zeldin, Saydean, Bruce McCoy, and Phyllis Rye, "The Programming Language HAL-An Evaluation for Space Shuttle Applications", CSDL, E-2716, October, 1972.
15. Snowden, R. A., "PEARL: An Interactive System for the Preparation and Validation of Structured Programs", Sigplan Notices, March, 1972.

Preceding page blank

16. Hamilton, M., "First Draft of a Report on the Analysis of Apollo System Problems During Flight", CSDL Shuttle Management Note No. 14, October 23, 1972.
17. Hamilton, M., "The AGC Executive and its Influence on Software Management", CSDL Shuttle Management Note No. 2, Feb. 10, 1972.
18. Hamilton, M., "Executive and its Influence on Software Management", CSDL Shuttle Management Note No. 7, March 21, 1972.
19. Densmore, Dana, "Evaluation of Synchronous vs. Asynchronous Executive/Operating System for the Space Shuttle (Study Task 31S), CSDL E-2687, June, 1972.

APPENDIX 1

Application of Top-Down Structured Programming to Shuttle Algorithms—A First Attempt at Structured Flow Charting

Structured flow charts are compared to conventional flow charts from the Space Shuttle GN&C equation document No. 12 (Revision 2), "Entry, Approach, and Landing Navigation". The flow charts in the Shuttle document are reproduced here in Appendix 2 for ease of comparison. The Shuttle document flow charts in Appendix 2 represent a FORTRAN program, while the flow charts in Appendix 1 are structured and represent a HAL program. The FORTRAN program was written with no concept of structured programming in mind. Some variable names have been changed to be more meaningful since the full name of a variable should be used in the actual HAL program. Wherever this occurs, it is noted for ease of comparison. For consistency, many original names are used here even though HAL names cannot have subscripts as part of the name. Notes are also marked on the structured flow charts where necessary to clarify changes from the original program flow.

The functional flow in Fig.1-1 is slightly different from that in the original document (c.f. Fig. 1 Appendix 2). The changes reflect the actual specification.

It is to be noted that this navigation program has been incorporated into the Unified Navigation Filter program for the Shuttle. But it is used here as a simple example to show how to structure a program. The structured flow should be easily understood even by people not familiar with the Shuttle navigation concept. It is suggested that the entire structured flow be read first before comparing the flow from Appendix 2 .

In Figure 1-2 an attempt has been made to show the modular nature of the data. The division of data modules presented here should not be taken as the final specification for entry module data for Shuttle. To do this, the specifications for the entire Shuttle data structure would have to have been defined. This division is presented as an example to show data modularity. All program constants appear in the COMPOOL even though they may pertain only to the navigation program. This rule makes all constants visible and easily checked by such systems as the data management system that will be a part of the verification process. This diagram indicates if a variable is accessible to other programs and therefore is an aid in the program structuring process. It also adds some understanding as to the type and amount of information required by the program.

The functional flow of the program and the first page of the actual program should be similar in program structure. Therefore, Fig. 1-3, the actual program flow can be compared to Fig. 1-1, the functional flow and to Fig. 1-2, the data flow.

The overview presented in Fig. 1-3 is much easier to follow than flipping back and forth through pages 5-5 through 5-9 of the Shuttle document. This is especially true concerning the data-good switches loop, which is hard to follow in the document; but here the overall picture of the loop is presented on one page, and then expanded in other procedures. The notation used in the original document (although partly repeated here for comparison) is not clear. For example, Fig. 2c of Appendix 2 shows $\rho(t)$ to be a density calculation and $\bar{\rho}$ to be an average density while Fig. 2g shows $\bar{\rho}_T = \bar{r} - \bar{r}_T$. The symbols are so closely related that the reader finds it difficult to follow the program flow.

This particular Shuttle flow chart example was chosen because the logic in the original flow diagrams was not too complicated. Even in this simple example, many of the interface problems inherent in the unstructured approach can be overcome. An attempt is made here to show the advantages of a structured approach.

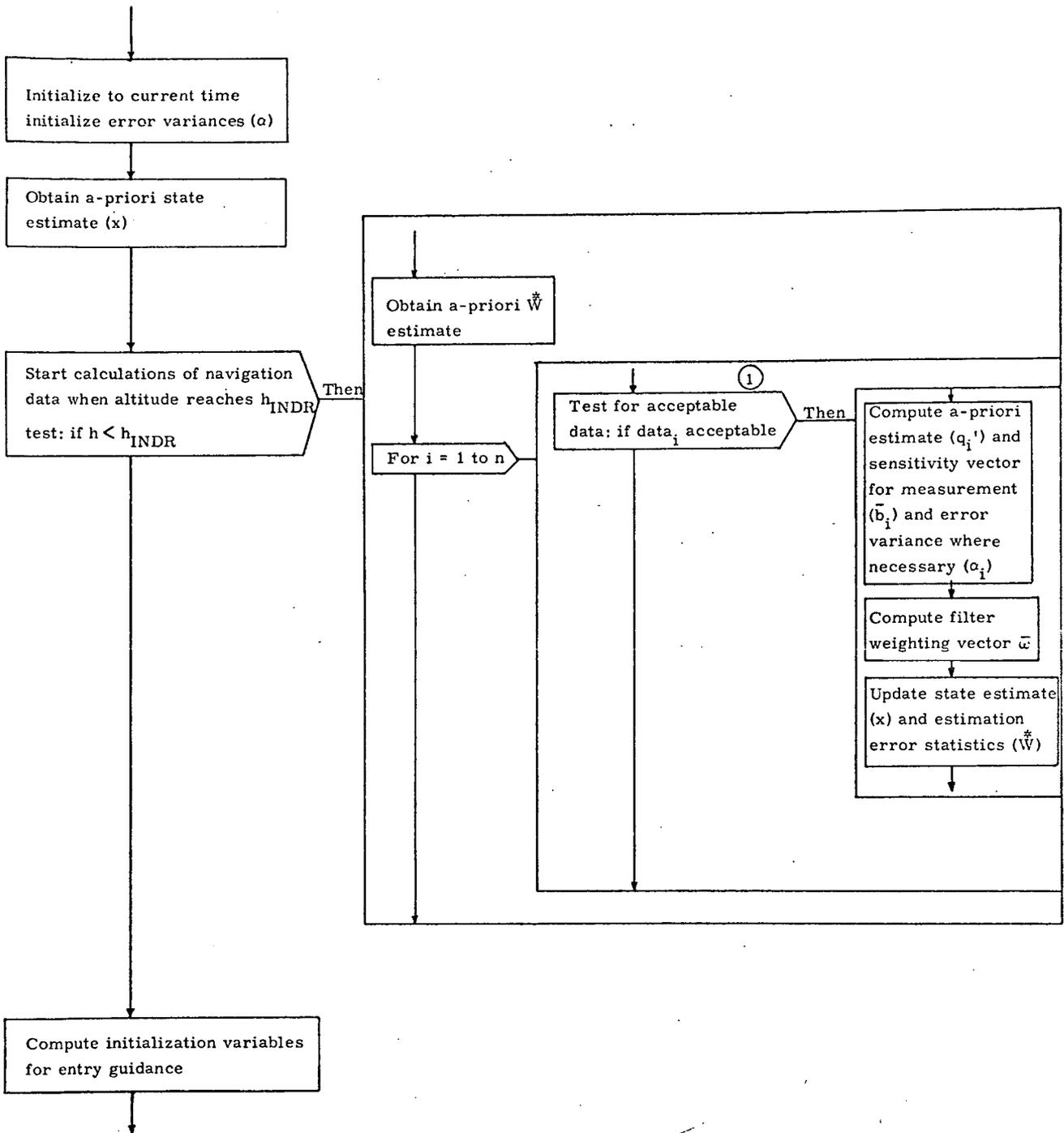


Fig. 1-1. ENTRY NAVIGATION FUNCTIONAL FLOW DIAGRAM

Fig. 1-1 NOTES:

- ① The test on the data good switches in the original functional flow is misleading. The specifications call for more than one criterion for data acceptability: data good switches and possible state constraints. (c. f. Appendix 2, Fig.1)

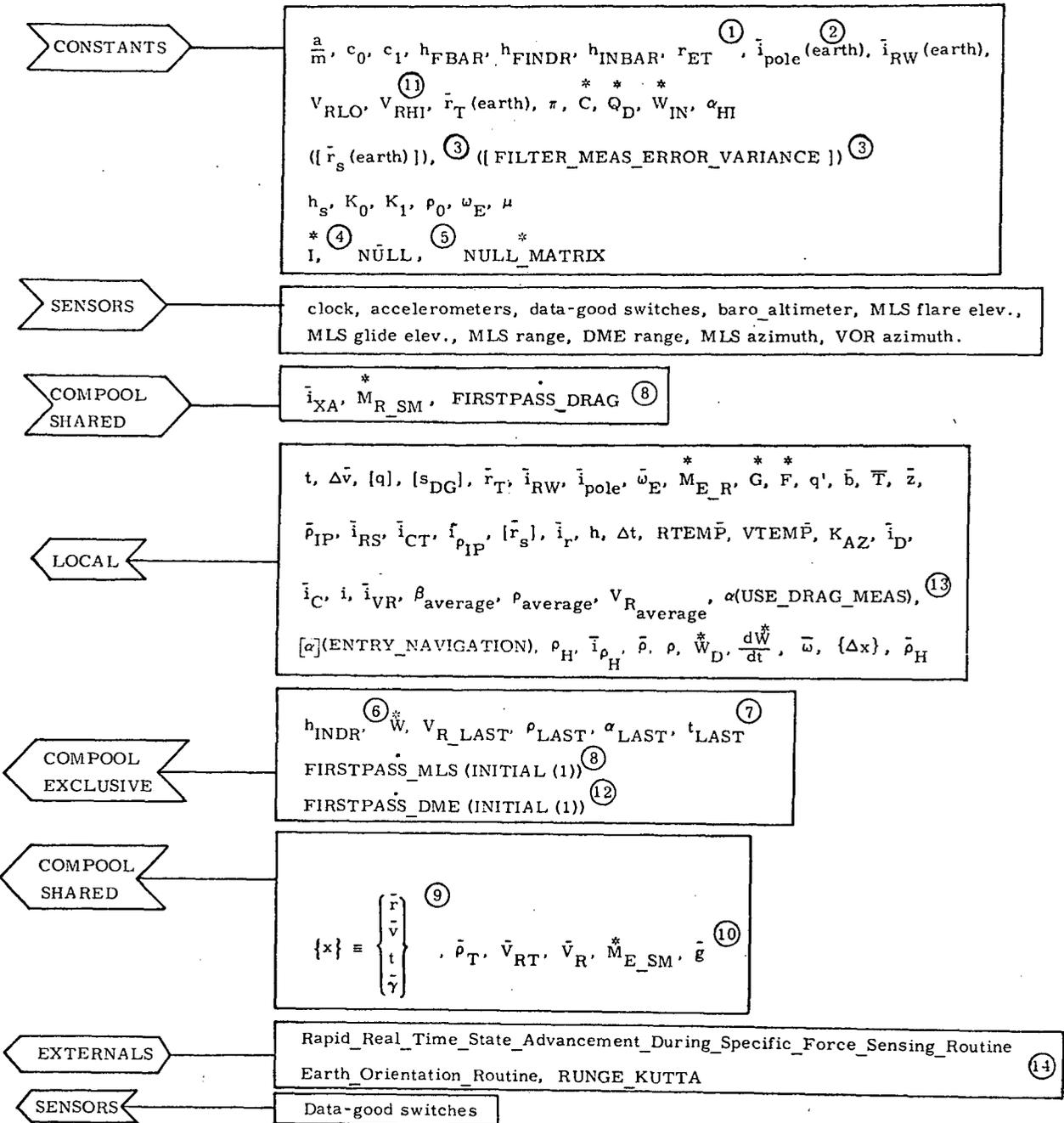


Fig. 1-2. ENTRY_NAVIGATION DATA MODULE

Fig. 1-2 NOTES

- ① r_{ET} is missing from the program constants in the original flow-chart (c.f. Appendix 2, Fig. 2A). It does appear in the original specifications - a case for using original specifications as guidelines.
- ② For consistency and visibility, all vectors are represented in Appendix 1 with bars above the variable name.
- ③ Since each filter measurement error variance corresponds directly to the list of measurements indicated by S_{DG_i} , it is advantageous to make an array. This array indicates the actual relationship instead of a list of constants that appear to be used in many different ways. (c.f. Appendix 2, Fig. 2A, the α^2 list, the r_s (earth) list) Note that the new array name is not ambiguous in the case of the error variances and clearly defines the contents of the array. In the case of α_{HI} (original notation), this is kept as a separate constant, since it indicates an altitude dependence for the baro-altimeter error variance case. The name was changed to α_{HI} (structured notation) to prevent a mistaken vector notation.
- ④ I^* (an identity matrix) is included as a constant here, but does not appear in the original. Although this symbol for an identity matrix is commonly used, the definition should be explicit to avoid confusion.
- ⑤ A null vector (\overline{NULL}) seems more clear than 0 (c.f. Appendix 2, Fig. 2.)
- ⑥ h_{INDR} is incorrectly placed in the constant list (c.f. Appendix 2, Fig. 2A) since it is actually re-assigned in the program (c.f. Appendix 2, Fig. 2b)
- ⑦ These variables were omitted from the original flowchart list of input variables.

Fig. 1-2 NOTES (cont.)

- ⑧ There is one FIRSTPASS variable in the original document. By referring to the specifications, it can be seen that there really should be 3 FIRSTPASS flags as shown here - one for every time the W matrix is to be re-initialized. FIRSTPASS is not initialized in the original document, but the initialization must be defined here, since the value of this variable is tested each time the program is entered. Since programs have no static variables from CALL to CALL, FIRSTPASS must be in the COMPOOL. The intention is to set FIRSTPASS=1 when entry guidance is initiated and then to reset FIRSTPASS to zero for each succeeding pass. There are two ways to accomplish this. One way is to have an external program on a higher level set and reset this variable; (obviously this was the intent from the flow of the original program and is the method shown in the data flow here for FIRSTPASS-DRAG). The second method is to initialize FIRSTPASS in the appropriate DECLARE statement in the COMPOOL and to let the Entry Navigation program reset it. This second option is used to initialize FIRSTPASS-MLS in the "COMPOOL EXCLUSIVE" data sub-module.
- ⑨ Indicating the state {x} as a structure makes the intent of the code for updating the state clear (c.f. Appendix 2, Fig. 2h) and recognizes the organization of the state {x}.
- ⑩ g is the notation throughout Appendix 1 for the variable sometimes specified as g and other times as g (t) (c.f. Appendix 2, Fig. 2g). Also note that the output variables for this program appear in Fig. 2g, Appendix 2, and are, therefore not visible. The output variables should appear in the data module associated with the program.

Fig. 1-2 NOTES (cont.)

- ⑪ Error in original document. V_{RHI} shown as a vector, but used as a scalar (c. f. Fig. 2a, Fig. 2d, Appendix 2).
- ⑫ The dot above the variable `FIRSTPASS_DME` indicates that it is a bit string data-type.
- ⑬ When the same name is used for two different variables, indicate the procedure or program that the variable belongs to, in parentheses.
- ⑭ See note ① Figure 1-7.

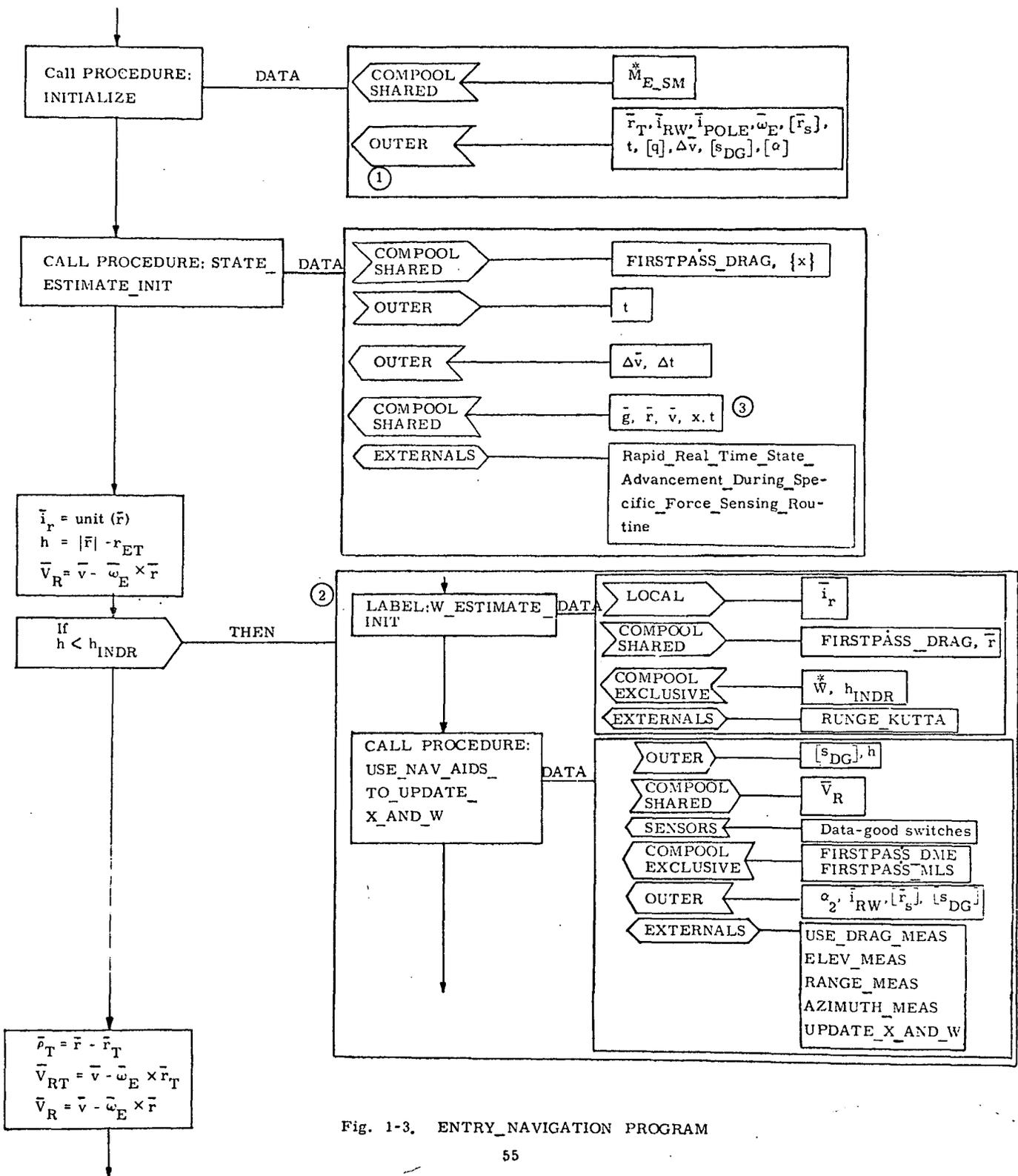


Fig. 1-3. ENTRY_NAVIGATION PROGRAM

Fig. 1-3 NOTES

- ① This indicates that these variables must be declared on the program level.
- ② W_ESTIMATE_INIT refers to a program label which blocks off a block of statements with a particular thought in mind, but will be found in-line in the code. Since this particular page in the flow-chart is too small for the entire level, the W_ESTIMATE_INIT block is found on another flowchart page. The modularity expressed here is accomplished in HAL by a

```
W_ESTIMATE_INIT: DO; statement ; ... END;
```

This label is to be accompanied by a data module to justify all data flow.

- ③ x.t qualifies the time of the permanent state vector and distinguishes it from t, the local variable.

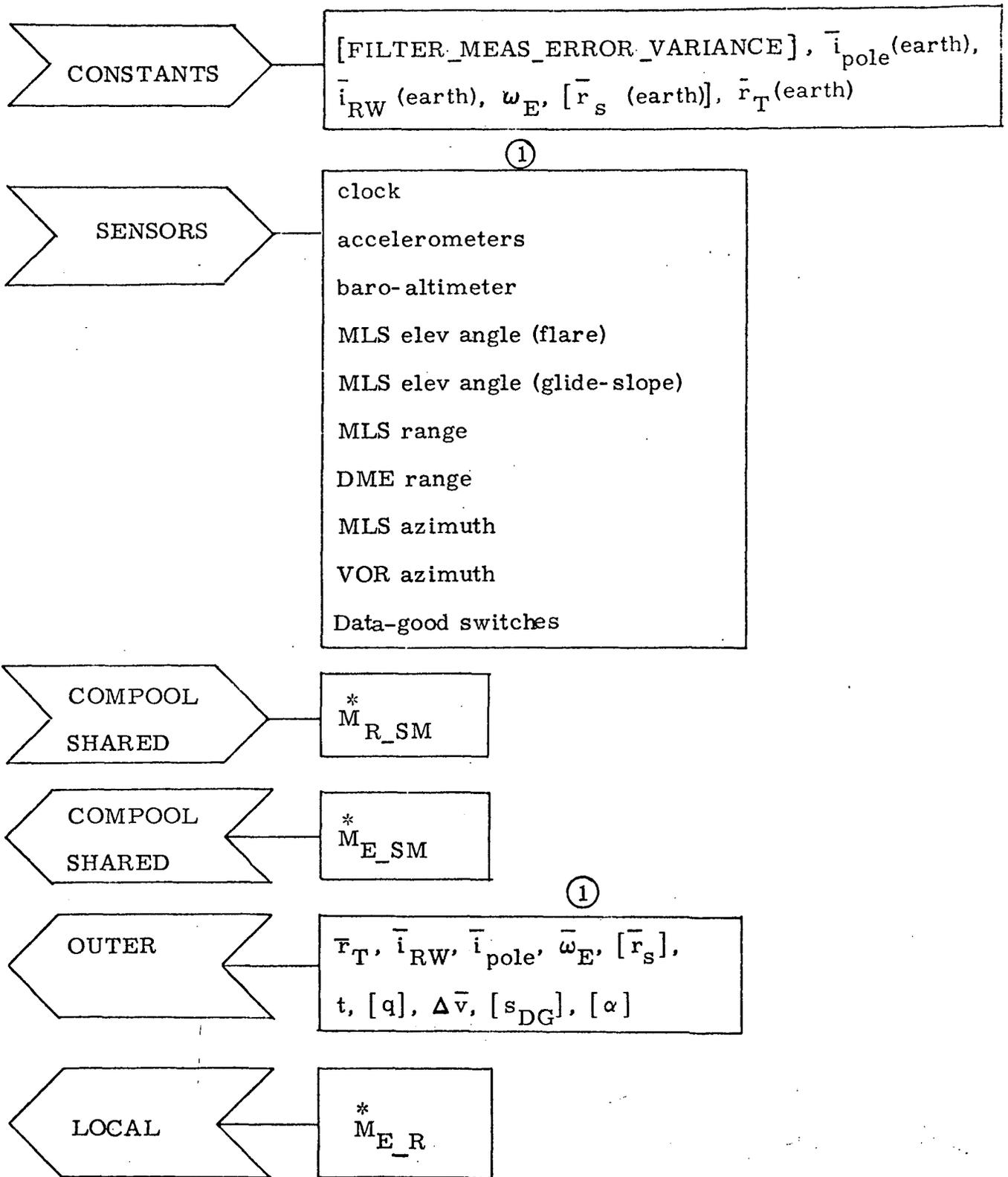


Fig. 1-4. INITIALIZE PROCEDURE DATA MODULE

Fig. 1-4 NOTES

- ① ω_E and $\bar{\omega}_E$ represent two separate data elements here. This notation is kept for ease in comparing the flow of Appendix 2. But the actual HAL program can not have the same name for two different variables declared within the same scope.

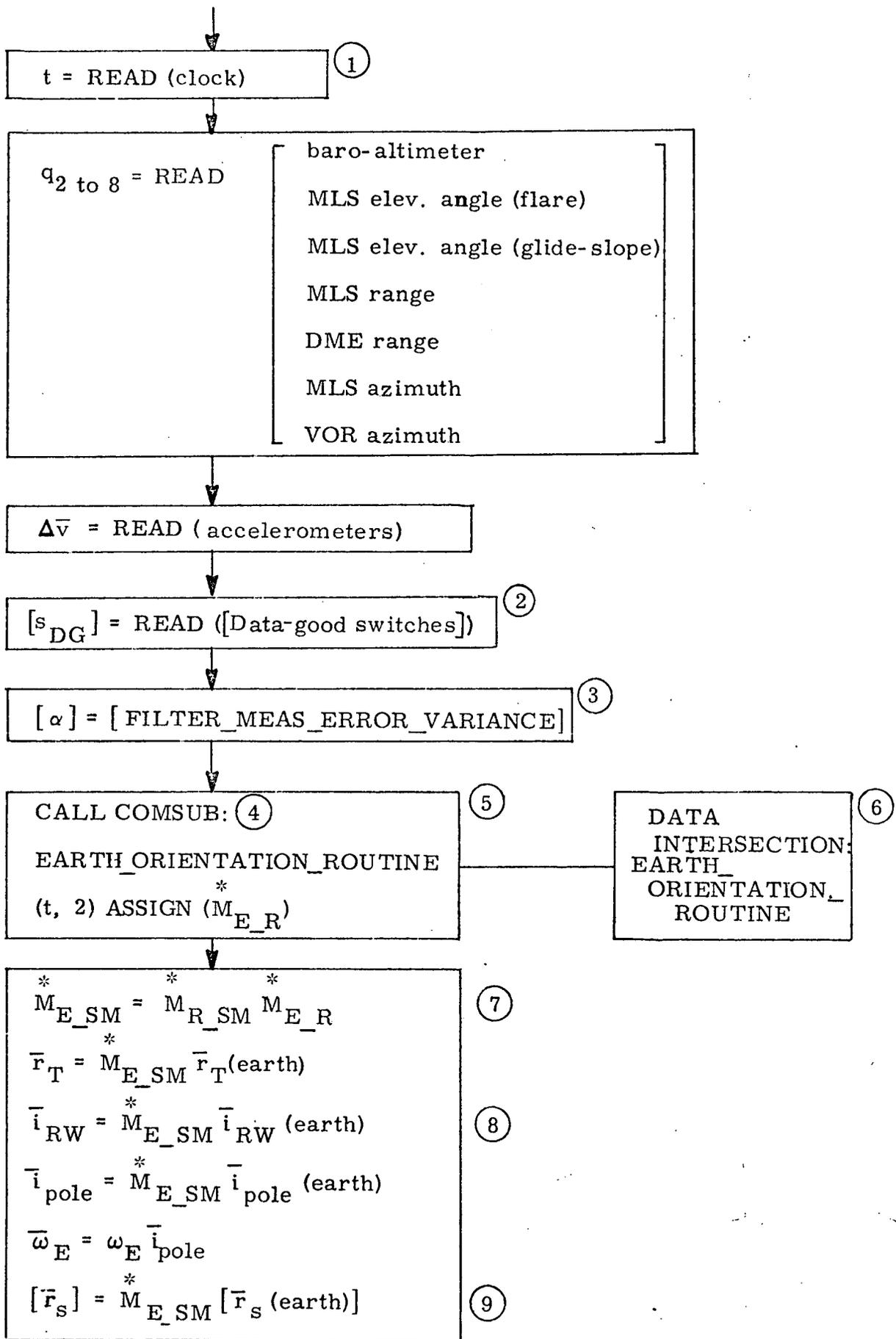


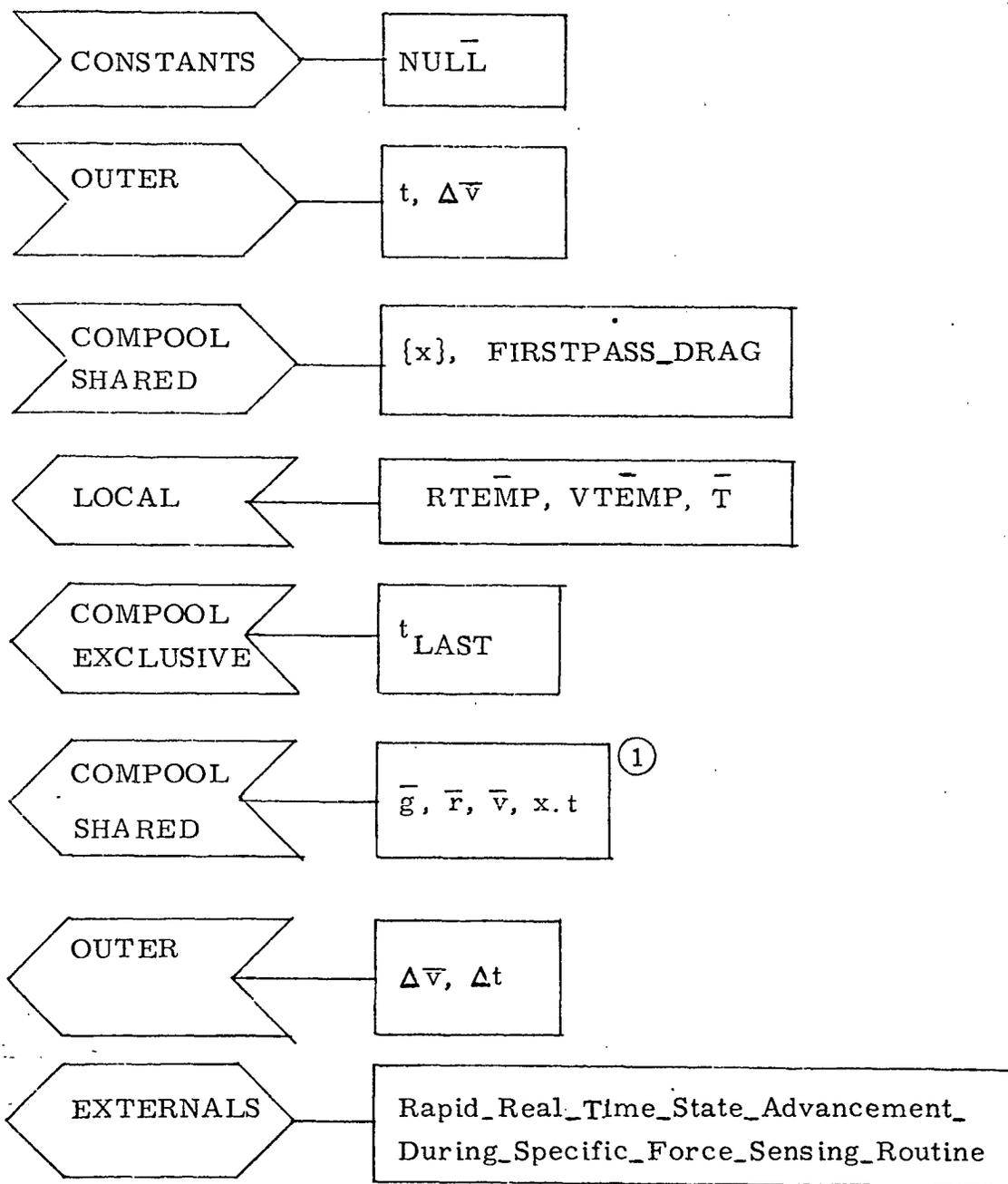
Fig. 1-5. INITIALIZE PROCEDURE

Fig. 1-5 NOTES

- ① Since there is little emphasis on data in the original document, some of the local variables described and initialized here (e. g. t, $\Delta\bar{v}$, [q],[s_{DG}]) were not described as completely in the original document. (c.f. Appendix 2 Fig. 2a). For example, the specifications (p. 4-2 of original document) require that the measurement to be incorporated (q) and its a priori estimate be at the same "effective" time. But the original flowcharts do not specify where this data read will occur.
- ② The decision to read all data good switches and assign them to a local array is made on a higher level. They could just as easily have been read during the USE_NAV_AIDS_TO_UPDATE X_AND_W procedure if it were felt that the later information would be more valid. Specifications in the original document (p. 4-2) indicate that the present decision is more valid.
- ③ The initialization for the set of α is scattered throughout Appendix 2. It can be simplified by initializing α as an array.
- ④ At present, there is no COMSUB implemented in HAL (i. e., a procedure that can be compiled separately and is used in common with many programs). The Shuttle HAL will include COMSUB. In the meantime, a module that is to become a COMSUB will be a PROCEDURE. Each user can "INCLUDE" the PROCEDURE as part of the compiled program and thus save recoding later.
- ⑤ The CALL to this routine is placed in the initialization routine because its only purpose is to compute $\bar{M}_{E R}^*$. This matrix is only a function of the present time and can be considered as "input" calculations for the Entry Navigation program.
- ⑥ The complete data module for this routine is defined as a part of the COMSUB specification. The fact that no COMSUB-PROGRAM intersection data is shown here indicates that the EARTH_ORIENTATION_ROUTINE has not yet been specified at that level. To actually compile and execute this page (the INITIALIZE procedure) at present, the EARTH_ORIENTATION_ROUTINE and its data module would be

simulated as pseudo modules.

- ⑦ This group of assignments is a subset of the group of assignments that appear in Appendix 2, fig. 2b. Since these assignments depend only on time, t , they belong in the initialization section. The remaining subset of assignments found in the original list are dependent on the current value of the state vector and, therefore, these assignments are made after the initial state estimate is made.
- ⑧ This equation is defined twice in the original (c. f. Appendix 2, fig. 2g).
- ⑨ The transformation of \bar{r}_s (earth) to SM coordinates is performed for each data-good unnecessarily in the original.



NOTES:

- ① The entire structure $\{x\}$ is referenced in this routine, but only a portion of the structure is assigned (\bar{r} and \bar{v} and t are assigned)

FIG 1-6 STATE_ESTIMATE_INIT DATA MODULE

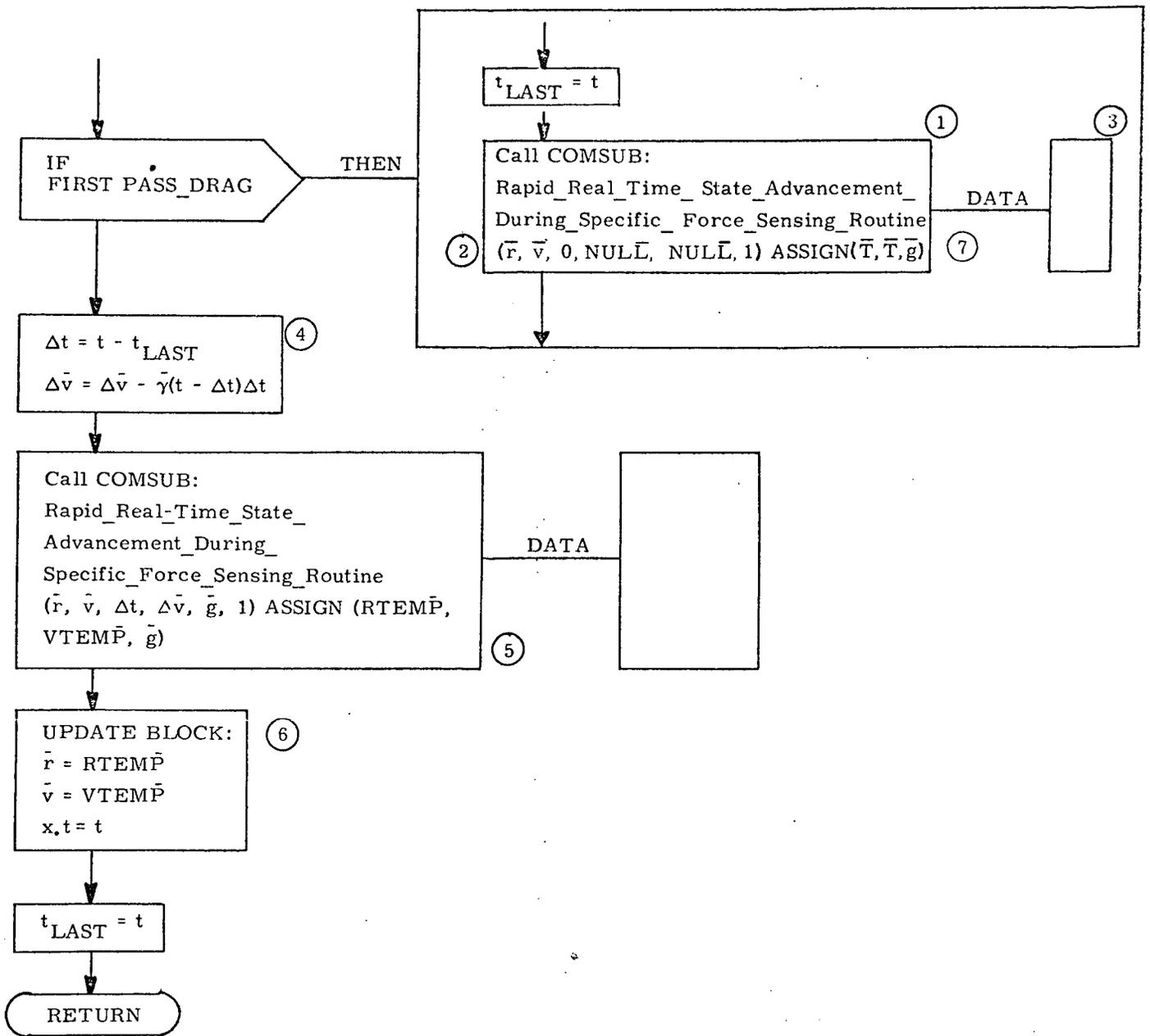


Fig. 1-7. STATE_ESTIMATE_INIT PROCEDURE

Fig. 1-7 NOTES:

- ① The name of this COMSUB must be reduced to a maximum of 31 characters for HAL. For comparison with the original document it has been kept in its present form.
- ② Since there is only one copy of $\{x\}$, the data element \bar{r} of the structure $\{x\}$ can clearly be referenced as just \bar{r} instead of qualifying \bar{r} as $x.\bar{r}$.
- ③ At present this is actually a null data intersection. This data module may not be complete. If this COMSUB becomes a program in the program structuring process, the formal parameters described in the program flow block will have to be incorporated into the data module.
- ④ Δt is not defined in the original document. (c.f. Appendix 2, fig 2a)
- ⑤ This ASSIGN is to be part of the HAL language for the COMSUB syntax (just as it is now for the PROCEDURE). The variables assigned are to be found in the ENTRY_NAVIGATION data module, not necessarily in the COMSUB data module. The permanent state $\{x\}$ may not be updated here because it is a locked variable. The HAL/s implementation for passing and assigning locked variables is now under consideration. Thus, the passing of \bar{r} , \bar{v} shown here would have to be modified for the current HAL implementation.
- ⑥ Here, the permanent state is updated in a special UPDATE block to avoid reading and writing conflicts with other programs that may also use the permanent state.
- ⑦ T is a local dummy variable for the purpose of matching the formal parameter list.

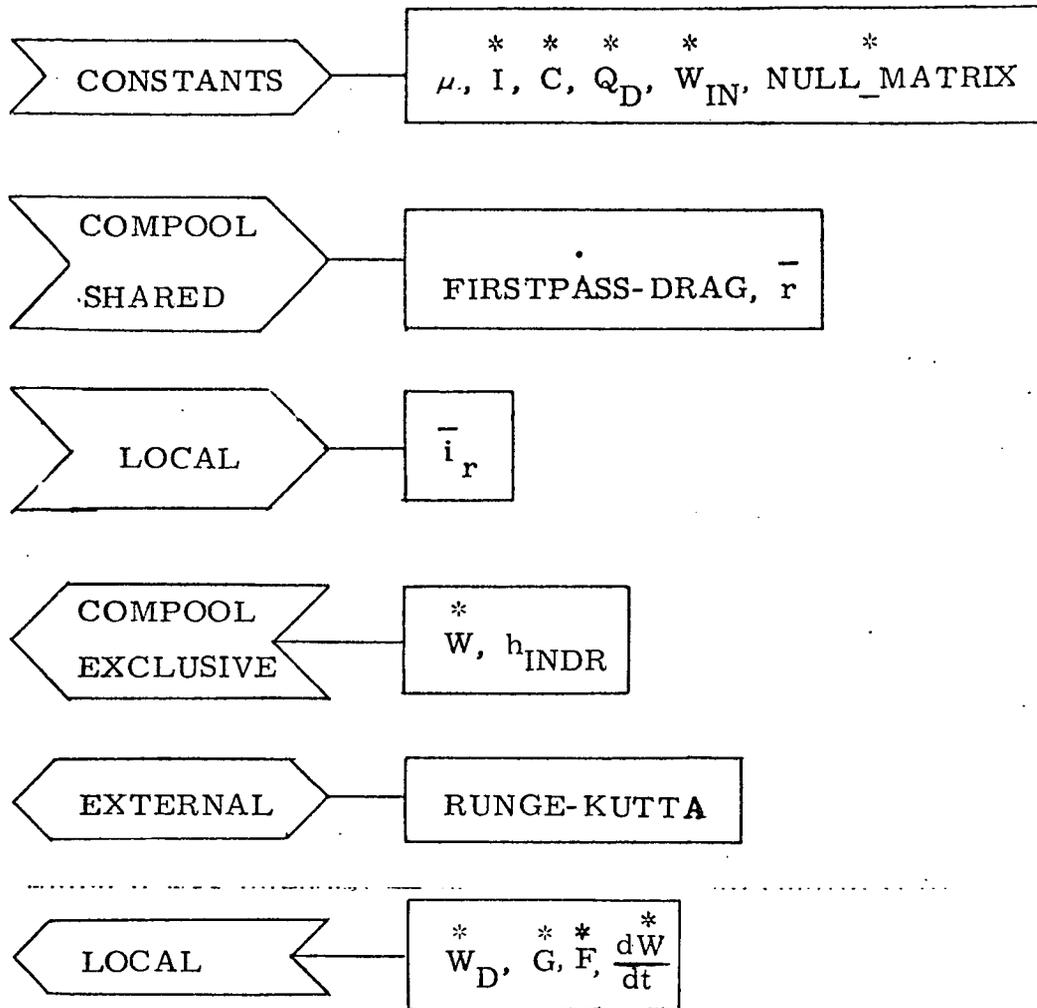


FIG. 1-8 W_ESTIMATE_INIT DATA MODULE

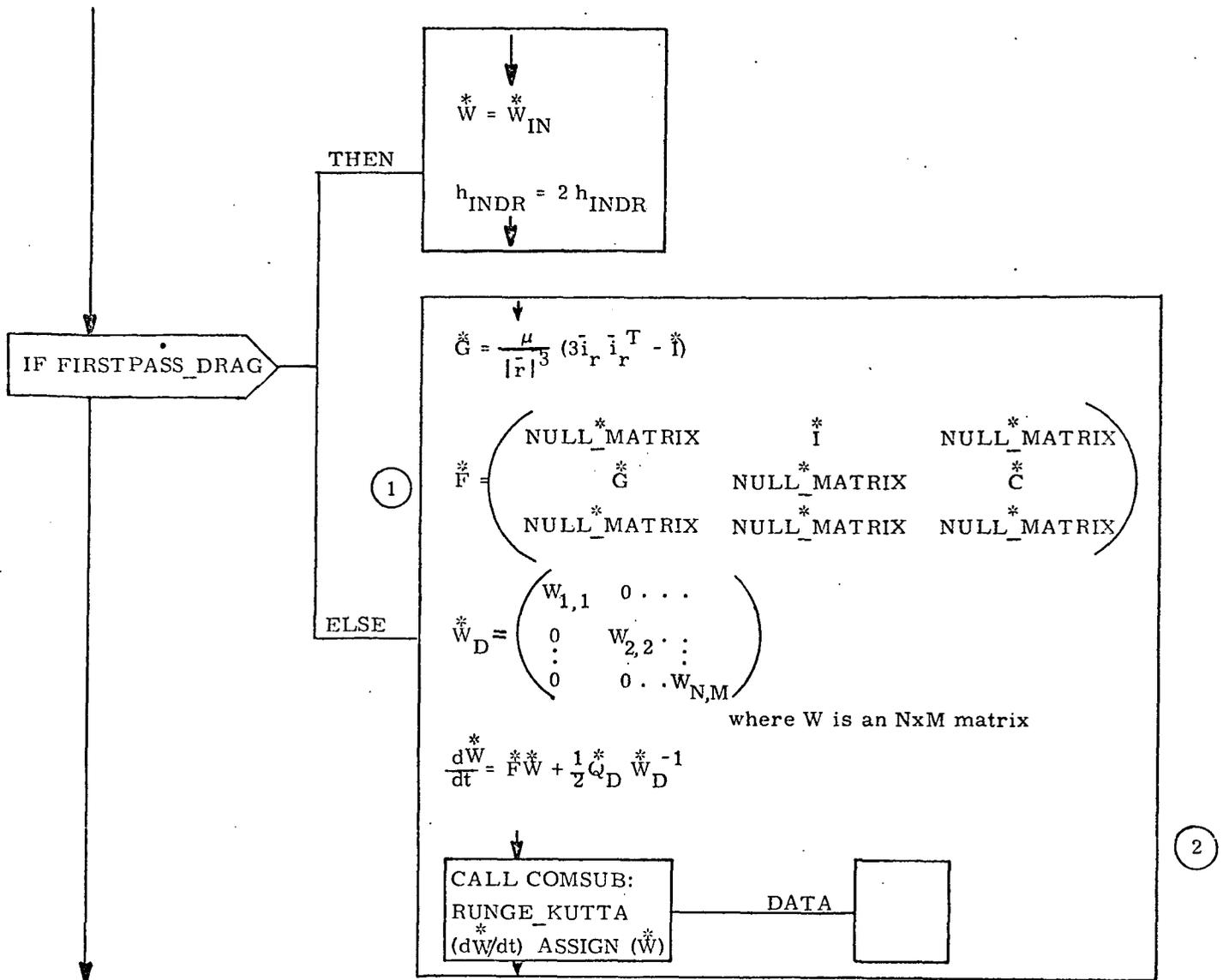


Fig. 1-9. W_ESTIMATE_INIT (LABEL)

Fig. 1-9 NOTES

- ① \dot{W}_D^* is not defined in the original flowchart.
- ② There is no mention of the integration step in the original document. (c. f. Appendix 2, fig. 2b). Also note that $\frac{d\dot{W}^*}{dt}$ is not an acceptable name for HAL.

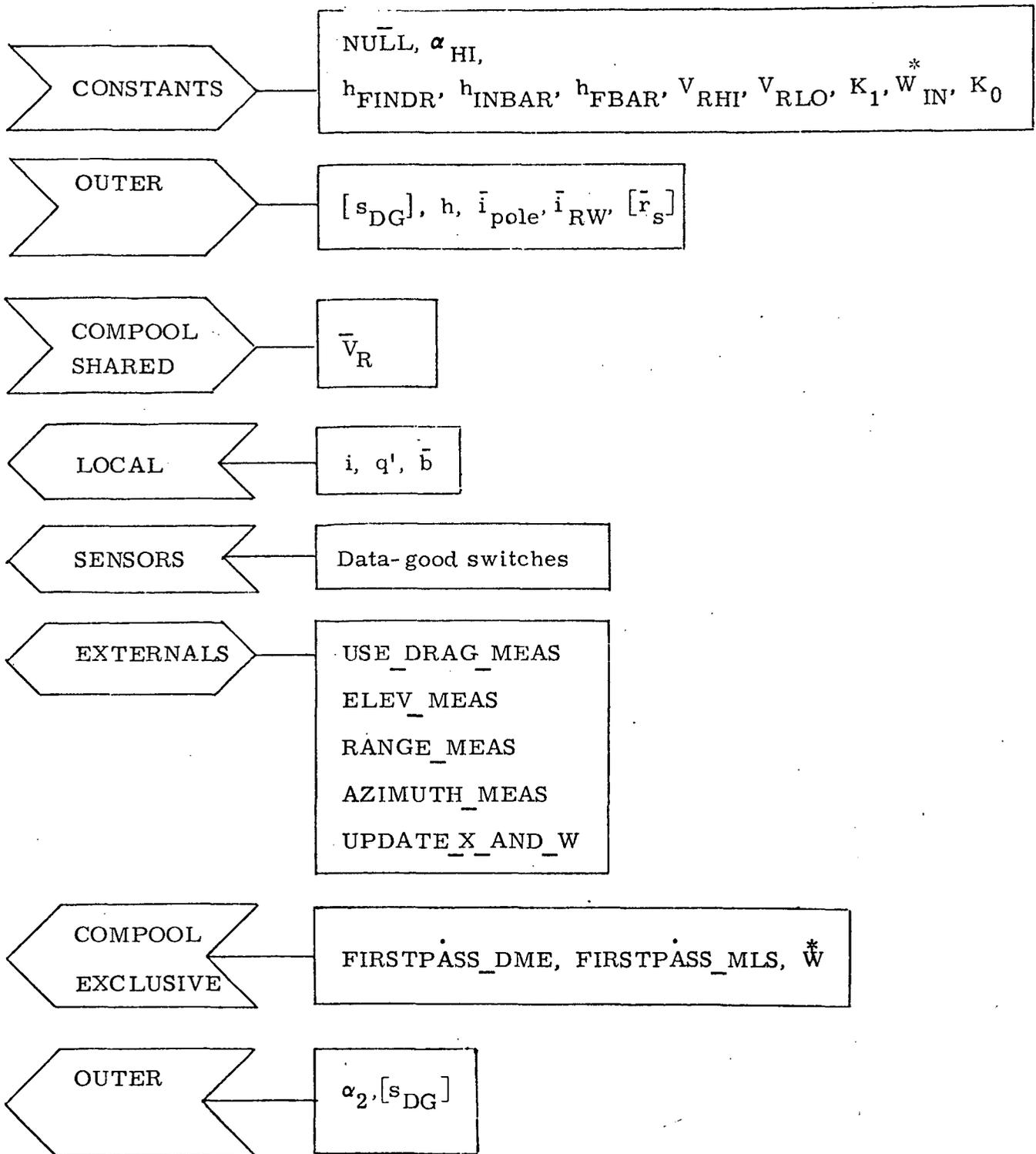


Fig. 1-10. USE_NAV_AIDS_TO_UPDATE_X_AND_W DATA MODULE

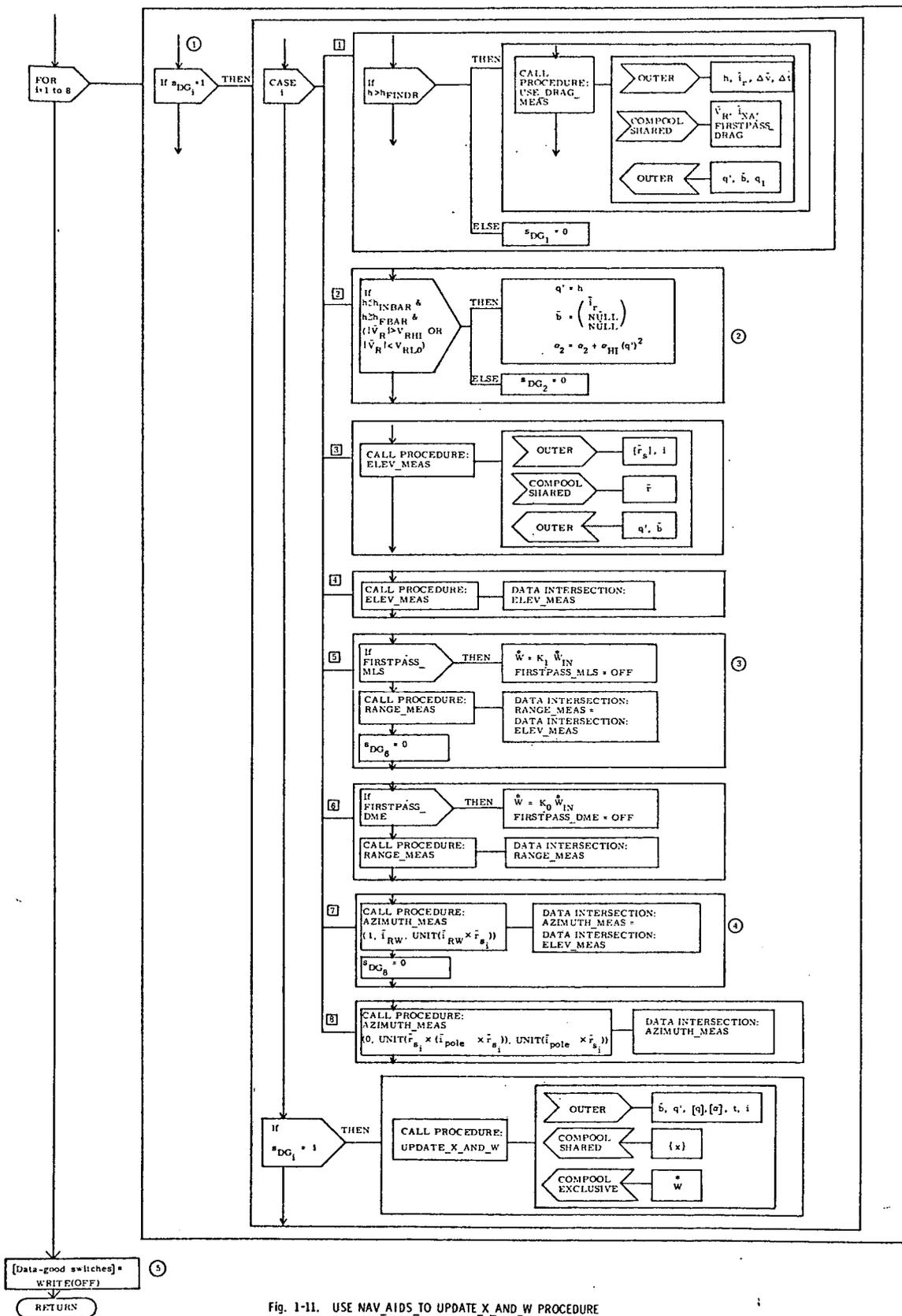


Fig. 1-11. USE NAV AIDS TO UPDATE X AND W PROCEDURE

Fig. 1-11 NOTES:

- ① S_{DG} for drag is never tested in original (c. f. Fig. 2c, Appendix 2). The specifications in the text indicate this test should be performed.
- ② The original indicates V_R without mentioning that it is the magnitude of the vector V_R .
- ③ $FIRSTPASS_MLS$ and $FIRSTPASS_DME$ are vaguely defined and never set to OFF in original document. (See discussion attached to Fig. 1-2)
- ④ $K_{AZ}, \bar{I}_C, \bar{I}_D$ are made into formal parameters for clarity. (c. f. Fig. 2g, Appendix 2 for original implementation)
- ⑤ The intent in the original document appears to be to turn off the data-good switches at the end of the cycle. Instead of scattering this information throughout the 8 cases, they are all turned off at once.

The context of S_{DG_i} is changed slightly from that presented in the original document. S_{DG_0} does not exist since a zero subscript reflects the MAC language and zero subscript does not exist in HAL. Therefore, S_{DG_i} is defined where $i = 1$ TO 8.

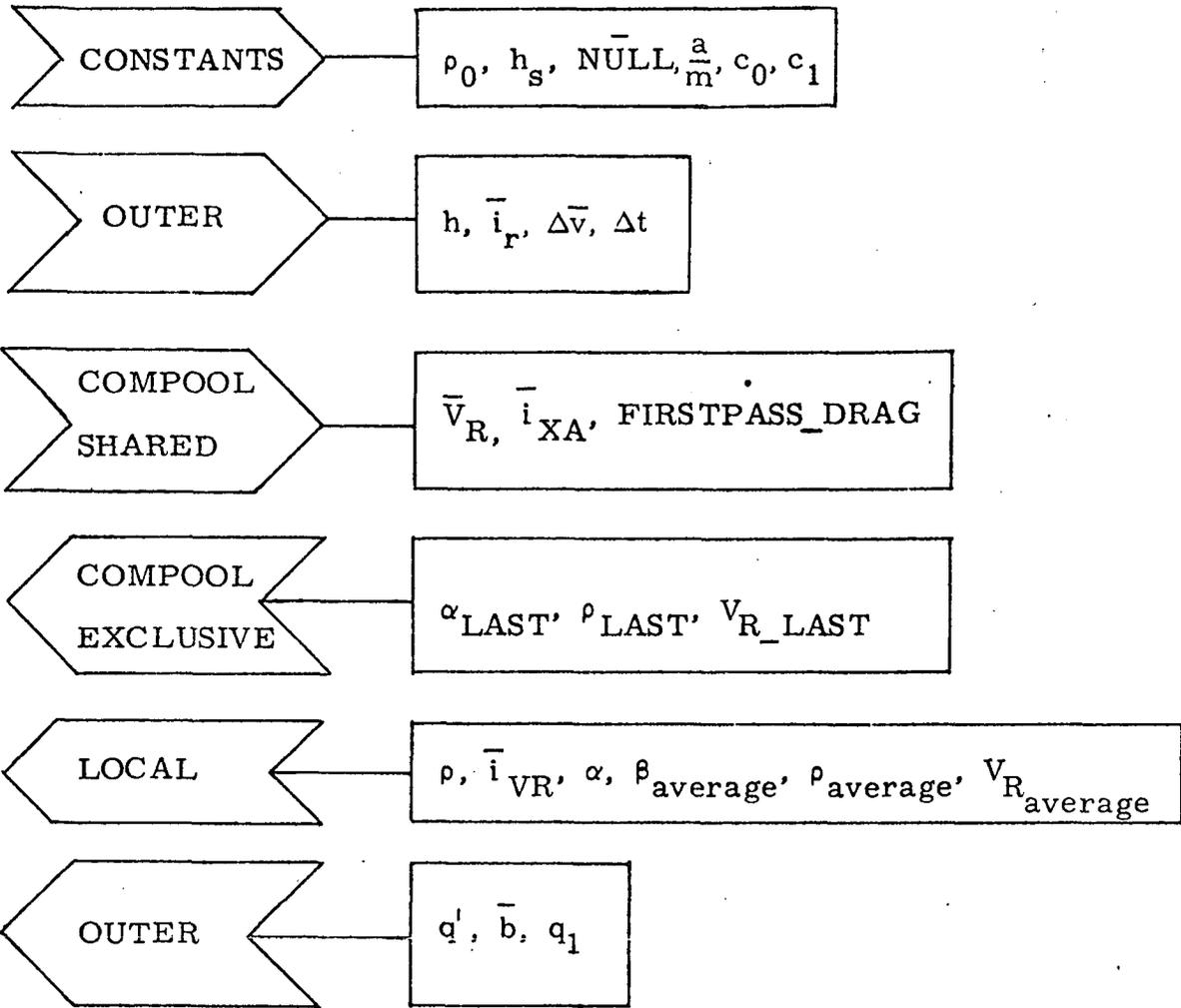


Fig. 1-12. USE_DRAG_MEAS DATA MODULE

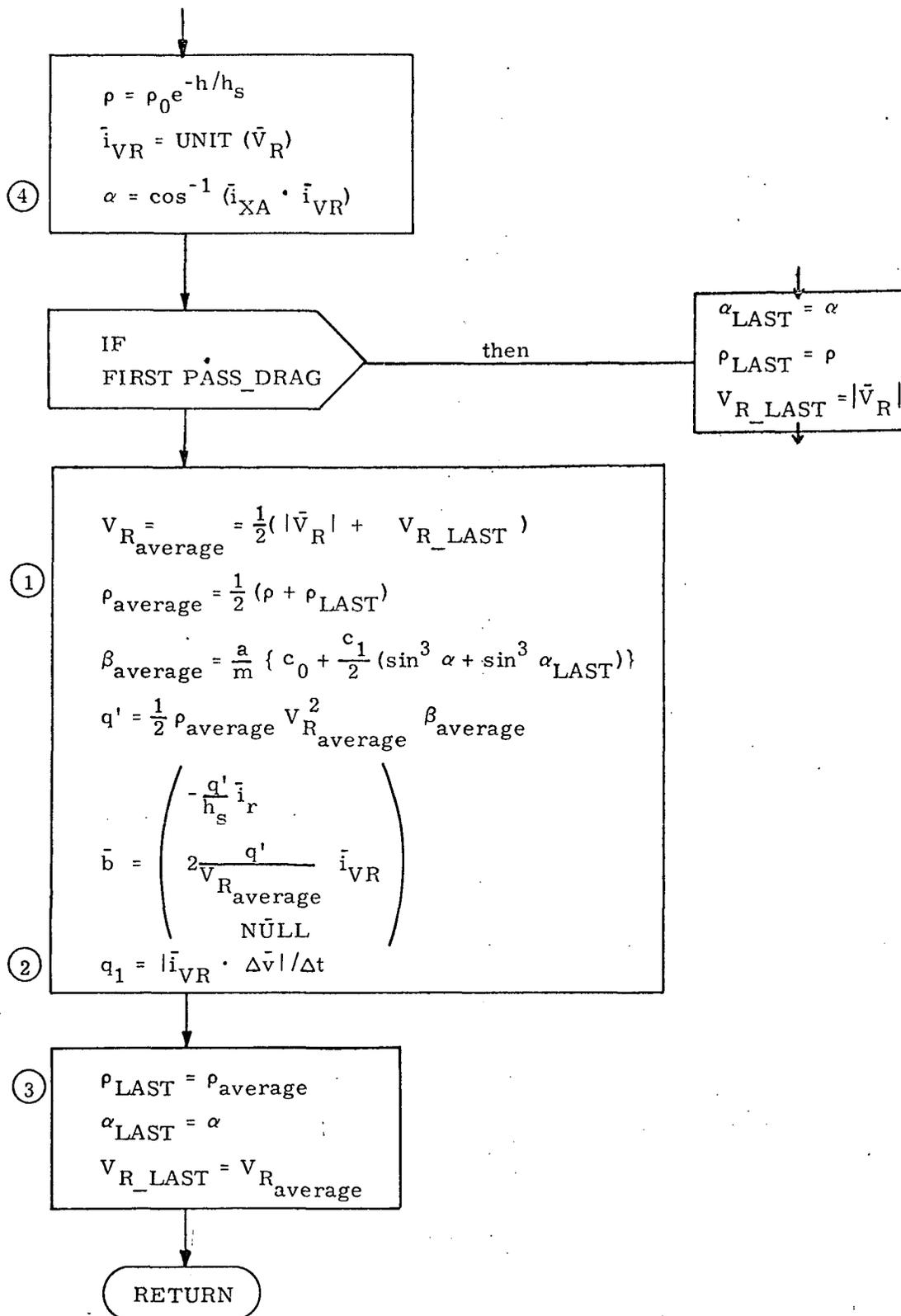


Fig. 1-13. USE_DRAG_MEAS PROCEDURE

Fig. 1-13 NOTES:

- ① Changed notation from \bar{f} to f average,
 \bar{V}_R to $V_{R\text{ average}}$
 \bar{f} to f average

This will avoid mistaking the notation in the original document for a vector.

- ② q_1 is not clearly assigned in the original flowchart. The specifications (p. 4-2) supplied the equation used here.
- ③ The assignment of these variables was omitted from the original document. (c. f. appendix 2, fig. 2c).
- ④ The data module for this PROCEDURE indicates that the α shown here is a local variable.

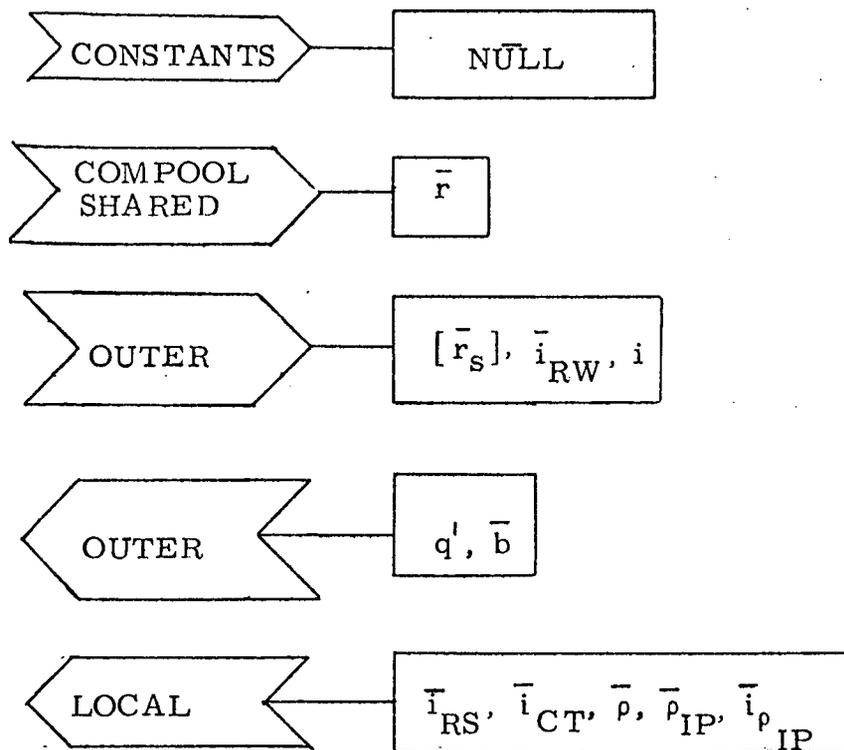


FIG. 1 - 14 ELEV_MEAS DATA MODULE

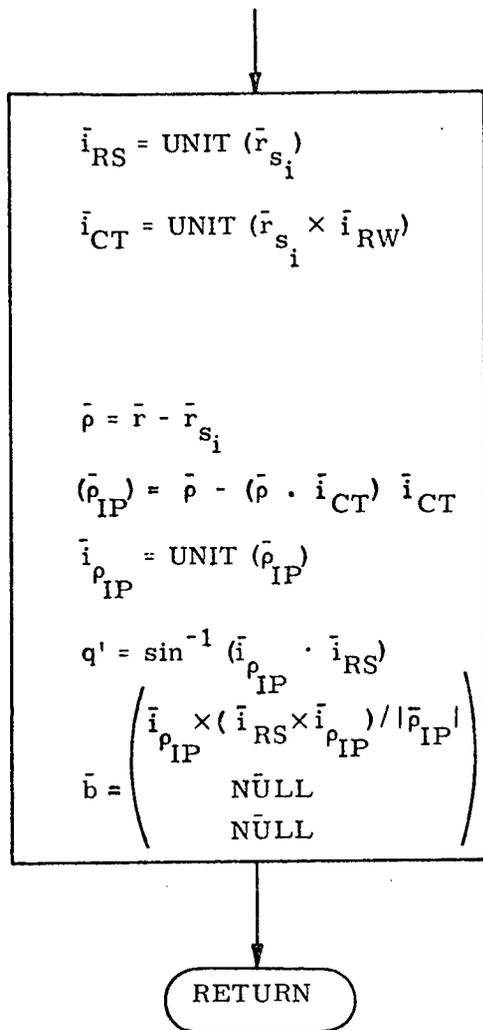


Fig. 1-15. ELEV_MEAS PROCEDURE

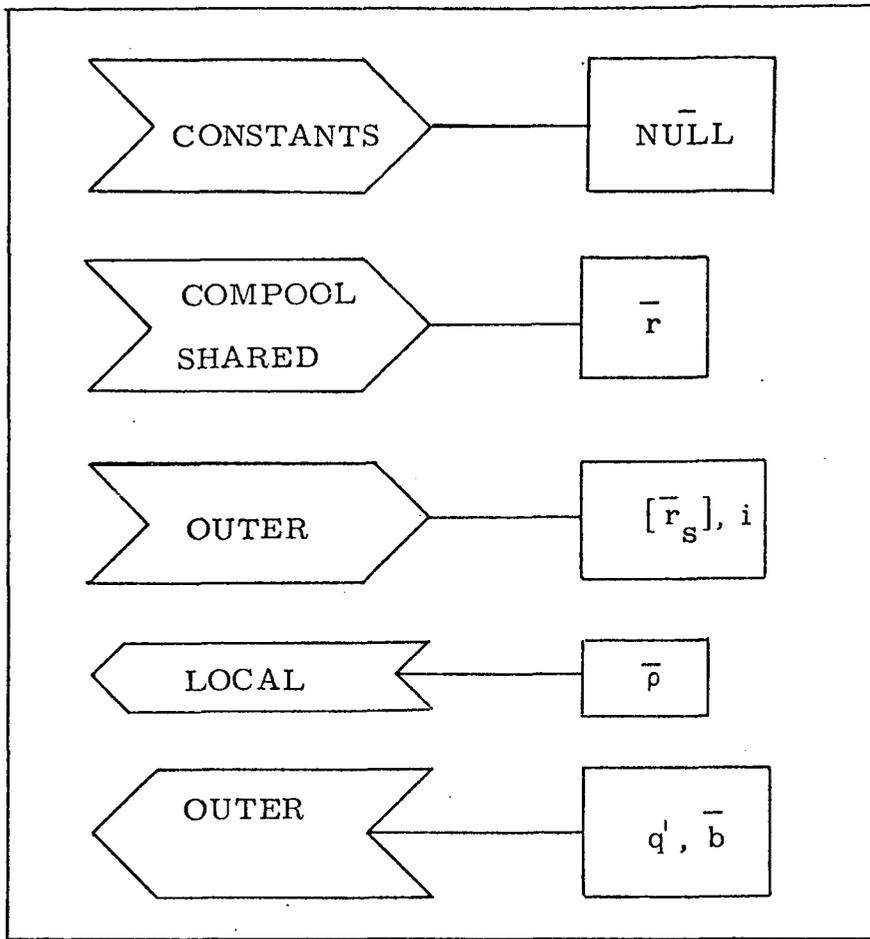


Fig. 1-16. RANGE_MEAS DATA MODULE

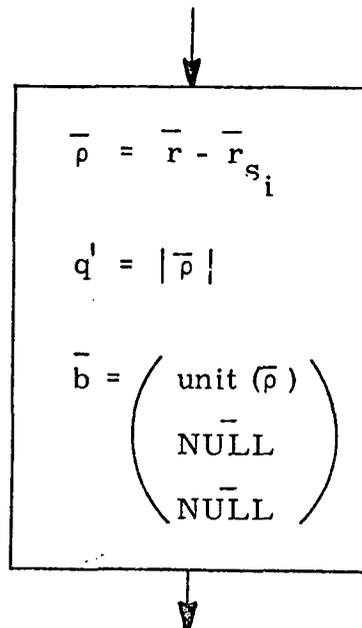


Fig. 1-17. RANGE_MEAS PROCEDURE

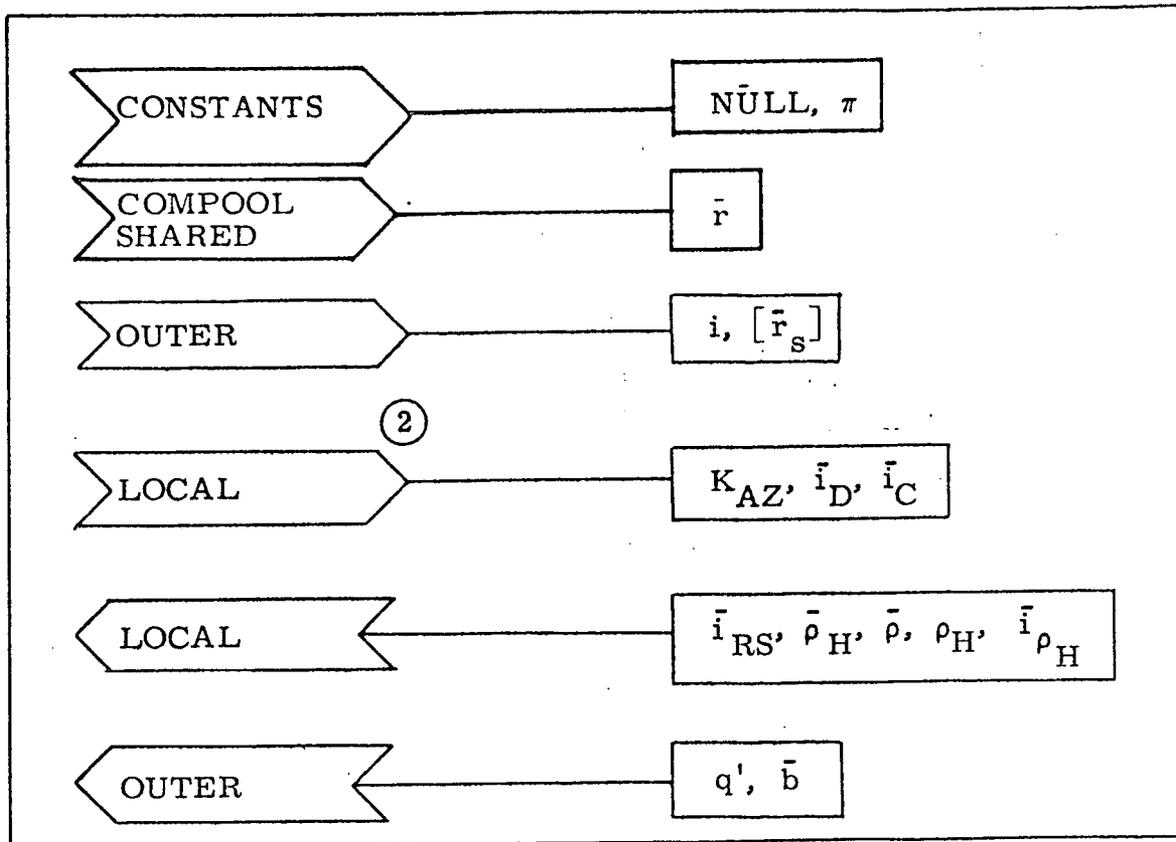


Fig. 1-18. AZIMUTH_MEAS DATA MODULE

$$\begin{aligned}
 \bar{i}_{RS} &= \text{UNIT}(\bar{r}_{s_i}) \\
 \bar{\rho} &= \bar{r} - \bar{r}_{s_i} \\
 \textcircled{1} \bar{\rho}_H &= \bar{\rho} - (\bar{\rho} \cdot \bar{i}_{RS}) \bar{i}_{RS} \\
 \rho_H &= |\bar{\rho}_H| \\
 \bar{i}_{\rho_H} &= \text{UNIT}(\bar{\rho}_H) \\
 q' &= \text{SIGN}(\bar{i}_{\rho_H} \cdot \bar{i}_D) [\sin^{-1}(\bar{i}_{\rho_H} \cdot \bar{i}_C) - \pi/2] - K_{AZ} \pi/2 \\
 \bar{b} &= \begin{pmatrix} (\bar{i}_{RS} \times \bar{i}_{\rho_H}) / \rho_H \\ \text{NULL} \\ \text{NULL} \end{pmatrix}
 \end{aligned}$$

Fig. 1-19. AZIMUTH_MEAS PROCEDURE ($K_{AZ}, \bar{i}_D, \bar{i}_C$) ^②

NOTE: ^① $\bar{\rho}_H$ and $\bar{\rho}$ equations are reversed in original (c.f. Appendix 2, Fig. 2g).

^② K_{AZ}, \bar{i}_D and \bar{i}_C are formal parameters indicated as local variables.

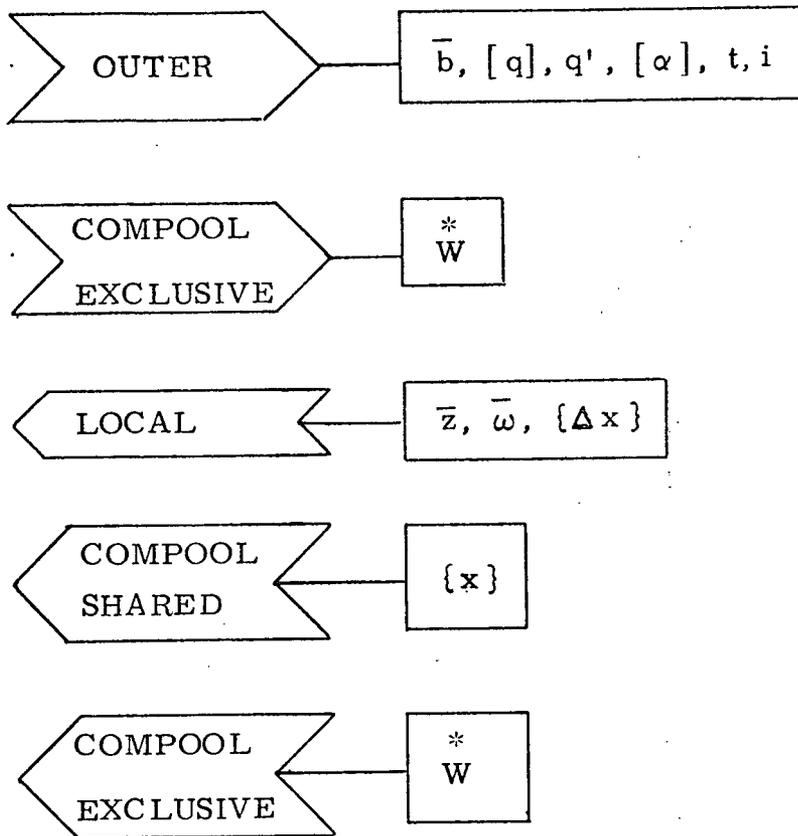
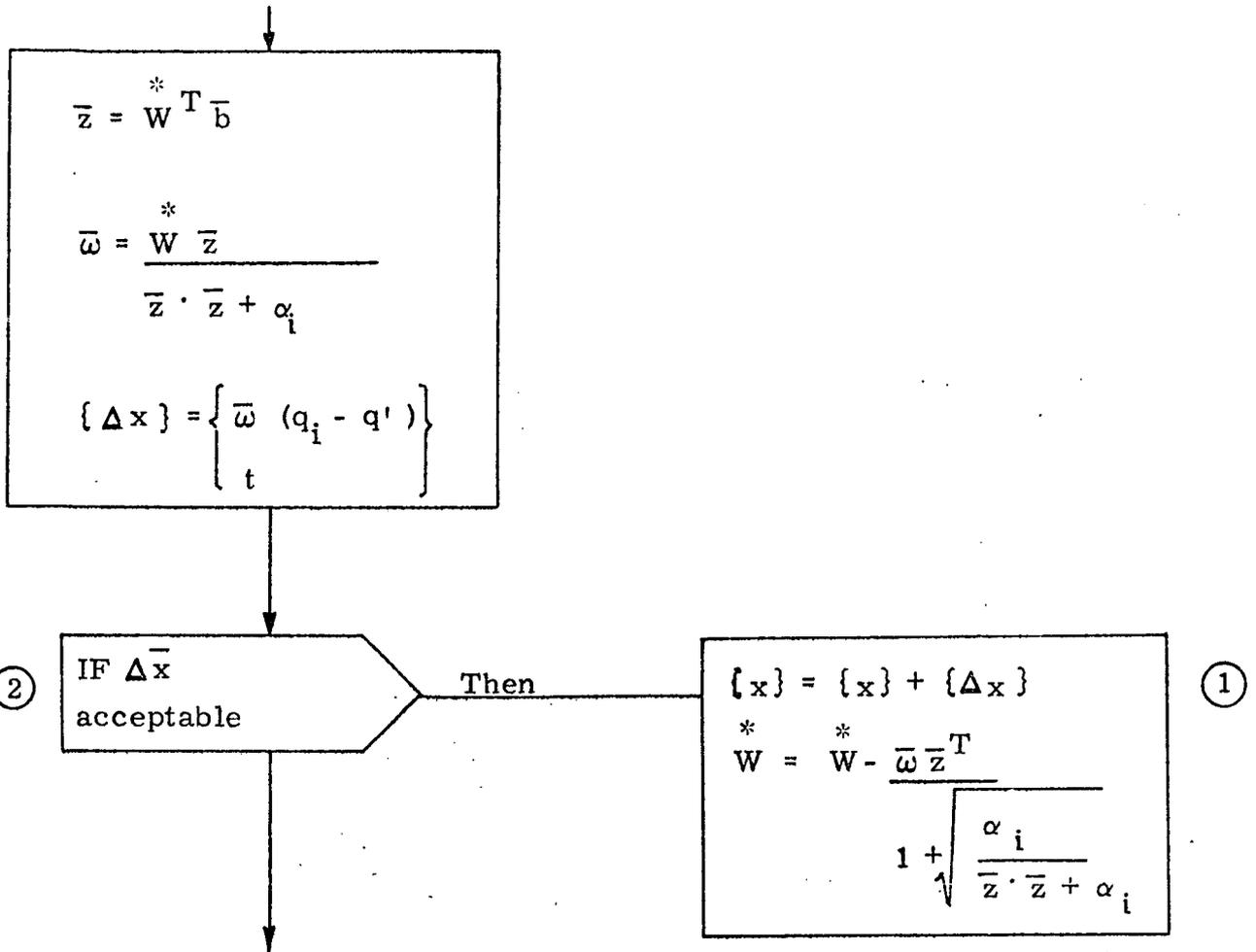


FIG. 1-20 UPDATE_X_AND_W_DATA MODULE



NOTE:

- ① The method of updating the state in the original is not necessary here. Also note that the time of the state was not updated in the original.
- ② This "acceptable" module is a pseudo-module at this level.

FIG. 1-21 UPDATE_X_AND_W PROCEDURE

APPENDIX 2

The following figures are included in this report for comparison with the figures in APPENDIX 1. They are reproduced from CSDL report, "Space Shuttle GN&C Equation Document No. 12 (Revision 2), Entry, Approach, and Landing Navigation".

Preceding page blank

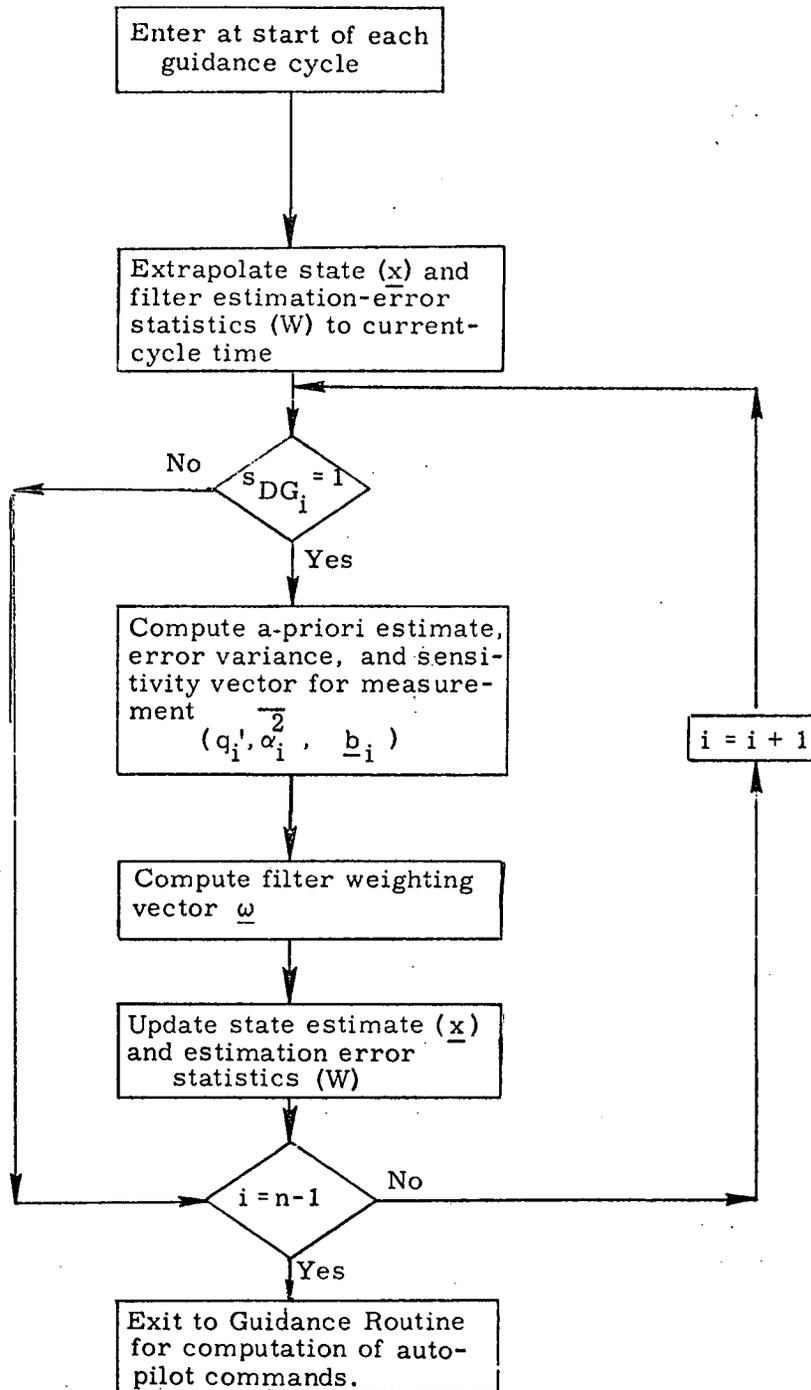


Figure 1. Entry Navigation Functional Flow Diagram

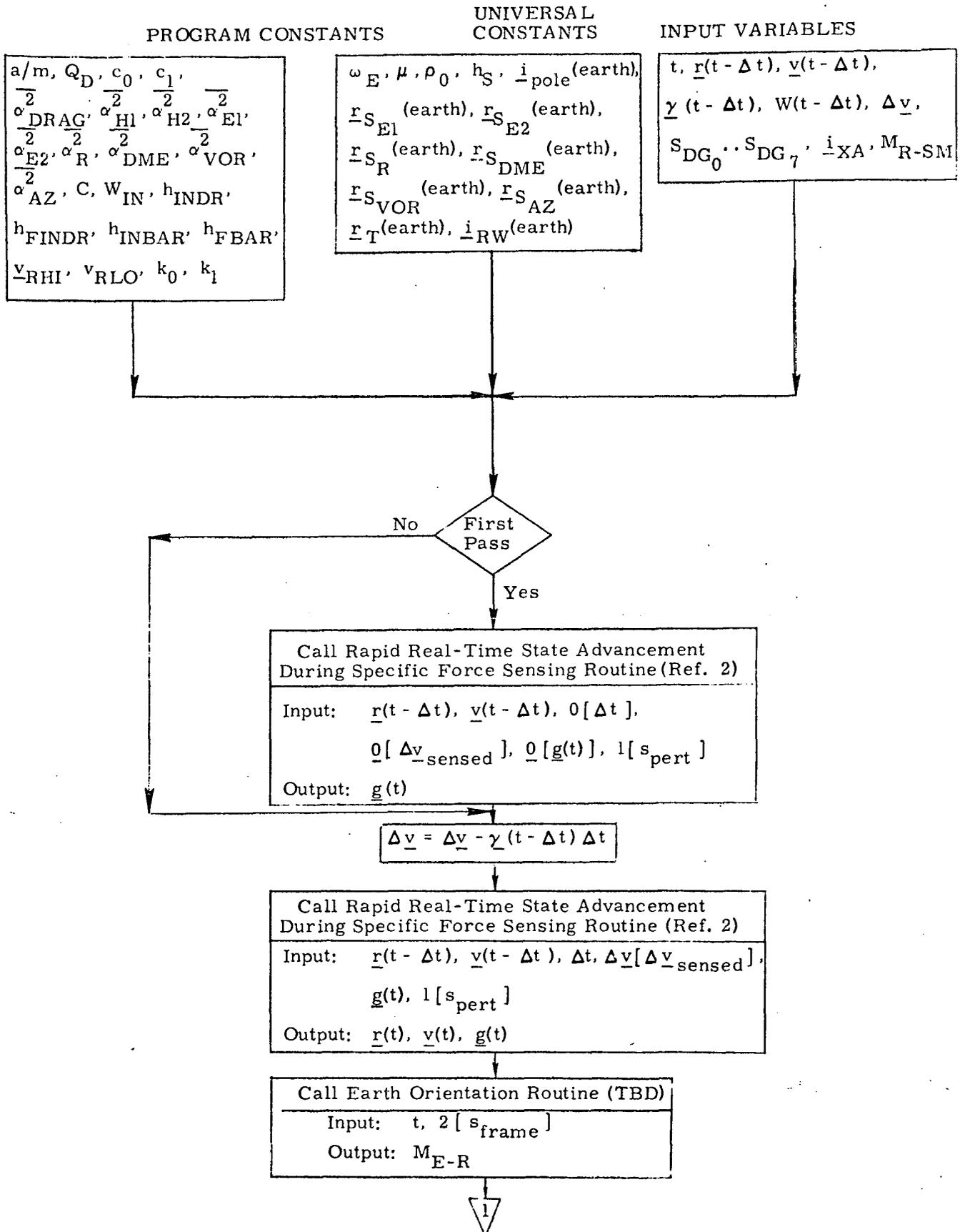


Figure 2a. Entry, Approach and Landing Navigation, Detailed Flow Diagram

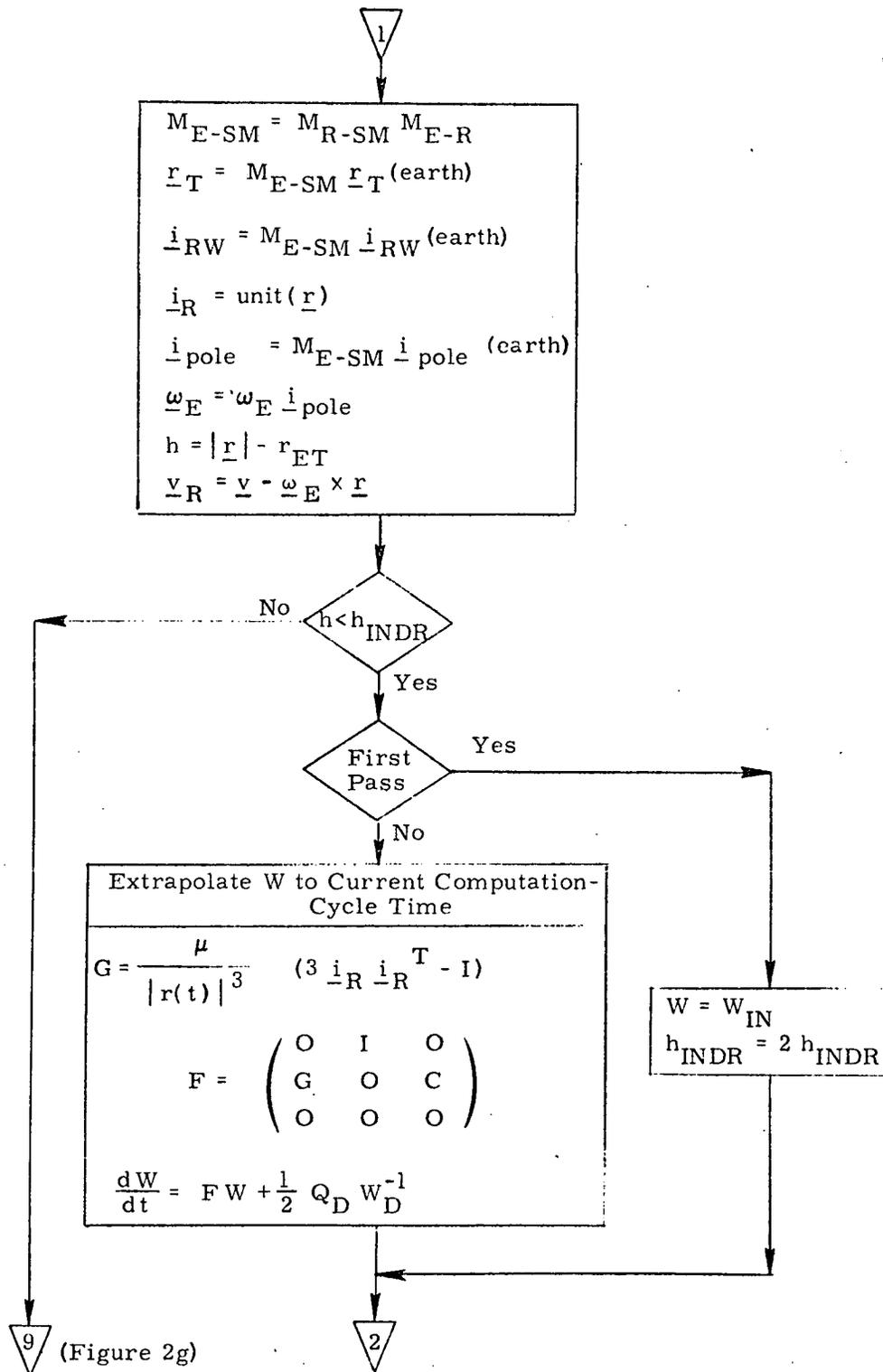


Figure 2b. Entry, Approach and Landing Navigation, Detailed Flow Diagram

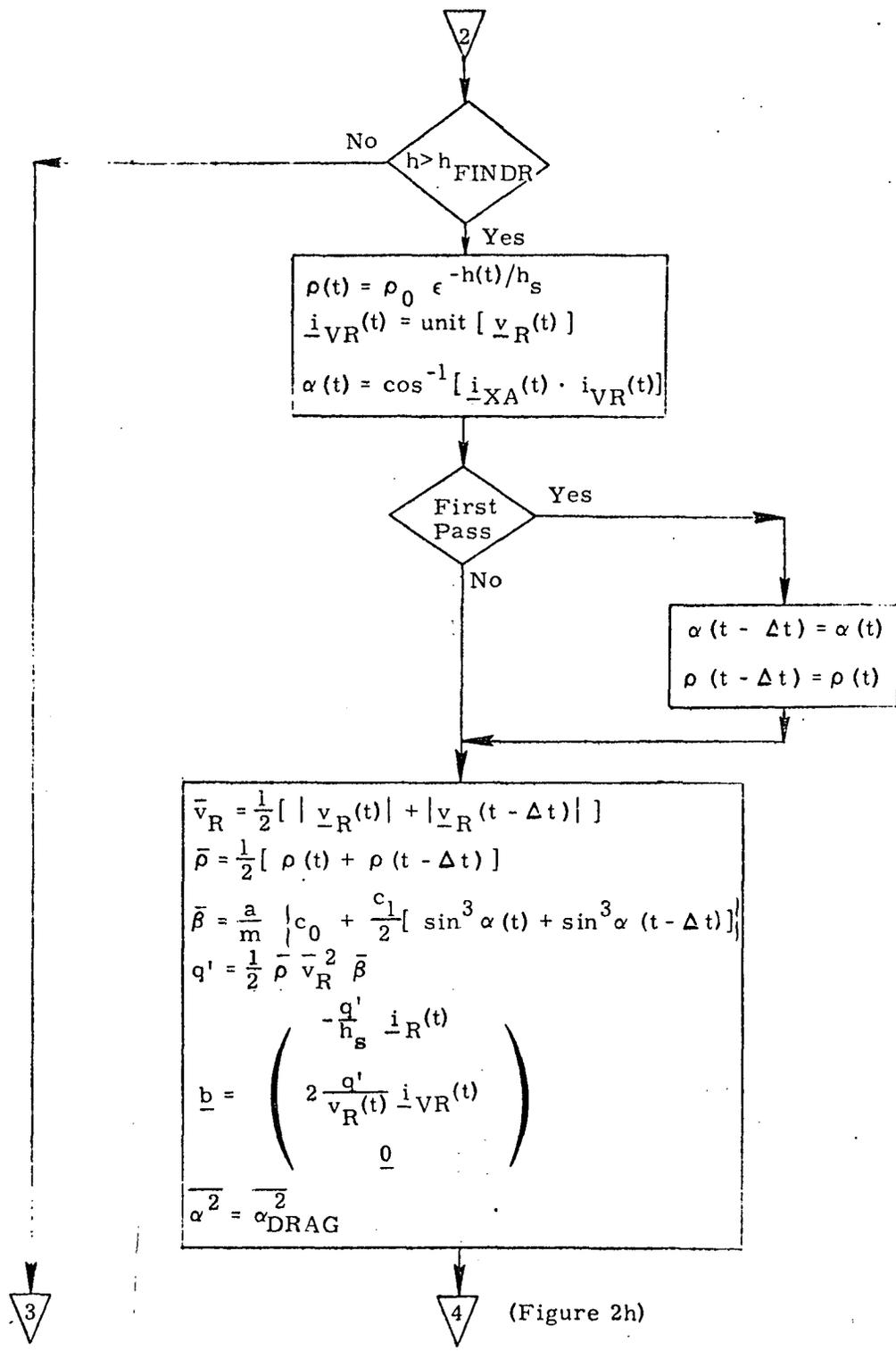


Figure 2c. Entry, Approach, and Landing Navigation, Detailed Flow Diagram

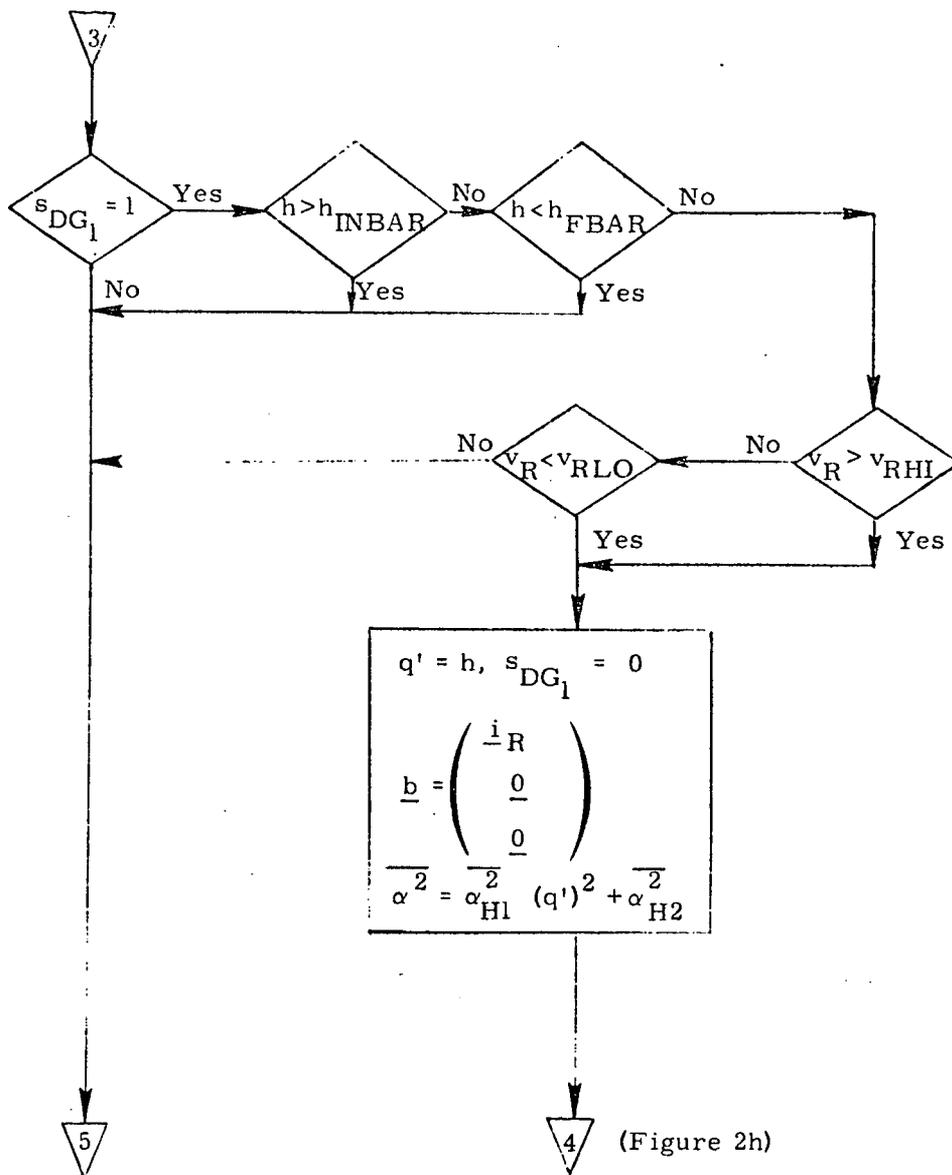


Figure 2d. Entry, Approach and Landing Navigation, Detailed Flow Diagram

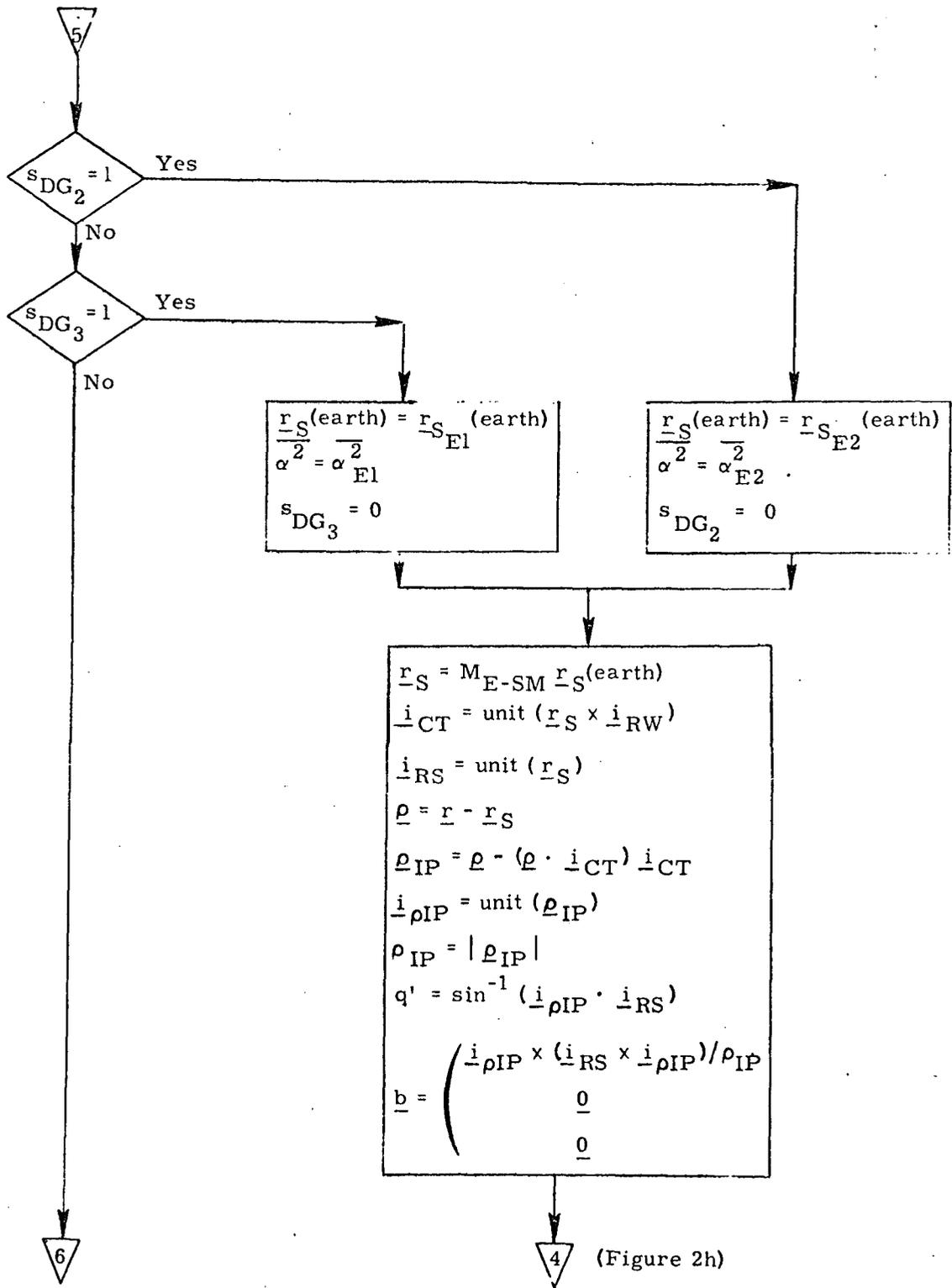


Figure 2e. Entry, Approach and Landing Navigation, Detailed Flow Diagram

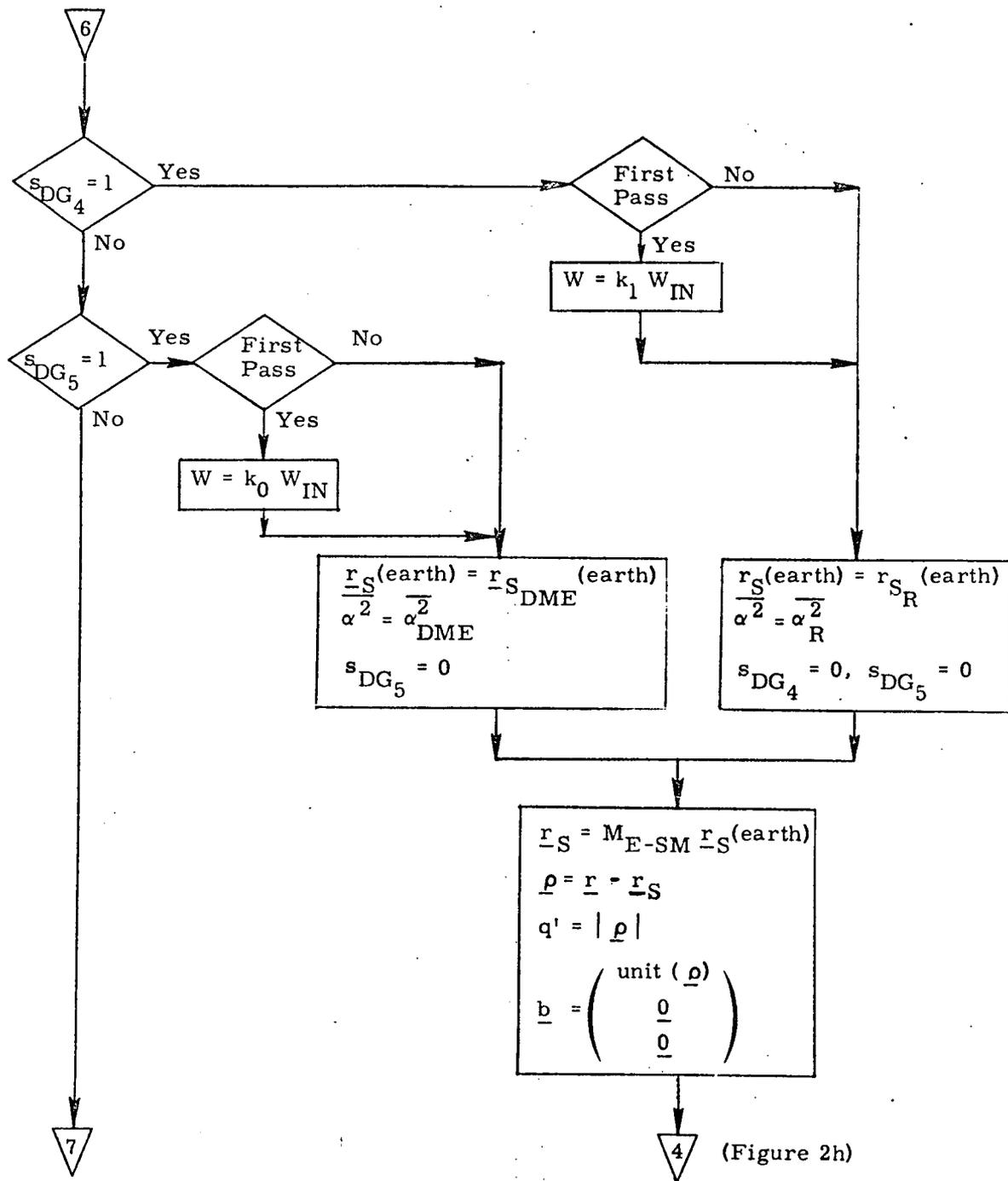


Figure 2f. Entry, Approach and Landing Navigation, Detailed Flow Diagram

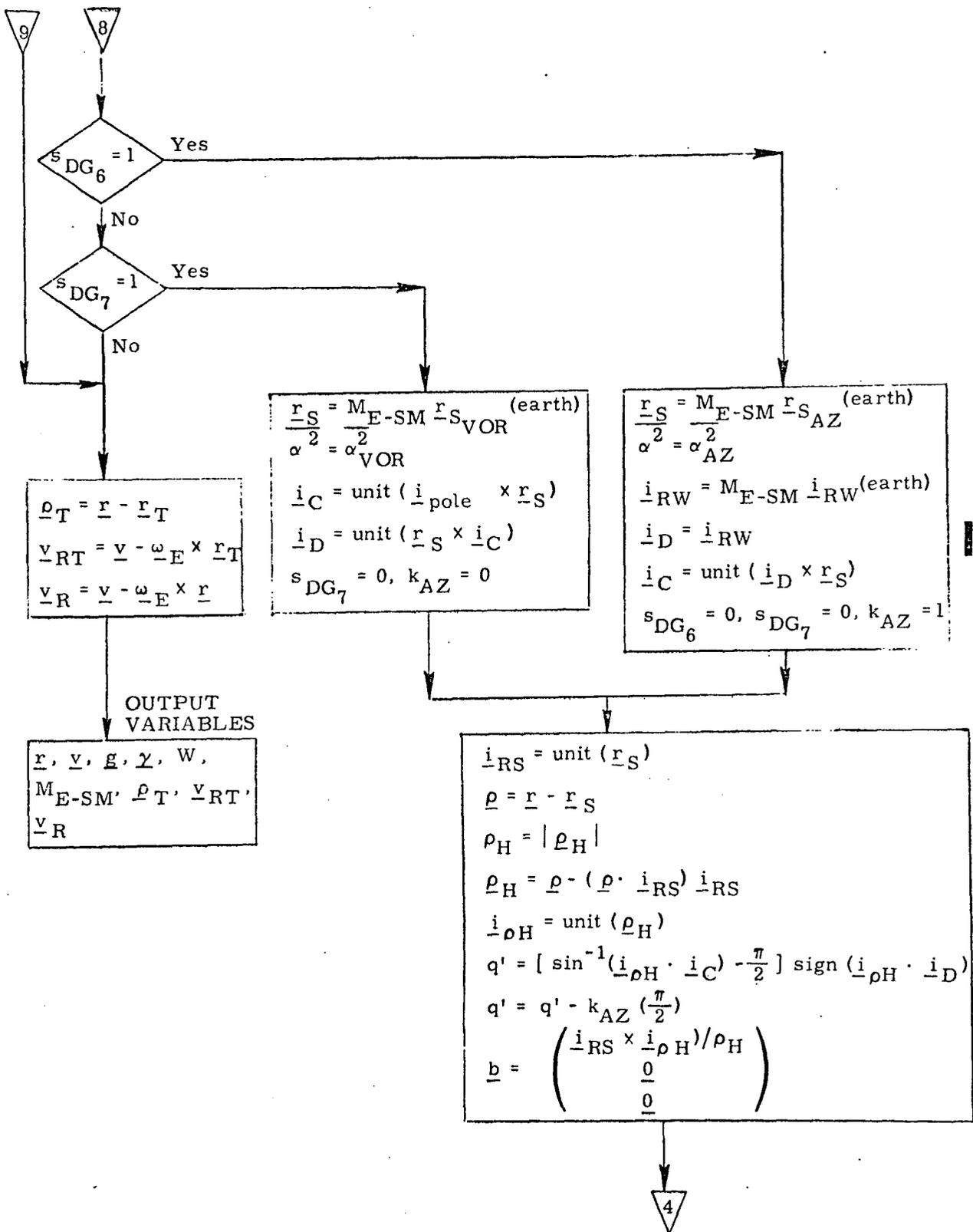
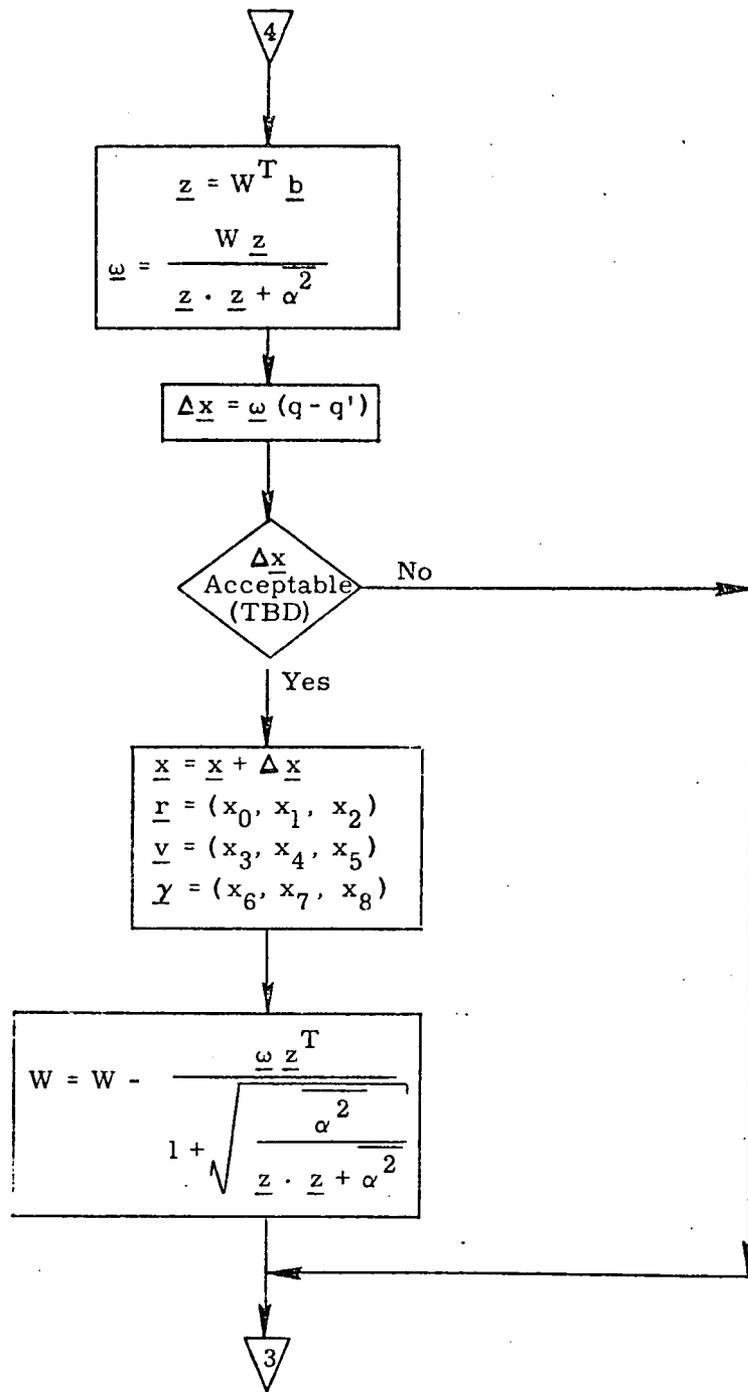


Figure 2g. Entry, Approach and Landing Navigation, Detailed Flow Diagram



(Figure 2d)

Figure 2h. Entry, Approach and Landing Navigation, Detailed Flow Diagram