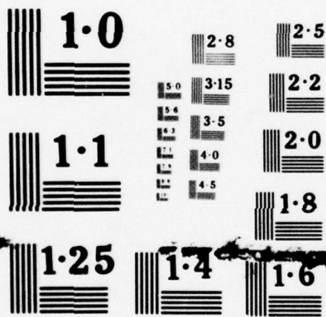
AD-A055 382 COMPUTER SCIENCES CORP HUNTSVILLE ALA F/G 9/2
HOL EVALUATION FOR MISSILE SOFTWARE APPLICATIONS.(U)
NOV 77 R PALMER, J W CRENSHAW, K D DANNENBERG DAAK40-77-C-0048
UNCLASSIFIED CSC/TR-77/5495 NL

1 OF 1
ADA
055382

END
DATE
FILMED
8 -78
DDC



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

This document has been approved for public release and sale; its distribution is unlimited.

AD A055382

CSC

COMPUTER SCIENCES CORPORATION

78 06 09 020

CSC

HOL EVALUATION FOR MISSILE SOFTWARE APPLICATIONS.

CONTRACT DAAK40-77-C-0048

Final rept.,

80 p.

Developed for
U.S. ARMY MISSILE COMMAND
GUIDANCE AND CONTROL DIRECTORATE
Missile Computer Software & Hardware Center
Redstone Arsenal, Alabama 35809

DDC
JUN 20 1978
F

CSC/TR-77/5495

8 NOV 1977

R. Palmer, J. W. Crenshaw, K. D. Dannenberg, D. R. Griffin

COMPUTER SCIENCES CORPORATION

515 Sparkman Drive, NW

Huntsville, Alabama 35806

This document has been approved for public release and sale; its distribution is unlimited.

Major Offices and Facilities Throughout the World

78 06 09 020

409 723

D.E. Copeland
Technical Monitor
MSL Computer Software and Hardware Center
Guidance and Control Directorate
U.S. Army Missile Command
Redstone Arsenal, Alabama

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION Per Ltr on file
BY
DISTRIBUTION/AVAILABILITY CODES
... /or SPECIAL
A
R78-0970

TABLE OF CONTENTS

Section 1 - Introduction 1-1

1.1 Scope 1-1
1.2 Background 1-2
1.3 Study Overview 1-6

Section 2 - HOL Life Cycle Economics and Effectiveness 2-1

2.1 Why Use an HOL? 2-1
2.1.1 Design Phase 2-1
2.1.2 Implementation Phase 2-2
2.1.3 Checkout/Testing Phase 2-3
2.1.4 Maintenance Phase 2-3
2.1.5 Arguments Against HOLs 2-4
2.2 When to Use an HOL 2-6
2.2.1 Analysis of ΔD (Development Cost Change) 2-8
2.2.2 Analysis of ΔM (Maintenance Cost Change) 2-8
2.2.3 Analysis of F (Front-End HOL Expenses) 2-9
2.3 Algorithm Alternatives 2-10

Section 3 - Requirements for Candidate HOLs 3-1

3.1 Approach 3-1
3.2 Missile Systems Requirements 3-1
3.2.1 Analog Autopilots 3-1
3.2.2 Function Generation 3-5
3.2.3 Preset Guidance 3-8
3.2.4 Timing 3-8
3.2.5 Targeting 3-10
3.2.6 Rotational Equations 3-14
3.2.7 Kalman Filtering 3-16
3.2.8 Data Representation 3-17
3.2.9 Summary 3-17
3.3 General Language Requirements 3-20

Section 4 - Candidate Language Evaluations 4-1

4.1 Language Requirements Analysis 4-1
4.2 Compiler Analysis 4-3
4.2.1 CMS-2 4-4
4.2.2 J3B 4-4
4.2.3 J73/I 4-5
4.2.4 TACPOL 4-5
4.2.5 SPL/1 4-6

TABLE OF CONTENTS (Continued)

4.2.6 Summary 4-6

Section 5 - Conclusions 5-1

Appendix A - Analysis of IRONMAN Requirements

Appendix B - Glossary

References

LIST OF FIGURES

Figure

3-1	Simple Autopilot	3-1
3-2	Command Guidance	3-3
3-3	Digital Version of Figure 3-1	3-3
3-4	Simple Limiter	3-6
3-5	Bang-Bang Switching Function	3-7
3-6	Simple Homing Guidance	3-11
3-7	Proportional Navigation	3-12
3-8	Covariance Guidance	3-13

LIST OF TABLES

Table

4-1	Summary of Analysis of Five Languages	4-2
4-2	Summary of Compiler Analysis	4-6
5-1	HOL Ranking	5-1

PREFACE

This report documents the results of a study tasked under Contract DAAK40-77-C-0048. Acknowledgement and appreciation for information furnished on existing compilers go to the following:

CMS-2	Lester Mayfeld, UNIVAC
J3B	John Walsh, Softech
J73/I	Terry Dunbar, Software Engineering Associates
TACPOL	Jamie Ritter, Litton
SPL1	Dr. James Pepe, Intermetrics

SECTION 1 - INTRODUCTION

1.1 SCOPE

This report is the result of Task Order 007 to Contract DAAK40-77-C-0048. The scope of the task is:

1. Identify candidate Higher Order Languages (HOLs) which are applicable to the missile systems environments, i.e., Pershing II and Digital Autopilot. All available versions of JOVIAL shall be included in the candidates. Characteristics of the candidate HOLs shall also be identified and shall include the following features.

   - o Language including: Structured capability, data types, compilation features, common data, COMPOOL, etc.
   - o Translator including: Error diagnostics, debug aids, efficiency, size, etc.
   - o Management including: Standardization, legacy, rehosting costs, re-targeting costs, etc.

2. Determine the life cycle economic and effectiveness benefit of the candidate HOLs as compared to assembly language. This comparison shall consider such attributes as:

   - o Development Environment
   - o Maintenance Environment
   - o Optimization
   - o Memory Utilization
   - o Performance

3. Determine and rank, using a matrix technique, the optimum HOL version(s) for identified environment and include consideration of design and implementation methodology, reliability history, efficiency, existing code generators, implementation cost/schedule, and available support/ V&V tools.

1.2 BACKGROUND

In recent years there has been considerable interest by DOD in the streamlining of the software development process. This has been brought about by several trends:

- The trend away from analog guidance and control techniques to digital systems.
- The increasing use of "embedded" digital systems; i.e., dedicated processors within other hardware.
- Multiprocessor or distributed processor techniques using microprocessors.
- Increasing complexity of both digital avionics systems and ground test or support systems.
- The decreasing cost of microprocessors and other digital hardware.
- The increasing costs of software development.
- The realization of the true costs associated with software maintenance.
- The emphasis upon total life cycle costs as the measure of system cost-effectiveness.
- Increased concern with software reliability.

As a result of these trends, the cost of software development and maintenance has become a significant, and often controlling, factor in overall life cycle cost. To control this cost, the following actions have been taken by the DOD community:

- To reduce the need for rewriting software, effort has been initiated to define a "Software Compatible" family of computers. This family will be upward compatible, so that software written for one member of the family can be run on any larger member. At present the instruction sets for the family are still being defined. Recognizing that the software cost is the key feature, the family is being designed from software rather than hardware considerations. It is hoped that this

approach will keep the family viable in the face of technology advances.

- It has been recognized that the cost of software development (and more so for maintenance) is reduced through the use of HOL. Execution efficiency is often a key factor in DOD systems. However, it has been realized that it is often cheaper to procure more powerful hardware than to resort to Assembler-language programming in order to accommodate marginal hardware performance. As a result, the use of HOL for DOD software is expected to increase greatly.

- A significant deterrent to wider use of HOL is the disturbing proliferation of HOL languages and dialects. This situation tends to negate much of the portability of HOL-written software. To combat this effect, the High Order Language Working Group (HOLWG) was formed in January 1975.

The HOLWG consists of representatives from the three services, plus the Office of the Assistant Secretary of Defense (OASD). This office will coordinate language specifications for the two languages with the American National Standards Institute (ANSI) and National Bureau of Standards (NBS). The HOLWG is engaged in the definition of a new, multi-purpose HOL called DOD-1. The intent is that the services will adopt DOD-1 for all new software efforts by the mid-1980's. Through DOD directives 5000.29 and 5000.31, the HOLWG has issued a list of seven interim approved HOLs which are: FORTRAN, COBOL, JOVIAL J3, JOVIAL J73, TACPOL, CMS-2, and SPL-1. Once DOD-1 is operational (approximately the end of 1980) it will be added to the approved list, and phaseout of the other languages will begin.

The HOLWG also proceeded to develop specifications for DOD-1. A tentative specification, "Strawman" was written and circulated among the various agencies. After several iterations, these specifications were refined into a "Woodenman" and subsequently "Tinman." The Tinman specifications consist of 98 specific requirements.

Although it was virtually certain that no existing language could meet the requirements of Tinman, it was desired to select an existing language which could (hopefully) serve as a basis for development. To this end, six contractors were selected (two by each service) to evaluate 23 candidate languages against the Tinman. The six contractors selected were:

Softech CSC	Army
Intermetrics RLG	Navy
IBM SAI	Air Force

The evaluated languages are listed below:

FORTRAN:	Developed by IBM in 1954 - 58
COBOL:	Business data processing language developed in 1959 - 61
JOVIAL J3B:	Air Force language developed in 1972
JOVIAL J73:	Air Force language devcloped in 1969 - 73
TACPOL:	Army language developed for TACFIRE in late 1960's
CMS-2:	Navy language developed in 1966 - 69 by CSC
SPL-1:	PASCAL-based NRL real-time signal processing language
PL/1:	Developed by IBM. Includes concepts from FORTRAN, COBOL and ALGOL
HAL/S:	PL/1-based, NASA language developed for Shuttle
C/S-4:	Intermetrics PASCAL-based language, with real-time and extension facilities
ALGOL 60:	Block structure language developed in 1957 - 60
ALGOL 68:	A successor of ALGOL 60, emphasizing generality
PASCAL:	A successor of ALGOL 60 emphasizing simplicity

CORAL 66:	British common language for real-time applications
MORAL:	New British language for embedded computer applications
SIMULA 67:	Simulation language developed in Norway
LIS:	PASCAL-based French system implementation language
LTR:	PASCAL-based official French common language
RTL/2:	Real-time British language developed at ICI
PEARL:	PL/1-based German process control language
EUCLID:	PASCAL-based experimental language emphasizing verification
ECL:	Harvard extensible language with good support environment
PDL/2:	PASCAL with parallel processing independent module facilities, developed by Texas Instruments

On 14 January 1977, the Language Evaluation Coordinating Committee issued a report to the HOLWG summarizing the results of the evaluations. Of the 23 languages, it found nine to be unacceptable as base languages: FORTRAN, COBOL, J3B, J73, TACPOL, CMS-2, ALGOL 60, CORAL 66, and SIMULA 67. (It is interesting that of the seven interim languages, six are considered unacceptable as a base for DOD-1.) The committee recommended three languages as potential bases: PL/1, ALGOL 68, and PASCAL.

Following the various evaluations of the languages, a workshop on Tinman was held at Cornell Univeristy (September 1976). As a result of this workshop, the requirements of Tinman were rewritten into a new specification, "Ironman." The major part of the rewrite, however, had to do with organizing the requirements more in keeping with contractor language description documents. The requirements themselves are essentially unchanged.

Until such time as a functional compiler for DOD-1 becomes available, the Government requires a HOL compiler capable of supporting software for embedded missile system applications such as Digital Autopilot (DAP) and Pershing II. It is desired that this compiler be based upon one of the DOD interim languages.

The purpose of the current study is to evaluate candidate HOLs for short- and medium-term applications in missile systems.

1.3 STUDY OVERVIEW

The applicability of HOLs to missile systems software is to be analyzed for the immediate future (prior to the implementation of DOD-1. All of the HOLWG approved interim languages except FORTRAN and COBOL are considered in this study.

In Section 2 a general discussion of HOLs and the implications of their use are presented. The issue of language "efficiency" is considered in a general context, making the argument that program efficiency is best achieved by proper algorithm design and improvement of critical sections of code. Subsequently, questions concerning reliability and performance are analyzed. A discussion is given of economic issues involved in a choice between high order languages. The parameters to the education are budget estimations of software development costs at various stages in the project's life-cycle. A sensitivity analysis is performed on the equation.

An analysis of software life-cycle costs in light of the high-order language decision is presented. The indications are that a substantial savings may be expected if a high-order language is used, with the rate of savings increasing as the project matures.

In Section 3, the requirements for an HOL for missile systems programming is analyzed. Typical missile system applications are discussed, and the functional requirements defined. Language requirements are defined, based upon the IRONMAN specifications for DOD-1.

In Section 4, five candidate languages are evaluated against the requirements. These languages are CMS-2, JOVIAL J-3B, JOVIAL J-73/I, TACPOL, and SPL/1.

A summary of the results is presented in matrix tabulated form. The conclusion of the study is that JOVIAL J73 appears to be a viable interim high-order language for missile system applications from the following view points:

- For programs with more than $300K allocated to software development, an HOL will be economically effective
- For the development of a missile system applications compiler, J73 has a slight technical edge over the other candidate languages
- J73 compilers exist which offer a better base and at least as economic an approach for retargeting and rehosting as any other candidate compiler.

SECTION 2 - HOL LIFE CYCLE ECONOMICS AND EFFECTIVENESS

2.1 WHY USE AN HOL?

One of the earliest software controversies was the debate over assembly language versus HOL. HOLs were introduced to solve the problems of complexity and error that seemed to haunt assembly language code. Further, since they were thought to be "simpler," it was supposed that such languages would make computers available to a much wider audience. The proposition of increased HOL reliability was never much in dispute; the question was always the cost.

HOLs are used for many reasons. Various studies have indicated that, in general, they seem to be easier to learn, to program in, and to debug than computer programming languages that are not HOLs. This seems to be due to their simpler semantic base and their orientation toward algorithms rather than machines. Programs written in HOLs are usually simpler to maintain. Again, this seems to be due to their algorithmic orientation. Programs come closer to representing the algorithm than the machine's representation of the solution. All of the attributes mentioned above serve to shorten the time required to develop, code, debug, and maintain programs using an HOL.

On the other hand, opponents of the use of HOLs point to the sacrifices in execution time and memory utilization which are required. Ultimately, the choice between the use of a HOL or machine-oriented language (MOL) reduces to one of minimizing overall life cycle costs. Software life cycle costs may be separated into four primary phases--design, implementation (coding), checkout/testing, and maintenance. Use of an HOL versus machine-oriented language (MOL) can and does have impact in each of these phases although to varying degrees.

2.1.1 Design Phase

The design phase seems to be least affected by use of an HOL as one would expect. However, there is some indication that, as familiarity with an HOL and its usage

increases, the design time indeed decreases. This reduction seems most significant if the design is expressed in terms of HOL algorithms. Admittedly, there is a fuzzy line here between where design stops and implementation begins, even more than with MOLs. However, regardless of this separation, it is apparent that expression of algorithms (and data) in the HOL provides for clearer communication of the design and reduced overall cost. This is particularly important where implementation is not performed by the designing personnel or where time has elapsed between design and implementation.

2.1.2 Implementation Phase

It is within the implementation phase that the most dramatic reduction percentagewise results from HOL usage. Explanation of this reduction seems clearly attributable to the number of source "statements" that must be written in the two languages. Although quantification is muddied by macro facilities at the MOL level and by the unpredictable complexity of the HOL statements, a good rule of thumb is that five MOL statements must be written for every HOL statement. Clerical type errors are reduced proportionately by this source volume reduction. A less obvious and less significant error reduction is obtained from the readability of the HOL statement versus the MOL statement; the typically more familiar syntax of the HOL statement makes many clerical errors conspicuous to the writer. In addition, a whole level of detail in the implementation is obviated by the use of a compiler. This is especially important for machines that possess difficult instruction level characteristics, such as instruction scheduling (CDC 6600), incomplete addressing (IBM 360), inconsistent instruction options (many machines), special registers rather than general purpose, etc. Although these complications affect compiler construction and cost in the same manner as programming costs, with the use of an HOL, these problems need only be faced by the compiler builders who, in general, are very senior specialists experienced with dealing with hardware ideosyncrasies.

2.1.3 Checkout/Testing Phase

A significant benefit of HOL usage is realized during the checkout phase. An important consideration here is availability of an integrated HOL facility which provides a symbolic, language level checkout/testing capability. The readability of the programs and the density of the HOL operations permit quicker comprehension of the total picture, manifests the program logic, and hence, promotes more productive checkout.

The typical cost and schedule constraints placed on software developments apply even more pressure on the MOL programmer than one using an HOL. The additional volume of source and increased level of detail necessitated by the MOL compromises the attention that may be concentrated on algorithms and efficiency. In addition, the increased level of errors and checkout time associated with MOL usage infringes upon the time allotted to program tuning.

An often overlooked factor alluded to above is the experience disparity between the typical compiler developer and application programmers. Compiler developers are usually specialists in instruction and instruction sequence generation. Consequently, a compiler is apt to produce more clever local code than the average programmer.

2.1.4 Maintenance Phase

The largest software cost reduction associated with HOL usage ensues during the maintenance phase. Program reliability is definitely enhanced by HOL usage. By eliminating many types of errors, by decreasing program source volume and proportionately decreasing all errors, by improving checkout productivity, and by increasing testing effectiveness, software products are delivered for less cost, earlier, and with greater stability. Concomitant with this increased reliability is an obvious decrease in maintenance requirements.

An important factor contributing to maintenance cost in general is the introduction of new, often relatively inexperienced personnel for this task. The readability of an HOL program versus one written in an MOL affords an immediate cost reduction by facilitating the training of the new staff. A performance benefit of HOL language and compiler usage that is difficult to quantify surfaces as programs undergo modifications whether performed during checkout or maintenance. With the exception of those changes aimed

specifically at performance improvement, MOL changes tend to degrade the performance of programs, whereas HOL changes tend to have little effect. This relative degradation appears to result from the fact that MOL fixes are less complete in terms of the overall coding, that MOL fixes tend to cancel, indeed they often even conflict with many of the original optimizations designed for the earlier structure. This is particularly true of register allocation. In contrast, a compiler will provide the same level of optimization for every compilation and determine register allocation according to the needs of each version of a modified program. A similar side-effect results from the reduced documentation requirement for HOL programs (at least a reduced detail level). Internal program logic documentation all too frequently does not keep pace with program evolution. Using an HOL, the reduced volume of documentation and the increased reliance upon the program listing decreases the related maintenance costs and minimizes the problems due to faulty and obsolete documentation.

2.1.5 Arguments Against HOL's

The most common supposition presented in arguments against HOL utilization involves the purported degration of program performance, decreased efficiency, and increased memory requirements. This position is easily supported by observing that nearly any programmer familiar with assembly language can make noticeable improvements in a compiler generated program; this remains true even for those compilers containing state-of-the-art sophisticated optimizers. It is further acknowledged that a senior programming specialist will make even more significant optimizations--optimizations still infeasible for present day compilers. What is not so readily recognized is that in practice the efficiency of an MOL program rarely exceeds that of the optimizing compiler processed HOL program. There are many factors that explain this phenomenon.

2.1.5.1 Efficiency

One aspect of High Order Language (HOL) programming that generates controversy is the relative efficiency of HOL-generated code versus code written in Assembler language.

The usual (and the only valid) argument for the use of Assembler language is its higher relative efficiency. Proponents of MOL coding will quote relative execution times of 200 - 300 percent between good HOL and MOL code. On the other hand, proponents of HOL have claimed efficiencies within 15 percent of MOL. Which is correct?

When comparing efficiencies, care must be taken to compare equal quantities. It has been said that any program written in HOL can be improved by some percentage by rewriting it in MOL. However, bear in mind that the rewriting is being done with the specific goal of improving the efficiency. On that basis, it is a rare HOL program that cannot be improved 30 percent by rewriting it in HOL, or an MOL program in MOL. For that matter, there are certainly Assembler language programs written so badly that rewriting them in HOL would improve their efficiency. The only valid comparison is between two programs written for the first time by equally competent programmers. The 10 to 15 percent degradation witnessed when comparing J73 object programs with MOL coded programs for the AFAL DAIS effort illustrates that little sacrifice needs to be made. This comparison was made on original programs before any maintenance was performed; after any substantial maintenance has been performed, the differential can be expected to decrease.

Although the compiler used for these comparisons contains a regional optimizer and generates efficient code sequences, it does not perform the level of improvement that can be achieved by the more sophisticated global optimizers. These optimizers perform total flow analysis, delete unreferenced code and procedures (important for heavily maintained programs), straighten code to improve other optimizations and minimize branches, eliminate linkage and parameter code for single entrance procedures, perform dead variable analysis to reduce data allocation and delete unnecessary stores and computations, redistribute code from loops and parallel paths, reduce operator strength, and more important, determine more optimal register assignment and dedication. These optimizers not only make significant improvements in object speed and space but also increase programmer productivity by eliminating the source optimization task which often degrades source readability.

2.1.5.2 Memory Utilization

The convenience of an HOL has its cost in increased memory utilization. The increase in the number of instructions generated by an HOL over those generated using an MOL is generally insignificant in the context of memory utilization. The real significant differences arise in the storage and retrieval of data. The use of exotic data structures (e.g., strings, linked lists, queues, stack) available in most HOLs carry with it a high overhead in terms of memory usage. Fortunately, this is not a big problem. Most HOLs have simple one-to-one array storage data structures also. Thus, if storage usage becomes a problem because of the data structures chosen, it is possible to drop back to a more memory efficient data storage system which would compare favorably with a comparable MOL structure.

In the evaluation of overall life cycle costs, hardware as well as software costs must be considered. However the current upward trend in software cost, coupled with the drastic reduction in memory cost, tends to invalidate arguments against HOLs based upon increased memory requirements.

2.1.5.3 Data Representation

Sometimes problems arise from the fact that program data cannot be represented in an efficient way inside the HOL. If this problem is envisioned, it is important to ensure that the HOL being used can manipulate the required data resources. (This usually implies that partial word data manipulation capabilities are built into the language.)

2.2 WHEN TO USE AN HOL

How does one make the final decision as to whether to use an HOL for a given project? This section describes a technique which may be useful in making the economic analysis

that is involved. It should be clear by now that many non-economic factors may be involved in this decision. In fact, factors such as strong constraints on the processor's memory size, compiler availability, and the like may override any of the economic issues that seem relevant. We assume here that an economic decision is reasonable.

Any economic analysis is based on issues summarized in the following equation:

$$S = \Delta D + \Delta M - F$$

where

$S =$ the assumed savings from using an HOL rather than a machine oriented language (MOL). (S may be negative)

$\Delta D =$ (software costs for MOL development) - (software costs for HOL development)

$\Delta M =$ (software costs for MOL system maintenance) - (software costs for HOL system maintenance)

$F =$ additional front-end expenses incurred from using an HOL

If a mean and standard deviation can be determined for each of the delta terms, simple probabilistic considerations give us the range for values of S, i.e. we know that

$$\mu_S = \mu_{\Delta D} + \mu_{\Delta M} - \mu_F$$

and

$$\sigma_S = (\sigma^2_{\Delta D} + \sigma^2_{\Delta M} + \sigma^2_F)^{\frac{1}{2}} .$$

(Note there is approximately a 67 percent probability that random values lie within one standard deviation from the mean in normally distributed random data.)

How can these formulas be used? Suppose for a given project

$$(\mu_S - \sigma_S) \geq 0 .$$

Then, with 80 - 85 percent confidence, one can claim that the choice of an HOL will not use additional money and will probably result in a net savings. We can make the same claim with 98 - 99 percent certainty if

$$(\mu_S - 2\sigma_S) \geq 0 \quad .$$

2.2.1 Analysis of ΔD (Development Cost Change)

Eventually, the market will determine ΔD very accurately. This will happen when HOLs develop some history with guidance and control applications. Current software experience leads to the following grass roots analysis.

A general rule of thumb indicates that project costs for software are spent in the following way: 40 percent, Design; 20 percent, Coding; and 40 percent, Checkout. Language choice seems to have little effect on design. With the required confidence, a 40 - 60 percent reduction in coding costs and a 35 - 70 percent reduction in checkout costs can be predicted when using an HOL. This leads to a 22 - 40 percent reduction in overall development costs. Hence, if

D = Development Costs for MOL System

then by using the mean of the estimates above

$$\mu_{\Delta D} = .31D$$

and

$$\sigma_{\Delta D} = .0728D \quad .$$

In other words, one can claim that HOL development costs are between 62 - 76 percent of MOL development costs with 65 - 70 percent confidence.

2.2.2 Analysis of ΔM (Maintenance Cost Change)

This is the hardest to estimate. Again by a grass roots analysis, experience and intuition suggest that a 30-80 percent cost reduction is achievable within the required confidence interval. Hence, if

M = Maintenance Cost for MOL System

then

$$\mu_{\Delta M} = .55M$$

and

$$\sigma_{\Delta M} = .25M \quad .$$

Thus, with 65 – 70 percent confidence, HOL Maintenance Costs are estimated at 20 – 70 percent of MOL Maintenance Costs.

We assume that some life cycle labor forecast exists to make the above analysis feasible.

2.2.3 Analysis of F (Front-End HOL Expenses)

Compiler cost seems to be the only major component of F. If a good compiler is being used, then a 5 - 15 percent decrease in speed and a 10 - 15 percent increase in space can be envisioned. Hence, a "reasonable" MOL choice for a machine configuration will also handle an equivalent HOL program.

It seems fairly clear that the market can determine $\mu_{\Delta F}$ with great accuracy. If only a language retargeting is involved, the price is expected to be between $75,000 and $100,000. This means that

$$\mu_F = \$87,500$$

and

$$\sigma_F = \$12,500.$$

Combining the terms before, we get

$$\mu_S = .31D + .55M - 87500$$

and

$$\sigma_S = \sqrt{(.0053D^2 + .0625M^2 + 12500^2)} \; .$$

Now suppose an HOL versus MOL decision is needed on a particular project where the front-end costs were as projected. The HOL would be indicated wherever

$$\mu_S - \sigma_S \geq 0$$

In particular, if equal amounts are allocated to development and maintenance (a reasonable ratio) then any project with more than $149,198 allocated to software development should use an HOL. With a 80 - 85% confidence level the lifetime savings to be expected in this case by use of an HOL are

$$\mu_S = \$40,810$$

with a total software cost of

$$C = \$257,586 .$$

Thus, for generic planning purposes, it seems safe to say that if funds are available for front end compiler development (not exceeding $100K) and $300K or more is planned for software development, then the use of an HOL is warranted on an economic basis.

2.3 ALGORITHM ALTERNATIVES

The use of an HOL is a particular solution to the problem of getting from an algorithm to a machine language computer program (MLCP) which represents a realization of that algorithm on a computer. The HOL needs two attributes for this to work. First, a valid HOL program must be amenable to a "mechanical" translation to a MLCP, i.e., the HOL must be capable of being "compiled." Second, the HOL must make it "easy" for the programmer to encode his algorithm in that language.

The mechanical nature of the translation process practically guarantees that the translated MLCP will be free of many types of "clerical" errors. This happens because many of the opportunities for error are removed. This fact, coupled with the empirical observation that the number of errors in a program is more closely correlated with the size of the source program than with the size of the MLCP, suggests that the "average" MLCP created from a translation of an HOL program will have fewer errors to remove than its equivalent produced by other techniques.

However, this freedom from errors has a cost. MLCPs exist that cannot be described by an HOL. In mathematical terms, this is equivalent to stating that while all valid HOL programs can be mapped (compiled) into MLCPs, an inverse mapping may not exist. Let us call such MLCPs "indescribable." If some indescribable MLCP exists that

is a more efficient implementation of a desired algorithm, the cost in efficiency of using the HOL is the difference between the cost of the HOL's implementation of the algorithm and the cost of the more efficient implementation. Here, program efficiency is measured in terms of program size or execution time. Let us consider the nature of this loss in efficiency. This issue is becoming minimized by advances in hardware technology, language design, and compiler design. Advances in hardware technology have greatly increased the amount of memory space available, increased processor speed and allowed us to change the basic configuration of the machine through microprogramming. Advances in language design have resulted in languages which have facilitated the optimization of compiled code. Furthermore, an understanding of the programming process indicates the way to better programs is through better algorithms rather than more efficient code strings.

The better algorithm issue is important. Consider the following example where the time needed to execute an algorithm as a function of the problem size is defined as its "time complexity." Suppose four algorithms have the following time complexity:

Algorithm	Time Complexity	Example
A1	N	Transverse Linear List
A2	$N \, Log_2 N$	Sorting
A3	N^2	Add Two N x N Matrices
A4	2^N	Traveling Salesman Problem

Here the time complexity is the time required to process an input of size N. If we assume one unit of time is one millisecond we have:

Time		Maximum Problem Size		
Algorithm	Complexity	1 Second	1 Minute	1 Hour
A1	N	1000	$6*10^4$	$3.6*10^5$
A2	$N \, Log_2 N$	140	4895	204094
A3	N^2	31	244	1897
A4	2^N	9	15	21

Suppose we can make a "hand-tailored' version of our algorithm that runs twice as fast. Then we have

Time		Maximum Problem Size Old-New					
Algorithm	Complexity	1 Second		1 Minute		1 Hour	
A1	N	1000	2000	$6*10^4$	$1.2*10^5$	$3.6*10^5$	$7.2*10^5$
A2	$N \, Log_2 N$	140	251	4895	9122	204094	327825
A3	N^2	31	44	244	346	1897	2683
A4	2^N	9	10	15	16	21	22

In all cases, improving the algorithm gives better results than improving the code.

There are some additional considerations to be taken into account on the optimization issue. If a program executes in a highly skewed fashion, work expanded to optimize portions of the code may not pay off in kind. Various researchers have noted (as a rule of thumb) that 10 percent of a programs code is executed 90 percent of the time.

Consider the effect of doubling the speed of the "hot" 10 percent as opposed to the "cold" 90 percent. (Here we assume the program takes 10X units of time to execute.)

Situation	Time for Hot 10%	Time for Cold 90%	Total Time	Index of Efficiency [10X / Total Time]
Normal	9X	X	10X	1
Speedup Hot	4.5X	X	5.5X	1.82
Speedup Cold	9X	.5X	9.5X	1.05
Speedup Both	4.5X	.5X	5X	2

Notice that the only significant improvement comes from working on the hot 10 percent.

The above consideration indicates that manual code throughput optimization can be safely performed in a two-stage process. The first step consists of choosing the best algorithm at design time. Coding can then proceed in an appropriate HOL without regard to optimization issues. Then, after a working system is complete, optimization can be performed on those elements of a system where it is required, and the execution characteristics of the system suggest that code optimization is worthwhile. In these situations, an assembly language might be a suitable tool.

Space complexity is also an issue in HOL use. It isn't uncommon to see HOL programs that are twice as large as equivalent assembly language versions of the same algorithm. However, some research has claimed good optimizing compilers produce only 10 - 15 percent larger programs than assembly language counterparts. Fortunately, there are a variety of methods which can attack the problem. The simple expedient of moving more common code into subroutines is often a solution. One is also able to "throw hardware" at the problem either by giving the host more memory or adding virtual memory capabilities to the processor.

Sometimes problems arise from the fact that program data cannot be represented in an efficient way inside the HOL. If this problem is envisioned, it is important to ensure that the HOL being used can manipulate the required data resources. (This usually implies that partial word data manipulation capabilities are built into the language.)

SECTION 3 - REQUIREMENTS FOR CANDIDATE HOLS

3.1 APPROACH

In this study, the aim is to evaluate various candidate HOLs for their applicability to embedded missile system applications. To accomplish this, it is first necessary to define the criteria against which the HOLs are to be judged. This is done in two steps. First, the functions that must be performed are defined by examining typical missile system applications. Second, the IRONMAN requirements for DOD-1 are examined and modified for the missile system environment.

3.2 MISSILE SYSTEMS REQUIREMENTS

In this subsection we shall consider the operations which must be performed by a missile system. We shall begin with a very simple case, and then consider increasing degrees of complexity.

3.2.1 Analog Autopilots

3.2.1.1 Fixed Attitude

Consider first the simple analog autopilot shown in Figure 3-1.

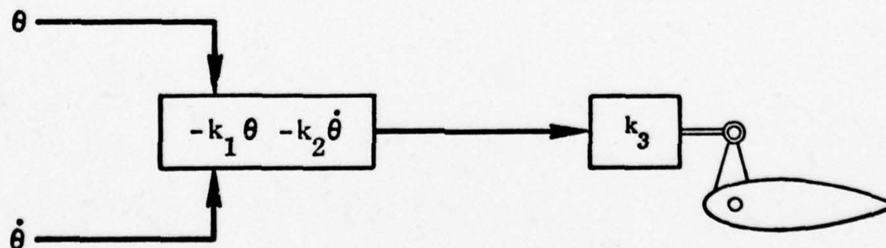


Figure 3-1. Simple Autopilot

One channel of an attitude control link is shown. Such a link would be suitable for an airplane autopilot or that for a cruise missile. Although only one channel is shown, the control law can easily be generalized to a vector form:

$$\underline{F} = -K_1 \underline{X} - K_2 \dot{\underline{X}} \tag{3-1}$$

or, even more generally,

$$\underline{F} = -\underline{\underline{K}}_1 \underline{X} - \underline{\underline{K}}_2 \dot{\underline{X}} \tag{3-2}$$

where $\underline{\underline{K}}_1$ and $\underline{\underline{K}}_2$ are matrices.

In Figure 3-1 a separate rate sensor is implied. If one is not available, its output can be synthesized by differentiating the "θ" signal.

Even in the rudimentary system shown, there are three functions required which are common to all practical G&C systems. These functions are:

(1) Read sensor signals

(2) Operate on these signals via a control law to obtain a command response

(3) Output the commanded response

3.2.1.2 Command Guidance

The autopilot of Figure 3-1 is of little use in missile systems since few, if any, missiles fly with constant attitude throughout the flight. More generally, a commanded, time dependent attitude is required. Such a system is shown in Figure 3-2.

In Figure 3-2 the open-loop actuator has been replaced by a servo, which is the more common practice. In command guidance the input commands θ_c and $\dot{\theta}_c$ are obtained by ground radio link.

Note that the functions to be performed are essentially the same as in the previous case. The only essential difference is a larger number of sensors to be sensed and a more complex algorithm.

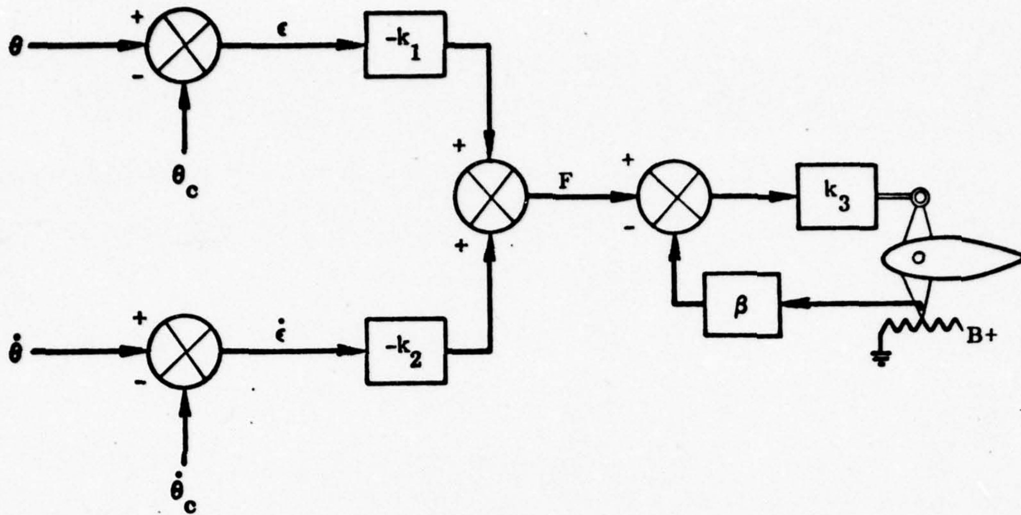


Figure 3-2. Command Guidance

3.2.1.3 Digital Mechanization

The two autopilots shown in Figures 3-1 and 3-2 are analog, yet the requirements under discussion are those for digital G&C applications. The implication is that the equations given will be mechanized using digital techniques. Indeed, this is often done; a successful analog system can be converted to digital by appropriate application of sampled data theory and, if the sampling rate is sufficiently high, the converted system will accurately mimic the original analog one. For example, Figure 3-3 shows a simple digital representation of the autopilot of Figure 3-1.

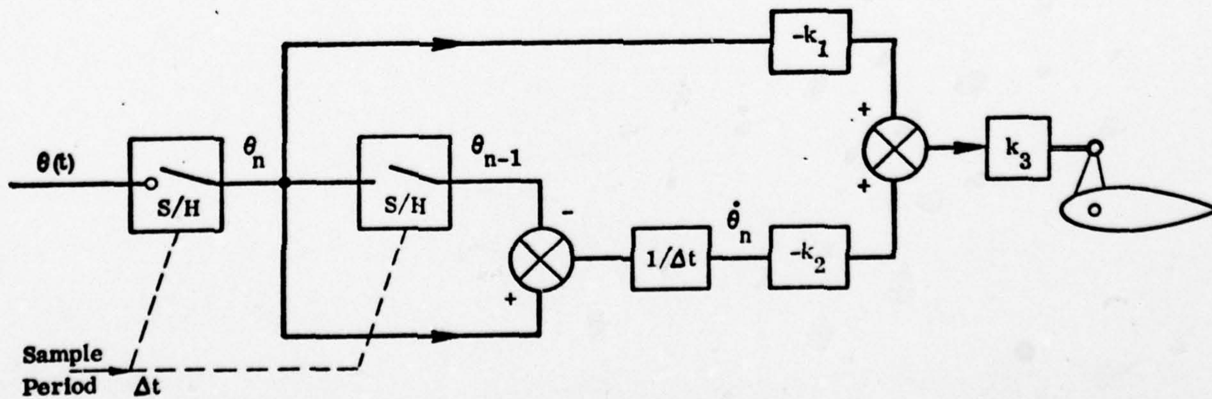


Figure 3-3. Digital Version of Figure 3-1

In practice, this approach is not necessarily best. It is generally preferable to consider the entire system as an integrated digital system and optimize gains at that level, than to attempt to digitally emulate an analog system.

It is not the purpose of this study to resolve the issue of analog versus digital approaches. Since analog block diagrams are more easily visualized than software flowcharts, we shall continue to represent the guidance systems as quasi-analog ones, with the understanding that the figures are merely graphic representations of basically digital systems.

Digital signal processing techniques such as filters and Z-transforms are required. These all have the form

$$f_n(x) = \sum_{j=1}^{n} a_j x_{n-j} \qquad 3\text{-}3$$

that is, an n-dimensional dot product.

3.2.1.4 Functions Required

On the basis of the two examples given so far, we can already identify certain mathematical functions that must be supported. These are:

- o *Addition, subtraction and multiplication of scalars*
- o Addition and subtraction of vectors
- o Multiplication of a vector by a matrix

Although the need for vector and matrix operations has been identified and they are regarded as highly desirable for a missile-oriented HOL, it is recognized that none of the candidate languages support such operations, and they are not required for DOD-1 by Ironman. Hence, these operations are regarded only as useful, but not as hard requirements. They can always be implemented by subroutines. In order to do this, we must require:

- o Data structures including array types
- o Efficient subscript and index constructs
- o Index counters and loop structures
- o Efficient routine or procedure calls with formal parameters

3.2.2 Function Generation

3.2.2.1 Nonlinear Elements

Additional functions are required if nonlinear relationships are needed, as is often the case. Such relationships are used to approximate nonlinear functions, and are the digital equivalent of analog function generators. Some examples are:

$$y = a_1x + a_2x^2 + a_3x^3 + a_4x^4 \quad \text{(Polynomial)} \qquad 3\text{-}4$$

$$y = \frac{(a_0 + a_1x + a_2x^2)}{(b_0 + b_1x)} \quad \text{(Rational Fraction)} \qquad 3\text{-}5$$

$$r = \sqrt{x^2 + y^2} \quad \text{(RMS)} \qquad 3\text{-}6$$

$$x = \cos(\theta) \quad \text{(Transcendental)} \qquad 3\text{-}7$$

While it may be possible to avoid certain of these operations by clever programming, we shall not place undue restrictions upon either the HOL or the programmer. Instead, we shall require an adequate set of mathematical operations to be supported. The efficient use of these is left to the programmer. An adequate set of operations is felt to be:

- Algebraic functions
  - Divide
  - Exponentiate
  - Square Root
- Trigonometric Functions and Inverses
  - Sine
  - Cosine
  - Tangent (2- and 4-quadrant)
- Logarithmic Functions
  - ln x
  - e^x

3.2.2.2 Discontinuous Elements

The transcendental functions do not exhaust the possible functional relationships between variables. There is also a large class of useful functions that have discontinuities within the range of interest, such as the simple limiter of Figure 3-4.

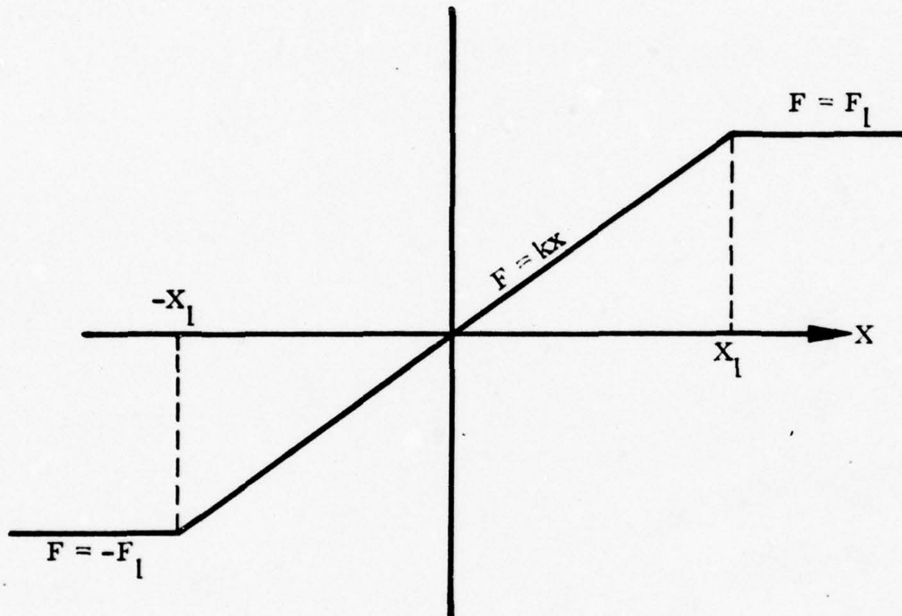


Figure 3-4. Simple Limiter

Limiters are often used in avionics systems to avoid overdriving hardware actuators. The limiting of variables is particularly important in digital systems, where an integer multiply can cause an overflow and consequent system failure.

Another type of control system which requires discontinuous switching is the bang-bang system, which contains actuators that are either off or on, rather than proportional. A typical switching function drawn on the phase plane is shown in Figure 3-5. The numbers in each region indicate whether the actuator is on forward (+1), on reverse (-1), or off (0).

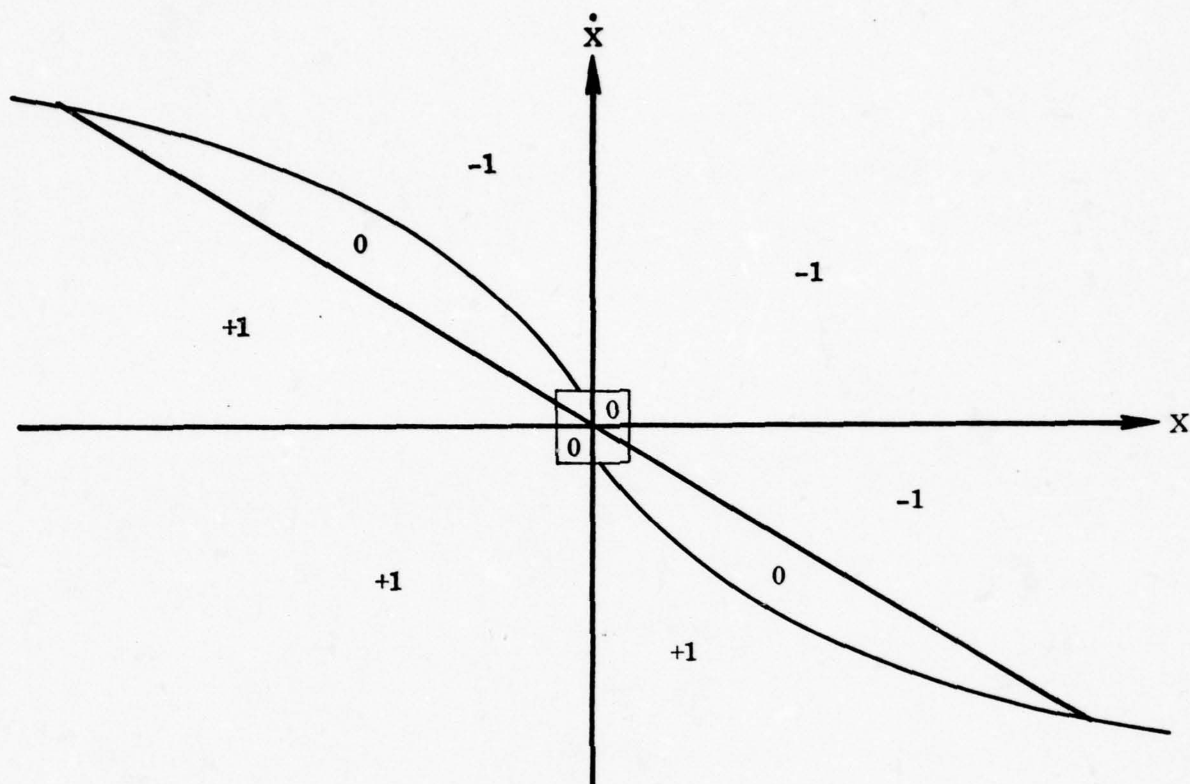


Figure 3-5. Bang-Bang Switching Function

3.2.2.3 Functions Required

In order to implement discontinuous functions, the following capabilities are needed:

- Relational operators
- Conditional branch

In order to support bang-bang controllers and conditional branches, the following capabilities are considered useful:

- Logical variables
- Logical expressions

3.2.3 Preset Guidance

In the system of Figure 3-2, it was assumed that the command attitudes and rates were obtained by ground command. An alternative approach could obtain these by table look-up:

$$\theta_c = \theta_c(t) ; \qquad \dot{\theta}_c = \dot{\theta}_c(t) \quad . \qquad 3\text{-}8$$

This presupposes, of course, that the desired path of the missile is known prior to launch. For this to be a feasible approach, it must be possible to preload the table based upon ground targeting. The table lookup formula has the general form (for first order)

$$\begin{aligned} \theta_c &= \theta_{ci} + \dot{\theta}_{ci}(t - t_i) \\ \dot{\theta}_c &= \dot{\theta}_{ci} + \ddot{\theta}_{ci}(t - t_i) \quad . \end{aligned} \qquad 3\text{-}9$$

No new operations are required for the table lookup per se. What is new, however, is the relationship with respect to time. A real time clock (RTC) is required and the capability to read it. This time dependence is discussed further in the next section.

3.2.4 Timing

The table lookup discussed above is only one of many examples of time-dependent relationships in G&C applications. Indeed, the whole concept of sampled-data control is based upon regular evaluations of input variables, with a "guaranteed" time interval between them. Other examples are sequencing (changing the program flow at discrete time intervals), delays (such as used for thruster minimum on-time constraints), and timing for data I/O interfaces.

A delay can be implemented by setting up a software delay loop, in which the CPU simply counts to a specified number before exiting the loop. The only problem here is that the CPU can do nothing else while looping. Therefore, for all practical purposes a hardware clock must be provided. There are two ways to implement such a clock. First, we can connect an oscillator of appropriate frequency to an interrupt input. The

computer is required to update a software clock each time the interrupt is serviced. Alternatively, we can provide a true hardware clock which can be read as an input device by the computer. A useful language should support both alternatives.

Event sequencing and staging is another area requiring software support. Such operations involve the real-time clock and require the ability to interrupt at a specified value of the RTC.

In addition, efficient management of an event table requires the ability to insert or delete entries in the table. These operations have similarity with string processing functions, and require the ability to relocate data blocks.

3.2.4.1 Interrupts and I/O

The input of data from the RTC is one example of the broader operation of data I/O. In addition to ground checkout equipment and other communications I/O, the sensors and actuators appear to the CPU as specialized I/O devices. Each of these devices tends to possess very distinct characteristics which must be accommodated by the flight software.

As with the RTC, there are two ways to process I/O devices:

- Interrupt-oriented
- Interrogated

In the case of interrupt-oriented I/O, an input device interrupts the CPU when it has input available. The CPU sends data to an output device, which interrupts when it can accept more data. In general, this approach is not compatible with the interrupt-oriented RTC. In the interrogated I/O case, as the name implies, the CPU sends data to output or reads data from input at its own pace. Handshaking is required to synchronize the CPU with slow I/O devices.

The degree to which interrupt processing is supported by an HOL is subject to some debate. Interrupts and I/O are highly hardware-dependent, so it is difficult to make general statements about them. Normally, G&C software includes a certain amount of machine language code to process interrupts and I/O devices, and this deviation from HOL standards is accepted as a necessary evil. However, there is nothing

inherent in this software that forces it to be written in MOL. Interrupt processing routines look just like ordinary ones, with two exceptions:

(1) They are normally not called by any other software, but only entered when the CPU is interrupted.

(2) They must be given absolute locations, since the interrupt forces a branch to a specific address.

Similarly, I/O routines can often be conveniently written in HOL. However, since the handshaking requires access to data at the bit level, such operations as masking (AND and OR) are required.

3.2.4.2 Functions Required

The functions of timing, interrupt processing, and I/O processing can be adequately handled by the HOL if it permits the following operations:

- Read input ports
- Write to output ports
- Mask AND, OR
- Shift left, right
- Enable and disable interrupts

Although the implementation of an RTC can be accomplished using these operations, the RTC concept is universal enough in missile system applications to support the following suggested special operations, which are considered desirable but not necessary:

- Read RTC
- Reset, start and stop RTC
- Delay for specified real time
- Interrupt on specified value of RTC

3.2.5 Targeting

In the examples given so far, the missile path has been assumed to be specified, either prior to launch or externally, through ground command. Such a system is open loop and subject to perturbations due to engine performance variations, wind gusts, etc.

Many guidance systems have targeting capability; they can sense variations in the motion, predict the resulting error, and correct for it. Some examples are examined next.

3.2.5.1 Homing

The simplest type of targeting (see Figure 3-6) is very similar to the system of Figure 3-1, except that the angle input (θ) is replaced by a targeting sensor.

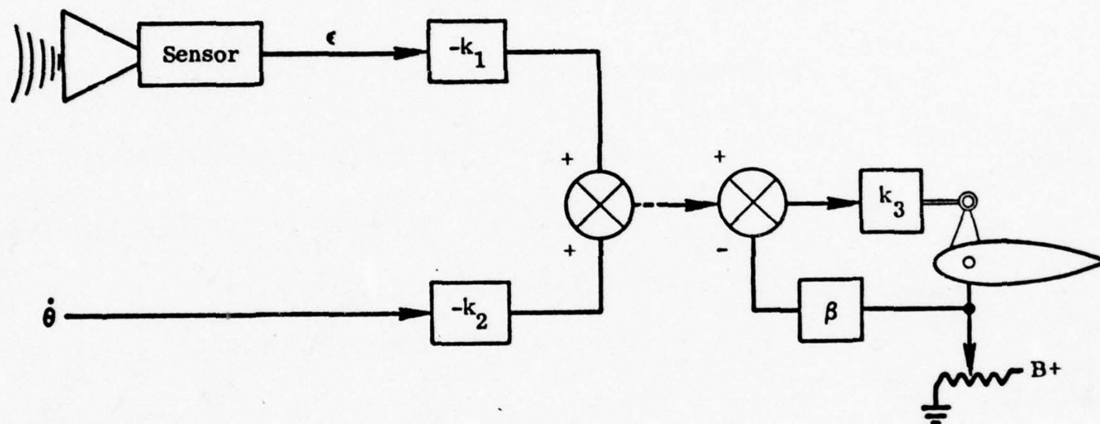


Figure 3-6. Simple Homing Guidance

Such a system will home; i.e., fly line-of-sight to a target. It could be used in a simple ARM or IR-seeking missile. Since the system is similar to Figure 3-1, no new functions are required.

3.2.5.2 Proportional Navigation

A more efficient targeting scheme is one which attempts to control $\dot{\theta}$, rather than θ. An example is shown in Figure 3-7.

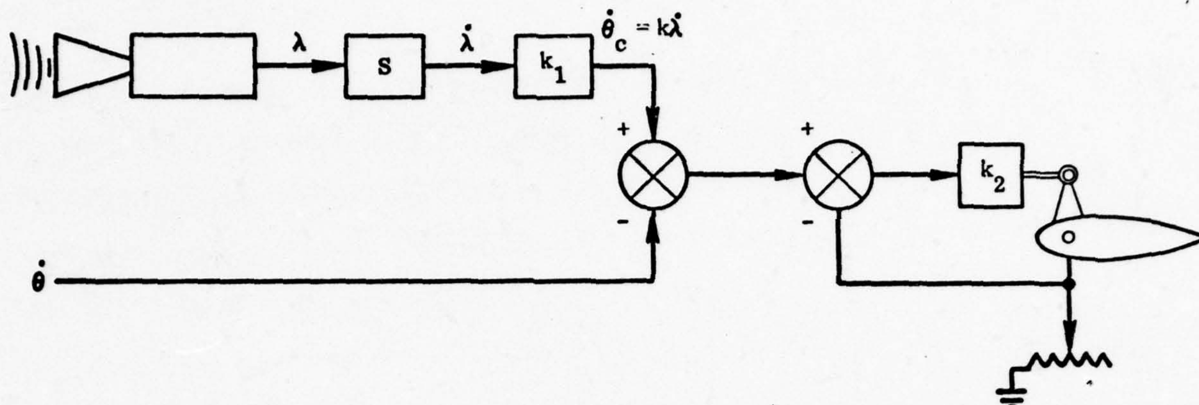


Figure 3-7. Proportional Navigation

The new operation required here is the differentiation denoted by the "S" block. Note that it is not proposed here to supply differentiation (S) and integration ($\frac{1}{S}$) functions in the HOL. To do so would invite terribly inefficient use of the computer resources. Rather, we observe that both S and $\frac{1}{S}$ can be approximated by their Z-transform representations, which require no new operations. Note, however, that these representations are time-dependent, and therefore, involve the RTC.

3.2.5.3 Covariance Guidance

A very flexible guidance scheme is shown in Figure 3-8. In this system the missile acceleration $\ddot{\underline{X}}$ is sensed and integrated to obtain current state. This is then compared to the nominal state defined by $\underline{X}_n$, $\dot{\underline{X}}_n$ and processed through the state transition matrix $\underline{\underline{\Phi}}$ to obtain the estimated target error. This is then propagated back to a desired acceleration and, ultimately, a new commanded attitude. Once again, the operations required include multiplication of a matrix by a vector, as well as the $\frac{1}{S}$ mechanization.

In Figure 3-8, it is not necessary to invert the influence matrix $\underline{\underline{M}}$. It can be stored in inverted form.

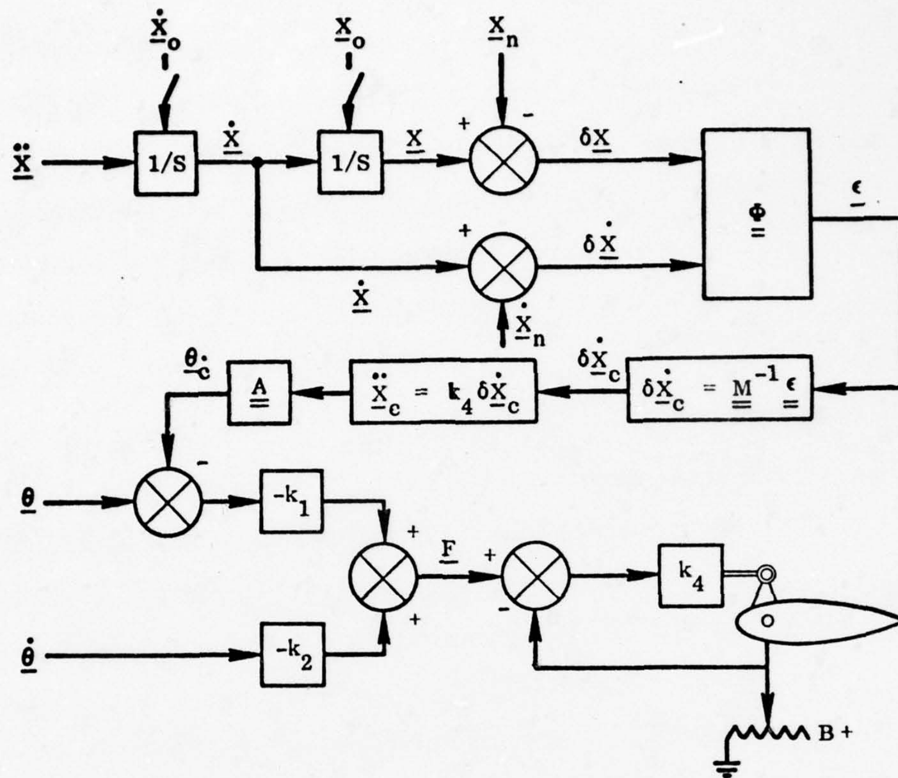


Figure 3-8. Covariance Guidance

To obtain the algorithm used in Figure 3-8, a linear relationship between δx and $\delta\dot{\underline{x}}$, and the error $\underline{\epsilon}$ is assumed. The values of $\underline{\underline{\Phi}}$ and $\underline{\underline{M}}^{-1}$ must be precomputed and may or may not be time dependent.

In some cases (e.g., highly parabolic flight paths) this may not be sufficient. The same is true for moving target trackers, in which the target motion, as well as the motion of the missile itself is considered. In order to predict the targeting error, the equations of motion for both bodies must be integrated forward. Thus,

$$\underline{x}_f = \underline{x} + \iint \ddot{\underline{x}}(\underline{x}, \dot{\underline{x}}, t)dt \quad . \qquad 3\text{-}11$$

The matrices $\underline{\underline{\Phi}}$ and $\underline{\underline{M}}^{-1}$ also have differential equations:

$$\dot{\underline{\underline{\Phi}}} = \dot{\underline{\underline{\Phi}}}(\underline{x}, \dot{\underline{x}}, t)$$

$$\dot{\underline{\underline{M}}}^{-1} = \dot{\underline{\underline{M}}}^{-1}(\underline{x}, \dot{\underline{x}}, t) \quad . \tag{3-12}$$

These must be numerically integrated. In this case, although time integrals, these operations must proceed faster than real time. While this requirement places a severe load upon the CPU and requires a high-speed numerical integration algorithm, it removes dependence upon the RTC.

3.2.6 Rotational Equations

The system of Figure 3-8 has been oversimplified in that no cross-coupling between rotational and translational dynamics is shown. In actuality, the measurements $\ddot{\underline{\theta}}$ and $\ddot{\underline{x}}$ are measured in body axes, whereas the target position and nominal motion are represented in inertial axes. A transformation is required of the form

$$\ddot{\underline{x}}_{inertial} = \underline{\underline{T}}\,\ddot{\underline{x}}_{body}$$

$$\ddot{\underline{x}}_{body} = \underline{\underline{T}}^{T}\,\ddot{\underline{x}}_{inertial} \quad . \tag{3-13}$$

In terms of Euler angles, the matrix $\underline{\underline{T}}$ has a form involving trigonometric functions. For example:

$$T_{11} = \cos\Psi\cos\Phi - \sin\Psi\sin\Phi\cos\theta \quad . \tag{3-14}$$

In practice, Euler angles are rarely used anymore; instead, quaternions are used, which do not require trigonometric functions. In terms of quaternions, Eq. 3-14 becomes

$$T_{11} = q_1^2 - q_2^2 - q_3^2 + q_4^2 \quad . \tag{3-15}$$

Even so, the use of trigonometric functions in certain parts of the computation are essentially unavoidable. A quaternion is a four-vector which has a norm of one and certain multiplicative properties. Some typical relations are given below.

$$\underline{\underline{T}} = 2\,\underline{e}\,\underline{e}^T + (2d^2 - 1)\underline{\underline{I}}_3 + 2\,d\,\tilde{e} \qquad 3\text{-}16$$

where

$$\underline{e} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} \qquad d = q_4 \qquad 3\text{-}17$$

$$\tilde{e} = \text{cross product operator} = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \qquad 3\text{-}18$$

$$\underline{\underline{T}}\,\underline{A} = 2(\underline{e} \cdot \underline{A})\underline{e} + (2d^2 - 1)\underline{A} + 2d(\underline{e} \cdot \underline{A}) \qquad 3\text{-}19$$

$$\underline{q}_{AC} = \underline{q}_{AB} \boxed{x}\, \underline{q}_{AC} \qquad 3\text{-}20$$

where $\boxed{x}$ indicates the four-dimensional cross product:

$$\underline{A} \boxed{x} \underline{B} = \begin{bmatrix} A_1B_4 + A_2B_3 - A_3B_2 + A_4B_1 \\ -A_1B_3 + A_2B_4 + A_3B_1 + A_4B_2 \\ A_1B_2 - A_2B_1 + A_3B_4 + A_4B_3 \\ -A_1B_1 - A_2B_2 - A_3B_3 + A_4B_4 \end{bmatrix} \qquad 3\text{-}21$$

$$\underline{L} = -k_1\underline{e} - k_2\underline{\omega} \quad \text{(Control law)} \quad . \qquad 3\text{-}22$$

In order to support these relations, the following operations are required:

- o Matrix product
- o Vector dot product
- o Vector cross product
- o Cross product operator (3 and 4 dimensions)
- o Truncation of a four-vector (to three)

As mentioned previously, all of these operations can be performed by applications software, providing that adequate subroutine structure, array processing, and looping facilities are provided by the HOL.

3.2.7 Kalman Filtering

In Figure 3-8 the acceleration is input directly into the guidance algorithm. The system has no memory of the past acceleration history. This tends to make the system sensitive to noise in the acceleration or its measurement. To avoid such problems and permit an optimal smoothing of the input data, a Kalman filter can be used. A typical representation of this filter is:

$$\Delta X^*_{n+1} = \Phi^*_{n+1} \Delta X^*_n + \Delta^*_n \Delta Y_n \qquad 3\text{-}23$$

where ΔX^*_{n+1} represents the estimated state at t_{n+1}, ΔY_n the measured errors at t_n, and Φ^* is the estimated state transition matrix.

$$\Delta^*_n = \Phi_{n+1} \Lambda_{x,n+1} M^T_n (M_n \Lambda_{x,n} M^T_n + \Lambda_\epsilon)^{-1} \qquad 3\text{-}24$$

where Λ_x is the covariance matrix for x and Λ_ϵ is that for noise. This algorithm represents one application of the theory of pseudo-inverse matrices, which have the general form:

$$\underline{\underline{A}}^T (\underline{\underline{A}}\,\underline{\underline{A}}^T)^{-1}$$

or $\qquad$ 3-25

$$(\underline{\underline{A}}^T \underline{\underline{A}})^{-1} \underline{\underline{A}}^T$$

depending upon the form of $\underline{\underline{A}}$.

The same construct appears for any control situation in which there are more degrees-of-freedom than there are variables to be controlled. The new operation required here is the matrix inverse, which also must be supported in applications software.

3.2.8 Data Representation

For best run-time efficiency, the best data representation is integer binary, with word length just enough to permit the desired accuracy. To reduce scaling problems to some degree, fixed point data (integer with relocated binary point) is often permitted. However, few existing HOLs have very flexible operations using fixed-point data; the proper representation of mixed-mode data is left up to the programmer.

Experience has indicated that about 50 percent of software errors in flight software is associated with scaling errors and data overflows. For this reason, heavy use of floating point data is recommended. However, the use of both integers and fixed-point variables should be supported for areas in which execution time is critical.

3.2.9 Summary

A representative range of guidance and control applications has been examined to attempt to define the functions which are required in an HOL suitable for G&C uses. At the application level, the following capabilities were found to be necessary:

- Data Representation
  - Integer
  - Fixed point
  - Floating point

- Scalar Arithmetic
  - Multiply
  - Add
  - Subtract
  - Divide
  - Exponentiate
  - Square Root
- Transcendental Functions
  - Sin
  - Cos
  - Tan
  - $\sin^{-1}$
  - $\cos^{-1}$
  - $\tan^{-1}$
  - e^x
  - ln x
- Vector Arithmetic
  - Add (3 or n dimensions)
  - Subtract (3 or n dimensions)
  - Dot product (3 or n dimensions)
  - Cross product
  - Four-D cross product
- Matrix Arithmetic
  - Multiply matrix by vector
  - Multiply matrix by matrix
  - Multiply vector by vector ($\underline{A}\ \underline{A}^T$)
  - Generate cross-product matrix (3 or 4 dimensions)
  - Inverse

- Control Functions
  - Conditional branch
  - Relational operators
  - Interrupt processing
- Real-time Functions
  - Enable interrupt
  - Disable interrupt
  - Reset RTC
  - Start RTC
  - Stop RTC
  - Read RTC
  - Interrupt on value of RTC
  - Delay for specified RT
- Logical Functions
  - Test discretes
  - Output discretes
  - Logical variables
  - Logical operators (AND, OR, NOT, XOR, EQ)
- I/O
  - Input data word
  - Output data word
  - Input data block
  - Output data block
- Array Functions
  - Move array
  - Move subarray
  - Insert element
  - Delete element

For the sake of execution time efficiency, it is desirable that as many of these functions as possible be supported directly by the HOL. Those that are not must be supported by applications software written in the HOL. To accomplish this, we require:

- o Data structures including vector and matrix array types
- o Efficient subscripts and index constructs
- o Index (loop) counters and loop structures
- o Efficient subroutine or procedure calls with formal parameters

3.3 GENERAL LANGUAGE REQUIREMENTS

One of the planned uses for the common language DOD-1 is in embedded missile system applications. Therefore, the requirements for a language acceptable for such applications must be contained in the DOD-1 requirements, which are represented by the IRONMAN specification. However, this specification also contains requirements pertinent to other applications. Thus, the requirements of interest in this study are represented by some subset of the IRONMAN specification. In order to define this subset, the IRONMAN specification was analyzed and evaluated against the functional requirements given in the previous section.

The IRONMAN specification consists of 112 specific requirements, grouped into the following 13 categories:

1. General Requirements (8)
2. Syntax and Comment Conventions (9)
3. Data Types (31)
4. Expressions (7)
5. Constants, Variables, and Declarations (7)
6. Control Structures (7)
7. Functions and Procedures (9)
8. Input-Output (5)
9. Parallel Processing (6)
10. Exception Handling (7)
11. Machine Dependent Specifications (6)
12. Library, Separate Compilation, Generic Definitions (4)
13. Standards, Translation and Support (6)

Of these requirements, some were eliminated as being inappropriate for missile systems applications. In other cases, it was felt that while the requirements were not inappropriate, they were of negligible value for the application and should not be given weight in the evaluations. Finally, a few requirements were eliminated as being too vague to properly evaluate. The remaining specifications, which make up the requirements for missile system applications, are given in Appendix A. Most of these are copied verbatim from IRONMAN. A few have been altered or rewritten to more fully conform to the specific needs of missile systems.

SECTION 4 - CANDIDATE LANGUAGE EVALUATIONS

4.1 LANGUAGE REQUIREMENTS ANALYSIS

The five HOLs chosen for evaluation in this study are:

LANGUAGE	SOURCE FOR INFORMATION	CONTROL AGENT
CMS-2	M-5049 and NAVLEX 0967LP-598-2210	Navy
JOVIAL-J73/I	MIL-STD-1589	Air Force
SPL-1	Intermetrics Report 172-1	Navy
TACPOL	Specification No. EL-CG-00043082C	Army
JOVIAL-J3B	Softech Report 7072.1	Air Force

These languages represent, with the exception of FORTRAN and COBOL, the interim languages specified in DOD Instruction 5000.31. The J3B version of JOVIAL, rather than J3, was chosen as representing the current state of the art in J3 compilers. This dialect was used successfully for imbedded avionics software in the B-1 program. JOVIAL J73/I was chosen as the dialect most nearly meeting the specifications of the full J73 language, which does not yet exist. The analysis of the five languages under consideration based on the modified IRONMAN requirements is included in Appendix A. For each paragraph of this specification, each language was rated with a zero (0), one (1), or two (2) depending on whether the requirement is not met, partially met, or fully met. No attempt was made to distinguish between degrees of partial fulfillment since such an attempt introduces too much subjectivity into the evaluations. The general requirements section was not evaluated for the same reason. The resulting scores are given in Table 4-1.

The cumulative rating scores for the various languages are given below.

	SCORE	RANK
CMS-2	88	3 (tie)
J-3B	90	2
J-73	103	1
TACPOL	85	5
SPL/I	88	3 (tie)

Max Score	210
Mean Score	90.8
Standard Deviation	7.05
τ Statistic Adjustment for 90% confidence and 4 degrees of freedom	9.79

Table 4-1. Summary of Analysis of Five Languages

Requirement	CMS-2	J-3B	J-73	TACPOL	SPL/1
2A	2	2	2	2	2
2B	2	2	2	2	2
2C	2	2	2	2	2
2E	1	2	2	1	2
2F	1	1	1	1	2
2G	1	1	2	1	1
3A	1	1	1	1	2
3-1A	1	1	1	1	1
3-1B	2	2	2	2	2
3-1C	0	0	1	1	0
3-1D	2	2	2	0	2
3-1E	1	2	2	1	1
3-1F	0	0	0	0	0
3-1G	2	2	2	2	0
3-1H	1	1	1	1	1
3-2C	2	2	2	2	2
3-3A	1	2	2	1	2
3-3B	2	2	2	2	2
3-3C	1	1	1	1	1
3-3D	2	2	2	2	2
3-3E	0	0	0	2	0
3-3F	2	2	2	2	2
3-5A	0	0	0	0	0
3-5B	0	0	1	0	0
3-5C	0	0	1	0	0
3-5D	0	0	0	0	0
4A	2	2	2	2	2
4B	2	2	2	2	2
4D	1	2	2	1	2
4E	2	1	2	1	1
4F	2	2	2	2	2
4G	2	2	2	2	2
5A	1	1	1	1	1
5B	1	2	2	1	2
5C	2	2	2	2	2
5D	1	1	1	1	2
5E	2	2	2	1	1
5F	2	2	2	2	2
5G	0	1	1	0	0
6B	2	2	2	2	2
6C	2	2	2	2	2
6D	0	0	2	0	0
6E	2	2	2	2	2
6F	1	1	1	1	1
6G	1	1	1	1	2
7A	1	1	1	1	1
7C	2	2	2	2	2
7D	2	2	2	2	2
7E	0	0	0	0	2
7F	1	2	2	1	2
7G	1	1	1	1	1
7H	1	1	1	1	1
7I	0	0	0	0	0
10A	1	0	1	1	0
10B	2	2	2	2	2
10C	1	0	1	1	0
10D	0	0	0	1	0
10E	2	2	2	2	2
10F	0	0	0	0	0
10G	0	0	0	0	0
11A	2	2	2	1	0
11B	2	2	2	1	1
11E	2	1	2	1	0
11F	0	0	2	0	0
12A	2	2	2	2	2
12B	2	2	2	2	2
12C	2	2	2	2	2
13A	1	1	1	1	1
13B	2	1	2	2	2
13D	1	1	1	1	1
13F	1	1	1	1	1
Total	88	90	103	85	88

This gives a 95 percent range from 81.01 - 100.59. The statistic indicates that the J-73 score is significant with 95 percent confidence (actually around 98 percent confidence).

It is interesting to note TINMAN scores for these same languages.

	SCORE	RANK
CMS-2	394	1
J-3B	354	4
J-73	379	3
TACPOL	336	5
SPL/I	383	2

Mean Score	369.2
Standard Deviation	23.6328
τ Statistic/95%	32.81

Here the 95 percent range is 336-402 and the hypothesis that no significant difference exists between the languages cannot be rejected at the 95 percent level.

Two conclusions can be safely drawn from the statistics:

- None of the languages closely satisfies the technical requirements
- J-73 comes as close as any of the languages under consideration

From this, it seems fair to claim that a rational choice between the candidate languages will be made on the basic of economic issues and historical issues.

4.2 COMPILER ANALYSIS

A major consideration in HOL utilization is associated with compiler availability. If a compiler does not exist on and for the machine(s) desired, one must be constructed. This can be done by design and development of an entirely new compiler or by rehosting and/or retargeting an existing one. In either case, this one-time cost must be justified by the life cycle savings from use of the HOL.

First, consider new compiler development. Little difference exists between the five languages for compiler development cost purposes. Compilation ease was a goal of the J73 language design and some reduction in cost can be expected. In any case, a new compiler matching the capability of the J3B or J73 compilers would require approximately

eight to ten man years of effort by qualified compiler writers. This approach would result in a compiler tailored to the needs of the user but should not be significantly better than existing compilers. However, an additional two to four man years would provide a comprehensive global optimizer that would noticeably out-perform the current compilers.

The alternate approach is to adapt an existing compiler. Toward this end a survey was made to assess the relative viability and desirability of adapting the existing compilers. Ignoring for the moment the merits of the individual languages, the compilers are compared below. For the purpose of the comparison, the UNIVAC 1108 was assumed to be the desired host.

Because of the proprietary nature of compilers, the information available on compilers for the five candidate HOLs is at best sketchy. Thus, the ranking of the compilers under consideration is more heuristic than deterministic.

4.2.1 CMS-2

Versions of this compiler exist in both a self-maintainable form written in CMS-2 and a more rehostable form in FORTRAN. Since they both provide essentially identical capability, the FORTRAN version was chosen for evaluation. The compiler exists on the UNIVAC 1108, CDC 6600, IBM 360/370, HIS 6000, and the AN-UYK/20. No rehosting is required.

A target capability exists for the AN-UYK/20. Retargeting for a missile borne computer is estimated at three to four man years. The resultant compiler would produce relocatable object modules.

The compiler is approximately 56K on the 1108 and compiles at 700 to 900 lines/ minute.

4.2.2 J3B

The compiler is written in AED and exists on the 360/370. Rehosting costs are estimated at two to three man years if AED exists on the new host (AED does exist on the 1108 and 6600).

The compiler generates for the SKC2070, IBM 360, LITTON 4516B, and the DELCO

Magic M362F-2. Retargeting costs are estimated at one man year exclusive of documentation. This compiler produces assembly language source so an assembler is required for the target computer.

The compiler requires 320K to 384K bytes of memory on a 370. Compilation speed is 600 to 800 lines/minute on a 370/155. The compiler includes basic block optimization.

4.2.3 J73/I

The compiler is written in a proper subset of J73/I termed J73/S. The compiler exists on the DEC-10. However, it was originally developed on the UNIVAC 1108 under CSC's time sharing system using a J73/S compiler. Rehosting J73/S and then the J73/I compiler to EXEC 8 on the 1108 is estimated at one man year.

The compiler generates code for the DEC-10, AN-UYK/14, SAMSO FTSC and the DELCO Magic M362F-2 computers. Retargeting costs are estimated at one and a half to two man years to produce relocatable binary programs.

This compiler is approximately 50K on the DEC-10 and compiles at 1500 lines/minute on a model KI-10.

This compiler performs basic block optimization.

4.2.4 TACPOL

The compiler is written in TACPOL. The compiler exists on the Litton L3050. Rehosting estimates are two man years if a PL/1 exists on the host; otherwise, seven to eight man years are required.

The compiler also produces code for the L3050. Retargeting is estimated at five to six man years.

The compiler size is 32K and it compiles at 140 to 150 lines/minute.

The compiler performs basic block optimizations and some special strength reduction for loop variables used as subscripts.

4.2.5 SPL/1

The compiler is written in FORTRAN. The compiler operates on the CDC 6600, DEC-10 and IBM 360. Rehosting is estimated at four to six man months if a suitable FORTRAN exists.

Code generators exist for the AN-UYK/20, AN-UYS/1 and AN-UYS/32. Retargeting is estimated at 21 to 24 man months plus assembler costs of three to six man months.

The compiler is 40K on the CDC 6600, 300KB on the 360 and 60K on the DEC-10. The compiler compiles at 800 to 900 lines/minute on the 6600.

The compiler performs register memory of generalized expressions and local optimizations.

4.2.6 Summary

A summary of the compiler analysis based on available information is given in Table 4-2.

Table 4-2. Summary of Compiler Analysis

	CMS-2	J-3B	J-73	TACPOL	SPL/1
Rehosting (Man/Years)	0	2-3	1	2	1/2
Retargeting (Man/Years)	3-4	1	1-1/2 - 2	5-6	2
Compiler Speed (line/minute)	700-900	600-800	1500	140-150	800-900
Compiler size (words estimated for 1108)	56K	50K	50K	32K	40K
Optimization	Fair	Good	Good	Good	Fair

The J3B compiler is the least expensive to retarget at one man year but requires use of an assembler to complete the compilations. An additional two to three man years is required to move it to the 1108. The J73/I compiler is next at one and one half man years. An additional man year is required to rehost under the 1108 EXEC 8 operating system. The J73/I compiler performs best, compiling considerably faster than all

other compilers. This compiler also provides the most complete set of user aids: statistics collection, source reformatting, multi-level diagnostics, combined environment/set-used listing, relocatable output, assembler format side-by-side listing, multiple compools, nested copies, and symbolic debugging.

The J73/I, J3B, and TACPOL compilers perform the most thorough and essentially equal levels of optimization.

SECTION 5 - CONCLUSIONS

The authors conclude that,

As discussed in Section 2.2, for significant software projects such as missile systems, use of an HOL is justified by life-cycle savings in spite of the initial one-time costs. Based upon the requirements obtained by modifying the DOD-1 IRONMAN specification, the five candidate languages were evaluated in Subsection 4.1. The compilers for these languages were also evaluated in Subsection 4.2. A matrix of the rankings is given in Table 5.1.

Table 5.1 - HOL Ranking

Item Considered	Ranking				
	CMS-2	J-3B	J-73	TACPOL	SPL/I
Language Requirements	3 (tie)	2	1	5	3 (tie)
Hosting Cost	1	5	3	4	2
Retargeting Cost	4	1	2	5	3
Compiler Speed	3	4	1	5	2
Compiler Size	5	3 (tie)	3 (tie)	1	2
Optimization	4 (tie)	1 (tie)	1 (tie)	1 (tie)	4 (tie)

If a new compiler is developed tailored to the needs of missile systems, J73 appears to be the better language. Compiler costs are not affected substantially by the choice of language.

The existing J73 compiler offers a better base and at least as economic an approach for rehosting and retargeting as any other candidate compiler. This compiler should also sufficiently satisfy the user requirements such that a totally new compiler would not be warranted. These considerations suggest J73 is the most reasonable and viable alternative.

APPENDIX A - ANALYSIS OF IRONMAN REQUIREMENTS

In this appendix the detailed requirements for missile systems applications are given. For the most part, these requirements are taken verbatim from IRONMAN. Requirements were deleted if they were not considered pertinent to embedded missile systems, or were impossible to evaluate were omitted. The paragraph numbers in this appendix correspond to the original IRONMAN numbering system.

Following each paragraph, the five languages are rated with a zero (0), one (1), or (2) depending on whether the requirement is not met, partially met, or fully met. No attempt was made to distinguish between degrees of partial fulfillment since such an attempt introduces too much subjectivity into the evaluations. The general requirements section was not evaluated for the same reason. Items 8 and 9 of the IRONMAN requirements concerning Input-Output and parallel processing are not applicable to the missile systems application and were not evaluated. Although many other methods of identifying the requirements for a G&C HOL can be aimed at, tying these requirements to the IRONMAN requirements allows ready access to and use of many man years worth of effort funded by the HOLWG in their studies. Thus, a more cost effective and objective analysis of the five languages of interest resulted from the use of this existing information (in the form of the IRONMAN requirements).

1. GENERAL DESIGN CRITERIA

1.A Generality. The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications require real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing. The language shall not contain features that are unnecessary to satisfy the requirements.

1.B Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to

correct programming errors.

1.C Maintainability. The language should promote ease of program maintenance. It should emphasize program readability over writability. That is, it should emphasize the clarity, understandability, and modifiability of programs over programming ease. The language should encourage user documentation of programs. It shall require explicit specification of programmer decisions and shall provide defaults only for instances where the default is stated in the language definition, is always meaningful, reflects common usuage, and can be explicitly overridden.

1.D Efficiency. The language design should aid the production of efficient object programs. Constructs that have exceptionally expensive or exceptionally inexpensive implementations should be easily recognizable by translators and by users. Users shall be able to specify the time space trade offs in a program. Where possible, features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs. Execution time support packages of the language shall not be included in object code unless they are called.

1.E Simplicity. The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications. It should have few special cases and should be composed from features that are individually simple in their semantics. The language should have uniform syntactic conventions and should not provide several notations for the same concept.

1.F Implementability. The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other

requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.

1.G <u>Machine Independence</u>. The language shall strive for machine independence. It shall not dictate the characteristics of object machines or operating systems. The design of the language shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. There shall be a facility for specifying those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.

1.H <u>Formal Definition.</u> To the extent that a formal definition assists in achieving the above goals, the language shall be formally defined. [Note that formal definitions are of most value during language design; and that the same method may not be appropriate for defining all aspects of a language.]

2. GENERAL SYNTAX

2.A <u>Character Set.</u> Every construct of the language shall have a representation that uses only the 64 character subset of ASCII:

```
 !"#$%&'()*+,-./
Ø123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
```

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

2.B <u>Grammar.</u> The language shall have a simple grammar and lexical structure. The program format should not be column restricted.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

2.C <u>Syntactic Extensions</u>. The language shall not be syntactically extensible. The base syntax shall not be modifiable. [Note that this does not disallow macro definitions

within the language].

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

2.D Not Evaluated.

2.1 IDENTIFIERS

2.E Mnemonic Identifiers. Mnemonically significant identifiers shall be allowed. There shall be a break character for use within identifiers. The language and its translators shall not permit identifiers or reserved words to be abbreviated.

RATINGS

CMS-2(1) J-3B(2) J-73(2) TACPOL(1) SPL/1(2)

2.F Reserved Words. The only reserved words shall be those that introduce special syntactic forms or that are otherwise used as delimiters. Words that can be used in place of identifiers shall not be reserved (e.g., names of built-in or predefined functions, types, constants, and the like shall not be reserved). All reserved words shall be listed in the language definition.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(2)

2.2 LITERALS

2.G Numeric Literals. There shall be built-in-numeric literals. Numeric literals shall have the same values in programs as in data.

RATINGS

CMS-2(1) J-3B(1) J-73(2) TACPOL(1) SPL/1(1)

[Proper satisfaction of this requirement is usually a function of the translator implementation rather than the language definition. In J-73, proper handling of this issue follows directly from the definition].

3. TYPES

3.A Strong Typing. The type of all objects within the language shall be determinable

at translation time.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(2)

3.1 NUMERIC TYPES

3.1.A Numeric Values. The language shall provide types for integer, fixed point, and floating point numbers. Numeric operations and assignment that would cause the most significant digits of numeric values to be truncated (e.g., when overflow occurs) shall constitute an exception situation.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

[Both CMS-2 and SPL/1 lack fixed point data types. TACPOL doesn't implement floating point. Fixed point is described in the J-73 specification. However, it is not currently implemented by any compiler. Neither version of JOVIAL supports an exception situation for fixed point overflow. (Indeed, few if any compilers do.)]

3.1.B Numeric Operations. There shall be built-in operations for conversion between numeric types. There shall be built-in operations for addition, subtraction, multiplication, division and negation for all numeric types. There shall be built-in equality (i.e., equal and unequal) and ordering operations (i.e., less than, greater than, less or equal, and greater or equal) between elements of each numeric type. Numeric values shall be equal if and only if they represent exactly the same abstract value. The semantics of all built-in numeric operations shall be included in the language definition. [Note that there might also be standard library definitions for numeric functions such as exponentiation.].

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

3.1.1 Floating Point Type

3.1.C Floating Point Precision. The precision of each floating point variable and expression shall be specifiable in programs and shall be determinable at translation

time. Precision specifications shall be required for each floating point variable. Precision shall be interpreted as the minimum precision to be implemented in the object machine. Floating point results shall be implicitly rounded (or on some machines truncated) to the implemented precision. Explicit conversion operations shall not be required between floating point precisions.

RATINGS

CMS-2(0) J-3B(0) J-73(1) TACPOL(1) SPL/1(0)

3.1.D Floating Point Implementation. A floating point computation may be implemented using the actual precision, radix, and exponent range available in the object machine hardware. There shall be built-in operations to access the actual precision, radix, and exponent range with which floating point variables and expressions are implemented.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(0) SPL/1(2)

[This requirement can be trivially satisfied in any language supporting floating point data types].

3.1.2 Integer and Fixed Point Types

3.1.E Integer and Fixed Point Numbers. Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.

RATINGS

CMS-2(1) J-3B(2) J-73(2) TACPOL(1) SPL/1(1)

[SPL/1 lacks fixed point data. TACPOL and CMS-2 allow implicit truncation in expression].

3.1.F Integer and Fixed Point Variables. The range of each integer and fixed point variable must be specified in programs and determinable at translation time. Such specifications shall be interpreted as the minimum range to be implemented.

Explicit conversion operations shall not be required between numeric ranges.

R A T I N G S

CMS-2(0)	J-3B(0)	J-73(0)	TACPOL(0)	SPL/1(0)

3.1.G Fixed Point Scale. The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable at translation time.

R A T I N G S

CMS-2(2)	J-3B(2)	J-73(2)	TACPOL(2)	SPL/1(0)

3.1.H Integer and Fixed Point Operations. There shall be built-in operations for integer and fixed point division with remainder and for conversion between fix point scale factors. The language shall require explicit scale conversion operations whenever the scale of a value must be changed to properly perform some operation (e.g., assignment, comparison, or parameter passing).

R A T I N G S

CMS-2(1)	J-3B(1)	J-73(1)	TACPOL(1)	SPL/1(1)

[None of the languages requires the explicit scale conversion].

3.2 BOOLEAN TYPE

There shall be a predefined unordered enumeration type for Boolean values. The Boolean type shall have operations for conjunction, inclusive disjunction, and negation.

R A T I N G S

CMS-2(2)	J-3B(2)	J-73(2)	TACPOL(2)	SPL/1(2)

3.3 COMPOSITE TYPES

3.3.A Composite Type Definitions. It shall be possible to define types that are Cartesian products of other types. Composite types shall include arrays (i.e., composite data with indexible components of homogeneous types) and records (i.e., composite data with labeled components of heterogeneous type).

RATINGS

CMS-2(1) J-3B(2) J-73(2) TACPOL(1) SPL/1(2)

[This is somewhat more generous than TINMAN evaluators. Here, with the exception of SPL/1, we assume that the "group" or "table" concept found in these languages approximates the record concept described above. In TACPOL and CMS-2, records are not strictly types].

3.3.B Component Specifications. For elements of composite types, the type of each component (i.e., field) must be explicitly specified in programs and determinable at translation time. Components may be of any type (including array and record types). Range, precision and scale specifications shall be required for each component of appropriate numeric types.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

3.3.C Operations on Composite Types. A value accessing operation shall be automatically defined for each component of composite data elements. Assignment shall be automatically defined for components that have alterable values. A constructor operation (i.e., an operation that constructs an element of a type from its constituent parts) shall be automatically defined for each composite type. An assignable component may be used anywhere in a program that a variable of the component's type is permitted.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

[None of the languages has a general constructor operation].

3.3.1 Arrays

3.3.D Array Specifications. The number of dimensions for each array must be specified in programs and shall be determinable at translation time. The range of subscript values for each dimension must be specified in programs and shall be determinable by the time of array allocation. The range of subscript values shall be restricted to a contiguous sequence of integers or to a contiguous sequence from

an enumeration type. [Note that translators may be able to produce more efficient object programs where subscript ranges are determinable at translation time.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

3.3.E Operations on Subarrays. There shall be built-in operations for value access, assignment, and catenation of contiguous sections of one-dimensional arrays of the same component type.

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(2) SPL/1(0)

3.3.2 Records

3.3.F Operations on Records. The assignment operation is to be defined between variables of type record, where both record definitions have identical names and components.

RATINGS

CMS(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

[Several of the languages do not enforce the checking feature implied above. If this judgment errs, it is on the generous side].

3.4 NOT EVALUATED

3.5 ENCAPSULATED TYPES

3.5.A Encapsulated Definitions. It shall be possible to encapsulate definitions. Encapsulations may contain definitions of the data elements comprising a type and of operations.

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(0) SPL/1(0)

3.5.B Effect of Encapsulation. The effect of encapsulation shall be to inhibit external access to implementation properties of the definition. In particular, declarations made within an encapsulation shall not automatically be accessible outside the encapsulation. Data elements defined in an encapsulation shall not automatically inherit the operations of the types with which they are represented.

RATINGS

CMS-2(0) J-3B(0) J-73(1) TACPOL(0) SPL/1(0)

[J-73 provides both own (RESERVE) data allocation as well as "IN an outer scope." This latter attribute permits definition of data referencable only within the inner scope but is allocated at the level of the specified outer scope].

3.5.C Own Variables. It shall be possible within encapsulations to declare variables that are accessible only within the encapsulation but remain allocated throughout the scope in which the encapsulation is declared. Such variables shall retain their values between entries to the encapsulation. It shall be possible to initialize such variables at the time of their apparent allocation.

RATINGS

CMS-2(0) J-3B(0) J-73(1) TACPOL(0) SPL/1(0)

[See 3.4.B].

3.5.D Operations Between Types. It shall be possible to define operations, like type conversion, that require access to local properties of more than one encapsulated definition. [Note that this capability violates the purpose of encapsulation and thus its use should be avoided wherever possible.]

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(0) SPL/1(0)

4. EXPRESSIONS

4.A Form of Expressions. The form (i.e., context free syntax) of expressions shall not depend on the types of their operands or on whether the types of the operands are built into the language.

RATINGS

CMS-2(2)   J-3B(2)   J-73(2)   TACPOL(2)   SPL/1(2)

4.B Type of Expressions. The language shall require that the type of each expression be determinable at translation time. It shall be possible to specify the type of an expression explicitly. [Note that the latter requirement provides a way to resolve ambiguities in the types of literals and to assert the type of results; it does not provide a mechanism for type conversion.]

RATINGS

CMS-2(2)   J-3B(2)   J-73(2)   TACPOL(2)   SPL/1(2)

[This requirement is met by any language which does not allow the type of its data to be changed during execution, and which requires or defaults declarations on both declared and parametric data].

4.C Not Evaluated

4.D Allowed Usage. Expressions of a given type shall be allowed wherever both constants and variables of the type are allowed.

RATINGS

CMS-2(1)   J-3B(2)   J-73(2)   TACPOL(1)   SPL/1(2)

4.E Constant Valued Expressions. Constant valued expressions (i.e., expressions whose values are determinable at translation time) shall be allowed wherever constants of the type are allowed. Such expressions shall be evaluated before execution time.

RATINGS

CMS-2(2)   J-3B(1)   J-73(2)   TACPOL(1)   SPL/1(1)

4.F Operator Precedence Levels. The precedence levels (i.e., binding strengths) of all infix operators shall be specified in the language definition, shall not be alterable by the user, shall be few in number, (e.g., three or four), and shall not depend on the types of the operands. [Note that there might be built-in operator symbols whose meaning is entirely specified by the user.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

4.G Effect of Parentheses. Parentheses shall override normal precedence rules.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

5. CONSTANT, VARIABLES, AND DECLARATIONS

5.A Declarations of Constants. It shall be possible to associate indentifiers with constant values of any type that is not dynamically allocated. Constants shall include both those whose values are determinable at translation time and those whose value cannot be determined until scope entry time. A translation time error shall be reported whenever a program attempts to assign to a constant valued identifier.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

5.B Declarations of Variables. There shall be no default declarations for variables. The type of each variable must be explicitly specified in programs and shall be determinable at translation time. Variables may be of any type.

RATINGS

CMS-2(1) J-3B(2) J-73(2) TACPOL(1) SPL/1(2)

[Note that J-73 has default attributes, but not default declarations].

5.C Scope of Declarations. The intended scope of a declaration shall be determinable from the program at translation time. Scopes may be lexically

embedded. Translators shall provide a warning wherever a local definition masks a more global definition. [Note that a function need not mask a more global function if they differ in name, number of parameters, or formal parameter types.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

5.D Restrictions on Values. Procedures, functions, types, labels, exception situations, and statements shall not be assignable to variables, computable as values of expressions, or usable as parameters to procedures or functions.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(2)

[Several of the languages can pass procedures as parameters. TACPOL and the JOVIALS also allow point labels to be passed].

5.E Initial Values. There shall be no default initial values for variables. The same syntactic form shall not be used both to declare constants and to initialize variables. [Note that initialization of variables must (except for some global variables) be accomplished during execution, not translation.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(1) SPL/1(1)

5.F Operations on Variables. Assignment and an implicit value access operation shall be automatically defined for each variable.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

5.G Other Declarations. It shall be possible to associate identifiers with specifications of type and representation (including range, scale, and precision). Such identifiers may be used in declarations of variables, to specify components of elements of composite types, and in formal parameter specifications.

RATINGS

CMS-2(0) J-3B(1) J-73(1) TACPOL(0) SPL/1(0)

[The JOVIAL's approximate this concept via "parametric defines"].

6. CONTROL STRUCTURES

6.A Not Evaluated

6.B Sequential Control. There shall be a sequential control mechanism (i.e., a mechanism for sequencing statements). Explicit statement delimiters shall be required. [Note the choice between terminators and separators can be left to the user.]

RATINGS

CMS-2(2)   J-3B(2)   J-73(2)   TACPOL(2)   SPL/1(2)

6.C Conditional Control. There shall be conditional control structures that permit selection among alternative control paths. The selected path may depend on the value of a conditional expression, on a computed choice among labeled alternatives, or on the true condition in a set of mutually exclusive conditions. The control action must be specified for all values of the discriminating condition.

RATINGS

CMS-2(2)   J-3B(2)   J-73(2)   TACPOL(2)   SPL/1(2)

6.D Short Circuit Evaluation. There shall be forms for short circuit conjunction and disjunction of Boolean expressions in conditional and iterative control structures.

RATINGS

CMS-2(0)   J-3B(0)   J-73(2)   TACPOL(0)   SPL/1(0)

[This capability could be easily added to any of the above languages].

6.E Iterative Control. There shall be an iterative control structure that permits a loop to have several explicit termination conditions and permits termination anywhere in the loop. Iterative control structures may be entered only at the head of the loop. [Note that when the number of iterations is zero or one and is determinable at translation time, the translator can omit any unnecessary object code.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

6.F Loop Control Variables. Loop control variables, if any, shall be local to the iterative control statement. Assignment shall not be allowed to control variables from the loop body. It shall be possible to iterate over sequences of integers and over elements of an enumeration type.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

6.G Explicit Control Transfer. There shall be an explicit mechanism for control transfer (i.e., the GO TO). However, the mechanism shall not permit:

- o transfers out of declarations, functions, procedures, or encapsulated definitions.
- o transfers into narrower scopes,
- o transfers into control structures,
- o transfers in the form of switches, designational expressions, label variables, label parameters, or alter statements.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(2)

7. FUNCTIONS AND PROCEDURES

7.A Function and Procedure Definitions. Functions (which return values to expressions) and procedures (which can be called as statements) shall be definable in programs. Existing functions (including those called using infix forms) and procedures shall be extensible to new data types (i.e., overloading shall be permitted).

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

[None of the languages seem to allow overloading].

7.B Not Evaluated

7.C Scope Rules. A reference to an identifier (other than an identifier for an exception situation) that is not declared in the most local scope shall refer to a program element that is lexically global, rather than to one that is global through the dynamic calling structure.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

7.1 FUNCTIONS

7.D Function Declarations. The result type for each function must be explicitly specified in the function declaration and shall be determinable at translation time. A function of two arguments may be specified as associative in this declaration. [Note that the latter requirement reduces the need for explicit parentheses.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

[This requirement is satisfied by almost every modern language].

7.E Restrictions on Functions. A function may only have input parameters and may not be called in a scope that contains variables that are referenced or assigned directly or indirectly within the body of the function. [Note that this requirement guarantees that parameters to functions can be implemented safely with either value or reference passing.]

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(0) SPL/1(2)

7.2 PARAMETERS

7.F Formal Parameter Classes. There shall be three classes of formal parameters: 1) input parameters, which act as constants that are initialized to the value of corresponding actual parameters at the time of call, 2) input-output parameters, which enable access and assignment to the corresponding actual parameters, and 3) output parameters, which act as local variables whose values are transferred to the corresponding actual parameter only at the time of normal exit. In the latter two cases the corresponding actual parameter must be

variable or an assignable component of a composite type.

RATINGS

CMS-2(1) J-3B(2) J-73(2) TACPOL(1) SPL/1(2)

7.G Parameter Specifications. The type of each formal parameter must be explicitly specified in programs and shall be determinable at translation time. Parameters may be of any type. Range, precision, and scale specifications shall be required for each formal parameter of appropriate numeric types. A translation time error shall be reported wherever corresponding formal and actual parameters are of different types and wherever a program attempts to use a constant or an expression where a variable is required.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

[TACPOL does not satisfy the requirement for explicit specification. None of the languages satisfy the range requirement. J-73 does not give an error on a mismatch--only a warning.]

7.H Formal Array Parameters. The number of dimensions for formal array parameters must be specified in programs and shall be determinable at translation time. Determination of the subscript range for formal array parameters may be delayed until execution and may vary from call to call. Subscript ranges shall be accessible within function and procedure bodies without being passed as an explicit argument.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

[All languages satisfy the first sentence. Only SPL/1 satisfies the second. None satisfy the third].

7.I Restrictions to Prevent Aliasing. Aliasing (i.e., multiple access paths to the same variable from a given scope) shall not be permitted. In particular, a variable may not be used as two output arguments in the same call to a procedure, and a nonlocal variable that is accessed or assigned within a procedure body may not be used as an output argument to that procedure.

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(0) SPL/1(0)

[This requirement attempts to prevent problems with side-effects. Note that it is relatively straightforward (although expensive) to include it in any of the languages evaluated].

8. INPUT OUTPUT FACILITIES

Deleted. Not appropriate for imbedded systems.

9. PARALLEL PROCESSING

Deleted. Not applicable to imbedded systems.

10. EXCEPTION HANDLING

10.A Exception Handling Facility. There shall be an exception handling mechanism for responding to unplanned error situations detected during program execution. The exception situations shall include errors detected by hardware, software errors detected during execution, error situations in built-in operations, and attempts to execute portions of programs that are not present in main memory. Exceptions should add to the execution time of programs only if they are invoked.

RATINGS

CMS-2(1) J-3B(0) J-73(1) TACPOL(1) SPL/1(0)

10.B Software Error Situations. The software errors detectable during execution shall include exceeding the specified range of an array subscript, exceeding the specified range of a variable, exceeding the implemented range of a variable, attempting to access an uninitialized variable, and failing to satisfy a program specified assertion [Note that many range checks can be done during translation thereby reducing execution costs.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

[Satisfaction of this requirement is totally a function of the compiler and the

run-time error software. It can be included in a straight forward manner in any of the languages evaluated].

10.C Invoking Exceptions. During any function or procedure execution it shall be possible to invoke an exception situation in the calling statement. This exception shall cause termination of the routine and an immediate transfer of control in the caller. Such exceptions must be specified in the definition of the function or procedure. Exceptions that can be invoked by built-in operations shall be given in the language definition.

RATINGS

CMS-2(1) J-3B(0) J-73(1) TACPOL(1) SPL/1(0)

10.D Processing Exceptions. There shall be a control structure for discriminating among the exceptions that can occur in a specified portion of a program. Exceptions that are not processed at a given function or procedure level shall cause termination of the function or procedure and shall invoke an exception in its caller.

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(1) SPL/1(0)

[Note that the action implied by the last sentence is equivalent to passing a label to a procedure. The concept can be dangerous and expensive to implement].

10.E Order of Exceptions. The order in which exceptions in different parts of an expression are detected shall not be guaranteed by the language or by the translator.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

10.F Assertions. It shall be possible to include assertions in programs. If an assertion is false when encountered during execution, it shall invoke an exception. [Note that assertions can also be used to aid optimization and maintenance.]

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(0) SPL/1(0)

[Although not currently met by any of the languages evaluated, this requirement would be trivial to implement in any of them].

10.G Suppressing Exceptions. It shall be possible to suppress individually the detection of exceptions for software error situations. Should such a situation occur when its detection is suppressed, the consequences will be unpredictable.

RATINGS

CMS-2(0) J-3B(0) J-73(0) TACPOL(0) SPL/1(0)

11. SPECIFICATIONS OF OBJECT REPRESENTATION

11.A Data Representation. The language shall permit but not require programs to specify the physical representation of data. These specifications shall be distinct from the logical descriptions. Specifications for the order of fields, the width of fields, the presence of "don't care" fields, the positions of word boundaries, and the object representation of atomic data shall be allowed. If object representations are not specified, they shall be determined by the translator.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(1) SPL/1(0)

11.B Multiple Representations. It shall be possible in programs to define more than one physical representation (e.g., packed and unpacked) for elements of a given type, and to associate a specific representation with each variable of that type. [Note that changes of representation can be accomplished through assignment.]

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(1) SPL/1(1)

11.C Not Evaluated

11.D Not Evaluated

11.E Code Insertions. For some object machines it shall be possible to write programs that include encapsulated code written in machine language or in other established programming languages. Such facilities shall be modest and shall attempt to maximize safety. The language should be designed to minimize the need for code insertions.

RATINGS

CMS-2(2) J-3B(1) J-73(2) TACPOL(1) SPL/1(0)

[For practical reasons the insertable code must be limited to specific languages, such as Assembler Language for the target machine].

11.F Optimization Specifications. It shall be possible in programs to specify the optimization criteria to be used. It shall be possible to specify whether minimum translation costs or minimum execution costs are more important. In the latter case the user may also specify whether execution time or memory space is to be given preference. The meanings of program constructs (other than execution time and space) shall not depend on the optimizations that are applied.

RATINGS

CMS-2(0) J-3B(0) J-73(2) TACPOL(0) SPL/1(0)

12. LIBRARY, SEPARATE COMPILATION, AND GENERIC DEFINITIONS

12.A Library Entries. The language shall support the use of an external library of definitions and separately compiled segments. Library entries shall include type definitions, input-output packages, common pools of shared declarations, and application oriented software packages. The library shall be structured to allow entries to be associated with a particular application, project or user.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

12.B Separately Compiled Segments. The language shall support the assembly of separately compiled program segments into an operational program.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

12.C Restrictions on Separate Compilation. Separate compilation shall not change the meaning of a program.

RATINGS

CMS-2(2) J-3B(2) J-73(2) TACPOL(2) SPL/1(2)

13. SUPPORT FOR THE LANGUAGE

13.A Defining Documents. The language shall have a complete and unambiguous definition. It should be possible to predict the complete action of any syntactically correct program from the language definition. The language documentation shall include the syntax, semantics, and appropriate examples of each feature including those for standard library definitions. The defining documentation might point out the relative efficiency of alternative constructs.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

13.B Standards. There will be a standard definition of the language. Procedures will be established for standards control and for certification that implementations meet the standard.

RATINGS

CMS-2(2) J-3B(1) J-73(2) TACPOL(2) SPL/1(2)

[These ratings are driven by DoD's own actions in DoD Directive 5000.31].

13.C Not Evaluated

13.D Translator Diagnostics. Translators shall be responsible for reporting errors that are detectable at translation time and for optimizing object code. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and shall provide warnings of unusually expensive constructs. A representative set of translation time diagnostic and warning messages shall be included in the language definition.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

13.E Translator Characteristics. Translators should fail to compile correct programs only when the program exceeds the resources or capabilities of the intended object machine or when the program requires more resources during the translation than are available on the host machine. Translators shall report an error when a program requires memory, devices, or special hardware that are unavailable in the object machine. Neither the language nor its translators shall impose arbitrary restrictions on language features. That is, they shall not impose restrictions or the number of array dimensions, on the size of data structures, on the size of set types, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels unless such restrictions are dictated by the limitations of the host or object machine and are documented in user accessible manuals.

RATINGS

CMS-2(1) J-3B(1) J-73(1) TACPOL(1) SPL/1(1)

APPENDIX B - GLOSSARY

ANSI	-	American National Standards Institute
DOD	-	Department of Defense
DOD-1	-	Multipurpose HOL being defined by HOLWG
HOL	-	High Order Language
HOLWG	-	High Order Language Working Group
Macros	-	Sequence of coding available by mnemonic reference with parameters.
Micro-programmable	-	The ability to build various analytic instructions as needed from the subcommand structure of a computer.
MOL	-	Machine Oriented Language
NBS	-	National Bureau of Standards
OASD	-	Office of the Assistant Secretary of Defense
Syntax	-	The rules governing the structure, expressions and relationships in a language.