HIGH LEVEL AEROSPACE COMPUTER PROGRAMMING LANGUAGE CONFERENCE

Naval Research Laboratory
Washington, D.C.
29 and 30 June 1970

Sponsored by the Naval Air Systems Command
Washington, D.C.

INDEX

INTRODUCTORY REMARKS — Page 3

PAPERS

Digital Computers: A Decade of Advancement — 7
Bernard A. Zempolich

The Advanced Avionic Digital Computer — 15
Ronald S. Entner

The Inclusion of Test-Type Instructions In High Level Language Syntax — 31
Roger W. Peretti

Providing An Efficient Match Between a High Level Programming Language and a Computer Instruction Repertoire — 55
Ralph Jenkins

CLASP - Its Role in AADC Software Development — 81
Edward H. Bersoff

Space Programming Language: Flight Software Comes of AGE — 103
Robert E. Nimensky

A Technical Overview of Compiler Monitor System 2 — 111
John P. O'Brien

PANELS

High Level Language Compatibility — 151

Computer / Compiler Standardization — 171

APPENDIX A: CMS-2 Compiler Design — 202

APPENDIX B: List of Attendees — 241

THIS PAGE INTENTIONALLY LEFT BLANK

INTRODUCTORY REMARKS

As digital systems grow in size and complexity, and as their applications increase in scope and dynamics, the need for an inclusive high level programming language becomes increasingly desirable. Unfortunately, digital computers procured for avionic and aerospace applications are seldom used in more than a single host system, thereby militating against general high level programming languages and the large, complex compilers required to efficiently translate programs into object code. This circumstance often results in one or more of the following situations;

a. Programmers are required to work with lower level assembly or machine languages, resulting in long lead times, high cost, high error rates, inflexible programs and poor documentation.

b. Programmers are required to use limited compiler languages, resulting in inefficient problem definition and, hence, inefficient object code which often requires tedious hand correction and recoding. This, again, leads to high cost, low flexibility and poor documentation.

c. The computer system purchaser, i.e. the Government, must defray the development costs of an adequate high level language and compiler which, in turn, leads to long lead times and tremendous expense. In addition, it is now necessary to train personnel to use the new language, as well as find a suitable computer upon which to run the new compiler.

The above situations are particularly striking when one considers the coding efficiency required for avionic applications. Aerospace computers have historically been memory limited. It is unlikely, therefore, that any compiler with a coding efficiency which is much less than hand coded efficiencies would be considered acceptable for such

applications. A suitable aerospace compiler, therefore, must, by definition, be both extensive and expensive. Several organizations have attempted to approach this problem by establishing a high level language compatibility requirement. The chief advantages of this approach reside in the areas of documentation and format, and to some extent, in the compiler front end, or "syntax analyser". This approach does not, however, contribute significantly toward the standardization of the larger body of the compiler itself. This portion of the compiler, which fits the operational program to the architecture and control structure of the object machine, greatly depends on the structural vagaries of the interfacing object computer. The metacompiler concept was developed to cope with this complication.

A metacompiler accepts, in addition to a high level language program, a definition of an object machine's architecture and control structure, thereby permitting the metacompiler to "adapt" to alternate object computers. While this approach permits the compiler to accommodate more than a single computer design, the metacompiler can be significantly more expensive to develop and implement than a conventional compiler.

On 29 and 30 June, 1970 a High Level Aerospace Computer Programming Language Conference was held at the Naval Research Laboratory, Washington, D.C.. The languages discussed at the conference were Compiler Monitor System -2, Space Programming Language, and Computer Language for Aeronautics and Space Programming. The purpose of the conference was to address the relative merits of each language with respect to avionic applications, as well as discuss high level aerospace programming language compatibility and computer hardware requirements (i.e. common instruction repertoires, standard word formats, etc.) which could lead to some measure of compiler standardization.

These proceedings provide a record of that conference. It is

hoped that this document will be reviewed with an eye toward future digital computer technology and requirements. Any attempt to remedy the problems of the past without attempting to prevent new problems in the future is, undoubtably, an inappropriate modus operandi.

The participation of the following parties is greatfully acknowledged:

- -- the authors for their papers and presentations
- -- the panelists and panel chairmen for their time and cooperation
- -- the Naval Research Laboratory for providing more than adequate facilities
- -- the Navy personnel who graciously contributed toward coffee and donuts.

RONALD S. ENTNER

Conference Coordinator

THIS PAGE INTENTIONALLY LEFT BLANK

DIGITAL COMPUTERS: A DECADE OF ADVANCEMENT

By Bernard A. Zempolich
Head, Tactical Computer Section
Avionics Division
Naval Air Systems Command

The digital computer is playing an increasing important role in Naval Aviation, the forces afloat, and all shore activities. The A-6 Intruder is a good example of the use of a digital computer in Naval Aviation and serves as an indicator of the progress made in the computer technology in recent years. A digital computer was first introduced as part of the A-6 system in the late 1950's. This computer had a limited functional capacity when compared with the computers of today but it played an important part in the increased operational capability which the A-6 system brought to the fleet. Now, nearly 10 years later, a second-generation digital computer with a much greater functional capacity and performance capability is being developed for the current A-6 system.

Digital computers of varying degrees of complexity were similarly introduced in a variety of naval aircraft applications and significant advances in computer capabilities have been made in succeeding years. These advances are due to the substantial progress made in the overall computer technology in recent years and particularly to the advances made in digital micro-electronic circuits. The ability of today's computers to handle more functions and operate at higher speeds has resulted in an "explosion" of applications for many different types of naval aircraft.

The paper presents an overview of this advancing computer technology as it relates to tactical functions, applications, test and evaluation, training and simulation. The reasons for the proliferation of many different kinds of computers of special design are also presented.

Tactical Functions:

The basic tactical functions performed by naval aircraft that incorporate general-purpose, programmable, digital computers as part of the avionics system are:

- Fleet air defense.
- Distant air superiority.
- Airborne early warning and control.
- Reconnaissance.
- Antisubmarine warfare (ASW)
- Attack.
- Ground support-tactical.
- Transportation.
- Medical evacuation.
- Test and checkout equipment.
- Countermeasures.
- Intelligence processing.

In each of these functions the computer is used for extremely fast calculations of arithmetic operations performed on input data, storage of data, and transfer of either raw or "operated upon" data. The digital computer used to perform these operations varies from system to system.

Applications:

The data-processing applications performed by computers in processing the functional tasks described previously are:

(1) Navigation

Doppler navigation
Flight control
Autopilot
Air data computations
Terrain-following
Stationkeeping

(2) System test

Built-in self-test
In-flight performance monitor
Automatic fault-detection
Operator training

(3) Weapons delivery and control

Weapon assignment
Ballistics data storage
Intercept solutions
Air-to-Air
Air-to-Ground
Air-to-Subsurface

(4) Target data processing

(5) Threat evaluation

(6) IFF

(7) IR

(8) Radar

(9) Signal processing

(10) Data display

(11) Sensor correlation

(12) Data link

(13) Intelligence generation

Depending on the specific program, any given computer system may perform more than one of the aforementioned "task area" groups. It can be expected that with future aircraft, computers will be responsible for more multimode applications. Applications that are not listed but which are contained within current planning are: (1) low-light-level television (LLLTV), (2) laser target designation (LTD), (3) battle-damage assessment (BDA), (4) airframe-performance monitoring (APM), and (5) recording of in-flight voice communications.

Test and Calibration:

The sophistication and complexity of the newer avionics systems have led to test and calibration methods which require the automaticity and speed of a programmable, general-purpose digital computer. Present systems being developed to provide the important task of test and calibration are: (1) versatile avionics shop test (VAST) for CVA avionics support; (2) The Naval Air Rework Facility NARF-550 test station for depot repair; and (3) computer-controlled test stations for specific systems such as the Carrier Aircraft Inertial Navigation System (CAINS) and certain electronic countermeasures (ECM) applications

Training:

The same factors that led to the incorporation of computers in avionics systems have created the need for computer-controlled trainers such as the operational flight trainers (OFT) and weapons system trainers (WST). Although the number of OFTs and WSTs that now use computers is small, it is reasonable to expect that most weapons systems currently being developed

and most certainly future systems - will utilize WSTs with general-purpose digital computers. A recent example is that of the WST for the Phoenix Missile System AN/AWG-9 Airborne Missile Control System (AMCS).

The utility of computers in naval aircraft maintenance trainers (NAMTs) is somewhat undefined at this time. However, again it is reasonable to expect that computers may very well be used to help train personnel in automatic fault-isolation oftroubles. It would appear that this application would be a direct outgrowth of the on-board built-in test and automatic fault-detection capabilities now being incorporated into certain weapons systems.

Simulation:

Digital computers are being used extensively at a number of NAVAIR field activities as well as in private industry to simulate a myriad of operational environments in which the system will be employed. In general, most of these computers are commercial in nature except for those being used for operational programming (software) by the Fleet Computer Centers (FCPCs).

Proliferation:

The many advances made in the general computer technology and the realization that these devices have the potential for widespread application have created a demand for the development of a variety of digital computers to meet specific operational needs. This has resulted in a proliferation of computer designs which do not have a general utility

but instead are "tailored" for a specific purpose in a specific system
The reasons for this proliferation of computers throughout the fleet in the last decade are:

a. Different memory capacity required. The data-processing requirements differ widely among systems and subsystems. Each single computer is normally designed to contain sufficient memory capacity to fulfill the specific weapons system operational requirements.

b. Use in different Vehicles. The space and weight restrictions imposed on each system have led to new and/or repackaged designs. In addition, maintenance factors (i.e., human factors) and on-board turnaround times have also created the need for new design and development.

c. Different procurement methods. Under the total package procurement plan, the Navy does not necessarily have direct control over all components which go to make up the total weapons system.

d. Equipment procured at different periods. Computers procured for a 1961 development obviously will not be the machines wanted for a project starting development in 1971.

e. Urgency to implement systems to meet operation al requirements. In many cases, the urgency of the operational requirement dictates quick-re action capability (QRC) procurement and use of whatever is available, from all sources, that will do the job.

f. Procurement by different activities. No two separate engineering groups can be expected to reach identical technical conclusions on every problem. In addition, it was only recently that within the Naval Air Systems Command (NAVAIR) cognizance for digital computer development has been vested in one section.

Efforts are being made to reduce the proliferation of computers of special design. General purpose-type computer designs are being utilized when possible and certain common elements of the computer systems such as program formats, circuits, and packaging are being developed as standardized elements for use in new digital computer designs.

Summary:

During the past 10 years there has been a dramatic growth in the application of general-purpose programmable digital computers in Naval Aviation. Their inherent advantages--speed of calculation and storage capability for voluminous data in limited space--are responsible for their increasing use in various weapons systems. An equally significant factor is the ease with which the microelectronic technology has been successfully incorporated in new computer systems designs.

THIS PAGE INTENTIONALLY LEFT BLANK

THE ADVANCED AVIONIC DIGITAL COMPUTER

By Ronald S. Entner
Naval Air Systems Command
Washington, D.C.

The Naval Air Systems Command hopes to develop a digital computer system which will optimize avionic subsystem performance through improved systems integration. This system should be responsive to dynamic operational requirements. The system should employ modular hardware and software and provide the means to utilize new magnetic and semiconductor technologies, which could lead to improved performance at reduced cost. Examples of these technologies are: Large Scale Integration, Medium Scale Integration, Electron Mask Generation, Ferroacoustics, etc..

The AADC program is, in essence, the outgrowth of a search for a cost-effective application of LSI technology in the area of computer systems. To that effect, it is hoped that an appropriate level of system modularity can be established to optimize on this new technology. Furthermore, since the computer will be more than the sum of its logic and storage, an approach to component interface is also an essential development.

Because the design and set-up costs for LSI production may actually exceed the cost of fabrication, itself, it would be useful to develop a system which utilizes a minimum of LSI types. This goal, in turn, requires that these types be general purpose enough to be used in a collection of larger systems. By quantizing at the byte-functional or functional levels, it is believed that the necessary goal of universality can be

realistically achieved. Byte-functional modularity is additionally compatible with LSI design requirements for partitioning and interconnection.

NAVAIR is currently developing the necessary package to support monolithic LSI wafers up to three inches in diameter, operating within a military aircraft environment (i.e. MIL-E-5400 Class 4). The package will support multichip and hybrid technologies, as well. The package is about four inches square, a half inch thick, provides three hundred points of electrical contact, and will dissipate 25 watts. Hopefully, prototype packages will undergo environmental testing by the end of fiscal year 1971.

The second level of packaging employs a zero force module insertion methodology. This approach will prevent undue stress during module insertion from fracturing the delicate, single crystal wafer. This capability is provided by a cam actuated contact mechanism which couples to the module's electrical contacts with sufficient force to meet military specifications.

Two years ago, when it was decided to proceed with a hardware development, it was necessary to chose a system architecture which would provide performance and permit modularity. Two design roads were open: the first, to establish basic computer functions, which could then be translated into LSI hardware and ultimately assembled into large macrosystems; the second, to begin with a worst case estimate and partition down to elemental modules. The latter road was chosen, as it provided assurance of meeting worst case processing requirements, as well as they could be predicted. Several computer architectures were analyzed in the attempt to define the

worst case design. After considerable design and analysis, the AADC Baseline Organization evolved.

The Optimized Simplex Processor (OSP), while chronologically out of order, does help explain the memory partitioning and hierarchy techniques used in some of the more complex AADC structures, of which the Baseline Organization is one. Briefly, there are three memory elements found in the OSP: Bulk Store, Main Store and Task Memory. The Bulk Store is a Block Organized Random Access Memory (BORAM) employing ferroacoustic technology. Ferroacoustics provides NDRO readout, 1 usec block access time, 70 nsec word cycle time, and non-volatility. This memory is used to store invariant programs and data, which constitute as much as 90% of an aerospace computer's storage burden. This, in turn, results in paged software structures. The Random Access Main-store Memory is NDRO or protected DRO, exhibits an approximately 250 nsec cycle time (due to system interfaces), and is non-volatile. The RAMM is used for variable data and Input/Output buffer storage. The Task Memory is a high speed random access memory with a cycle time small enough to permit unbuffered transmission from BORAM. This memory provides random access to Program Modules (PM's) transfered from Bulk Storage. In operation, tasks which are stored as pages in BORAM are transfered into Task Memory, followed by data transfers from Main Store. The processor then executes directly from Task Memory. The capability is provided, however, to operate directly from Main Store in the event that large volumes of variable data are encountered (e.g. matrix computations).

The Time Division Multiplex Block Transfer Multiprocessor is, essentially, a cluster of OSP's on a shared bus system.

In this architecture, each OSP operates asynchronously. A program is brought down just as before from Bulk Store to Task Memory, the variable data accessed from the Main Store, and then computation begins, after which the next processor is loaded, etc.. The primary advantages of this approach are 1) a significant reduction in memory conflicts, assuming correct scheduling, 2) the ability to share data and routines without resorting to multiple storage areas and 3) the locations of programs and data are non-critical since multiple simultaneous accesses are nearly non-existant, which, in turn, leads to 4) the ability to dynamically reconfigure software in the on-line system. Among the disadvantages of this approach are 1) time loss due to memory transfers, 2) storage problems for tasks generating or utilizing large volumes of variable data, which is compounded by the fact that there are now multiple subscribers to the Main Store and 3) optimal resource utilization.

To improve the transfer to processing time ratio, AADC will attempt to employ multiple addressing, macro instructions and, when necessary, wide intermemory buses. To reduce the problem of operating with large volumes of variable data, a Matrix-Parallel Processor (MPP) has been postulated. The MPP consists of an Associative or Array Processor, an Associative or Pseudo-Associative Memory and some form of frequency analysis and synthesis device. All these elements are modular.

The Master Executive Control (MEC) component of the AADC is, essentially, responsible for schduling and system resource utilization. Depending on the sophistication of a

particular version of the AADC, the MEC may be implemented as software, hardware or some suitable combination. The TDM multiprocessor would use a software, floating executive, for example.

The AADC Baseline Organization illustrates an architecture employing all hardware elements embodied in the AADC development. The Baseline combines the TDM multiprocessor, the Matrix-Parallel Processor, a hardware Master Executive Control and two I/O systems: a high speed multiplexed I/O and additional dedicated channels for special I/O requirements.

The Processing Elements used in the Baseline are configured along functional and byte-functional lines. The arithmetic function is partioned by byte, permitting whole and part word operations, as well as providing a convienent means of developing independent hardware mantissa and exponent processors for floating point operations. The control unit, however, would not be enhanced by byte partitioning, as only whole instruction words are meaningful (half word control is not currently anticipated for AADC). Task Memory is configured by byte and numbers of words. The Task Memory byte may be an integral number of bytes greater than the arithmetic unit byte. Optimal byte size will be determined by simulation. Ultimately, byte size may very well be determined by available technology.

While the MPP will functionally consist of the three items already addressed, work is in progress to provide the capability to perform matrix operations within the AP itself. This will, hopefully, eliminate the need for a frequency analysis (i.e. FFP) element.

The AADC will interface with a Generalized Multiplexed

Communication System. This system will be under Master Executive Control, thereby permitting dynamic reallocation of communication resources. The approach will, in addition, extend AADC processing availability to each system on the communication bus. This situation could be used to provide back-up computational and control capability in the event of certain subsystem malfunctions.

The Master Executive Control element found in the Baseline Organization consists of an executive processor and an associative memory for task and resource status keeping. In its final implementation, the MEC processor may be the same control unit used in the Processing Element. The reliability of the MEC need only be a few times greater than the PE reliability to provide adequate operation at minimal risk to overall system reliability when compared to a floated executive implementation. In any event, the use of a hardware executive should only be considered as a worst case alternative, the necessity of which is currently under study.

By coordinating the development of AADC hardware and software, it is believed that both hardware and software goals can be more easily achieved. An important element in the development of both is the availability of an algorithm bank. The algorithms stored in such a bank can be used to judge the applicability of various programming languages, can be used to establish the throughput and special features required of the hardware, and can be used to write application and simulation programs.

The AADC program addresses the use of a metacompiler system for translating source code into object code for

various system configurations. This element will be part of a system synthesizer which, in turn, will hopefully be part of an Automated Design Facility (ADF). It is hoped that the ADF will be able to reduce a Specific Operational Requirement (SOR) into useful hardware and software in a fraction of the time required by conventional proceedures. In addition to compiling applications programs, the synthesizer will generate the necessary executive parameters to enable the MEC of a particular version of the AADC to schedule the execution of the problem oriented tasks. Scheduling will occur on-line and in real time.

Microprogramming, it is believed, will prove a useful attribute in AADC. It will be possible, through the use of microprogramming techniques, to modify, on a task to task basis, the instruction repertoire of each AADC Processing Element. This will permit the use of high level instructions which will provide a better match between object and source code than heretofore possible. Microprogramming may also lead to some measure of computer emulation capability, thereby permitting the use of old, but nevertheless useful software. The exact means of achieving cost-effective microprogram control is currently under study.

The AADC program presently straddles a region between system design and hardware/software considerations. Memory and packaging work are about to enter a hardware development stage. Current plans are to build the Optimized Simplex Processor as a first milestone. It is projected that this first version of the OSP will be, effectively, a two millon operations per second machine, based on a 30:70 long to short instruction

mix. This ratio is based on analysis of conventional avionic program instruction mixes. Future programs will probably exhibit a greater preponderance of long instructions due, in part, to the use of microprogrammed control. The execution time of these long instructions will probably be less than conventional long instruction execution times, however, because of the nature of microprogramming.

ADVANCED LSI/MSI PACKAGING

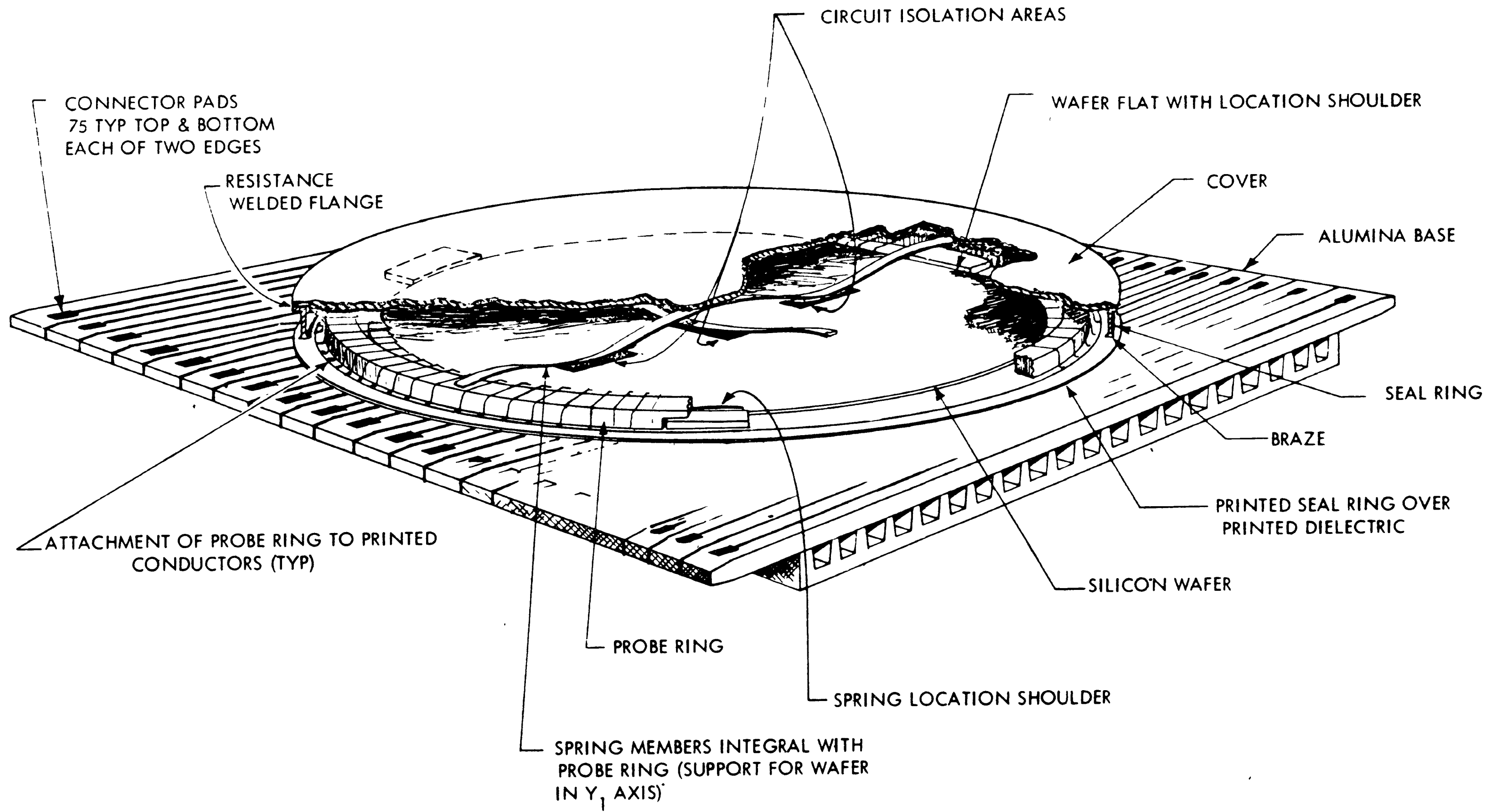
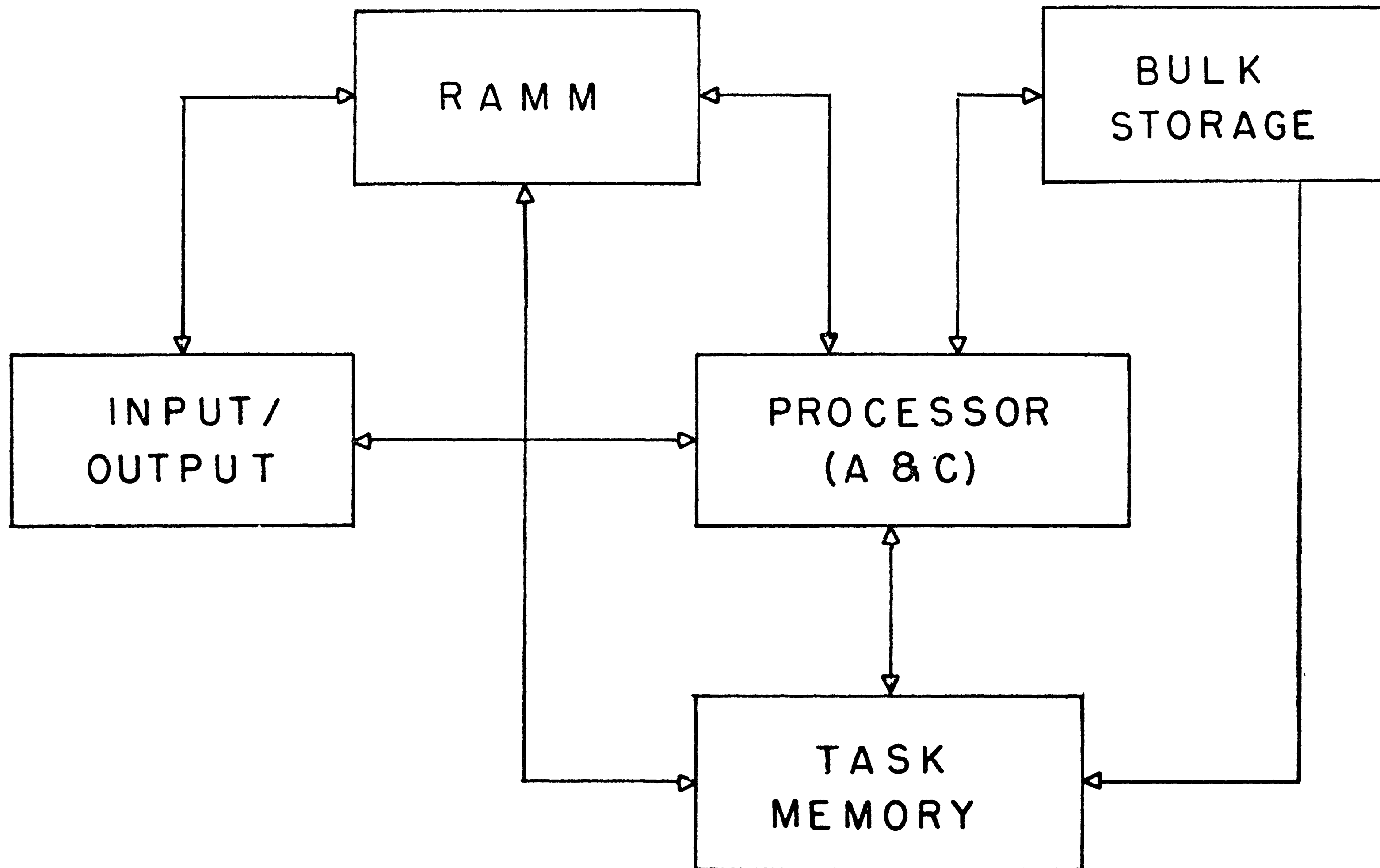
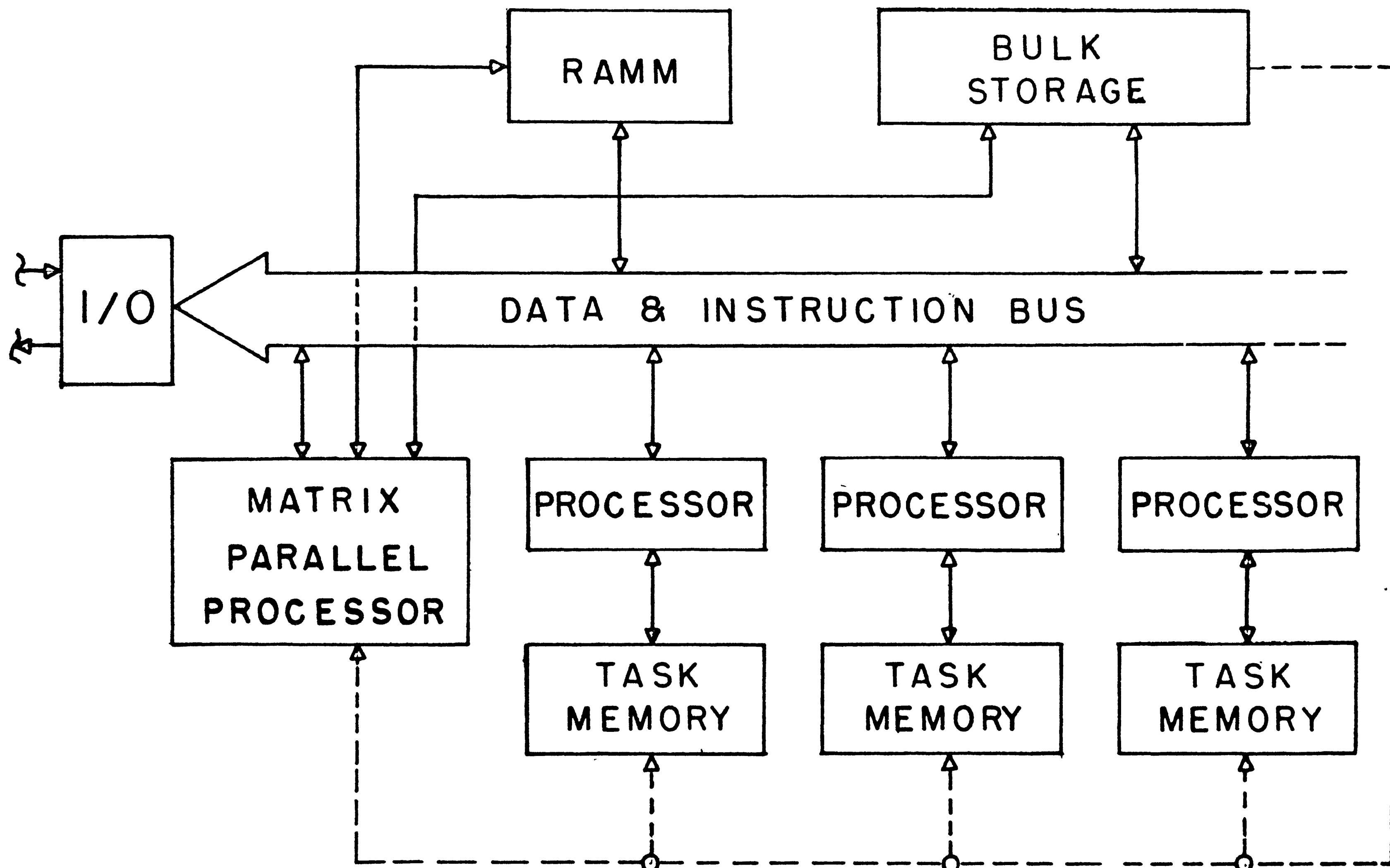


FIG. 1



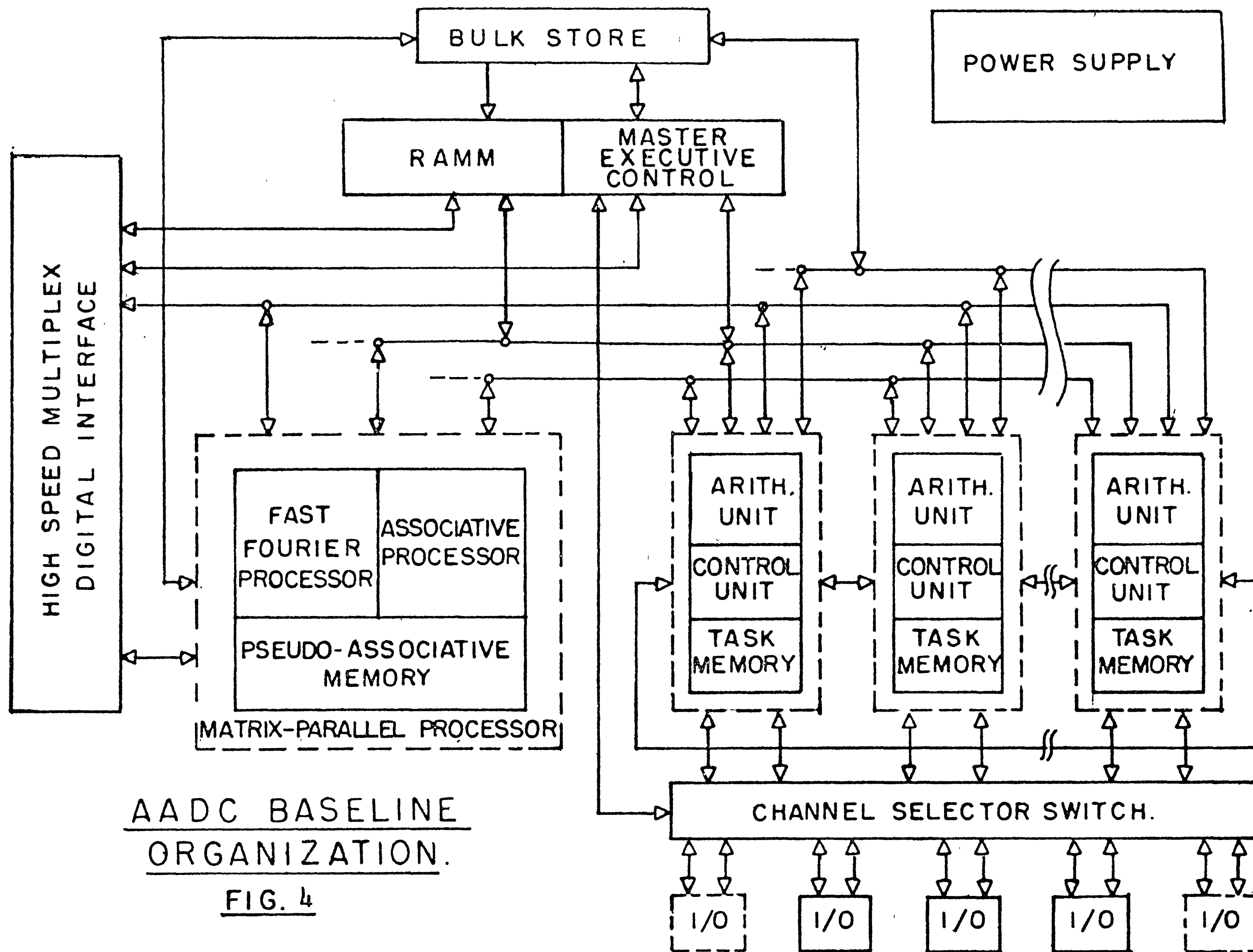
OPTIMIZED SIMPLEX PROCESSOR.

FIG. 2



TDM BLOCK TRANSFER MULTIPROCESSOR.

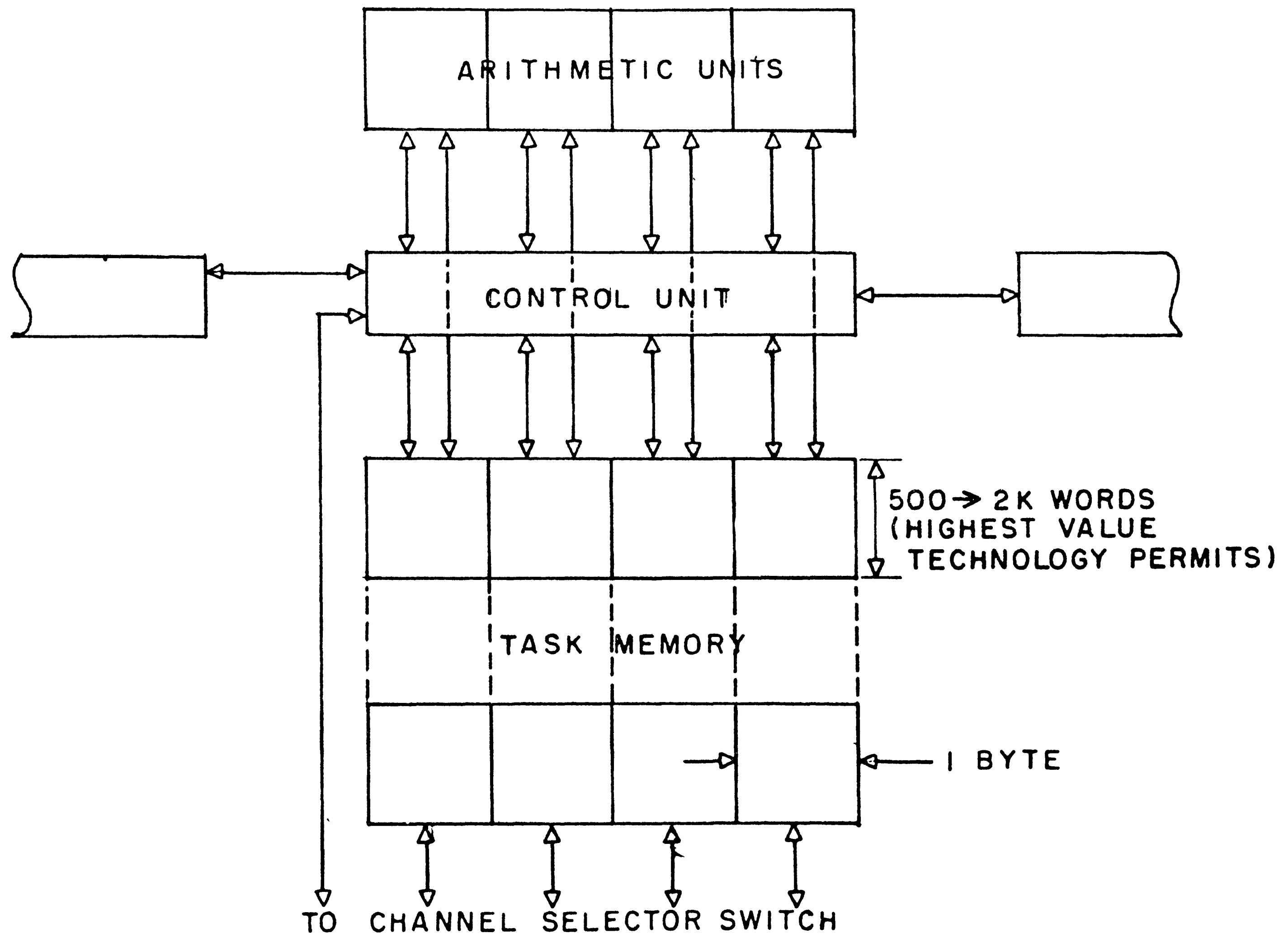
FIG. 3



AADC BASELINE ORGANIZATION.

FIG. 4

JJH AEDC OCT.20,1969



PROCESSING ELEMENT DETAIL.

FIG. 5

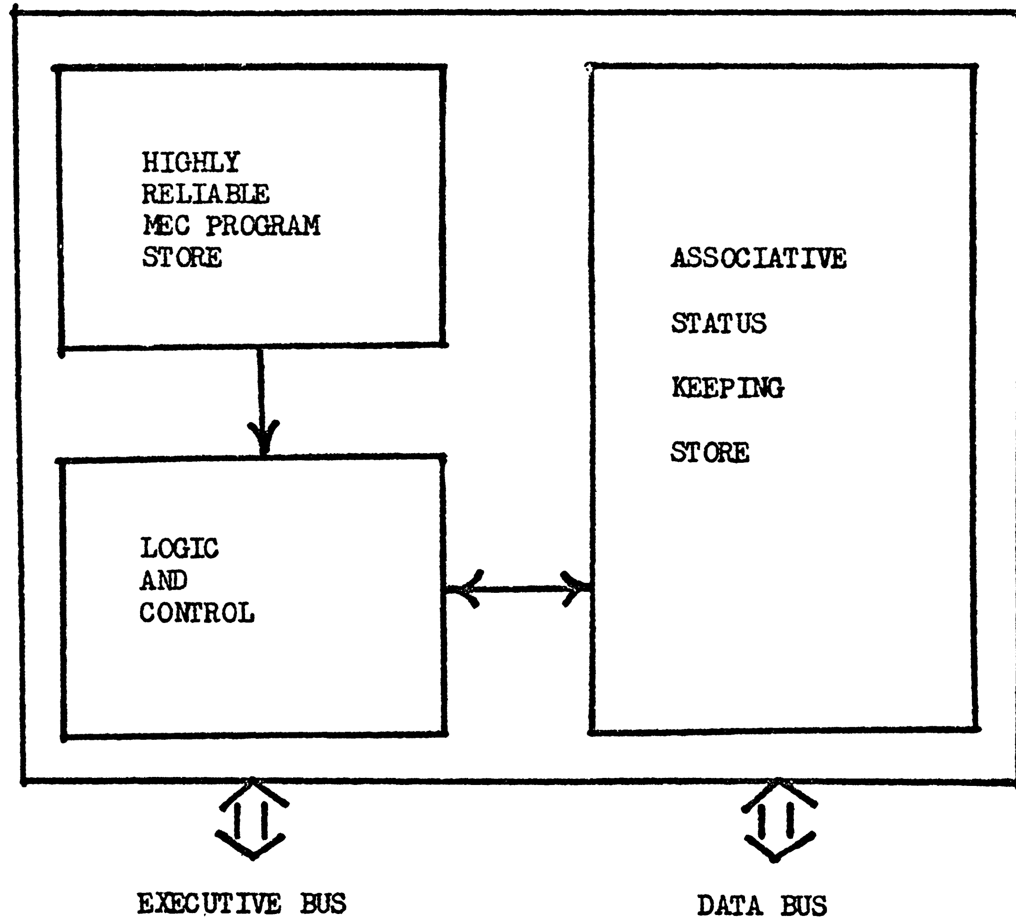


FIGURE 6: Baseline Master Executive Control Hardware Elements

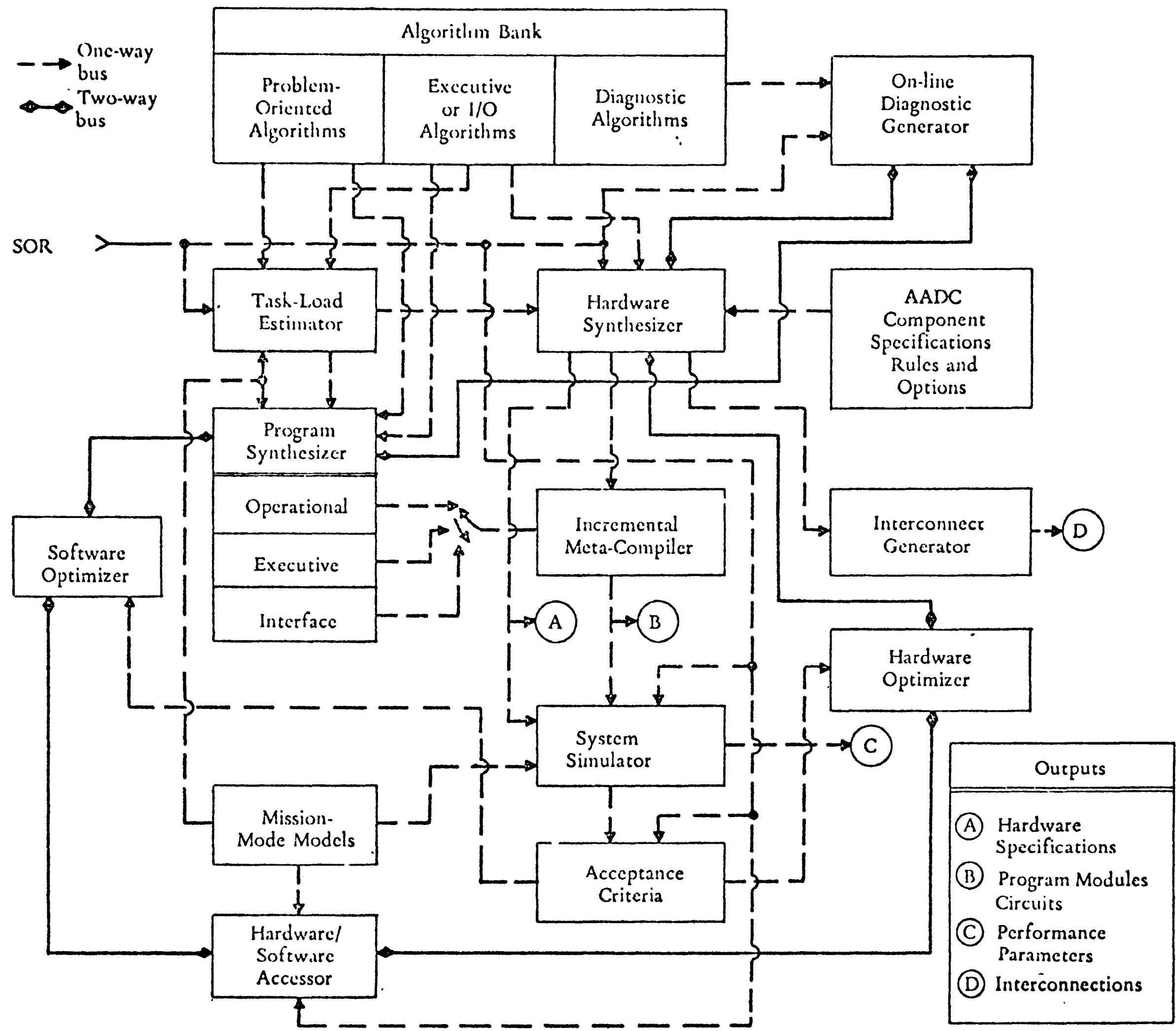


FIG. 7: Preliminary AADC Automated Design Facility Functional Block Diagram

THIS PAGE INTENTIONALLY LEFT BLANK

THE INCLUSION OF TEST-TYPE INSTRUCTIONS IN HIGH LEVEL LANGUAGE SYNTAX

ROGER W. PERETTI

29 JUNE 1970

Roger W. Peretti RWP | Computer Engineering | 35 | 1083 | DATE 7 August 1969
NAME | DEPT. NAME | PLANT NO. | EXT.

SUBJECT: USER ORIENTED AEROSPACE LANGUAGE DEVELOPMENT

o INTRODUCTION AND SUMMARY

In general, language development can be broken down into three phases -- definition, interpretation, implementation. There are at least two major steps involved in the definition phase, namely:

(1) Performing a detailed analysis of the presumed language requirements.

(2) Development of the actual statements and syntax* for the language.

Once the syntax has been defined and found to be unambiguous, we are prepared to determine the semantical interpretation which is to be associated with each statement. The final step in developing a language involve implementation and testing. Each of these tasks can be monumental.

During May 1969, an A.D. program was initiated in order to develop a user oriented aerospace language. The primary goal of this program was to analyze the O.B.C. software requirements for the 1975-80 time period, and either incorporate these requirements (in the form of statements) into the repertoire of an existing language, or, develop a new language which would incorporate them. The criteria for going one way or the other are explained at the end of this memo, and largely depend on language efficiency. Ranked according to their presumed importance, the features to be incorporated into these statements include:

---

* A language consists of a set of valid or admissible strings, and the meanings or interpretation associated with these strings. The syntactical definition of a language refers to the rules that define the structure of valid statements and programs.

(A) Must have English-like input statements -- to make routines accessible to non-programmers.

(B) Must produce efficient code -- to optimize speed and storage.

(C) Should be flexible -- to allow for future expansion.

(D) Should contain hardware implemented routines -- required for code optimization.

The A.D. project is still in the definition phase. Corresponding to (1), several test-type languages* were examined in order to take advantage of the analysis that was involved in their development. Each language was determined to be too hardware oriented, too "ground-based" oriented, and in general, too complex for the intended airborne application. Using the F-14 O.B.C. system as a baseline, and projecting into the 1975-80 time frame, we see a requirement for statements which TEST the GO/NO-GO status of an avionic system, and DISPLAY failure information. Other statements would be available to perform a LIMIT TEST, and TRENDing ANALYsis**.

After a preliminary analysis it was determined that CMS-II and METAPLAN should be examined, to determine whether either of these languages could be used as a base language in which to incorporate the O.B.C. statements. METAPLAN was chosen because:

(A) There was an immediate need for these changes. Experience on the F-14 has shown that METAPLAN's instruction repertoire is insufficient to perform the O.B.C. functions.

---

* Including ATLAS, BAGIES, DIMATE, GAELIC, PLACE, UTEC and VTRAN. (cf. App.B)

** As the reader may already have inferred, these statements would take the form TEST ( ), DISPLAY ( ), LIMIT TEST ( ), and TREND ANALYZ( )

(B) CMS-II is less user oriented and less English-like in structure, thereby defeating the goal of this program.

The second part of this memo examines problem (2) in the definition phase with respect to the language METAPLAN. In that part, a syntax is developed which allows for the incorporation of the statements:

TEST - - -

DISPLA - - -

where the unspecified letters refer to the acronym for the equipment in question.

The interpretation phase has not been started and therefore, although we have defined the two statements, we have not incorporated the meaning to be associated with these statements, nor have we defined the recognizer which will make these statements legal utterances in METAPLAN. This is the next step in the program.

One of the main problems involved with adding statements to an existing language concerns the possibility of introducing ambiguity into that language. This problem is dealt with in great detail in the following section. In summary form, the items which were concluded from the work involved in expanding the syntax of METAPLAN to include test-type instructions, are:

(1) METAPLAN is not a simple precedence language*, and therefore, we are unable to determine whether it is unambiguous in its present form.

---

* To date, the only way a grammar can be proven unambiguous, is if it falls into the simple precedence category. Precedence grammars and matrices will be developed in the remainder of this appendix.

(2) We can redefine METAPLAN such that it is a simple precedence grammar, but that would involve generating the precedence matrix for the entire language (a tremendous undertaking).

(3) The effect of changing a small part of METAPLAN such that it becomes simple precedence in form, cannot be evaluated without further investigation.

(4) The overall efficiency of the resulting language will be an important factor in determining whether a new language is needed.

The problems posed by the above conclusions will be resolved before entering the interpretation phase.

o EXPANDING THE SYNTAX OF METAPLAN TO INCLUDE TEST-TYPE INSTRUCTIONS

The problem at hand is that of introducing new statements into the METAPLAN language. In part, this is handled by introducing a series of productions which syntactically define* the additional statements (the meaning to be associated with these statements is within the realm of semantics and will not be discussed here). Specifications of the form $\langle S \rangle ::= \langle a \rangle$ with only one alternative on the right-hand side are referred to as productions. The symbol on the left-hand side is referred to as the defined symbol associated with the production. The symbols which appear on the right-hand

---

* Reiterating, a language consists of a set of valid or admissible strings and the meanings or interpretation associated with these strings. The syntactical definition of a language refers to the rules that define the structure of valid statements and programs.

side of a production may be classified into terminal symbols, which are symbols of the language being specified, and nonterminal symbols, which are names of sets used in specifying the language. When adding new productions to a given language, one of the prime concerns is that the structure of the language is unaltered, and ambiguities* are not introduced.

Experience has shown that the following instructions should be added to the METAPLAN language:

(1) TEST ---

(2) DISPLA ---

The three (or more) unspecified symbols are the programmer-defined name which refers to the equipment being either tested or displayed. When dealing with METAPLAN, this name can be from one to six alphanumeric characters in length, the first character of which must be alphabetic. In general, these two commands will respectively test the GO/NO-GO status of a piece of equipment, and display failed equipment.

Although not dealt with from a recognition point of view within this paper, the following commands will be defined and could easily be incorporated into the METAPLAN repertoire.

(1') STATUS TEST ---

(2') DISPLA STATUS ---

(3) LIMIT TEST ---

(4) DISPLA TEST DATA ---

(5) TEST ROC ---

---

* For our purposes, ambiguities may be thought of in the same light as not knowing whether to interpret A + B/C as (A + B)/C or A + (B/C), or worse, hoping to get one, while actually gettin; the other.

The last two, which presumably are the only commands requiring explanation, deal with displaying the numeric result of a specified test, and testing based on a stored rate-of-change.

The set of productions P which define these commands are as follows*:

(1) <test> ::= <TEST><blank><symbol><blank><ceoc>

(2) <display> ::= <DISPLA><blank><symbol><blank><ceoc>

(1') <test 1> ::= <STATUS><blank><TEST><blank><symbol><blank><ceoc>

(2') <display 1> ::= <DISPLA><blank><STATUS><blank><symbol><blank><ceoc>

(3) <test 2> ::= <LIMIT><blank><TEST><blank><symbol><blank><ceoc>

(4) <display 2> ::= <DISPLA><blank><TEST><blank><DATA><blank><symbol><blank><ceoc>

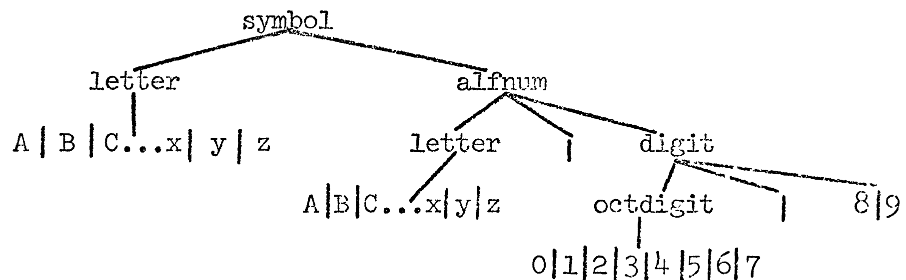
(5) <test 3> ::= <TEST><blank><ROC><blank><symbol><blank><ceoc>

The essential backbone in this definition is "symbol" which derives its interpretation from the following productions:

<symbol> ::= <letter><[alfnum]$^{5}_{0}$>

<alfnum> ::= <letter>|<digit>

This reads as: Symbol consists of a letter, followed by from zero to 5 alphanumeric characters, where an alphanumeric character is defined as either a letter or a digit. A defining-tree respresentation of these two productions looks like:



* Production specification is not according to the notation of Programmatics but rather according to the original specification of Algol 60.

An initial aim in performing our additions, was to make certain that the structure of the language is unaltered, and ambiguities are not introduced. In order to determine whether or not the language (grammar) is ambiguous, we first test to see if it is a simple precedence grammar. If the grammar is not of simple precedence type, then there are no formal rules for determining ambiguity, and we are merely guided by the fact that: A grammar G is unambiguous if it has precisely one canonical generating sequence for every string which it generates. It is said to be ambiguous if there is at least one terminal string of the language for which there is more than one canonical generating sequence. For a language that is as involved as METAPLAN, it is an impossible task to generate and perform canonical parses on every terminal string which the language generates.

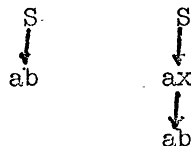
It might do well to stop for a moment and examine the idea of ambiguity a little more closely. If we define a grammar where:

$$\langle S \rangle ::= \langle a \rangle \langle b \rangle$$

$$\langle S \rangle ::= \langle a \rangle \langle x \rangle$$

$$\langle x \rangle ::= \langle b \rangle$$

and we attempt to parse the string $\langle a \rangle \langle b \rangle$, we find that the following two canonical parses produce $\langle a \rangle \langle b \rangle$:



Therefore, the grammar which we have just defined* is ambiguous. Ambiguity is to be avoided because if the terminal string has two canonical generating

---

* From this example it should be obvious that the class of sentences S that belong to this grammar consist of ab and ax. The replacement rules which exercise a set of productions are demonstrated in this example, and should be understood before continuing.

sequences, it will be associated with two different sets of semantic actions, and will therefore be semantically ambiguous.

When we talk about grammars, it is usually in terms of bounded context, context-free, or simple precedence. In a <u>context-free</u> grammar, when we apply any production, we don't care what context it is in. Therefore, we have a free choice in choosing a production in forming or reducing strings. A context-free grammar is said to be of <u>bounded context</u> (m,n) if it is always possible to avoid backtracking* by examining the m characters preceding the matched string and the n characters following the matched string**. A <u>simple precedence</u> grammar is an unambiguous grammar in which not more than one relation holds between each pair of symbols within a precedence matrix.

In order to determine whether the instructions just added to METAPLAN have an undesirable effect on the overall structure of the language, it is necessary to determine whether the expanded METAPLAN grammar is a simple precedence grammar***.

In detecting a simple precedence grammar, we must list the set of productions P which are affected by the additions. In our case, the set P is as follows:

(1) <test> ::= <TEST><blank><symbol><blank><ceoc>

(2) <display> ::= <DISPLA><blank><symbol><blank><ceoc>

---

* Backtracking occurs when we arbitrarily apply productions. If at some time we apply the wrong one, we are forced to "back-up" and try available alternatives.

** In the examples which follow, when we form a context matrix, we will be dealing with a bounded context (1,1) grammar.

*** Simultaneously, we will perform the same evaluation on the unexpanded METAPLAN.

(3) ⟨ceoc⟩ ::= ⟨eoc⟩

⟨ceoc⟩ ::= ⟨blank⟩

(4) ⟨symbol⟩ ::= ⟨letter⟩

⟨symbol⟩ ::= ⟨letter⟩⟨alfnum⟩

⟨symbol⟩ ::= ⟨letter⟩⟨alfnum⟩⟨alfnum⟩

⟨symbol⟩ ::= ⟨letter⟩⟨alfnum⟩⟨alfnum⟩ ⟨alfnum⟩

⟨symbol⟩ ::= ⟨letter⟩⟨alfnum⟩⟨alfnum⟩⟨alfnum⟩⟨alfnum⟩

⟨symbol⟩ ::= ⟨letter⟩⟨alfnum⟩⟨alfnum⟩⟨alfnum⟩⟨alfnum⟩⟨alfnum⟩

(5) ⟨alfnum⟩ ::= ⟨letter⟩*

⟨alfnum⟩ ::= ⟨digit⟩*

The next step is to produce the leftmost and rightmost symbols of a non-basic symbol U, where L (U) and R (U) are defined as:

$$L\ (U) = \{S \mid \exists\ z\ (U \xrightarrow{*} Sz)\}$$

$$R\ (U) = \{S \mid \exists\ z\ (U \xrightarrow{*} zS)\}$$

Translating this, we find that if X is a leftmost symbol of a production U, then leftmost symbols of X are also leftmost symbols of U. The following table gives L (U) and R (U) for the grammar defined above, and is probably best understood if it is read from the bottom-up.

U	L (U)	R (U)
test	TEST	ceoc, eoc, blank
display	DISPLAY	ceoc, eoc, blank
ceoc	eoc, blank	eoc, blank
symbol	letter	alfnum, letter, digit
alfnum	letter, digit	letter, digit

* Letter and digit are as we suspect, i.e., letter = A,B,C,...Y,Z and digit = 0, 1, 2,...9.

Using this information (i.e., leftmost and rightmost derivatives of a symbol U), the next step is to generate a precedence matrix in which the following rules hold true:

(i) The relation $\doteq$ holds between all adjacent symbols within a string which is directly reducible.

(ii) The relation $\lessdot$ holds between the symbol immediately preceding a reducible string, and the leftmost symbol of that string.

(iii) The relation $\gtrdot$ holds between the rightmost symbol of a reducible string, and the symbol immediately following the string.

In more explicit terms, the relation $\doteq$ holds between any two symbols appearing next to each other in the set of productions.

<letter><alfnum> ⇒ <letter> $\doteq$ <alfnum>

The relation $\lessdot$ holds in the case where the left-hand symbol is any symbol, and the right-hand symbol belongs to the set U. $\lessdot$ is then applied to the left-hand symbol and the leftmost derivatives of the element of U.

<letter><alfnum> ⇒ <letter> $\lessdot$ <letter>
<letter> $\lessdot$ <digit>

The relation $\gtrdot$ holds in the case where the right-hand symbol is any symbol, and the left-hand symbol belongs to the set U. $\gtrdot$ is then applied to the right-hand symbol and the rightmost derivatives of the elements of U.

<alfnum><alfnum> ⇒ <letter> $\gtrdot$ <alfnum>
<digit> $\gtrdot$ <alfnum>

Applying these rules to the set of productions P, we arrive at the following precedence matrix:

	test	display	ceoc	symbol	alfnum	TEST	blank	DISPLA	eoc	letter	digit
test											
display											
ceoc											
symbol							≐				
alfnum					≐		⋗			⋖	⋖
TEST							≐				
blank			≐	≐			⋖		⋖	⋖	
DISPLA							≐				
eoc											
letter					⋗ ≐		⋗			⋖	⋖
digit					⋗		⋗				

If we look at the relation which holds between letter followed by alfnum we can see that either of the following two relations is acceptable:

⟨letter⟩ ⋗ ⟨alfnum⟩

⟨letter⟩ ≐ ⟨alfnum⟩

Since this dual relationship occurs within the unexpanded portion of METAPLAN, and not between any of the relations in the expanded version, we can conclude that METAPLAN is not of simple precedence type, and unfortunately, there are no rules for determining whether this grammar is unambiguous.

In order to gain an appreciation for the problems involved with having two relations holding between adjacent symbols, we will attempt to reduce the string:

TEST APC

Translating this into the notation we've been using, we have:

⟨TEST⟩⟨blank⟩⟨A⟩⟨P⟩⟨C⟩⟨blank⟩⟨blank⟩

Examining our precedence matrix, and inserting the relations which exist between the adjacent symbols we have:

<· TEST ≐ blank <· letter <· letter <· letter ·> blank <· blank ·>

In the reduction process, we scan the string from left to right and attempt to reduce the first (and innermost) sub-string enclosed in <· ·> .
Examining this string, we find that we should first reduce <· letter ·> .
The following replacement would be performed:

<·TEST ≐ blank <·letter <·letter <·letter·> blank <· blank·>

<·TEST ≐ blank <· letter <· letter ·>/≐ alfnum ·> blank <· blank·>

At this point, we don't know whether to reduce <·letter·> to alfnum or <· letter ≐ alfnum ·> to symbol. Reducing <·blank·> to ceoc doesn't solve our problem, and the wrong decision would lead to backtracking.

In order to make the correct substitution, we must develop a set of context relations based on the precedence matrix. If we examine the productions

⟨ceoc⟩ ::= ⟨eoc⟩

⟨ceoc⟩ ::= ⟨blank⟩

in light of the precedence matrix, we find that ceoc must have a blank on its left.* Therefore, eoc or blank can only be reduced to ceoc if, after performing

---

* In actual practice, it must also be followed by a string terminal ⊣

the reduction, ceoc is preceded by a blank. We can represent this reduction as

$$\{\text{blank} \ldots \dashv\}$$

Similarly, we can develop context relations for the complete set of productions. The following relations represent the result of this work:

(1) Production #1 can be applied whenever we encounter the string

$\text{TEST} \doteq \text{blank} \doteq \text{symbol} \doteq \text{blank} \doteq \text{ceoc}$

(2) Production #2 can be applied whenever we encounter the string

$\text{DISPLA} \doteq \text{blank} \doteq \text{symbol} \doteq \text{blank} \doteq \text{ceoc}$

(3) Production #3 can be applied whenever the right side appears in the context:

$$\{\text{blank} \ldots \dashv\}$$

(4) Production #4 can be applied whenever the right side appears in the context:

$$\{\text{blank} \ldots \text{blank}\}$$

(5) Production #5 can be applied whenever the right side appears in the context:

$$\left\{\begin{matrix}\text{alfnum}\\ \text{letter}\\ \text{digit}\end{matrix}\right. \quad \cdots \quad \left.\begin{matrix}\text{alfnum}\\ \text{blank}\\ \text{letter}\\ \text{digit}\end{matrix}\right\}$$

From (4) above, we see that $\lessdot$ letter $\doteq$ alfnum $\gtrdot$ can't be reduced to symbol because this string is not in the context: blank...blank. Therefore, $\lessdot$ letter $\gtrdot$ is reduced to alfnum, and the entire reduction results in the following:

⋖ TEST ≐ blank ⋖ letter ⋖ letter ⋖ letter ⋗ blank ⋖ blank ⋗

⋖ TEST ≐ blank ⋖ letter ⋖ letter ≧ alfnum ⋗ blank ⋖ blank ⋗

⋖ TEST ≐ blank ⋖ letter ≧ alfnum ≐ alfnum ⋗ blank ⋖ blank ⋗

⋖ TEST ≐ blank ≐ symbol ≐ blank ⋖ blank ⋗

⋖ TEST ≐ blank ≐ symbol ≐ blank ≐ ceoc ⋗

test

The end result, is that we recognize the string TEST APC belongs to the syntactic class ⟨test⟩ and is therefore a legal utterance within the expanded METAPLAN language.

As an interesting aside, had the original syntax of METAPLAN been defined in a slightly different manner, the portion of the grammar which we have been examining would be simple precedence in form. The alteration is not difficult, and the set of productions P' which satisfy this condition are as follows:

(1) ⟨test⟩ ::= ⟨TEST⟩⟨blank⟩⟨symbol⟩⟨blank⟩⟨ceoc⟩

(2) ⟨display⟩ ::= ⟨DISPLA⟩⟨blank⟩⟨symbol⟩⟨blank⟩⟨ceoc⟩

(3) ⟨ceoc⟩ ::= ⟨eoc⟩

⟨ceoc⟩ ::= ⟨blank⟩

(4) ⟨symbol⟩ ::= ⟨letter⟩

⟨symbol⟩ ::= ⟨letter⟩ ⟨alfnum⟩ *

(5) ⟨alfnum⟩ ::= ⟨letter⟩

⟨alfnum⟩ ::= ⟨letter⟩⟨alfnum⟩*

⟨alfnum⟩ ::= ⟨digit⟩

⟨alfnum⟩ ::= ⟨digit⟩ ⟨alfnum⟩

---

* Although the right-hand side of the productions for ⟨symbol⟩ and ⟨alfnum⟩ are identical, the context for application is different.

Preceding as before, L (U) and R (U) are defined as:

U	L (U)	R (U)
test	TEST	ceoc, eoc, blank
display	DISPLA	ceoc, eoc, blank
ceoc	eoc, blank	eoc, blank
symbol	letter	letter, alfnum, digit
alfnum	digit, letter	digit, alfnum, letter

Based on the above information, the following precedence matrix is developed:

	test	display	ceoc	symbol	alfnum	TEST	blank	DISPLA	eoc	letter	digit
test											
display											
ceoc											
symbol							≐				
alfnum							⋗				
TEST							≐				
blank			≐	≐			⋖		⋖	⋖	
DISPLA							≐				
eoc											
letter					≐		⋗			⋖	⋖
digit					≐		⋗			⋖	⋖

Because not more than one relation holds between each pair of symbols, the grammar described by P' is an unambiguous simple precedence grammar. The context relations for the first four productions are identical to those for the previous grammar. However, a significant change is noted in applying P'(5). The complete list of context relations is as follows:

(1) <test> ::= <TEST> <blank> <symbol> <blank> <ceoc>

{⊢ ... ⊣}

(2) <display> ::= <DISPLA> <blank> <symbol> <blank> <ceoc>

{⊢ ... ⊣}

(3) <ceoc> ::= <eoc>

<ceoc> ::= <blank>

{blank ... ⊣}

(4) <symbol> ::= <letter>

<symbol> ::= <letter> <alfnum>

{blank...blank}

(5) <alfnum> ::= <letter>

<alfnum> ::= <letter> <alfnum>

<alfnum> ::= <digit>

<alfnum> ::= <digit> <alfnum>

{letter / digit ... blank}

Applying these rules to the string TEST APC yields:

<TEST><blank><letter><letter><letter><blank><blank>

⋖ TEST ≐ blank ⋖ letter ⋖ letter ⋖ letter ⋗ blank ⋖ blank ⋗

⋖ TEST ≐ blank ⋖ letter ⋖ letter ≐ alfnum ⋗ blank ⋖ blank ⋗

⋖ TEST ≐ blank ⋖ letter ≐ alfnum ⋗ blank ⋖ blank ⋗

⋖ TEST ≐ blank ≐ symbol ≐ blank ⋖ blank ⋗

⋖ TEST ≐ blank ≐ symbol ≐ blank ≐ ceoc ⋗

test

Therefore, the TEST string is recognized as belonging to the class test .

In P', we find that if we encounter ⋖ letter ≐ alfnum ⋗ we still must rely on the context relations in order to reduce the string to either <symbol> or <alfnum>. Of significant importance, however, is the fact that we now have an unambiguous grammar. This property insures the occurrence of predicted results. A similar analysis performed on the five test-type statements presented at the beginning of this report will show that their addition will not add ambiguity to the existing METAPLAN language.

The problem which necessitated production redefinition concerns the length of symbol strings. We have seen that the original specification of <symbol> led to a grammar which was not of simple precedence type. If we don't worry about the length of symbol strings, we find that the original syntax of METAPLAN is unacceptable. That is, if we attempt to reduce the string:

<letter><digit><letter><letter>

using the productions:

<symbol> ::= <letter> <alfnum>

<alfnum> ::= <letter>

<alfnum> ::= <digit>

we arrive at the following:

<letter> <digit> <letter> <letter>
<letter> <digit> <letter> <alfnum>
<letter> <digit> <symbol>
<letter> <alfnum> <symbol>
<symbol> <symbol>

Using these productions, a string which represents the class <symbol> is reduced to the class <symbol><symbol> .

If we apply the productions P' to this same string, we find the following reduction:

<letter> <digit> <letter> <letter>
<letter> <digit> <letter> <alfnum>
<letter> <digit> <alfnum>
<letter> <alfnum>
<symbol>

The problem with accepting this set of productions, is that the right-hand side of the production for <symbol> and <alfnum> are identical, i.e., in P'

<symbol> ::= <letter>
<alfnum> ::= <letter>

and

<symbol> ::= <letter> <alfnum>
<alfnum> ::= <letter> <alfnum>

This condition is known as indeterminancy, and should be avoided. Without the context relations, we would not know whether to reduce <letter><alfnum> or <letter> to <symbol> or <alfnum>. Thus, the grammar P' is of bounded context form. By a slight redefinition of P' we can eliminate this indeterminancy, and generate a context free grammar. The affected productions within the modified grammar P" are as follows:

<symbol> ::= <letter>
<symbol> ::= <letter> <alfnum>
<alfnum> ::= <symbol>
<alfnum> ::= <digit>
<alfnum> ::= <digit> <alfnum>

Using P" to reduce the same string, we have:

<letter> <digit> <letter> <letter>
<letter> <digit> <letter> <symbol>
<letter> <digit> <letter> <alfnum>
<letter> <digit> <symbol>
<letter> <digit> <alfnum>
<letter> <alfnum>
<symbol>

The advantage of P" over P' is that we have eliminated the indeterminancies. The left and right derivatives for P" are:

U	L(U)	R(U)
symbol	letter	letter, alfnum, symbol, digit
alfnum	symbol, digit, letter	letter, alfnum, symbol, digit

These lead to the following precedence matrix:

	symbol	letter	digit	alfnum	$\dashv$
symbol					$\gtrdot$
letter	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\gtrdot$
digit	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\gtrdot$
alfnum					$\gtrdot$
$\vdash$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	

Examining the precedence matrix, we find that P" is a simple precedence grammar. Using the relations found in this matrix, we have the basis for reducing the string in a right-to-left fashion. Applying these relations to the previous example, we have:

$$\vdash \lessdot \text{letter} \lessdot \text{digit} \lessdot \text{letter} \underbrace{\lessdot \text{letter} \gtrdot} \dashv$$

$$\vdash \lessdot \text{letter} \lessdot \text{digit} \lessdot \text{letter} \underbrace{\lessdot \text{symbol} \gtrdot} \dashv$$

$$\vdash \lessdot \text{letter} \lessdot \text{digit} \underbrace{\lessdot \text{letter} \doteq \text{alfnum} \gtrdot} \dashv$$

$$\vdash \lessdot \text{letter} \lessdot \text{digit} \underbrace{\lessdot \text{symbol} \gtrdot} \dashv$$

$$\vdash \lessdot \text{letter} \underbrace{\lessdot \text{digit} \doteq \text{alfnum} \gtrdot} \dashv$$

$$\vdash \lessdot \text{letter} \underbrace{\lessdot \text{symbol} \gtrdot} \dashv$$

$$\vdash \underbrace{\lessdot \text{letter} \doteq \text{alfnum} \gtrdot} \dashv$$

$$\vdash \lessdot \text{symbol} \gtrdot \dashv$$

Thus, P" is not only a simple precedence grammar, but because the context relations are not necessary in order to reduce a string, P" is also a context free grammar. Context free is a desirable quality because it allows the rules for reduction to be more lenient.

Further examination of the instruction repertoire yields the fact that maybe the conditional IF command could use modification. Currently, if we want to perform an operation only if the value of a boolean expression is true (or false), we must represent the value "true" in memory, and compare the boolean expression with the stored "true" value. The reason for this, is that the first non-arithmetic command following the IF command must be a relational* command. I believe METAPLAN should allow statements of the form:

IF <expression>|<relational expression> THEN

which is essentially defining:

<if clause> ::= IF <expression>|<relational expression> THEN

In addition to expanding the syntax of METAPLAN, an attempt will be made to produce a simple precedence grammar through syntactical redefinition. The semantic specification for these changes will then be attacked and the need for a new language will be evaluated. This evaluation will be strongly dependent on the flexibility of METAPLAN's semantics.

---

* EQ, LT, GE, NE, NL, and NG

In conclusion, METAPLAN is being thoroughly investigated, and "patched-up" to include pertinent test-type instructions. If the resulting language proves to be undesirable, or overly inefficient, other avenues will be explored, including that of language definition from scratch. The end product, in any event, will be a language intended for operation in the 1975-80 time frame which includes English-like test-type instructions.

RWP:sn

THIS PAGE INTENTIONALLY LEFT BLANK

Providing an Efficient Match Between a High Level Programming Language and Computer Instruction Repertoire

A paper given at the High Level Aerospace Computer Programming Language Conference on 29 June 1970.

By

Ralph Jenkins

SYSTEMS CONSULTANTS, INC.
1050 31st STREET, N.W.
WASHINGTON, D. C. 20007

Providing an Efficient Match Between a High Level Programming Language and Computer Instruction Repertoire

1.0 Introduction

Historically, computers have not incorporated instruction repertoires that were related to the implementation of any particular high level programming language. Typically, the logical designer selects the repertoire, and the compiler writer must create a compiler using the available repertoire. Consequently, compilers have employed considerable software logic to translate the intent of the high level programming language statements into executable computer instructions. In addition, instruction repertoires have been designed to meet a broad class of applications such as scientific or business uses, and the generality reduces efficiency in both the compiler and the application programs. Because the range of problems to be solved by the avionics computer can be established reasonably well, it should be possible to design an instruction repertoire which facilitates both the development of efficiently operating application programs and a high level language in which these programs can be easily written. An instruction repertoire having such properties would result in the following:

Efficient Compilation

Because the instruction repertoire would be designed with the tasks of translating source language to object code in mind, a more efficient compiler could be developed. Such a compiler would require fewer passes through the source language and would create object code in an efficient manner.

Efficient Object Code

By having an instruction repertoire matched to avionics problems and to the higher level language, the object code generated by the compiler would be very efficient. This would be due to a high degree of correspondence between the statements used in the high level language to describe problems and the actual machine repertoire used to implement the program.

New Compiling Techniques

Because the instruction repertoire would be both problem and compiler oriented, the possibility for the development of new techniques of compiling exists. In the past many compiler techniques arose to bridge the gap between the high level language and the machine instruction repertoire. Removal of this gap should result in a reexamination of these techniques and hopefully the development of new compiler techniques, such as micro-programs to implement new language requirements.

1.1 Past Efforts to Match the High Level Language and the Machine Instruction Repertoire

Though limited, there have been previous efforts in matching language and instruction repertoires. The most concerted previous effort to match a machine instruction repertoire and a high level language was the effort involved in the design of the Burroughs B5000. The basic approach was to implement certain of the compiler techniques (for ALGOL-60) in the hardware. The B5000 and its successors the B5500, B6500, and B7500 have the capability to perform the evaluation of parsed or "Polish notation" expressions through the use of hardware push-down/pop-up stacks. This approach greatly speeds the process of compilation but may result in inefficient generation of the object code.

Matching of instruction repertoires to the applications programs is achieved to some extent in all computers, but is usually not highly successful due to the wide range of applications faced by most machines.

1.2 Area of Interest

From the previous it can be seen that it is desirable to develop a computer instruction repertoire which leads to both efficient object code and to the easy translation of high level languages. In the case of the Advanced Avionics Digital Computer (AADC) we are in an enviable position in that the conceptualization of the computer has involved both hardware and software specialists. As a result

it has been possible to design an instruction repertoire to meet these goals.

2.0 The AADC Instruction Repertoire

Systems Consultants was contracted by the Naval Air Development Center to specify the instruction repertoire for the AADC. The following is a description of the approach used in specifying the instruction repertoire.

2.1 Technical Approach

The technical approach in specifying the instruction repertoire for the AADC was to examine and analyze selected avionic missions to determine computer requirements. From this analysis it was possible to determine the usage of specific computer operations for each mission, which in turn defines their importance. The approach consisted of the following three phases:

I. Determination of the computational and computer operation requirements for a selected group of digital avionics systems used in current aircraft.

II. Combining the individual sets of computer operations for each avionic system into one complete set which defined the present AADC requirements.

III. Specifying the AADC instruction repertoire based on the usage of specific computer operations for the present-day avionic systems, AADC hardware requirements, and projected

future additional processing requirements, i.e., increased computation for additional tasks and data handling.

These phases are illustrated in Figure 2-1 and are discussed in detail in the following paragraphs.

Determination of the Computational and Computer Operations Requirements for Selected Digital Avionic Systems

To determine the computational and computer operations requirements for existing digital avionic systems, a representative cross-section of Naval airborne missions was chosen, i.e., large tracking problems (AEW), weapons delivery (attack) and navigation and intercept (fighter). This cross-section of missions provided a representative sample of the current computational requirements for onboard digital avionic systems. The systems selected for examination were as follows:

(1) E-2B

(2) F-14/F-111B (Phoenix)

(3) A-6E

(4) P-3C

(5) A-7D/E

The procedure of determining the computer operations requirements consisted of progressively defining the structure of the requirements from a broad or mission requirement level down to specific computational and computer operations requirements necessary to accomplish the mission. Figure 2-2

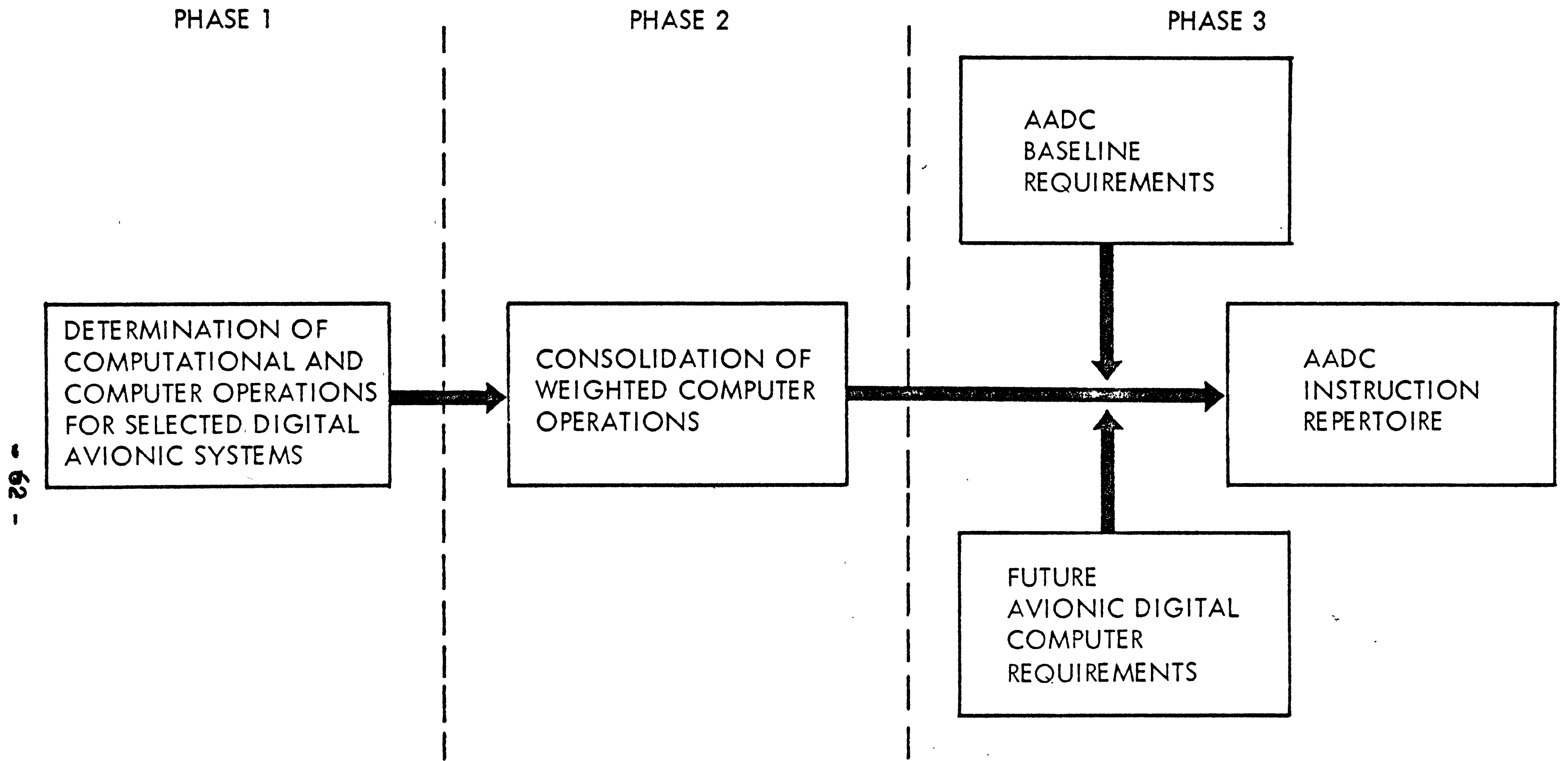


FIGURE 2-1. ANALYSIS OF THE AADC INSTRUCTION REPERTOIRE REQUIREMENTS

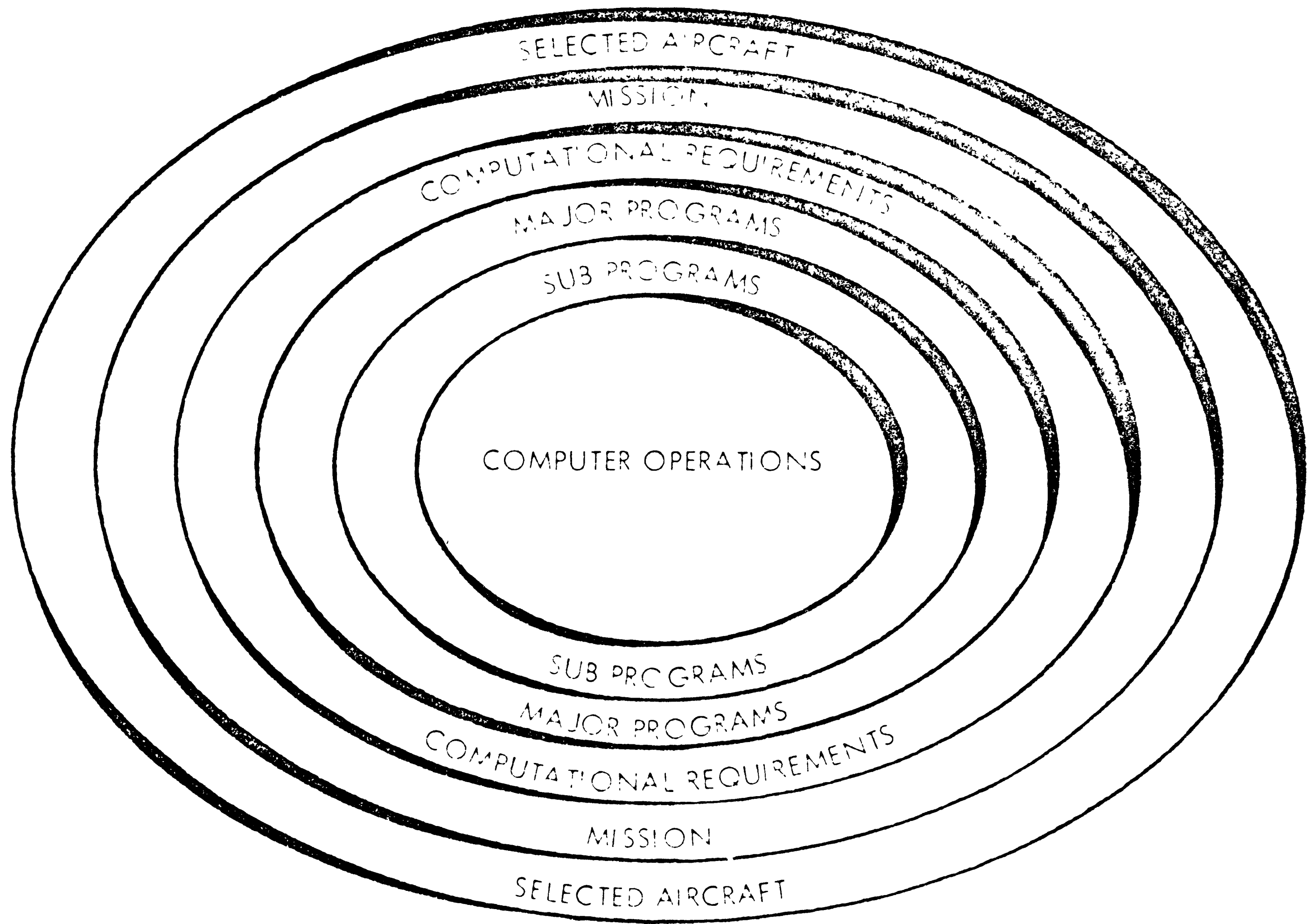


FIGURE 2-2 SEQUENCE TO DETERMINE COMPUTER OPERATIONS

illustrates the broad-to-specific structure, which begins with a selected aircraft, its misssion, computational requirements, major programs and sub-programs, and culminates with the computer operations.

The analysis for determining the computer operations was accomplished in two parts: (1) computational requirements; and (2) computer operations requirements. A description of each process is discussed in the following paragraphs.

Determination of Computational Requirements for a Selected Avionic System (Part 1)

The analysis consisted of determining the computational requirements of a selected aircraft, by examining the mission requirements to ascertain those computations that contribute significantly to the mission. Once the computational requirements were identified along with their major programs and sub-programs, they were categorized for a more exacting definition into the following three types:

- Functional

  Functional computations are those computations that are non-mathematical and contribute significantly to mission effectiveness; i.e., track control and display processing.

- Mathematical

  Mathematical computations are those computations that deal with numeric computations; i.e.,

velocity calculations.

° <u>Remaining</u>

The remaining computations are those that are not included in the functional and mathematical computations; i.e., Inflight Performance Monitor (IFPM).

These categories were chosen because avionic computer programs typically involve these three general types of computations; hence, they were very useful for establishing a common point of reference for comparing the selected aircraft computing requirements.

<u>The Determination of the Computer Operations for Selected Avionic Systems (Part 2)</u>

Once the computational requirements were determined, further analysis was conducted to determine the computer operations used by a particular aircraft system. The computer operations were taken from the flow charts of the individual avionic system programs and not the listings. By using this method the computer operations were program or problem related which was the goal of the instruction repertoire specification study. The first step in the process was to determine how often a particular major program is referenced in some time period, e.g., iteration rate per second. This was accomplished for periodic functions by assigning a value equal to the number of periodic references during the time interval, i.e., 4 times per second; for non-periodic references, an

assumed worst case value was assigned.

Once the "iteration rates" for all the major programs were established, the usage of the computer operations within the major programs were then determined. This was accomplished by thorough analysis of the major program and numerous sub-program flows which yielded the type and usage of computer operations for the particular aircraft. Therefore, the number of times a particular computer operation such as branching occurred within a major program or sub-program flow, defined a "usage weight" assigned to that particular operation. The "usage weight" determined the importance of each computer operation relative to the others. The avionic computer programs selected for evaluation may or may not represent the most efficient usage of computer operations. However, by investigating a wide range of systems, a representative set of computer operations was determined.

Once both the "iteration rate" for each major program and the "usage weight" for all the computer operations were determined, the "total weight" for each computer operation for each of the three types of computations (Functional, Mathematical and Remaining) was determined. The "total weight" was accomplished by multiplying the iteration rate times the usage weight. The resultant values defined the relative use of each computer operation during a mission. This phase of the analysis was completed for one aircraft when the computer operations for each computational task were combined to define

one set of computer operations for the particular aircraft.

Consolidation of the Present Computer Operations

This Phase (II) of the analysis was the combining of the computer operations for each aircraft into one set of computer operations for the AADC system. Since the time periods used to determine iteration rates differed for each aircraft, the iteration rates for all the selected aircraft were converted to a common time reference. These computer operations were then combined by weight, and, where duplication existed, the computer operation weights were added to form the total computer operations requirements of the AADC based on present requirements.

Specification of the AADC Instruction Repertoire

This was the final phase in the analysis and completed the task of defining an instruction repertoire for the AADC. The computer operations derived from Phase I and II dictated a particular set of instructions for present avionic digital computer systems. However, the instruction repertoire must be powerful enough to meet the requirements of new systems to be developed between now and the 1980's. Therefore, future projected computational requirements to be placed on the AADC were considered. Such areas of concern included increased inputs from phased array radar; handling of more tracks; increased need for display processing; improved navigation in the areas of terrain avoidance and

terrain following; compiler generation; and executive control over such areas as tracking, communications, I/O, etc. Also, the effects of the AADC hardware concept were considered to determine their instruction repertoire ramifications. After these areas were discussed their effects were weighted with the requirements derived in Phase I and II to define and specify the AADC instruction repertoire.

2.2 Results of the Analysis

The analysis of computer operations provided a definite indication of the type of operations required for avionic applications. It should be noted that computer operations are not computer instructions. Therefore, computer operations are indicative of high level language functions.

This led to the specification of instructions that performed the following:

Transmission - the transmission capabilities provide a means of transferring data between registers, storage, and register and storage utilizing both single and multiple addressing variations.

Branching - the branching instructions provide transfer of program control. This capability is provided by a variety of decision dependent operations that test the contents of any portion of a register or memory location, or compare the contents of a register or memory location against

another data element.

Arithmetic - the arithmetic instructions provide both a fixed and floating point capability that includes single and double precision operations for the add, subtract, multiply, divide and square root. A multiple address capability was included to provide for fast intermediate storage and retrieval of results.

Input/Output - the input/output instructions provide efficient control and monitoring of all input/output operations, i.e., buffer, external control and I/O decision.

Bit Manipulation - the bit manipulation instructions provide an efficient means for operation upon portions of words. Multiple addressing was also specified.

Indexing - the indexing capabilities provide a powerful addressing scheme defined by a direct, index, indirect and index indirect capability, along with the necessary instructions to provide for various decision controlled iterative operations.

In addition matrix and bulk processing instructions were specified for operation of the AADC matrix parallel processing, i.e., filter operations and list functions. The preliminary specification of the AADC instruction repertoire is shown in Table 2-1.

INSTRUCTION	OPERAND NOTE 1	DESCRIPTION NOTE 2
● INDEXING		
INDEX JUMP ZERO	IMPLICIT/IMMEDIATE	IF INDEX $\neq$ ZERO, DECREMENT INDEX BY ONE, TRANSFER TO OPERAND IF INDEX = ZERO, EXECUTE N.I.
INDEX JUMP VALUE	IMPLICIT/IMMEDIATE	IF INDEX $\neq$ OPERAND, INCREMENT INDEX BY ONE, EXECUTE N.I. IF INDEX = OPERAND SET INDEX TO ZERO, EXECUTE N.I.
REPEAT	IMPLICIT/IMMEDIATE	REPEATS N.I. A SPECIFIED NUMBER OF TIMES
INCREMENT	IMPLICIT	THIS CAPABILITY IS PROVIDED FOR BY THE FIXED POINT MULTI-ADDRESS ARITHMETIC INSTRUCTIONS
DECREMENT	IMPLICIT	THIS CAPABILITY IS PROVIDED FOR BY FIXED POINT MULTI-ADDRESS ARITHMETIC INSTRUCTIONS
● TRANSMISSION		
LOAD SINGLE REGISTER	IMPLICIT/IMMEDIATE	OPERAND → REGISTER
LOAD DOUBLE REGISTER	IMPLICIT/IMMEDIATE	$OPERAND_n, OPERAND_{n+1} \rightarrow REGISTER_n, REGISTER_{n+1}$
STORE SINGLE REGISTER	IMMEDIATE	REGISTER → STORAGE
STORE DOUBLE REGISTER	IMMEDIATE	$REGISTER_n, REGISTER_{n+1} \rightarrow STORAGE_n, STORAGE_{n+1}$
EXCHANGE REGISTER	IMMEDIATE	$REGISTER_m \rightarrow REGISTER_n$
N REGISTER STORAGE	IMMEDIATE	$REGISTER_n \rightarrow STORAGE_n$; $REGISTER_{n+1} \rightarrow STORAGE_{n+1}$ etc.
READ CLOCK	IMMEDIATE	CLOCK VALUE → REGISTER
SET CLOCK	IMMEDIATE	REGISTER → CLOCK
NO-OP		PERFORM NO OPERATION
*EXCHANGE MEMORY	IMPLICIT	$OPERAND_m \rightarrow STORAGE_n$
● BRANCHING		
TRANSFER UNCONDITIONAL	IMPLICIT/IMMEDIATE	TRANSFER TO OPERAND
TEST BIT N TRANSFER	IMPLICIT/IMMEDIATE	IF REGISTER BIT N = 1, TRANSFER TO OPERAND IF REGISTER BIT N = 0, EXECUTE N.I.
TRANSFER REGISTER POSITIVE	IMPLICIT/IMMEDIATE	IF $REGISTER_n \geq 0$, TRANSFER TO OPERAND IF $REGISTER_n$ NOT ≥ 0, EXECUTE N.I.
TRANSFER REGISTER NEGATIVE	IMPLICIT/IMMEDIATE	IF $REGISTER_n < 0$, TRANSFER TO OPERAND IF $REGISTER_n$ NOT < 0, EXECUTE N.I.
TRANSFER REGISTER ZERO	IMPLICIT/IMMEDIATE	IF $REGISTER_n = 0$, TRANSFER TO OPERAND IF REGISTER NOT = 0, EXECUTE N.I.
TRANSFER REGISTER NOT ZERO	IMPLICIT/IMMEDIATE	IF $REGISTER_n \neq 0$, TRANSFER TO OPERAND IF REGISTER = 0, EXECUTE N.I.
TRANSFER ON OVERFLOW	IMPLICIT/IMMEDIATE	IF OVERFLOW DESIGNATOR SET, TRANSFER TO OPERAND IF OVERFLOW DESIGNATOR NOT SET, EXECUTE N.I.
TRANSFER RETURN	IMPLICIT/IMMEDIATE	SAVE ADDRESS OF NEXT INSTRUCTION THEN TRANSFER TO OPERAND
MASTER EXECUTIVE REFERENCE	IMPLICIT/IMMEDIATE	NOTIFY MEC OF REQUEST
* TRANSFER EQUAL TO	IMPLICIT/IMMEDIATE	IF $OPERAND_1 = OPERAND_2$, TRANSFER TO $OPERAND_3$ IF $OPERAND_1 \neq OPERAND_2$, EXECUTE N.I.
*TRANSFER LESS THAN	IMPLICIT/IMMEDIATE	IF $OPERAND_1 < OPERAND_2$, TRANSFER TO $OPERAND_3$ IF $OPERAND_1$ NOT $< OPERAND_2$, EXECUTE N.I.
*TRANSFER GREATER THAN	IMPLICIT/IMMEDIATE	IF $OPERAND_1 > OPERAND_2$, TRANSFER TO $OPERAND_3$ IF $OPERAND_1$ NOT $> OPERAND_2$, EXECUTE N.I.
* TRANSFER LESS THAN OR EQUAL TO	IMPLICIT/IMMEDIATE	IF $OPERAND_1 \leq OPERAND_2$, TRANSFER TO $OPERAND_3$ IF $OPERAND_1$ NOT $\leq OPERAND_2$, EXECUTE N.I.
*TRANSFER GREATER THAN OR EQUAL TO	IMPLICIT/IMMEDIATE	IF $OPERAND_1 \geq OPERAND_2$, TRANSFER TO $OPERAND_3$ IF $OPERAND_1$ NOT $\geq OPERAND_2$, EXECUTE N.I.

TABLE 2-J INSTRUCTION REPERTOIRE FOR THE AADC
(SHEET 1 of 5)

INSTRUCTION	OPERAND NOTE 1	DESCRIPTION NOTE 2
*COMPARE BRANCH EQUAL TO	IMPLICIT/IMMEDIATE	IF $REGISTER_n = OPERAND_1$, TRANSFER TO $OPERAND_2$ IF $REGISTER_n \neq OPERAND_1$, EXECUTE N.I.
*COMPARE BRANCH LESS THAN	IMPLICIT/IMMEDIATE	IF $REGISTER_n < OPERAND_1$, TRANSFER TO $OPERAND_2$ IF $REGISTER_n$ NOT $< OPERAND_1$, EXECUTE N.I.
*COMPARE BRANCH GREATER THAN	IMPLICIT/IMMEDIATE	IF $REGISTER_n > OPERAND_1$, TRANSFER TO $OPERAND_2$ IF $REGISTER_n$ NOT $> OPERAND_1$, EXECUTE N.I.
*COMPARE BRANCH LESS THAN OR EQUAL TO	IMPLICIT/IMMEDIATE	IF $REGISTER_n \leq OPERAND_1$, TRANSFER TO $OPERAND_2$ IF $REGISTER_n$ NOT $\leq OPERAND_1$, EXECUTE N.I.
*COMPARE BRANCH GREATER THAN OR EQUAL TO	IMPLICIT/IMMEDIATE	IF $REGISTER_n \geq OPERAND_1$, TRANSFER TO $OPERAND_2$ IF $REGISTER_n$ NOT $\geq OPERAND_1$, EXECUTE N.I.
*COMPARE BRANCH LESS THAN OR EQUAL TO AND GREATER THAN OR EQUAL TO	IMPLICIT/IMMEDIATE	IF $REGISTER_n \geq OPERAND_1$ AND $\leq OPERAND_2$ TRANSFER TO $OPERAND_3$ IF $REGISTER_n < OPERAND_1$ OR $> OPERAND_2$, EXECUTE N.I.
● ARITHMETIC NOTE 3		
ADD FIXED POINT	IMPLICIT/IMMEDIATE	$REGISTER_n + OPERAND \rightarrow REGISTER_n$
SUBTRACT FIXED POINT	IMPLICIT/IMMEDIATE	$REGISTER_n - OPERAND \rightarrow REGISTER_n$
MULTIPLY FIXED POINT	IMPLICIT/IMMEDIATE	$REGISTER_n \times OPERAND \rightarrow REGISTER_n, REGISTER_{n+1}$
DIVIDE FIXED POINT	IMPLICIT/IMMEDIATE	$REGISTER_n, REGISTER_{n-1} - OPERAND \rightarrow REGISTER_n$, $REMAINDER \rightarrow REGISTER_{n+1}$
SQUARE ROOT FIXED POINT	IMMEDIATE	$\sqrt{\lvert REGISTER_n, REGISTER_{n+1} \rvert} \rightarrow REGISTER_n$
ADD FLOATING POINT	IMPLICIT	$REGISTER_n, REGISTER_{n+1} + OPERAND_n, OPERAND_{n+1} \rightarrow REGISTER_n, REGISTER_{n+1}$
SUBTRACT FLOATING POINT	IMPLICIT	$REGISTER_n, REGISTER_{n+1} - OPERAND_n, OPERAND_{n+1} \rightarrow REGISTER_n, REGISTER_{n+1}$
MULTIPLY FLOATING POINT	IMPLICIT	$REGISTER_n, REGISTER_{n+1} \times OPERAND_n, OPERAND_{n+1} \rightarrow REGISTER_n, REGISTER_{n+1}$
DIVIDE FLOATING POINT	IMPLICIT	$REGISTER_n, REGISTER_{n+1} \div OPERAND_n, OPERAND_{n+1} \rightarrow REGISTER_n, REGISTER_{n+1}$
SQUARE ROOT FLOATING POINT	IMMEDIATE	$\sqrt{\lvert REGISTER_n, REGISTER_{n+1} \rvert} \rightarrow REGISTER_n, REGISTER_{n+1}$
FIXED TO FLOAT	IMMEDIATE	$REGISTER_n, REGISTER_{n+1}$ (FX.PT.) $\rightarrow REGISTER_n, REGISTER_{n+1}$ (FL.PT.)
FLOAT TO FIXED	IMMEDIATE	$REGISTER_n, REGISTER_{n+1}$ (FL.PT.) $\rightarrow REGISTER_n, REGISTER_{n+1}$ (FX.PT.)
COMPLEMENT	IMMEDIATE	$REGISTER'_n \rightarrow REGISTER_n$
*LOAD - ADD(FX. PT.) - STORE	IMPLICIT/IMMEDIATE NOTE 4	$OPERAND_1 + OPERAND_2 \rightarrow OPERAND_3$
*LOAD - SUBTRACT(FX. PT.) - STORE	IMPLICIT/IMMEDIATE NOTE 4	$OPERAND_1 - OPERAND_2 \rightarrow OPERAND_3$
*LOAD - MULTIPLY(FX. PT.) - STORE	IMPLICIT/IMMEDIATE NOTE 4	$OPERAND_1 \times OPERAND_2 \rightarrow OPERAND_n, OPERAND_{n+1}$
*LOAD - DIVIDE(FX. PT.) - STORE	IMPLICIT/IMMEDIATE NOTE 4	$OPERAND_n, OPERAND_{n+1} \div OPERAND_m, \rightarrow OPERAND_q$
* LOAD-ADD(FL.PT.)-STORE	IMPLICIT/IMMEDIATE NOTE 4	$(OPERAND_x, OPERAND_{x+1}) + (OPERAND_y, OPERAND_{y+1}) \rightarrow (OPERAND_z, OPERAND_{z+1})$
* LOAD-SUBTRACT(FL.PT.)-STORE	IMPLICIT/IMMEDIATE NOTE 4	$(OPERAND_x, OPERAND_{x+1}) - (OPERAND_y, OPERAND_{y+1}) \rightarrow (OPERAND_z, OPERAND_{z+1})$
* LOAD-MULTIPLY(FL.PT.)-STORE	IMPLICIT/IMMEDIATE NOTE 4	$(OPERAND_x, OPERAND_{x+1}) \times (OPERAND_y, OPERAND_{y+1}) \rightarrow (OPERAND_z, OPERAND_{z+1})$
* LOAD-DIVIDE(FL.PT.)-STORE	IMPLICIT/IMMEDIATE NOTE 4	$(OPERAND_x, OPERAND_{x+1}) \div (OPERAND_y, OPERAND_{y+1}) \rightarrow (OPERAND_z, OPERAND_{z+1})$

TABLE 2-1 INSTRUCTION REPERTOIRE FOR THE AADC

(SHEET 2 OF 5)

INSTRUCTION	OPERAND	DESCRIPTION
** SINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** COSINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** TANGENT	MICRO PROGRAM	SELF DEFINING NOTE 5
** ARCSINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** ARCOSINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** ARCTANGENT	MICRO PROGRAM	SELF DEFINING NOTE 5
** HYPERBOLIC SINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** HYPERBOLIC COSINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** HYPERBOLIC TANGENT	MICRO PROGRAM	SELF DEFINING NOTE 5
** HYPERBOLIC ARCSINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** HYPERBOLIC ARCOSINE	MICRO PROGRAM	SELF DEFINING NOTE 5
** HYPERBOLIC ARCTANGENT	MICRO PROGRAM	SELF DEFINING NOTE 5
** MATRIX ADD	MICRO PROGRAM	SELF DEFINING NOTE 5
** MATRIX MULTIPLY	MICRO PROGRAM	SELF DEFINING NOTE 5
** MATRIX INVERT	MICRO PROGRAM	SELF DEFINING NOTE 5.
** VECTOR DOT PRODUCT	MICRO PROGRAM	SELF DEFINING NOTE 5
** VECTOR CROSS PRODUCT	MICRO PROGRAM	SELF DEFINING NOTE 5
** VECTOR ADDITION	MICRO PROGRAM	SELF DEFINING NOTE 5
** SQ. ROOT SUM OF SQUARES	MICRO PROGRAM	SELF DEFINING NOTE 5
** ARITHMETIC SCALE	MICRO PROGRAM	SELF DEFINING NOTE 5
** ARITHMETIC ROUND	MICRO PROGRAM	SELF DEFINING NOTE 5
** COMPLEX ADD	MICRO PROGRAM	SELF DEFINING NOTE 5
** COMPLEX MULTIPLY	MICRO PROGRAM	SELF DEFINING NOTE 5
** COMPLEX CONJUGATE	MICRO PROGRAM	SELF DEFINING NOTE 5
** HYPERBOLIC, POLAR, CARTESIAN CONVERSION	MICRO PROGRAM	SELF DEFINING NOTE 5
** BCD TO OCTAL	MICRO PROGRAM	SELF DEFINING NOTE 5
** OCTAL TO BCD	MICRO PROGRAM	SELF DEFINING NOTE 5
** UNITS CONVERSION	MICRO PROGRAM	SELF DEFINING NOTE 5
** LOGARITHM FOR ANY BASE N	MICRO PROGRAM	SELF DEFINING NOTE 5
** ANTILOGARITHM FOR ANY BASE N	MICRO PROGRAM	SELF DEFINING NOTE 5

TABLE 2-1 INSTRUCTION REPERTOIRE FOR THE AADC

(SHEET 3 OF 5)

INSTRUCTION	OPERAND NOTE 1	DESCRIPTION NOTE 2
● BIT MANIPULATION		
AND	IMPLICIT	$REGISTER_n \bullet OPERAND \rightarrow REGISTER_n$
OR	IMPLICIT	$REGISTER_n + OPERAND \rightarrow REGISTER_n$
EXCLUSIVE OR	IMPLICIT	$REGISTER_n \oplus OPERAND \rightarrow REGISTER_n$
*LOAD AND	IMPLICIT/IMMEDIATE	$OPERAND_n \bullet OPERAND_m \rightarrow REGISTER_n$
*STORE AND	IMPLICIT/IMMEDIATE	$OPERAND_n \bullet REGISTER_n \rightarrow OPERAND_m$
*LOAD OR	IMPLICIT/IMMEDIATE	SET $OPERAND_n$ FOR $OPERAND_m$ EQUAL TO ONE $\rightarrow REGISTER_n$
*STORE OR	IMPLICIT/IMMEDIATE	SET $OPERAND_n$ FOR $REGISTER_n$ EQUAL TO ONE $\rightarrow OPERAND_m$
*LOAD EXCLUSIVE OR	IMPLICIT/IMMEDIATE	COMPLEMENT $OPERAND_n$ FOR $OPERAND_m$ EQUAL TO ONE $\rightarrow REGISTER_n$
*STORE EXCLUSIVE OR	IMPLICIT/IMMEDIATE	COMPLEMENT $OPERAND_n$ FOR $REGISTER_n$ EQUAL TO ONE $\rightarrow OPERAND_m$
REVERSE BIT ORDER	IMMEDIATE	MIRROR IMAGE OF REGISTER $\rightarrow$ REGISTER
SHIFT LEFT CIRCULAR	IMPLICIT/IMMEDIATE	SHIFT $REGISTER_n$ LEFT, END AROUND
SHIFT LEFT DOUBLE CIRCULAR	IMPLICIT/IMMEDIATE	SHIFT $REGISTER_n$, $REGISTER_{n+1}$ LEFT, END AROUND
SHIFT RIGHT ZERO FILL	IMPLICIT/IMMEDIATE	SHIFT $REGISTER_n$ RIGHT, FILL ZEROS TO THE LEFT, TRUNCATE
SHIFT RIGHT DOUBLE ZERO FILL	IMPLICIT/IMMEDIATE	SHIFT $REGISTER_n$, $REGISTER_{n+1}$ RIGHT, FILL ZEROS TO THE LEFT, TRUNCATE
SHIFT RIGHT SIGN FILL	IMPLICIT/IMMEDIATE	SHIFT $REGISTER_n$ RIGHT, FILL SIGN TO THE LEFT, TRUNCATE
SHIFT RIGHT DOUBLE SIGN FILL	IMPLICIT/IMMEDIATE	SHIFT $REGISTER_n$, $REGISTER_{n+1}$ RIGHT, FILL SIGN TO THE LEFT, TRUNCATE
SET BIT N MEMORY	IMPLICIT/IMMEDIATE	SET BIT N OF THE OPERAND WHERE N IS ANY BIT OF A WORD
CLEAR BIT N MEMORY	IMPLICIT/IMMEDIATE	CLEAR BIT N OF THE OPERAND WHERE N IS ANY BIT OF A WORD
SET BIT N REGISTER	IMMEDIATE	SET BIT N OF THE REGISTER WHERE N IS ANY BIT OF THE REGISTER
CLEAR BIT N REGISTER	IMMEDIATE	CLEAR BIT N OF THE REGISTER WHERE N IS ANY BIT OF THE REGISTER
● INPUT/OUTPUT		
INITIATE INPUT	IMPLICIT/IMMEDIATE	INITIATES INPUT OF DATA OVER AN I/O CHANNEL INTO THE SPECIFIED AREA OF MEMORY
INITIATE OUTPUT	IMPLICIT/IMMEDIATE	INITIATES OUTPUT OF DATA OVER AN I/O CHANNEL FROM A SPECIFIED AREA OF MEMORY
TERMINATE INPUT	IMPLICIT/IMMEDIATE	TERMINATES AN INPUT PROCESS ON A SPECIFIED CHANNEL
TERMINATE OUTPUT	IMPLICIT/IMMEDIATE	TERMINATES AN OUTPUT PROCESS ON A SPECIFIED CHANNEL
TEST I/O	IMPLICIT/IMMEDIATE	IF THE SPECIFIED I/O CHANNEL IS ACTIVE, TRANSFER TO OPERAND IF THE SPECIFIED I/O CHANNEL IS NOT ACTIVE, EXECUTE N.I.
ENABLE INTERRUPT	IMPLICIT/IMMEDIATE	SELECTIVELY ALLOWS INTERRUPTS TO OCCUR OVER SPECIFIED CHANNELS
DISABLE INTERRUPT	IMPLICIT/IMMEDIATE	SELECTIVELY PREVENTS INTERRUPTS FROM OCCURRING OVER THE SPECIFIED CHANNELS
EXTERNAL EQUIPMENT COMMAND	IMPLICIT/IMMEDIATE	SPECIFIES COMMANDS FOR CONTROL OF EXTERNAL EQUIPMENT

TABLE 2-1 INSTRUCTION REPERTOIRE FOR THE AADC

(SHEET 4 OF 5)

INSTRUCTION	OPERAND	DESCRIPTION
● MATRIX AND BULK PROCESSING		
INITIATE TRANSFORM	MICRO PROGRAM	NOTE 5
INITIATE INVERSE TRANSFORM	MICRO PROGRAM	NOTE 5
INITIATE FILTER	MICRO PROGRAM	NOTE 5
INTERROGATE FILTER	MICRO PROGRAM	NOTE 5
TERMINATE FILTER	MICRO PROGRAM	NOTE 5
MATRIX MANIPULATION	MICRO PROGRAM	NOTE 5
SEARCH LIST	MICRO PROGRAM	NOTE 5
SORT LIST	MICRO PROGRAM	SELF DEFINING NOTE 5
MODIFY LIST	MICRO PROGRAM	SELF DEFINING NOTE 5
COMPUTE LIST	MICRO PROGRAM	SELF DEFINING NOTE 5
ENCODE	MICRO PROGRAM	SELF DEFINING NOTE 5
DECODE	MICRO PROGRAM	SELF DEFINING NOTE 5
MULTIPLEX	MICRO PROGRAM	SELF DEFINING NOTE 5

NOTE 1

IMMEDIATE OPERAND IS A SELF DEFINING VARIABLE

IMPLICIT OPERANDS DEFINE THE LOCATION OF THE OPERAND AND WILL INCLUDE AN INDEXING CAPABILITY

NOTE 2

N.I. = NEXT INSTRUCTION

NOTE 3

THE FIXED POINT OPERATIONS ILLUSTRATE THE SINGLE PRECISION FORM, AND THE FLOATING POINT OPERATIONS ILLUSTRATE THE DOUBLE PRECISION FORM; HOWEVER ALL ARITHMETIC OPERATIONS INCLUDE BOTH A SINGLE AND DOUBLE PRECISION CAPABILITY

NOTE 4

OPERAND THREE IS EITHER IMPLICIT OR IMMEDIATE AND OPERAND ONE AND TWO ARE IMPLICIT

NOTE 5

THIS INSTRUCTION WILL SPECIFY THE LOCATION OF THE DESIRED INPUTS AND OUTPUTS

* MULTIPLE ADDRESS INSTRUCTION

** SUPPLEMENTARY INSTRUCTIONS

TABLE 2-1 INSTRUCTION REPERTOIRE FOR THE AADC

(SHEET 5 OF 5)

3.0 Relation of the AADC Instruction Repertoire to a High Level Programming Language

Applied to Compiler Operation

A compiler used for generation of object programs for avionic applications must be capable of processing complex arithmetic operations (tracking equations, matrix manipulation, etc.) as well as numerous control functions (display, communications operator interface, etc). These functions stated in the high level language must be translated to a series of instructions executed by the computer. To do this, the compiler usually processes a source high level language statement by a series of "passes". These passes "crack" (sometimes called parsing) the statement into an internal form that can be interpreted by further passes in order to generate the object program instructions that will accomplish the original source statement.

An instruction repertoire that is similar to the high level language can reduce compiler passes. For example, Figure 3-1 illustrates the pseudo compiler generation for three different computer instruction repertoires (Burroughs B5500, a conventional class machine and the AADC). The source expression has been "parsed" in a format that is typical and favorable for instruction generation for either computer repertoire. The Burroughs B5500 will "unstring" the statement using an expression evaluation sequence which

- ARITHMETIC EXPRESSION
  H = A+(B-C) (D/E+F)
- CMS-2 STATEMENT
  SET H EQ A+(B+C)*(D/E+F)
- PARSED POLISH NOTATION
  HABC-DE/F+*+=

BURROUGHS B5500	CONVENTIONAL (OPTIMIZED)	AADC
ENTER ADDRESS H	ENTER B	LOAD-SUB-STORE B-C $\rightarrow T_1$
ENTER A	SUBTRACT C	LOAD-DIV-STORE D/E $\rightarrow T_2$
ENTER B	STORE B-C	LOAD-ADD-STORE T_2+F $\rightarrow T_2$
ENTER C	ENTER D	LOAD-MUL-STORE $T_1 \times T_2 \rightarrow T_1$
SUBTRACT	DIVIDE E	LOAD-ADD-STORE T_1+A $\rightarrow$ H
ENTER D	ADD F	
ENTER E	MULTIPLY (B-C)	
DIVIDE	ADD A	
ENTER F	STORE IN H	
ADD		
MULTIPLY		
ADD		
EXCHANGE		
STORE IN H		

FIGURE 3-1. INSTRUCTION REQUIREMENTS COMPARISON

utilizes a "tripling" technique (two operands and a one operator). The sequence yields a near one-for-one generation (one operand = one instruction and one operation = one instruction). The sequence is simple with respect to compiler logic because the instruction generation logic only needs to provide operand moves into the logic portion of the computer and execute each operation in the proper order. However, it is inefficient in object instruction generation and processing time and requires additional hardware logic in the object computer.

The conventional class machine requires fewer instructions but will require a more complex compiler logic in order to keep track of intermediate operands and provide optimization of code generation. Because the AADC instruction repertoire has been optimized to perform multiple operations, the AADC design requires even fewer instructions while simplifying compiler logic.

Another area of matching instruction repertoire to a language is the ability to provide various programmable micro-instructions. This enables the computer to be adaptable in providing high usage operations that cannot normally be accomplished with one instruction, i.e., sine, cosine, simple matrix manipulation, etc. The AADC instruction repertoire provides this capability.

Applied to Object Program Operations

The matching of instruction repertoire to high-level statements has been demonstrated to produce more efficient instruction generation. Another example of efficiency is shown in Figure 3-2. In this case the conventional instruction sequence will require three instructions to perform the operations. For the AADC, the entire process has been reduced to one instruction which, in this case, is an exact match to the high-level statement.

- CMS-2 STATEMENT

  IF A EQ B THEN GOTO C

CONVENTIONAL	AADC
ENTER A	TRANSFER EQ TO A=B BRANCH C
SUBTRACT B	
BRANCH C	

FIGURE 3-2 DECISION EXAMPLE

4.0 Summary

In the earlier part of the presentation it was indicated that the selection of a computer's instruction repertoire has many implications for computer software. The instruction repertoire has an impact upon both the executable programs and upon the compiler which translates a higher level language into such programs. The approach taken was to analyze the computing needs of a wide range of avionics computer applications to specify an instruction repertoire which was problem oriented. As a result, the AADC instructions are those underlying avionic problem solutions. At the same time they facilitate compiler development, as high level language operators and the instruction repertoire are well matched. This match results in more efficient object code, simpler, hence more effective compilers, and achieves the flexibility required by the AADC design concepts.

CLASP - ITS ROLE IN AADC SOFTWARE DEVELOPMENT

Edward H. Bersoff
Logicon, Inc.
Falls Church, Va. 22044

Background

"By properly coordinating the concurrent development of AADC hardware and software, it is believed that both system elements can be combined to create an 'integrated package'." These words, written in the NAVAIR document "AADC Software Considerations" dated December 1, 1969, serve to define the problem facing NAVAIR today. The "integrated package" results from some facility which will automatically translate operational requirements into a hardware configuration and a set of program modules together with an executive. For purposes of this paper this automated facility will be called a Synthesizer.

It is clear that the Synthesizer must be prepared to function some time before the AADC becomes operational. To be sure that the goal is reached on time it is necessary to formulate a software plan which takes into account the orderly evolution of the Synthesizer system. Figure 1 offers one possible approach to a Synthesizer design. Each block in the figure represents a software development task which must be performed in order for the system to operate. What is immediately obvious is that the Job Model either directly or indirectly provides an input to every critical portion of the Synthesizer and represents a first step in its development. It therefore becomes clear that the construction of the Job Model should be the first software task considered. This is currently being done by the Naval Air Development Center.

Figure 2 gives a recommended overall software plan. Its pyramidal shape accentuates the critical nature of early studies which have a profound effect on future ones. Ancillary studies appear outside the pyramid. These are either hardware studies which have an impact on the software or software studies already underway. For example, the instruction set already developed might also have been defined after a functional simulator had been developed and concurrently with benchmark program specification. The Job Model also affects the nature of the MEC so this study too could have been done at a later time. However, what is significant, is that software problems are being considered, and this consideration is taking place before hardware design is fixed. In any event, the overall plan should incorporate the results of these ancillary studies into the Synthesizer design.

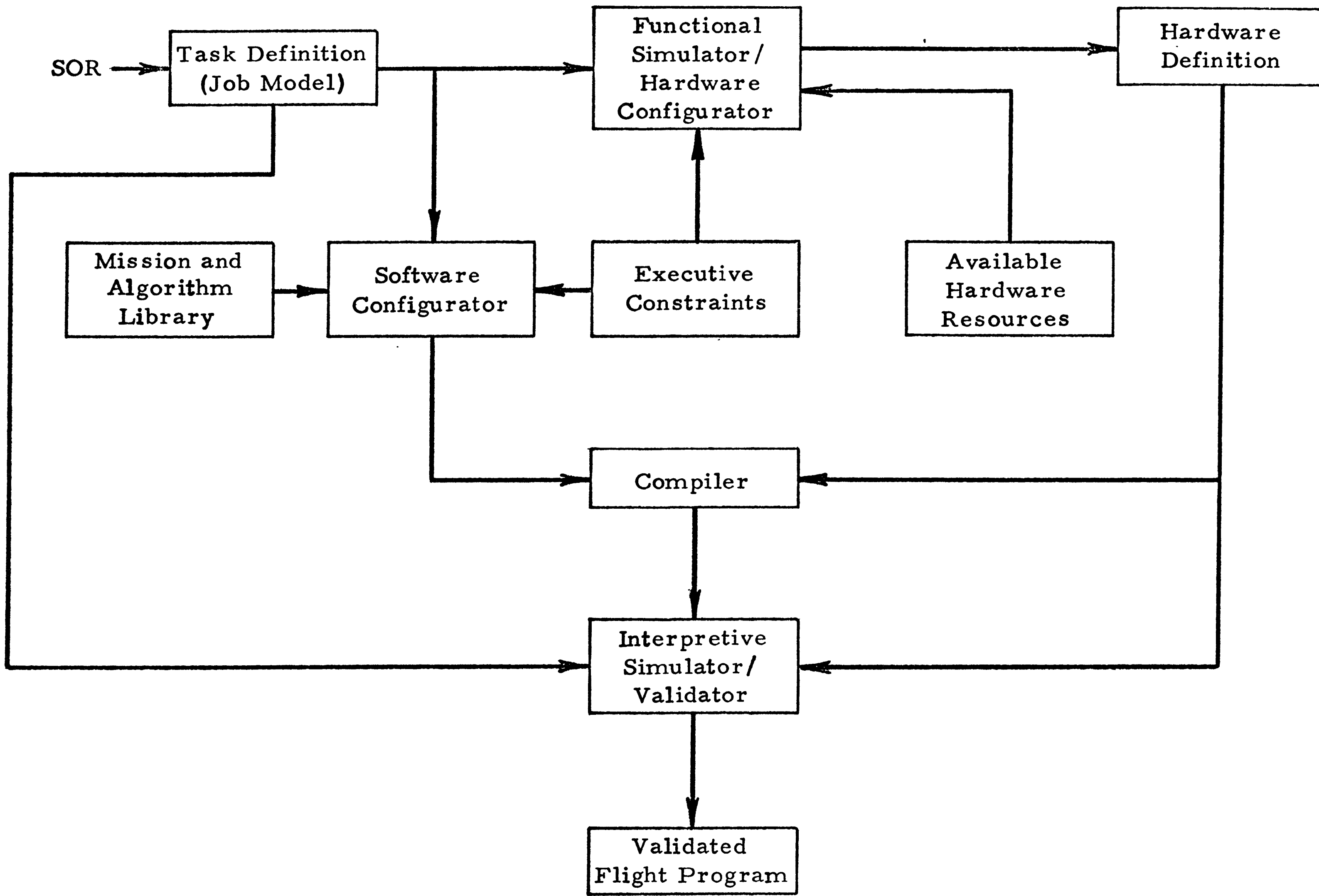


Figure 1. Synthesizer Functional Block Diagram

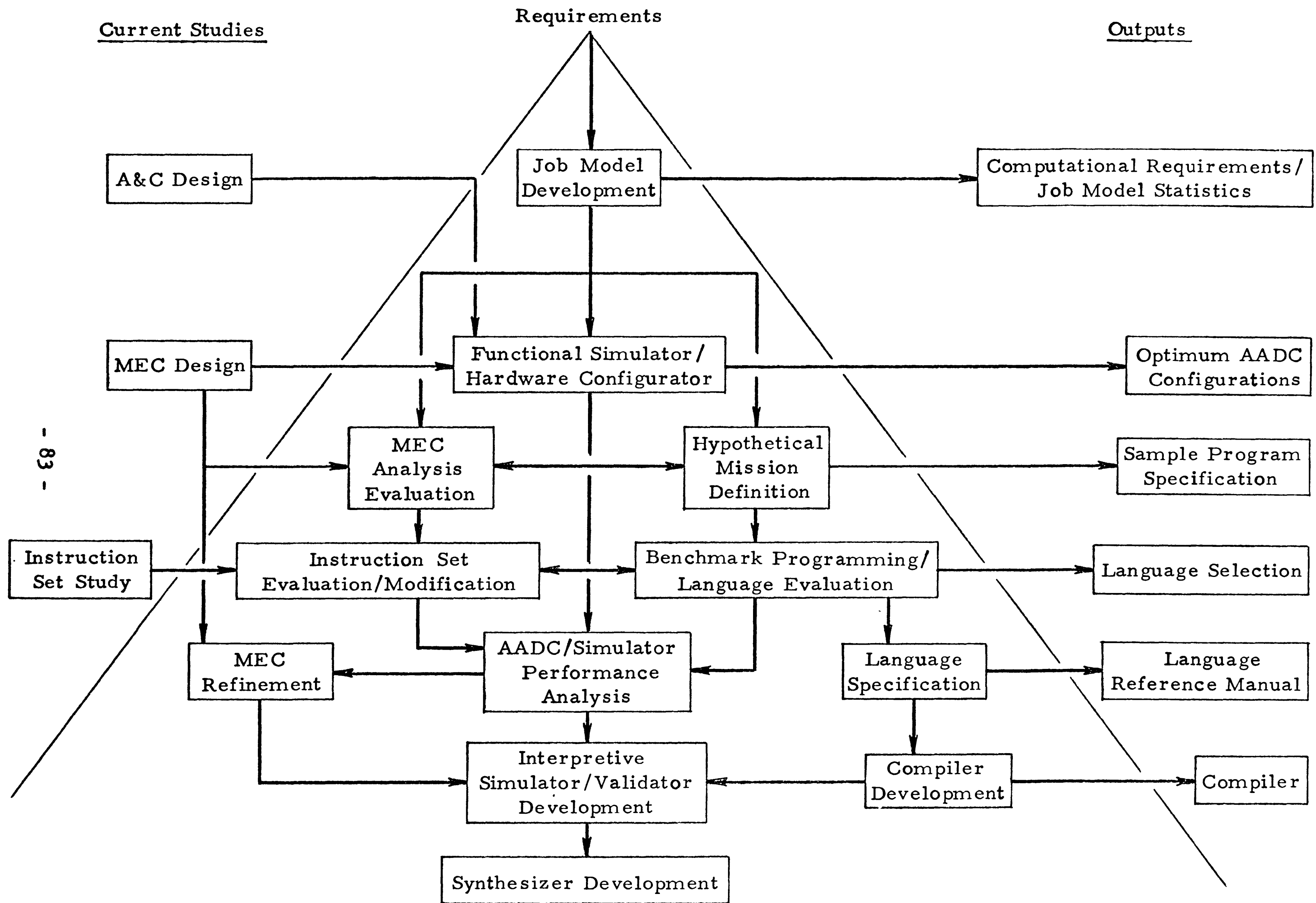


Figure 2. Software Development Plan

A critical decision point in the software development cycle with both technical and non-technical implications is the selection of a programming language. Until now the Navy has not played a significant role in the definition of an airborne programming language. The Navy candidate language, CMS-2, is ostensibly a fully compatible superset of CS-1. Recognizing the deficiencies of the CS-1/MS-1 system the Navy in 1967 tasked CSC, the designers of CS-1/MS-1 to create a new, more useful language. This language was called CMS-2. The CMS-2 compiler-monitor system presently operates on CP-642 series computers, and generates code for either these computers or for the CP-879. A version which will operate on, and generate code for, the AN/UYK-7 is under development. The system is very powerful, but relatively new and unexercised. It is intended to provide a tool for the development of all types of programs. From a NAVAIR point of view, however, CMS-2 has yet to be tested as to its suitability for airborne missions. This test can be performed only when it is known exactly what tasks are being carried out in the AADC. This information should be made available during the course of a Job Model development.

In summary the immediate questions that NAVAIR must answer are:

- What tasks must be performed by future Navy aircraft?
- What are the computational requirements of these tasks?
- What is a typical "worst case" operational program specification?
- Can CMS-2 satisfy programming requirements?
- What changes, if any, are required to CMS-2?

This paper is concerned with an approach intended to answer these questions and a description of CLASP, a programming language which should be considered along with CMS-2 for possible NAVAIR use.

Which Programming Language?

There is little doubt at the present time that some high order language will be used to program the AADC. From among the many candidates, two appear a priori most suitable for that effort, CMS-2 and SPL/CLASP. It is possible that one of the languages may prove to be far superior to the other. If this is so it should be selected as a base for future AADC programming. It is more likely, however, that neither will be significantly better than the other. The reason for this is that while CMS-2 is Navy oriented, it was not designed for avionics applications. The opposite may be said for SPL/CLASP. Thus in the event that no one language has a clear technical superiority it would

seem reasonable to adopt some form of CMS-2 for AADC use. This recommendation is made because of the significant amount of resources the Navy has already invested in the language. What is required now is a rational study of AADC needs as they relate to the capabilities of the two candidate languages. This may be done through a four phase plan as follows:

- Job Model Design
- Problem Specification
- Benchmark Programming/Language Evaluation
- Language Selection/Specification

The Job Model represents the results of a detailed analysis of the computational requirements of all possible AADC configurations. These requirements are manifested as tasks based on aircraft missions. Figure 3 defines some of these tasks. Although the figure shows that navigation is performed by all aircraft computers, the algorithm used for this task will likely be unique to a particular mission. For example, a fighter aircraft need not know his position as accurately as a reconnaissance aircraft, so it makes sense to use a less sophisticated algorithm for fighter navigation and concentrate the computational effort on a more important task like missile delivery. What this points out is that algorithms as well as tasks affect the Job Model Design. Much data can be gathered through a judicious study of the Job Model. These data should be used to answer questions such as:

- Which tasks, if any, will require paging? (This will depend on the algorithms used.)
- Can these large tasks be paged?
- How many instructions on the average are executed per task? What is the variance of this statistic?
- What is the extent of cross-referencing between tasks? How does this vary with time?
- What portions of which tasks may be executed in parallel?
- What tasks may or should be executed simultaneously?

The answers to these questions can be used in a variety of ways. First they will provide an essential input for any future Functional Simulator/Hardware Configurator development. The purpose of this simulator/configurator will be to iterate through various AADC configurations to determine if

Computer Task \ Aircraft Type	CARRIER BASED									LAND BASED		
	FIXED WING							ROTARY				
	Fighter	Attack	AEW	ASW	ECM	ELINT	RECON	ASW	SEARCH/ RESCUE	PATROL/ ASW	RECON/ ELINT	ECM
Navigation	X	X	X	X	X	X	X	X	X	X	X	X
Automatic Flight Control	X	X	X	X	X	X	X	X	X	X	X	X
On-Board Checkout	X	X	X	X	X	X	X	X	X	X	X	X
Environmental Control	X	X	X	X	X	X	X	X	X	X	X	X
Radar Signal Processing	X	X	X	X	X	X	X	X	X	X	X	X
Executive	X	X	X	X	X	X	X	X	X	X	X	X
Target Signal Recognition	X	X					X				X	
Target Tracking	X			X			X	X		X	X	
Sensor Monitor & Control			X	X	X	X	X	X		X	X	X
Sensor Correlation				X			X	X		X	X	
Data Compression			X	X	X	X	X	X		X	X	X
Countermeasure Monitor & Control			X	X	X	X	X	X		X	X	X
Display Signal Format & Control			X	X	X	X	X	X		X	X	X
Communication Format & Control			X	X	X	X	X	X		X	X	X

Figure 3. Computer Tasks by Aircraft

Computer Task \ Aircraft Type	CARRIER BASED									LAND BASED		
	FIXED WING							ROTARY				
	Fighter	Attack	AEW	ASW	ECM	ELINT	RECON	ASW	SEARCH/ RESCUE	PATROL/ ASW	RECON/ ELINT	ECM
Acoustic Signal Processing				X				X		X		
Bomb Delivery		X		X			X	X		X	X	
Missile Delivery	X	X					X				X	
Torpedo Delivery				X				X		X		

Figure 3. (cont'd.)

they can adequately perform the computations described by the input parameters. Allowing for some margin of safety the simulator/configurator should produce as output an appropriate AADC architecture. This use of the Job Model statistics produced by NADC will be studied by NRL. However they also provide a vital input to Problem Specification.

In order to evaluate programming languages one must program with them. Too often language studies are performed in a vacuum without regard to the application. As a result the conclusions drawn from the study are necessairly vacuous. It is therefore imperative that a representative AADC mission be specified based on the knowledge gained through the Job Model Design. This should be done in three steps as follows:

- Task selection
- Algorithm selection
- Detailed programming specification

During task selection, it should be decided which tasks might typically be performed by a multiprocessor version of the AADC. This configuration should be used as a baseline since it represents the most severe test of a programming language. The full AADC system, including the Matrix-Parallel Processor, will ultimately simplify the problem since many of the more arduous tasks will be done by the MPP. The next step will be to select appropriate computational algorithms for each task selected. In order to tax the system, the most sophisticated algorithms available for each task should be used. The last step is the production of a detailed programming specification. It should include flowcharts, accuracy and timing requirements, and data descriptions that will be required for the next phase, Benchmark Programming/ Language Evaluation.

With respect to programming language selection, the Navy now finds itself in a rahter unique position. Both the Air Force and NASA have developed essentially compatible languages (SPL and CLASP) without the benefit of knowing the nature of their target computers. NAVAIR has a computer design and an existing language (CMS-2) not specifically tailored to airborne applications. That there is some doubt as to the present suitability of CMS-2 is evidenced by statements made by Computer Sciences Corporation in a report concerning the language.* They say that CMS-2 was developed purely as a stopgap measure and to satisfy imposed CS-1 compatibility requirements.

---

* Recommendations for an Improved Compilation System, CSC Formal Report FR 3099, 30 November 1967.

".... However, a newly conceived, tailored language has several advantages.... Because of the time constraints, the current operational systems compatibility problem, and the present loading on the center [FCPCP], a new system, new language implementation effort is not recommended to solve the immediate problems [1967]. It is recommended on a longer term basis.... The implementation plan formulated and recommended for CMS-3 [the new language] is responsive to the long range objectives of FCPCP and the Chief of Naval Operations (CNO). The acquisition of this system will provide the Navy with a state-of-the-art compilation system that can be kept in the forefront of the technology. The acquisition of CMS-3 is not restricted to fulfilling the requirement of CS-1 compatibility. All new Navy systems that enter the system acquisition phase following the implementation of CMS-3 system will use it as the program production tool. Existing system compiled in the CMS-2 system will only be translated to CMS-3 System when the cost effectiveness formula results are favorable."

It may be, however, that CSC was wrong and the CMS-2 will prove adequate for AADC needs. It is obviously imperative that the Navy find out.

Using the detailed flight program specification programming should begin in both CMS-2 and SPL/CLASP; one programmer coding in CMS-2 and one in SPL/CLASP. As inadequacies in the candidate languages are discovered, they should be fully documented. If a new language feature might prove useful it should be fully defined. Upon completion of the benchmark programming, the participating programmers could report on their results. Data may be collected from the programmers through questionnaires containing such questions as those given in Figure 4. In addition users of both CMS-2 and SPL/CLASP should be interviewed in order to obtain their comments via the same questionnaire. This will allow NAVAIR to capitalize on the knowledge of those with in-depth experience.

The data gathered during the programming/evaluation task should then be used to establish a capabilities-deficiencies matrix. This matrix will clearly show in what ways CMS-2 and SPL/CLASP were suitable for the given problem and in what ways they each proved inadequate. Using the matrix and its supporting documentation a meaningful comparison may be made between the two languages and thereby a selection based on facts rather than speculation.

Whichever language is chosen the next step should be the preparation of language specification. This specification will strip away any unnecessary features of the base language and will add those capabilities not

PRELIMINARY LANGUAGE EVALUATION QUESTIONNAIRE

1. What elements or features of this language seemed to be especially suited for this problem?

2. What elements or features of this language seem to be weak and/or difficult to use for this problem?

3. Discuss the generality of this language in its ability to apply directly and effectively to many problems in this application area.

4. Discuss the simplicity of this language in its ease of learning and using.

5. Discuss the consistency of this language in terms of the constant application of the same rules in the same way.

6. Discuss the efficiency of this language in terms of your productivity of applying the rules of this language to writing the statements.

7. Discuss any specific and strict rules which you have found tend to make the user error prone in writing statements.

8. Discuss your feeling about using this language for nearly all problems which might occur in this particular application area.

9. Which areas of this language seem to be easiest to learn and apply?

10. Which areas seem to be the most difficult?

11. Discuss any features of this language which might tend to make parts of it machine dependent, such as I/O control, word length, precision, bits or characters per word, etc.

12. Discuss the possibility of using a natural subset of this language for your application.

Figure 4.

included in the base. After the preparation of a preliminary specification, portions of the problem may be reprogrammed in the modified base language. This should be done by the programmer who used the other candidate language in the previous task. The purpose of this switch is to obtain a fresh point of view and to assess the readability and understandability of the documentation produced. The goal of this reprogramming effort will be to "fine tune" the language specification. At the conclusion of this task a programmer's reference manual should be produced which will fully document modifications to the base language. An outline of this study approach is given in Figure 5.

What is CLASP?

Until very recently, computer programs for aeronautic and space applications had always been written in assembly language. Programmers in other areas had been continually rewarded with significant advances in their programming languages to simplify the coding task, but the aerospace programmer had been neglected to the point that his burden was too great to bear. It had been argued that there was no solution to the problem, that all aerospace programs must be written in assembly language. Those who have taken this stand maintained that there are too many different aerospace computers to justify the cost of developing a compiler for each. In addition, they have claimed that no compiler could produce code that was efficient enough to solve real-time guidance and navigation problems, and the advent of strapdown inertial sensors, with their increased computational requirements, tended to lend credence to this reasoning. Finally, mere mention of "the fixed-point problem" was always sufficient to silence anyone who continued to argue in favor of higher level languages. Almost all aerospace computers were fixed point, that is, a binary point was assumed between the first and second bits of a data word, and all numerical quantities represented as some fraction times a suitable power of 2. Seemingly, one had only to observe an aerospace programmer at work, constantly shifting bits and keeping track of binary points, to know that a compiler could never do the job.

There is no question that these arguments were valid. But it now appears that they can be challenged, as indeed they have been. The first important step taken to progress beyond assembly language coding was System Development Corporation's SPL (Space Programming Language), developed for the Air Force early in 1967. However, many observers, NASA among them, thought that a giant step had been taken when something smaller had been called for: SPL as originally defined was very large and included capabilities of questionable importance. NASA felt that what was needed was a concise, readily implementable language -- one that was oriented to the fixed-point computers of the near future, a language that, once having been proven satisfactory, could later be extended to accommodate more sophisticated computers and auxiliary support software functions. Accordingly, in

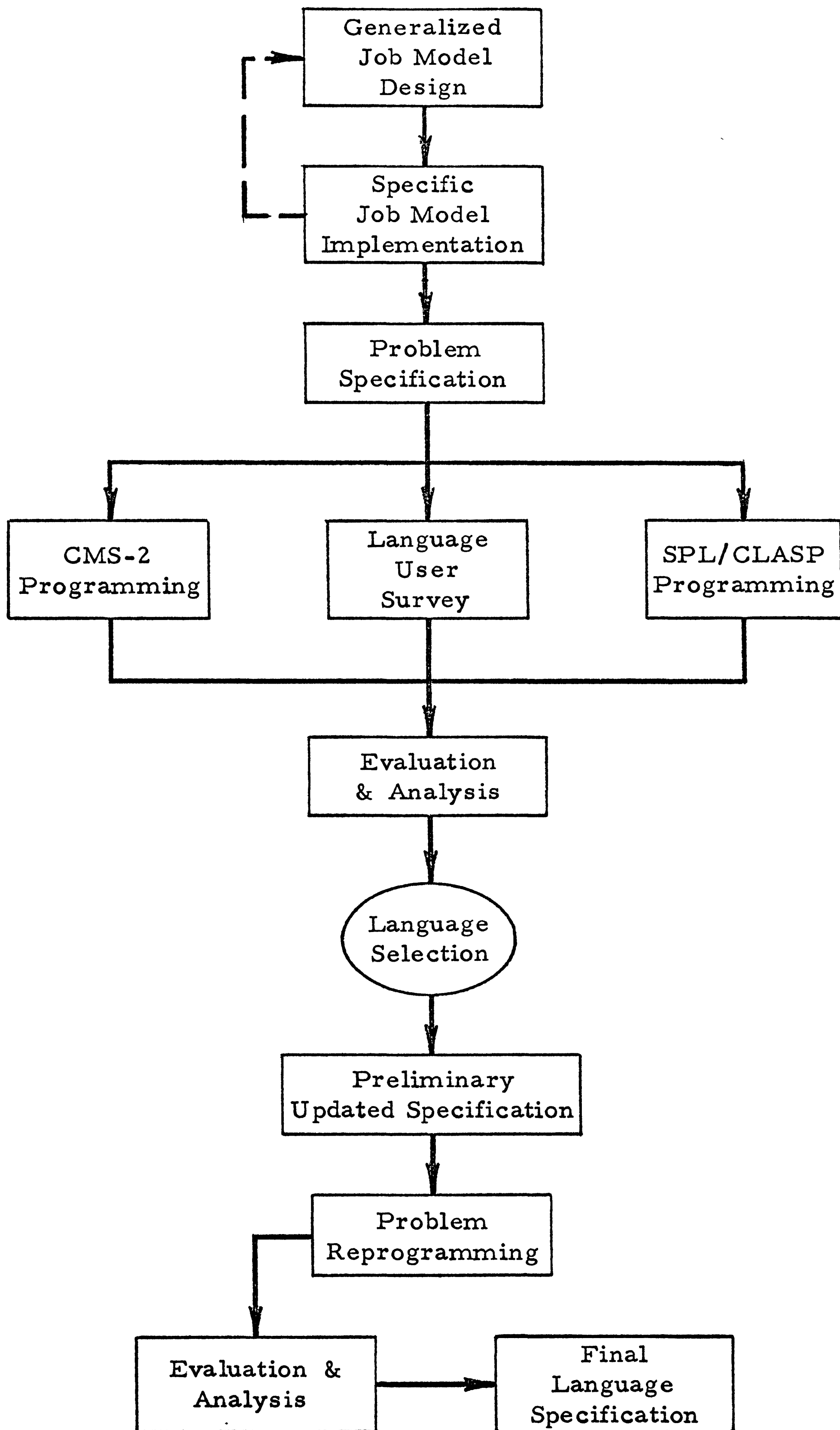


Figure 5. Proposed Work Flow

mid-1968 NASA and Logicon undertook a study with the goal of developing such a language.

Having been intended to be of immediate, practical benefit, the study was oriented toward computers and applications of the present and near future. While not considered a special-purpose machine, the typical present aerospace computer has a relatively small memory and a fairly limited instruction set, with no hardware floating-point operations and minimal instructions for logical decision functions and internal index register manipulation. With regard to the problems to be solved, these consist principally of arithmetic calculations and logical decisions, with significantly fewer text manipulation and table look-up operations. Guidance and navigation functions are performed by repeating the appropriate calculations at a relatively slow major cycle frequency; other functions such as input/output processing and control are performed at much higher minor cycle frequencies. The required response times between input and output, which are functions of the overall vehicle system design, are generally very short -- on the order of a few seconds at most.

The study plan itself consisted of using two candidate languages to code portions of a representative problem designed to be solved on a current computer; analyzing the resultant programs to select a base language for further definition; using the language thus defined to reprogram the same problem; and analyzing the second set of programs to enable further language definition and refinement. Thus the first step to be taken was to choose the candidate languages. SPL, which had been defined for the application area, was selected, as was PL/I, the latter because it included many of the real-time capabilities found in SPL. Other possible candidates were eliminated, FORTRAN, for example, because most of its functions can be accomplished using the richer PL/I, and JOVIAL because the developers of SPL had already indicated that it was inadequate and had found it necessary to make many extensions to basic JOVIAL in the definition of SPL. Then, to serve as the benchmark problem, portions of the Titan IIIC programming specification developed by The Aerospace Corporation were selected. These consisted of a set of typical guidance equations, program blocks for engine output command calculations and staging sequencing, and a portion of the main control logic flow diagram, this last to enable determining the languages suitability for coding executive functions. The overall 1-second major cycle logic was specified, along with the executive program to control proper phasing of the major cycle and 5-, 10-, and 20-millisecond minor cycle functions. As is typical of aerospace programming specifications, also included was complete information specifying the range and accuracy required for all program parameters, including where extended precision was to be used, and the critical timing requirements to be met.

Four programmers independently coded the representative problem, two in each language. They were instructed to spend as much time as

possible on its more difficult and aerospace-peculiar aspects, avoiding repetitive operations that would not lead to meaningful conclusions or that could be handled in some way not provided for in the assigned language. None of the programmers was able to code the entire benchmark problem in the two weeks allotted, but each succeeded in coding substantially more than would have been possible using assembly language. Thus both SPL and PL/I were shown to offer significant benefits, chiefly through their relatively simple assignment and control statements. Nevertheless, both languages had serious deficiencies which prevented complete, direct solution of the problem. It was clear that the tricks and circumlocutions necessary to overcome these deficiencies would negate the hoped-for benefits of using a higher level language.

While the programmers were able to specify the equations easily, describing the data attributes necessary to control precision throughout the steps of a calculation presented difficult problems. This reflects the fact that both languages are oriented to having the bulk of the calculations done in floating rather than fixed point. Many of the fixed-point difficulties were similar in that the relatively simple way of doing an operation in assembly language was not available in either SPL or PL/I; to overcome these problems, the programmer would have to break the program into steps almost as numerous and as detailed as he would when coding in assembly language. For example, one operation the programmers needed to be able to do but could not, at least not without a great deal of difficulty, was to define a fixed-point variable with the binary point outside the number of bits actually allocated to the item. This may be desirable for variables having either very small or very large values; as a hypothetical example of the latter, if the coordinates of a vehicle's position are measured in feet and it is sufficient to maintain each to a precision of only 16 feet, the programmer would wish to scale the coordinates so that the binary point is four places to the right of the computer word's least significant bit in order to allow the greatest range of values for position.

Another problem arose when it was necessary to give a variable one scaling for a particular mission phase and another scaling for a subsequent phase. For example, it may be necessary to maintain a vehicle's position to one precision during near-earth maneuvers and to a much less accurate precision subsequently. However, neither PL/I nor SPL offered a means of dynamically rescaling variables without a great deal of coding duplication.

A third problem came about with the use of double-precision accumulation of products. To exemplify the mechanism involved, the multiply operation in the typical aerospace computer will generate a double length for the intermediate products of the following equation:

$$(V)^2 = (V_x)(V_x) + (V_y)(V_y) + (V_z)(V_z)$$

Making use of the double-precision addition command available in many computers enables accumulating the sum in double precision to obtain improved accuracy; however, this requires one more instruction per term and causes some increase in execution time. In some cases it is not desirable to pay this penalty to achieve improved accuracy; in other cases using double precision is mandatory, such as in matrix operations when the intermediate products are of opposite sign but nearly equal in magnitude. The programmers required control over the choice -- something that neither PL/I nor SPL offered simply and directly.

A fourth problem area concerned the temporary saving of intermediate results in common locations. Neither language provided a convenient mechanism for doing this with fixed-point quantities.

These, then were a few of the problems encountered with fixed-point variables in coding the test problem. Turning now to constants, a peculiarity of aerospace programming is found in the existence of two distinct types of constants: absolute and modifiable. Absolute constants, such as the coefficients in a polynomial approximation for the sine of an angle, are those which are very unlikely to be altered when the program itself is changed to accommodate it to a new mission. Modifiable constants, for example, accelerometer nonlinearity compensation terms, are those which are expected to be different for different missions. The program is written and checked out using nominal values for such mission-dependent constants, and the actual values are loaded directly into memory when they become known, often shortly before the mission. To permit this to be done, the programmer must be able to specify the attributes of such constants (required precison and maximum value as well as the nominal numeric value) as completely as he can those of variables. That is, he needs to be able to specify enough information so that modifiable constants could be treated similarly to variables in the automatic scaling algorithms of a compiler; if he could not do so, the program would have to be recompiled every time the value of a single mission-dependent constant was changed. The alternative provided in PL/I and SPL -- defining such constants as variables having preset values -- was found undesirable because it would hinder any compiler optimization functions. Also pointed up by an analysis of the test problem programming was the need for an ability to define the value of a constant as a function of other constants; this would simplify programming in many cases and, by enabling automatic change of such dependent constants, would help to reduce errors introduced when constants are modified.

Other language deficiencies were found in many areas. While the aerospace programmer is concerned chiefly with incremental inputs, telemetry, and discrete output commands, the input/output capabilities of both languages were not easily related to the requirements of the benchmark problem, instead emphasizing files, records, and peripheral devices. Similarly,

review of the limited debugging capabilities showed that they were not oriented to aerospace programs, for which much of the debugging is done using computer simulators rather than the actual computers. Also, neither language provided sufficient programmer control over object code optimization; however, PL/I contained a built-in possibility for extending its existing capabilities in this area.

Both languages were discovered to have numerous features that were not required for coding the representative benchmark problem. That there should be many features in PL/I that were not particularly useful for the application area was expected, but SPL also had a great many features of little or no utility, among them complex array declarations, table declarations, alternative forms of many statement types, text manipulation, and automatic and controlled storage. Both provided many needed capabilities, and both were deficient in the areas of fixed-point data declaration and arithmetic control. PL/I was superior as regards optimization control but lacked desirable features of SPL, such as built-in matrix operations, decision tables, and a simple method of interfacing with direct code.

Overall, SPL was found to have more features that might be useful and fewer that were clearly unsuitable; hence it was selected as the base for development into a concise aerospace programming language. The choice was influenced by the fact that the Air Force and System Development Corporation were proceeding with the development of SPL, and it was expected that continuing cooperation among the two government agencies and their contractors might result in further modification of SPL to make it more suitable and at the same time compatible with the corresponding NASA language. The NASA language developed on the basis of this work and the subsequent study phases was given a distinct name, CLASP (Computer Language for Aeronautics and Space Programming), to minimize confusion between it and the continually evolving SPL. Many modifications to SPL have been initiated by System Development Corporation as a result of the work discussed here, and most of the deficiencies discussed above no longer exist in SPL. The objective of having CLASP be a proper subset of SPL has been achieved in large part. In the absence of a standardization control authority, however, the compatibility of various compiler implementations will almost certainly vary, particularly with regard to semantic differences.

While CLASP's basic structure is similar to that of other higher order languages -- the assignment and logical control statements, for example, would not be surprising to a FORTRAN programmer -- it provides many features that are either unique in themselves or are used in unique ways. Only these unique capabilities will be discussed here.

CLASP allows the programmer to declare the attributes of fixed-point data items such that the code generated by a compiler will perform the indicated

arithmetic with the required accuracy and without excessive penalities in object code size or execution time. The CLASP programmer can specify the minimum total number of bits and the minimum number of fractional bits to be allocated to each fixed-point item. In practice, he should specify for each data item only the minimum number of bits to be allocated for the expected range of values and the necessary accuracy. While the total bits needed to allow data storage at the required precision might not be a multiple of the number of bits per word, it would require fewer instructions and less execution time to allocate storage for the item in increments of full words. The present CLASP compiler does not use any excess bits to allocate more than one data item to a single computer word unless explicitly directed to do so, but instead uses them to generate a consistent set of scalings that minimizes intermediate shifting.

As mentioned in the discussion of the base language selection, another fixed-point problem existed with regard to the use of registers or data words for temporary storage. Normally, when a fixed-point assignment of the form $\alpha = \beta$ is made, before storage takes place the computed value of β must be adjusted by shifting in order that the binary points of a α and β will be aligned. Such a readjustment should not be made, however, if α is a temporary variable that might be used in many places in the program and with different attributes desired for each place. The solution provided for this problem in CLASP is to declare such temporary variables as data items having the attribute TEMP. Doing this has the result that such a variable will assume the temporary attributes of the expression to the right of the equals sign until such time as a new temporary assignment is made to that variable.

To solve the problems relating to modifiable constants, CLASP allows them to be specified as parameters; absolute constants are specified simply as constants. Parameters may be changed before program execution without requiring recompilation, while constants are fixed at compile time. Both are likely to be assigned to read-only storage if the aerospace computer has such a structure.

The fact that a constant's value does not change without recompilation means that a CLASP compiler will be able to determine the permitted range of scalings solely from the value given. If, for example, a constant's value is established as 2.5, only two bits would need to be allocated for the integer part and one for the fractional part, greatly increasing the flexibility available to the scaling heuristics and algorithms in finding an optimum set of scalings. Constants might not even appear explicitly in the program; for example, a multiplication by a constant might be replaced by a shift. Parameters, on the other hand, must appear and must be declared with attributes such as the range of values and the precision required, just as variables are declared.

In cases where the scalings derived through the use of the scaling algorithms may be undesirable, CLASP provides a scaling operator to make it possible for the programmer to specify the total number of bits and the number of fractional bits for any intermediate result, just as he can for declared items. Thus in the expression

A + (B - C) .S (10, 9) + D

the scaling operator .S(10, 9) specifies that the size of the intermediate result (B - C) is 10 bits, nine of which must be fractional. This capability has been provided for occasions when the programmer has information about intermediate results -- such as the fact that B and C are always about the same size -- which is not supplied in the data declaration but which may be required to generate code that produces results of the required precision. An abbreviated form of the scaling operator can also be used when the programmer wishes to accumulate products in double precision and assign the result to a single-precision variable.

Object code efficiency is of great interest because of the small memory size of the present typical aerospace computer and the strict real-time requirements to be achieved by the typical program. Accordingly, many features are incorporated to enable the generation of such efficient object code by a CLASP compiler. Primary among these are three optimization directives that can be applied to any desired area of code: OPTIMIZE SPACE (n), OPTIMIZE TIME (n), and SIC. The last of these is provided for indicating that no optimization is to be attempted by the compiler; it is included for use primarily in early stages when the programmer is interested in getting a rough idea about program correctness. By placing a SIC at the beginning of a program area, the programmer can direct that all other optimization directives within that area are to be ignored.

For the space and time optimization directives, the parameter n serves to specify the degree or level of optimization. Recall that most aerospace programs have functions which must be performed at a high frequency and others which are performed at lower frequencies. Clearly, it is very important to optimize the execution time of the higher frequency functions, and proportionately less important as the frequency becomes lower. For example, if a control function is to be performed 20 times each second and a guidance function but once each second, the programmer could specify OPTIMIZE TIME (20) for the former function and OPTIMIZE TIME (1) for the latter. In the event that program areas included in the higher frequency functions are to be executed only under special conditions, the programmer can assign to them a relatively low degree of time optimization. With regard to space optimization, this is most likely to be specified for compiling the lower frequency functions.

Among the CLASP features that also have a marked effect on the degree of optimization obtained is the nonscalar subscript, (*), by means of which all elements of a row, column, or plane of an array can be successively referenced. Consider the following set of equations:

$$P_x = k_{11} V_x + k_{12} V'_x + k_{13} V''_x$$

$$P_y = k_{21} V_y + k_{22} V'_y + k_{23} V''_y$$

$$P_z = k_{31} V_z + k_{32} V'_z + k_{33} V''_z$$

If variables V_x, $V_y, \ldots, V''_z$ and the constants $k_{11}, k_{12}, \ldots, k_{33}$ are declared as elements of arrays, the single CLASP statement

```
P(*) = K (*, 1) * V(*) + K(*, 2) * VP(*) + K(*, 3) * VPP(*)
```

will accomplish the computations for all three equations. This CLASP statement could be translated by a compiler in two ways. First, it could be replaced by an equivalent statement with normal, single-valued subscripts, and this statement preceded by a loop control statement that would cause it to be executed three times, with the subscript value, initially 0, incremented each time. This would result in compiled code of size s_1 and execution time t_1. Alternatively, three statements could be generated from the single statement written; these statements would be similar to the given equations (with no subscripting). This alternative would result in compiled code of size s_2 and execution time t_2. If S were the level of space optimization and T the level of time optimization specified in the appropriate optimization directive statements, the best choice would be the first method if $S \cdot s_1 + T \cdot t_1$ were less than $S \cdot s_2 + T \cdot t_2$ and the second method if not.

The optimization directives are also used in the generation of fixed-point code. In the absence of any other information (e.g., from the scaling operator), the intermediate rescalings that may be required during arithmetic expression evaluation to resolve otherwise conflicting scalings should be chosen such as to minimize the function

$$T_r = \sum_{i=1}^{n} T_i r_i$$

where

n = total number of rescaling operations

T_i = time optimization parameter n in effect for the ith rescaling operation

r_i = execution time for the ith rescaling operation

This function states that scaling readjustments should be done in the region where the degree of time optimization specified is the least. A similar function could be written for the effect of rescaling on size optimization. In a practical case, it will not be necessary to evaluate all possible scaling readjustments to determine the minimum T_r because the problem can be partitioned and individual scaling readjustments determined for individual variables or small groups of variables.

CLASP contains features which some might think of as retrograde steps to machine dependency. These features were added to promote efficiency and because an aerospace computer program of necessity has a close interrelationship with its hardware environment. It is possible in CLASP to assign an identifier to a machine register and declare the attributes (e.g., data type, number of integer and fractional bits) of that register when it is referenced by that identifier. For example, the statement

DECLARE HARDWARE INTEGER, INDEX1=2

would assign the identification INDEX1 to hardware address 2 and specify that it contained integer values. In conjunction with these hardware declarations, the programmer has the capability of reserving the use of registers for his own special purposes. The directive LOCK 2 would prevent the compiler from generating code using the indicated hardware register 2 except where the programmer explicitly referenced it by the declared identifier. He would return the use of that register to the compiler by the directive UNLOCK 2.

Several in-line arithmetic functions are provided for doing elementary operations such as absolute value, rounding, and limiting to a specified range. Logical operators are included for performing logical product, logical sum, exclusive OR, and shifting operations. In the event that the programmer cannot accomplish his objectives using these machine-like operations, he can lapse into in-line assembly code without any attendant inefficiencies due to linkages. The interrupt capability of the computer, utilized for most aerospace program executives, is handled in CLASP by means of the ON statement; this allows the programmer to declare the means by which the interrupt routine is entered and exited. After the entry mechanism has been declared, interrupt processing is handled by means similar to the normal subroutine capability of the language; thus the executive can be considered as a special case of a subroutine. The LOCK and UNLOCK statements used to reserve and restore the use of machine registers can also be used to inhibit or activate interrupts.

Some of the things that CLASP does not contain may seem surprising. Any superfluous features would be likely to make the language harder to learn and use, to make it more costly to implement, and, most important, to result

in concomitant losses in the efficiency of the generated object code. Much attention was therefore paid to the specification of a "bare bones" language adequate to do the job efficiently but containing no frills.

CLASP does not have any built-in input/output operations because of the wide differences in the input/output characteristics of aerospace computers, together with their very application-dependent nature. Their absence is justified by experience with other special-purpose higher level languages: regardless of what may have been specified in the language, actual implementations differ widely because of differing application needs. Input/output operations are accomplished in CLASP by lapsing into direct code and making use of hardware declarations.

CLASP does not have the built-in mathematical functions -- sine, arctangent, etc. -- common to other languages. Although these are present in the base language, SPL, they were eliminated in defining CLASP because they would introduce unacceptable inefficiencies: to implement them, it would be necessary to prepare either a general subroutine package containing all functions or individual subroutines for each function. The general package would be inefficient if only a few functions were required by a particular application program, and the individual subroutines inefficient if most functions were required. These inefficiencies are further compounded when such subroutines are required with fixed-point input and output parameters. For example, the fixed-point arctangent function satisfactory in one aerospace application program may be unsuitable in another because of differences in the permitted ranges of arguments, accuracy required, and allowable execution time. In CLASP, mathematical functions may be defined by the same means as any other subroutine; the programmer, however, must supply the procedure specifying how the function is to be calculated, including the precision, range of values, etc., for its arguments. In practice, a library of such functions will be maintained, to be drawn from as required for any specific program with additions to the library being made as a need for function subroutines with particular properties occurs.

Compared with SPL, CLASP has many other, although less significant simplifications. Such things as status variables, table declarations, qualified named variables, matrix inversion, and notational substitution directives have been deleted. The conditional statements, allowable subscript expressions, and assignment rules have been greatly simplified. Together with the additions discussed above, these simplifications make CLASP a language than can do the job in the aerospace programming area and can be implemented for the computers of today and the near future without great expense.

THIS PAGE IS INTENTIONALLY LEFT BLANK

SPACE PROGRAMMING LANGUAGE:
FLIGHT SOFTWARE COMES OF AGE

Robert E. Nimensky
System Development Corporation

Abstract

This paper describes SPL/J6, a Space Programming Language designed to replace machine-language programming in the flight software development process. This language is shown to be capable of expressing the complex vector mathematics and decision control of guidance and navigation equations by integrating the hardware characteristics of the computer into SPL/J6. The flight programs produced by SPL/J6 compilers are shown to be time and space competitive with comparable machine language programs. Finally, the paper describes SPLIT, an SPL Implementation Tool. SPLIT makes SPL/J6 compilers cost effective by reducing their cost and lead time. The SPL/UNIVAC 1824 Compiler became operational in less than six months at less than half the cost of a conventional compiler.

INTRODUCTION

Few television viewers were aware of one of the anachronisms of the Apollo 11 space mission. The most sophisticated space mission of our era employed software development techniques that were primitive by comparison with the flight hardware techniques. Specifically, the software techniques employed machine language programming rather than higher-level programming languages such as FORTRAN, ALGOL, JOVIAL, or PL/I. The failure to adopt one of these languages or to develop a space programming language has tended to cause increasingly longer delays in space missions. The increasing complexity of space missions and spaceborne hardware increases the difficulty of machine language programs meeting scheduled launch dates.

Three basic reasons have deterred the aerospace industry from adopting one of these higher-level languages or developing their own aerospace programming language:

1. Currently available higher level languages cannot express the aerospace problem in its real-time, fixed-point, small word size environment.

2. Since object code optimization has not been the primary concern of most commercially developed compilers, there is a fairly widespread belief that inefficient object code is an inherent attribute of higher-level languages.

3. The cost and lead time for building compilers (the programs that translate higher-level languages to machine language) by conventional techniques have been too high to be cost effective.

The Air Force System Command's Space and Missile Systems Organization believes it has solved these three problems by developing SPL/J6 and SPLIT. SPL/J6 is the higher-level language specifically designed to accommodate the vector equations of flight programs and the real-time, fixed-point hardware characteristics of aerospace hardware. SPLIT is an acronym for SPL Implementation Tool, which is a meta-compiling technique used to build SPL compilers for different computers. This compiler building technique has enhanced the code optimization of SPL compilers and made SPL compilers cost-effective. In less than 6 months the SPL/UNIVAC 1824 compiler was built using SPLIT, and at less than half the cost of an equivalent compiler built using conventional techniques.

This paper describes the unique capabilities of SPL-SPLIT that contribute to cost-effective flight software development.

LANGUAGE REQUIREMENTS

A flight programming language is a tool for expressing mission specifications as a flight program. This program is a set of instructions that controls the flight computer throughout a mission. A typical flight program may perform navigation, guidance, attitude control, fire control, event sequencing, data management, and/or hardware evaluation. These functions can change with the objectives of the mission, such as research and development or operational; support or tactical; and command control or subsystem control.

Since navigation, guidance, and attitude control are mathematical functions, a flight programming language should approximate standard mathematical notation to facilitate translating the mathematical specifications into flight programs. Also, since logic flow is usually expressed in flowchart notation, the language should embody a powerful decision logic form to ease translating decision logic into a flight program. Finally, the language should be capable of integrating the machine characteristics of the target computer into the language of mathematics and sequence control. How well a machine-independent flight program language integrates all the machine-dependent characteristics of a computer is the real measure of its success (whereas the coding efficiency is the measure of the compiler's success).

SPL MATHEMATICAL NOTATION

FORTRAN, JOVIAL and PL/I are procedure-oriented programming languages capable of communicating algorithms in a mathematical-like notation. SPL/J6 is a newcomer to this family of languages and embodies most of their features and mathematical notation. In fact, J6 represents the JOVIAL lineage of SPL. Since SPL was developed by the Air Force, compatibility with the Air Force Standard JOVIAL was an objective; the necessary aerospace extensions to JOVIAL give SPL its unique character and the designation Space Programming Language.

As an introduction to SPL, the SPL/J6 program 'INTERGUIDE' depicted in Figure 2 returns four parameters from nine guidance equations. The programmer usually receives these equations in conventional mathematical notation. They are readily expressible in SPL, which can be understood by the computer, as well as by the mission planners who write the guidance equations.

The SPL/J6 program consists of statements, which are instructions to, and the only part of the program actually read by, the computer; and comments, which are enclosed in quotation marks ("), ignored by the computer, and meant only for the human reader.

The program is identified by lines 1 through 5. Lines 6 through 18 represent data declarations--all data are declared to be fixed-point with 12 binary place precision; initial values are set within the standard declarations; and the "overlay" declaration (line 10) equates a list of vector elements to a vector. The mode declaration (line 9) causes all undeclared variables found between this statement and the next mode declaration to be declared fixed point with 12 binary place precision. Thus, JP, SP, QP, J, S, Q, PSI.X4P, Y4P, Z4P, and even the output values SINX, COSX, SINY, COSY are fixed point 12 binary precision variables. The executable statements encoded on lines 21-24 are based on the original equations, and are part of a declared "close" routine (lines 20-25). This close is called at lines 27 and 36. Line 26 begins a loop that continues until "time" exceeds "maxt." At that time, the statement following line 40 (the end of the loop) is executed. System functions "log" (line 21), "atan" (line 30), "cos" (line 38), and "sin" (line 39) are called up by simple statements as shown. Lines 28-29 and 38-39 show nonscalar and multiple assignments, respectively; 48 represents the exit from the program, and line 49 denotes the end of the program.

The line numbers are merely mnemonic entries included for reference in this example, and are not input to the computer. Also they do not necessarily indicate a one-to-one relationship with the statements and comments, and some may be left blank.

Vehicle guidance, navigation, and attitude control equations are based on gravitational, navigational, and inertial coordinate systems, which are three-element vectors. The use of vectors and matrices is clearly seen in equations E, F, and I in Figure 1. How easily these computations are expressed in SPL are shown by lines 28, 29, 38, and 39 (Figure 2).

A. $L = \ln \tau/(\tau - T)$

B. $J' = \tau L - T$

C. $S' = J' - TL$

D. $Q' = (T^2/2) + \tau S'$

E. $\begin{vmatrix} J \\ S \\ Q \end{vmatrix} = V_{ex} \cdot \begin{vmatrix} J' \\ S' \\ Q' \end{vmatrix}$

F. $\begin{vmatrix} X'''' \\ Y'''' \\ Z'''' \end{vmatrix} = G \begin{vmatrix} X \\ Y \\ Z \end{vmatrix}$

G. $\psi = \tan^{-1} (X''''/Y'''')$

H. $\psi_T = \psi + \frac{1}{\eta_T} VT - S + K_{g_\eta} T^2$

I. $|\psi_T| = \begin{vmatrix} C_{\psi_T} & -S_{\psi_T} & 0 \\ S_{\psi_T} & C_{\psi_T} & 0 \\ 0 & 0 & 1 \end{vmatrix}$

Figure 1. Typical Guidance Equations.

```
0100-START(INTERGUIDE)    ''BEGINNING OF PROGRAM''
0200-PROC .INTERGUIDE(=SINX,COSX,SINY,COSY)
0300-''SAMPLE SET OF GUIDANCE EQUATIONS CODED IN SPL.  THE
0400-PROCEDURE NAME IS INTERGUIDE AND IT RETURNS THE VALUE
0500-OF THE SINE AND COSINE OF X AND Y.  ''
0600-DECLARE FIXED 12, TIME=0, MAXT= 10000, TAU, T, L, JSQP(3),
0700-    VEX,DG(3),XYZ(3),       MAX=1.3659,CETA,
0800-       V,KGN,DPSIT(3,3) = (8(0),1)
0900-MODE FIXED 12
1000-OVERLAY JSQP=JP,SP,QP
1100-DISPLAY.  DECLARE FILE DISPLAY2,
1200-       DEVICE=CRT2,
1300-       ERROR=ERR2
1400-DISPLAY2. DECLARE TEXT,K1 4='PHI=',PHI 12 FIXED 4
1500-SIGNAL.  DECLARE FILE SIGNAL1,
1600-       DEVICE = THRUSTCONTROL,
1700-       ERROR=ERR1
1800-DECLARE BOOLEAN,SIGNAL1
1900-''CLOSE PROCEDURE CALC DEFINED NEXT''
2000-CLOSE CALC
2100-       L = .LOG(TAU/(TAU - T))              ''EQUATION A''
2200-       JP = TAU * L - T                     ''EQUATION B''
2300-       SP = JP - T * L                      ''EQUATION C''
2400-       QP = (T**2)/2 + TAU * SP             ''EQUATION D''
2500-EXIT
2600-WHILE TIME LS MAXT ''BEGIN EXECUTABLE STATEMENT IN LOOP''
2700-       GOTO CALC
2800-       J,S,Q = VEX * JSQP                   ''EQUATION E''
2900-       X4P,Y4P,Z4P = DG * XYZ               ''EQUATION F''
3000-       PSI = .ATAN(X4P/Y4P)                 ''EQUATION G''
3100-       IF PSI GR MAX
3200-            THEN WRITE DISPLAY
3300-           ELSE SIGNAL1 = ON
3400-                WRITE SIGNAL
3500-       END
3600-       GOTO CALC
3700-       PSIT = PSI + CETA * V * T - S + KGN * T**2  ''EQUATION H''
3800-       DPSIT(0,0)=DPSIT(1,1)=.COS(PSIT)  ''EQUATION I''
3900-       DPSIT(0,1)=-DPSIT(1,0)=.SIN(PSIT) ''EQUATION I''
4000-END   ''END OF WHILE LOOP INITIATED ON LINE 26''
4100-
4200-
4300-
4400-''REST OF PROGRAM USED TO CALCULATE SINE AND COSINE
4500-OF X AND Y.''
4600-
4700-
4800-EXIT
4900-TERM ()  ''END OF PROGRAM''
   -
  -.STOP
```

Figure 2. Sample SPL Program.[1]

SPL DECISION LOGIC

Elementary decision logic is demonstrated in lines 31 through 34 where file DISPLAY2 will be written on an output device if PHI > MAX. If PHI ≤ MAX then SIGNAL1 will be set to ON and its value will be output with the WRITE command. This conditional statement may contain other conditional statements and thus be nested to any level. Very complex logic can be represented with this statement form; however, after a few levels of nesting the logic becomes difficult to read and is prone to programming errors because of misplaced clauses and ENDs. For these complex decision processes SPL has implemented decision tables as part of its structure.

[1] Levi J. Carey and Walter A. Sturm, "Space Software: At the Crossroads", Space/Aeronautics, pp. 62-69, December 1968.

Figure 3 is a decision table divided into upper and lower sections by a horizontal double line. The conditions (Boolean formulas) appear in the upper half and the actions (THEN statements) in the lower half. The double vertical line divides the table into two more sections. The left side is called the stub and the right side contains entries. Each set of conditions and actions is called a rule.

Rule 1 is read: If A equals B and Q is less than R and ZR is greater than 100, then set SIGNAL equal to FALSE and finally go to ABLE. If the conditions do not satisfy rule 1, then rule 2 is tried, and then rule 3; if none of the rules apply, then the ELSE rule is invoked and control transferred to ERROR.

Problem 1 converts to SPL format with no loss in the tabular form, as the comma separators substitute for the table lines (see Figure 4). The ELSE rule has been moved so that the entry list for all conditions and actions does not have to code for the ELSE rule. Since ALERT was not to be set in rule 1, a blank followed by a comma was needed to place RED and CLEAR in the proper columns.

	PROBLEM 1					
	STUB		ENTRIES			
			Rule 1	Rule 2	Rule 3	Else
CONDITIONS	A	EQ	B	19	C/D	
	Q	LS	R	Z	J*9	
	ZR	GR	100	150	5	
ACTIONS	SIGNAL	=	FALSE	FALSE	TRUE	
	ALERT	=		RED	CLEAR	
	GOTO		ABLE	ABLE	ABLE	ERROR

Figure 3. Decision Table Program.

```
                     PROBLEM 1
CONDITIONS A        EQ [ B  , 19 , C/D ]
           Q        LS [ R  , Z  , J*9 ]
           ZR       GR [ 100 , 150 , 5 ]
ACTIONS    SIGNAL = [FALSE,FALSE,TRUE ]
           ALERT  = [     , RED ,CLEAR]
           GOTO     [ABLE ,ABLE ,ABLE ]
ELSE       GOTO                       ERROR
           END
```

Figure 4. Usage of SPL Decision Logic.

SPL HARDWARE INTERFACE

The SPL forms discussed so far are more or less machine independent and do not involve the hardware characteristics of a target computer. SPL's treatment of I/O, interrupt processing, bit and byte manipulation, arithmetic logical operations, fixed-point scaling operators and data declarations, hardware operands, and direct code give complete control of the hardware to the SPL programmer. To illustrate how these machine characteristics are handled in a machine-independent language two hardware-oriented SPL forms are described.

The SPL CHRONIC statement for interrupt processing was implemented on the UNIVAC 1824 for data and timing interrupts. The form also was used for the AGC Master Reset condition. Though the interrupts are peculiar to the UNIVAC 1824, the SPL compiler generates all the code necessary for saving registers and returning control to the instruction following the location where the interrupt occurred after the routine coded by the SPL programmer is executed.

The UNIVAC 1824 I/O is referenced by channel number and is governed by a direct, indirect, or incremental mode. All these features have been implemented in a machine-independent form. Thus, the SPL syntax can accept pairs of attributes describing an I/O file, though only the code generator pass can analyze these pairs to determine if they apply to the UNIVAC 1824 before using them to generate I/O commands.

There is nothing that a higher order language and its compiler can do to eliminate the hardware deficiencies of a given computer. It can, however, make coding "around" the hardware as painless and as efficient as possible. The fixed-point arithmetic in SPL cannot eliminate the shifting inherent in scaling fixed-point numbers but, it can ensure accuracy and do all the shifting for the programmer. In SPL the data declarations assign fractional bits and integer bits. This information is stored in a symbol table and used whenever fixed-point operations are performed.

A REAL PROBLEM

The UNIVAC 1824C computer for the Titan IIIC vehicle is a good illustration of the software-hardware interface.[2] Translating flight equations into the machine language of this computer is a constant battle against time, memory, and hardware constraints. This computer has a read-only, random-access, thin-film memory of 4096 48-bit words, each word being divided into three 16-bit instructions or two 24-bit data words. A read/write memory of 512 24-bit data words is provided for storage of intermediate results and the I/O buffer. The computer has three index registers: fixed-point arithmetic commands (table precision add and subtract), real-time interrupts, and input/output commands.

Some of the problems encountered in coding for this computer are:

1. Limited memory addressing by 8-bit operand. A single extension register is added to this operand to give a complete 15-bit address. About 10% of the operational program is devoted to setting this extension register.

2. Shifting because of fixed-point arithmetic accounts for about 6% of the total code.

3. The 512 read/write memory locations cannot hold the 900 computed intermediate results. This problem is solved by time-sharing the memory locations so that new results replace ones not required for future computations.

Many intricate coding techniques were employed, but a reduction in mission capability was still required to meet the tight timing and memory constraints of the Titan IIIC. How did SPL/J6 and its first operational compiler on the UNIVAC 1824 attack these problems?

SPL/UNIVAC 1824

In the Titan IIIC the extension register manipulation accounted for 10% of the coding effort. SPL cannot eliminate extension registers but it can automatically set the extension register; so, on the UNIVAC 1824, this one feature eliminates 10% of the code a programmer has to write.

Assigning 900 intermediate results to a space that holds less than 512 is readily handled by SPL. There is a language feature, an ephemeral data type, which allows an SPL programmer to change the attributes of a variable during the course of a program. Each time a value is stored in an ephemeral variable, the variable takes on the attributes of the value currently stored. Thus, if a fixed-point number, scaled with 15 fractional bits, is stored in ephemeral A, the next time A is used in an equation it will be used with a scaling factor of 15. Subsequently, if a value scaled 10 is stored in A, the next time it is used in an equation a scaling

[2] Raymond J. Rubey, R. Dean Hartwick, William C. Nielsen and Otis F. Tabler, "Definition and Evaluation of Merit in Spaceborne Software", SAMSO-TR-68-268, June 1968.

of 10 will be used. In addition to this data type, the SPL overlay statement allows the programmer to equate several results to the same memory location.

Another way the SPL compiler allows multiple use of the limited read/write memory is by using a push down stack for storing all input parameters and local variables for procedures. Thus, as each procedure is completed, the intermediate storage for all the variables used in the procedure is released and becomes available for another procedure.

SPL OPTIMIZATION

Language Features That Aid Optimization

The major responsibility for efficient object code belongs to the compiler, not the language itself, but SPL forms were designed to provide information which could be used by the compiler to generate good code. These language features include the following:

1. Preset value declaration allows the compiler to generate initial values at compile time rather than at run time.

2. The constant attribute allows the compiler to use immediate instructions when available. An immediate instruction contains the data in the instruction rather than in a memory location.

3. The declare index declaration tells the compiler to maintain values as indices thereby reducing setting and resetting of index registers.

4. The free-form, multiple-line, arithmetic assignment statement allows the compiler to optimize storage of intermediate results.

5. The decision table allows the compiler to optimize conditions and actions over a large segment of code.

6. The inclusion of logical operations, shifting, bit and byte manipulation, hardware operands, and direct code permits the programmer to solve his problem directly in SPL; whereas in other higher-level languages which lack these features inefficient coding techniques are used to simulate these operations.

7. Item and table data structures allow packing of data to optimize memory allocation.

8. Arrays and tables optimize the use of index registers for setting multiple values.

SOME COMPILER OPTIMIZATIONS

The SPL/UNIVAC 1824 Code Generator performs the following local optimizations (statement by statement):

Statement	Optimization
A = A+B	An 'add to memory' instruction is used.
A = 0	A 'clear memory' is used.
A = A+1	An increment instruction is used.
A+B	The operand which needs to be scaled is loaded first to save storing into, and restoring from, a temporary register.
A GR B	Relationals are done by a subtraction followed by a branch on the value in the accumulator. If B needs to be scaled, the operands are reversed and the branch logic is changed to LE to save storing into, and restoring from, a temporary register.
A (I+3) + B (I+3)	Register memory is used so that only one computation of the index is required.

These optimizations have all been implemented on the UNIVAC 1824. Although SDC has been unable to measure the quality of the object code, analytical studies indicate that the compiler generated code is within 10% of the theoretical optimum code. The theoretical optimum code is calculated by coding a sequence in assembly language using every known coding trick to minimize code. The quantitative measure of SPL's efficiency is currently being performed by Aerospace Corporation, which is coding part of the Titan IIIC guidance program in SPL. The code generated by the UNIVAC 1824 compiler will then be compared to the hand-coded version that employed intricate coding techniques.

SPL COST EFFECTIVENESS

The SPL Compilers built by SPLIT have three logical passes: 1) SPL Syntax Analyzer, 2) SPL Semantics, and 3) SPL Code Generator. The SPL Syntax Analyzer parses SPL statements into statement trees and declared data into dictionary trees. These two sets of trees are input to the SPL Semantics in a tree pruning pass. Scaling, mixed mode conversions, and reduction of complex tree forms to simpler forms constitute the semantics functions. The first two passes are machine indepen-

dent, whereas the last pass, the code generator, produces assembly language and is thus machine dependent.

The first cost saving factor of the SPL technique is the machine independence of the first two compiler passes. Since they can be used for any machine, only the code generator need be constructed for each new computer requiring an SPL Compiler. By placing the two machine-independent passes on a host machine such as a CDC 6600 or an IBM 360/65, a complete compiler is available for every computer already having a code generator.

The second cost saving factor is inherent in the SPLIT Compiler building technique. The use of SPLIT to build the SPL/UNIVAC 1824 compiler significantly reduced the time and cost as compared to building the same compiler with conventional coding techniques. This cost reduction was achieved by automating the compiler building process with the SPLIT metacompiler, which is one of the most advanced production techniques for translating a higher-order language to machine code.

The SPL compilers are written in the SPLIT language, a problem-oriented language in which compiler building is the problem. In problem-oriented languages, the user deals directly with his problem rather than with the details of a particular machine. Thus, SPLIT's Syntax Language is ideally suited for describing the syntax of any programming language. This problem-oriented language allows the programmer to think about the syntax of the language in the same language in which he codes his syntax equations. The code generator pass is also a descriptive graphical language which allows the programmer to 'see' the statement trees produced in syntax and then to 'see' the code generated directly from that tree. Again, the most significant attribute of this problem-oriented language is that the programmer can think about his problem in the same terms in which he codes the solution to his problem.

BENEFITS OF META TECHNIQUE

A compiler written in a higher-order language inherits the same benefits as any other program written in a higher-order language. The time and cost of building a compiler are greatly reduced by using JOVIAL instead of an assembly language. By the same token, time and costs are even further reduced by using a procedure-oriented language such as JOVIAL. As a measure of this savings, the syntax of SPL requires 528 lines of SPLIT language to parse a language more complex than JOVIAL. These 528 lines of code generate a compiler of some 35, 000 machine language instructions. The syntax analyzer of JOVIAL, written in JOVIAL to run on the IBM 360, required 8192 lines of code to generate 25, 000 lines of machine-language code.

The ratios of source statements to machine code are impressive but the real benefit lies in the nature of the SPLIT language. It allows the compiler writer to deal with the syntax and semantics of the source language rather than deal with all the tedious program tasks associated with building a compiler such as, how to write the scan program, how to build the symbol table, what attributes go where. He writes the syntax description in a language which readily lends itself to describing formal languages. He writes the code generator, again, in a language designed specifically for generating strings of code. Using languages which are natural to the problem actually assists the compiler writer in designing solutions for complex compiler problems.

Having a compiler (syntax and generator) completely described in 20 pages reduces the problem of compiler maintenance to such small proportions that one programmer can easily maintain a compiler.

OBJECT PROGRAM OPTIMIZATION

The SPLIT generator's language, which pictures the statement tree, allows the compiler writer to 'see' all the relationships between the operators and operands in a particular statement. With these pictures at hand it is relatively easy for the compiler writer to produce localized optimum code. As noted earlier, this has been proven in comparisons between code produced by the SPL SPLIT-built compiler version and the machine-language version of the Titan III missile guidance computer program for the UNIVAC 1824.

When the statement trees are connected to form a program tree the latter becomes a directed graph in which each edge represents a flow path and each node represents a 'basic block'; that is, a set of instructions in which if one instruction is executed, all are. By analyzing the strongly connected regions, basic blocks which dominate others, and basic blocks which can occur on a path from one basic block to another, we find that entire programs can be optimized.

The kinds of global program optimization that can be done are:[3]

1. Eliminating redundant instructions.

---

[3] F. E. Allen, "Program Optimization", 5th Annual Review of Automatic Programming, pp. 239-307, 1969.

2. Removing invariant instructions from inside loops.

3. Replacing or modifying test sequence to produce better code.

4. Eliminating unused definitions and computations upon which they depend.

5. Optimizing resource allocation, minimizing use of temporary storage, saving recomputed values, etc.

SDC is currently under contract with ARPA to implement these optimizations using the graph representation of a program.

SUMMARY

A Space Programming Language has been developed by the System Development Corporation for the Air Force Space and Missile Systems Organization. The SPL/J6 language has resolved most of the problems that prevented higher-order languages from being used in the flight software development process. SPL/J6 is suited to space and avionics applications by virtue of its powerful mathematical decision control language, which is supported by many machine-oriented features. The space problem is readily expressed in the SPL language and efficient object code is evident in the object flight program.

The cost effectiveness of the SPL-SPLIT compiling system has been demonstrated by the time and cost reductions achieved in the building of the SPL/UNIVAC 1824 Compiler.

It should also be apparent that by implementing the global optimizations described earlier, more efficient code can be produced. The amount of optimization that can finally be achieved will be determined largely by the money spent to get the additional optimization required. Training programmers in the use of SPL will probably bring about the greatest optimization of object code at the least cost.

The operational use of SPL is thus far somewhat limited and has been restricted to spaceborne programs; however, new uses of SPL in avionics are being studied.

THIS PAGE IS INTENTIONALLY LEFT BLANK

CSC-STD70-009

September 1970

A TECHNICAL OVERVIEW

OF

COMPILER MONITOR SYSTEM 2 (CMS-2)

Prepared by

Systems Technology Department
Computer Sciences Corporation
3065 Rosecrans Place, Suite 201
San Diego, California 92110

TABLE OF CONTENTS

Section		Page
1	Introduction	1
2	Background and Design Development	2
3	Basic Features of the CMS-2 System	7
4	A Description of the CMS-2 Language	15
5	System Operation	28

Section 1

INTRODUCTION

CMS-2 is the abbreviation for Compiler Monitor System 2, a computer programming system designed and implemented for the U.S. Navy by Computer Sciences Corporation.

The programming language of the system is also called CMS-2, and represents a state-of-the-art combination of the most desirable capabilities of such widely used languages as JOVIAL, CS-1, ALGOL and FORTRAN.

The purpose of this report is to provide a broad overview of the CMS-2 language features and system capabilities. This overview is presented at a level of detail which will allow evaluation of its adaptability to advanced avionics systems requirements.

CMS-2 is not a theoretical system but rather an operational system which exists in the Navy today. CMS-2 is not merely a high level language, but encompasses the complete scope of a computational system including a batch processing monitor for controlling the production of tactical programs and optimizing throughput, a librarian for reduction of input handling and source program storage, loaders to load and link the generated object programs, debug aids for maximum ease in debugging, and a flowcharter for a built-in documentation capability.

Section 2 of this report provides a brief background of CMS-2 development. Section 3 describes the components of the CMS-2 system and their functions. Section 4 contains a description of the CMS-2 high level language while Section 5 provides a description of the control operators for the various system components.

While this report is limited to providing a broad technical description of the elements of the CMS-2 system, in-depth details of the CMS-2 design are available through the Fleet Computer Programming Center, Pacific.

Section 2

BACKGROUND AND DESIGN DEVELOPMENT

Since the early 1960's, the Navy has developed many tactical data systems using the CS-1 compiler and the MS-1 monitor system. In 1966, because of the rapid advances in compiler design coupled with the impending arrival of third generation computers, the Chief of Naval Operations assigned the Fleet Computer Programming Center Pacific (FCPCP) the responsibility for updating the Navy's computational facilities to meet the needs of the fleet into the future. FCPCP, in turn, tasked Computer Sciences Corporation to perform an evaluation of these requirements.

CSC conducted an in-depth analysis of existing Navy programming capabilities, proposed future Navy computer systems, and existing compiler languages. Based upon the results of this study, authorization for implementation was given by the Chief of Naval Operations and CSC was tasked to implement the CMS-2 system.

The CMS-2 capability has now been developed and expanded to produce executable object code for five different military computers in use on various Navy and Marine Corps projects. These computers are:

Target Computers	Major Applications
CP642A and CP642B	NTDS, ASWSC&CS, TACDEW, TACS/TADS
Litton L304 (CP879)	ATDS, E2C
AN/UYK-7	AEGIS (ASMS), DX, DXGN, LHA, S3A
UNIVAC 1830A	A-NEW
UNIVAC 1218/1219	Fire Control Systems

The CMS-2 language has already been specified for use by the Navy for several advanced projects, including DXGN, AEGIS, LHA, and S3A.

DESIGN CRITERIA

In order to develop an advanced system while retaining the many existing tactical data systems written in CS-1, the objectives of the new system were established:

- To combine the best features of the existing CS-1/MS-1 and other new languages for new and future requirements
- To allow salvage of the maximum value from previously developed CS-1 programs, or facilitate their ready translation
- To provide generation of object code for existing and future computers without changing system tapes
- To include program debugging and testing features needed for quality and efficient performance

During the design phase of CMS-2, CSC studied the features of such languages as CS-1, JOVIAL, FORTRAN, and ALGOL. The most desirable features of these languages as they best suited the particular needs of Navy applications were incorporated into CMS-2.

The significant features thus incorporated into CMS-2 include the following:

<u>Procedure-Orientation</u>

Forward- and backward linking to procedures

Local and global ranges of definitions

Inter-system name linking within construct of language

Communication pool processing

Source language debugging capability

Absolute or relocatable output allocation

Free format source statements

Expanded data types and structures

Fixed Point, floating point, character, Boolean, and status elements

Multi-dimensional array structures

Complex equivalencing of storage areas

Procedure, index, and item switches

Definition of table lengths at load time

Selective data pooling

Flexible Processing statements

Powerful arithmetic-exponentiation, mixed mode

Intrinsic or user-developed functions

Algebraic evaluation of expressions

Indirect referencing of table structures

Bit and character string referencing

User-specified or automatic scaling

Sophisticated Input/Output Capabilities

High level file structures

Record and stream processing

Extensive formatting of data

Centralized I/O processing

ADAPTABILITY

As noted earlier, the development of CMS-2 was precipitated for the most part by the inability of the CS-1 system to adapt to the needs of the Navy within a

changing environment. Thus, adaptability was made one of the prime considerations of CMS-2 design.

Adaptability within CMS-2 is attained in two ways:

- Adaptability of the CMS-2 compiler system to run on many machines
- Adaptability of the CMS-2 compiler to generate code for many machines and many applications

Adaptability of the CMS-2 System

CMS-2 is composed of discrete passes to perform the functions of source code cracking, intermediate language generation and local optimization, and finally listing and object code generation. This modular approach allows the compiler to be modified to generate for a new target machine by replacing the machine dependent portions of the compiler.

Within certain limitations, the compiler itself can be transcribed into its own language, compiled through a CMS-2 compiler generating code for a target machine, and the resultant code run on the target machine to give CMS-2 compilation capability on any machine for which there is a code generator. While this procedure requires program modification for the machine dependent monitor interfaces and any other machine dependent areas of the compiler itself, this procedure is far superior to the process of rewriting a total compiler system.

Finally, the sub-modular structure of the compiler itself makes it adaptable for modification to run in a multi-pass overlay environment when space requirements dictate.

Adaptability of CMS-2 Generated Programs

While the modular structure of the CMS-2 program structure can be used to advantage to provide adaptability and flexibility within the compiler system

itself, these features are of even more importance to the users in regard to object programs.

The procedure concept of CMS-2 coupled with the linking capability of the CMS-2 loader allows the user maximum freedom in coding. Existing segments can be combined to create new systems. When changes are required only the modified segments need be recompiled. Finally, the CMS-2 librarian system allows input data for compilation to be maintained safely on a mass storage thus eliminating the time consuming task of deck manipulation and associated problems such as dropped card decks.

Finally, like CS-1, CMS-2 allows the insertion of direct machine code, properly bracketed, within the high level statements. This technique allows existing or specially written procedures to be included within CMS-2 programs by merely bracketing them with appropriate high level statements.

Section 3

BASIC FEATURES OF THE CMS-2 SYSTEM

A CMS-2 system is currently operational at the Fleet Computer Programming Center, Pacific (FCPCP), and at other Navy project sites, on the Univac CP-642B/USQ-20 computer. A CMS-2 system is also being developed to operate on the Univac AN/UYK-7 computer. The present CMS-2 system at FCPCP includes a monitor system (MS-2), the CMS-2 compiler, a librarian, loaders for CP-642 object code, tape utility routines, and a flowcharter.

THE MS-2 MONITOR SYSTEM

The MS-2 monitor system is a batch processing operating system designed to control execution of CMS-2 components and user's jobs being run on the CP-642 computer. The monitor coordinates all system job requests, and provides the external communication for all programs running under its direction. This communication includes a control card processor, an input/output system, operator communication package, and a debug package providing dump, patch, and snap capabilities. In addition, MS-2 maintains a library of system programs, which can be called upon request. Job accounting information is maintained and output for computer center use and a priority scheduling algorithm is available for job processing.

THE CMS-2 COMPILER

The compiler is a three-phased language processor that analyzes a dual syntax source program and generates object code for any one of five different computers used in military projects today. The three phases of the compiler are described below and illustrated in Figure 3-1.

a. Syntax Analyzer - A user's source program, consisting of high-level CMS-2 or CS-1 language statements and properly-bracketed machine code instructions, is input into the syntax analyzer phase. The source statements are checked for validity, and an internal

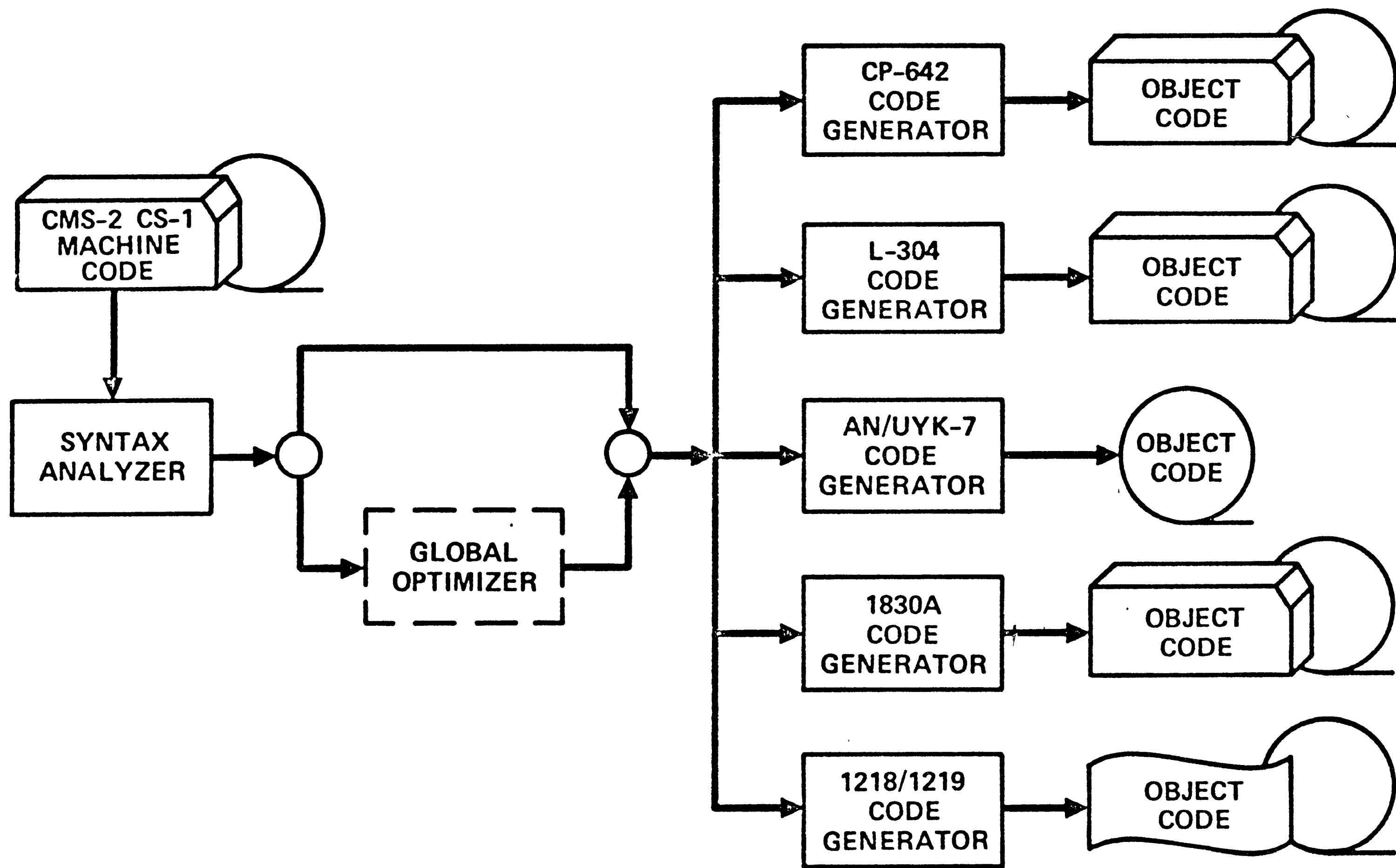


Figure 3-1. The CMS-2 Compiler

language (IL) and symbol table are generated. The IL and symbol table are completely independent of the target computer for which machine code will be generated.

b. Global Optimizer - The IL and symbol tables generated by the syntax analyzer phase can next be processed by the global optimizer phase. This phase identifies areas of the program which can be reorganized to reduce the program's memory requirements and execution time. Checks are made for loop improvement, simplification of arithmetic and logical computations, removal of arithmetic redundancies, and packing of data. No changes are made to the program structure. Instead, messages are printed so the programmer can make modifications if desired.

   The modular design of the compiler permits the global optimizer phase to be deleted if desired, and in fact this phase has not yet been implemented for the FCPCP CMS-2 system.

c. Code Generator - The code generator phase processes the IL and symbol tables to produce the final output listings and object code for the target computer. A separate code generator phase is used for each target computer. The CMS-2 system at FCPCP now includes code generators for the CP-642A and B, Litton L-304 (CP879), AN/UYK-7, Univac 1830A (CP-901), and 1218/1219 computers.

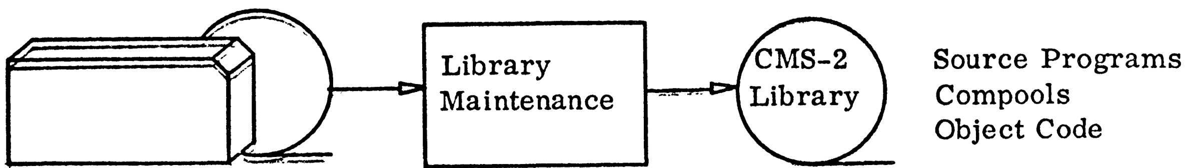
Object code is produced in a format compatible with the loaders used on each machine. Each code generator locally optimizes the object code. This includes utilization of instructions unique to the target computer, efficient register usage, and continuous analysis of object code strings for unnecessary or redundant instructions.

The code generators produce object code in one of two compilation modes: absolute or relocatable. In an absolute compilation, all instructions and data units are assigned absolute memory locations. The resulting object code represents an executable program. In the relocatable mode, each system element being compiled is given a starting address of zero. All memory locations and symbolic references between system elements must be assigned or resolved by a linking loader program. The loader itself joins various system elements, perhaps generated by separate compilations, to produce the final executable programs.

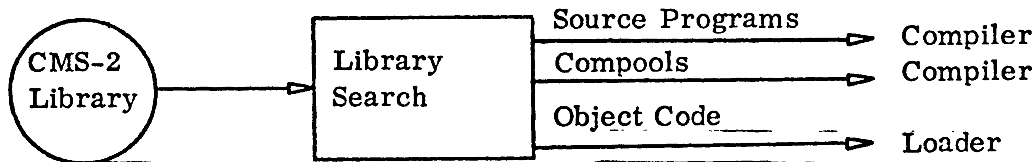
THE CMS-2 LIBRARIAN

The librarian is a file management system that provides storage, retrieval, and correction functions for a programmer's source programs and object code. Library operations are performed by three different routines.

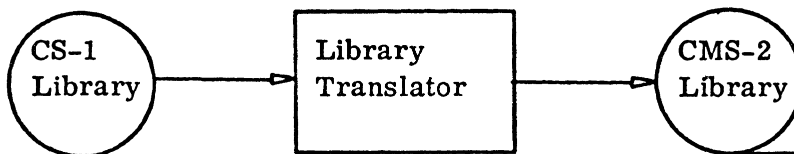
a. The library maintenance or executive routine (LIBEXEC) is used to create, modify, or reproduce libraries for CMS-2 programmers. A CMS-2 library is placed on magnetic tape and may contain source programs, object code or predefined data pools (compools).



b. The library search routine is responsible for retrieving data from a previously created CMS-2 library. Source program and compool elements may be retrieved for input to the CMS-2 compiler. CP-642 object code can be retrieved when requested by the CP-642 loader.



c. A library translator routine (LIBTRAN) is used to convert existing CS-1 program decks or libraries into a CMS-2 library format. Most CS-1 language statements are acceptable to the CMS-2 compiler; others are converted by LIBTRAN to equivalent acceptable statements. Those CS-1 statements that cannot be processed by the CMS-2 system are identified by LIBTRAN and must be changed by the programmer.



THE CP-642 OBJECT CODE LOADERS

The CMS-2 system includes two loader programs for CP-642 object code produced by the CMS-2 compiler. The absolute loader accepts object code generated by a compilation in the absolute mode. All instructions and data are loaded into computer memory at the addresses assigned during the compilation.

The relocatable loader processes only outputs from a relocatable compilation. Relocatable object code can come directly from the compiler or from a CMS-2 library. The relocatable loader assigns all memory addresses and links the program segments together to produce an executable object program.

TAPE UTILITY ROUTINES

The CMS-2 system provides a set of utility routines to assist a programmer with the manipulation of data recorded on magnetic tape. The routines provide the capability to construct, duplicate, compare, list, and reformat data files on tape.

THE CMS-2 FLOWCHARTER

The flowcharter is designed to process specific statements in a user's CMS-2 source program and output to the high speed printer a flowchart of the program logic.

CMS-2 JOB FLOW

To use the components of the CMS-2 system, a programmer must construct a job input deck to describe his requirements. The control cards in the job deck are processed by the MS-2 monitor. Based on the instructions specified on the control cards, the monitor can retrieve a CMS-2 component program from the CMS-2 system library and pass control to the component for further processing. Figure 3-2 illustrates the possible paths of job control.

Within a single job, several components of the CMS-2 system may be executed. Examples include:

a. Retrieving a source program from a user's library for transmittal to the CMS-2 compiler, and subsequent updating of the source program on the library (Utilizing the Library Search, Compiler, and Library Maintenance components).

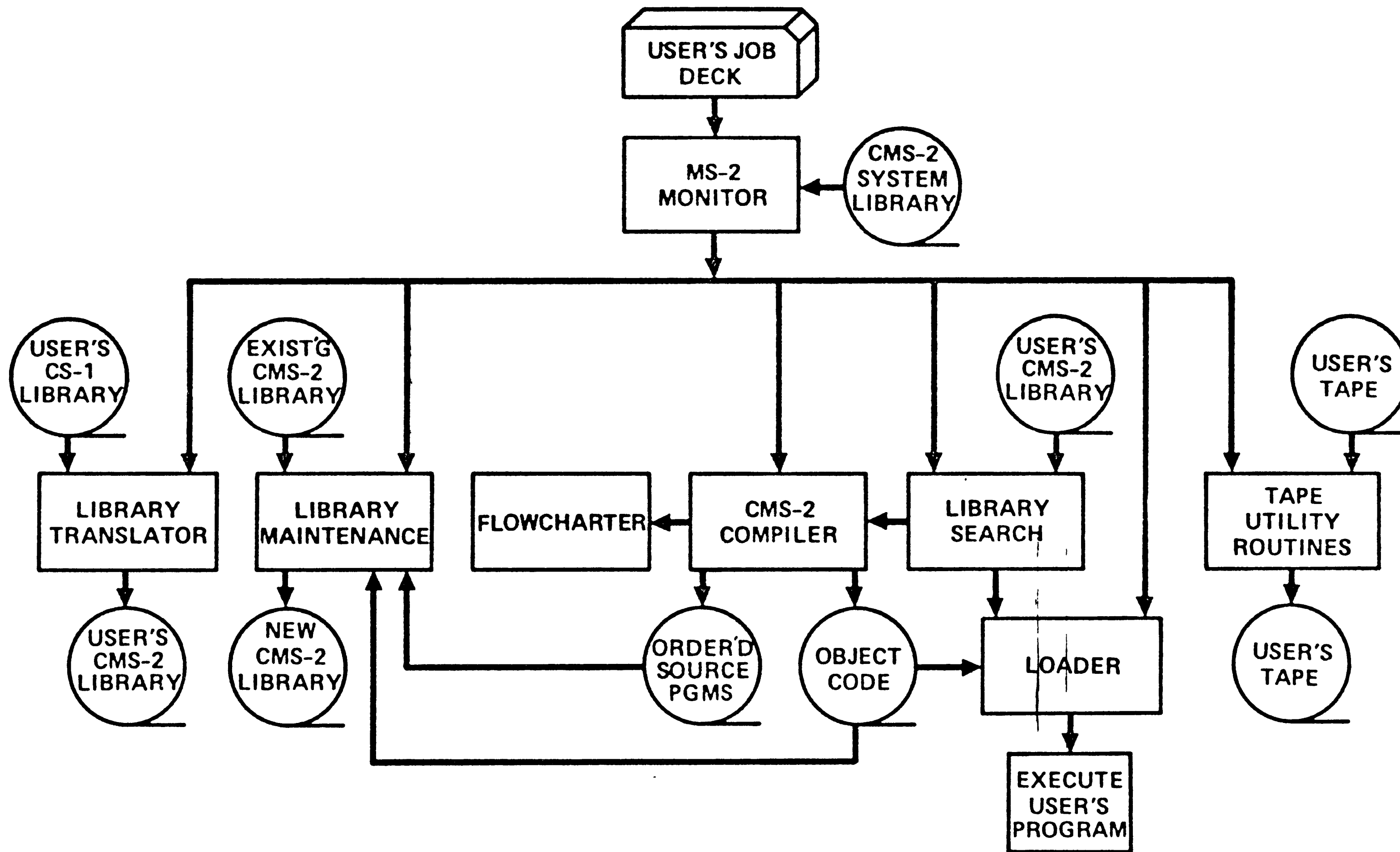


Figure 3-2. CMS-2 Job Flow

b. Compiling a source program for the CP-642 computer, producing a flowchart output, and loading the object code into the computer for execution (utilizing the Compiler, Flowcharter, and Loader).

c. Converting a CS-1 user's library into a CMS-2 user's library then listing the contents of the CS-1 library (utilizing the library translator and the tape utility routines).

Section 4

A DESCRIPTION OF THE CMS-2 LANGUAGE

The machine-independent, high-level language of CMS-2 has evolved from such well-used languages as JOVIAL, CS-1, and FORTRAN. The syntax includes a variety of operators and features to provide flexibility and capability to the experienced programmer. At the same time, the language is easy to understand and quickly learned by the novice.

A complete description of the language is found in Volume I of the CMS-2 Users Reference Manual (M-5012), published by the Fleet Computer Programming Center, Pacific. Below is a brief description of the CMS-2 syntax processed on the CP-642 CMS-2 system now operational at FCPCP. The CMS-2 compiler being developed to operate on the AN/UYK-7 computer will have expanded language features to better utilize the new capabilities of more sophisticated computers. These enhancements include reentrant procedures and their associated data structures, extensive program reallocation capabilities using address base registers, and other advanced language features.

LANGUAGE STRUCTURE

The CMS-2 language is composed of an orderly set of statements, or sentences. The statements are composed of various symbols that are separated by delimiters. Three categories of symbols are processed: operators, identifiers, and constants. The operators are language primitives assigned by the compiler to indicate specific operations or definitions within a program. Identifiers are the unique names assigned by the programmer to data units, program elements, and statement labels. Constants are known values, and may be numeric (octal, decimal, or hexadecimal), Hollerith codes, status values, or Boolean.

CMS-2 statements are written in a free format and terminated by a dollar sign. Several statements may be written on one card, or one statement may cover many cards. A statement label (followed by a period) may be placed at the beginning of a statement for reference purposes.

SOURCE PROGRAM STRUCTURE

The collection of program statements developed by the programmer for input to the CMS-2 compiler for compilation as a entity is identified as a compile-time SYSTEM. Those declarative statements within the system that define data to be manipulated are generally grouped into packages called Data Designs. The dynamic statements that cause manipulation of data or express calculations to solve the programmer's problems are grouped into procedures.

The two categories of system elements within a compile time system are the System Procedure (SYS-PROC) and System Data Design (SYS-DD). SYS-PROCs contain all procedure packages and may also include local data designs (LOC-DD) whose declaratives are "local" definitions. Local definitions may only be referenced within the SYS-PROC boundaries, unless flagged by external definition identifiers. SYS-DDs contain data declaratives valid throughout the compile-time system ("global" definitions). A typical compile-time system, therefore, may be represented as illustrated in Figure 4-1.

SYS-DDs A and D contain data which can be referenced throughout the system. The local data in SYS-PROC-C, however, cannot be referenced outside of C. A SYS-PROC may be entered through its "Prime procedure," whose name is automatically global to the system.

The system structure illustrated in Figure 4-1, may represent an entire tactical or application program. It can be compiled as an entity or in individual segments. For example, SYS-DD "D" and SYS-PROC "E" may be modified and recompiled separately from the other elements. References to data or symbols in elements A, B, or C can be processed by use of external

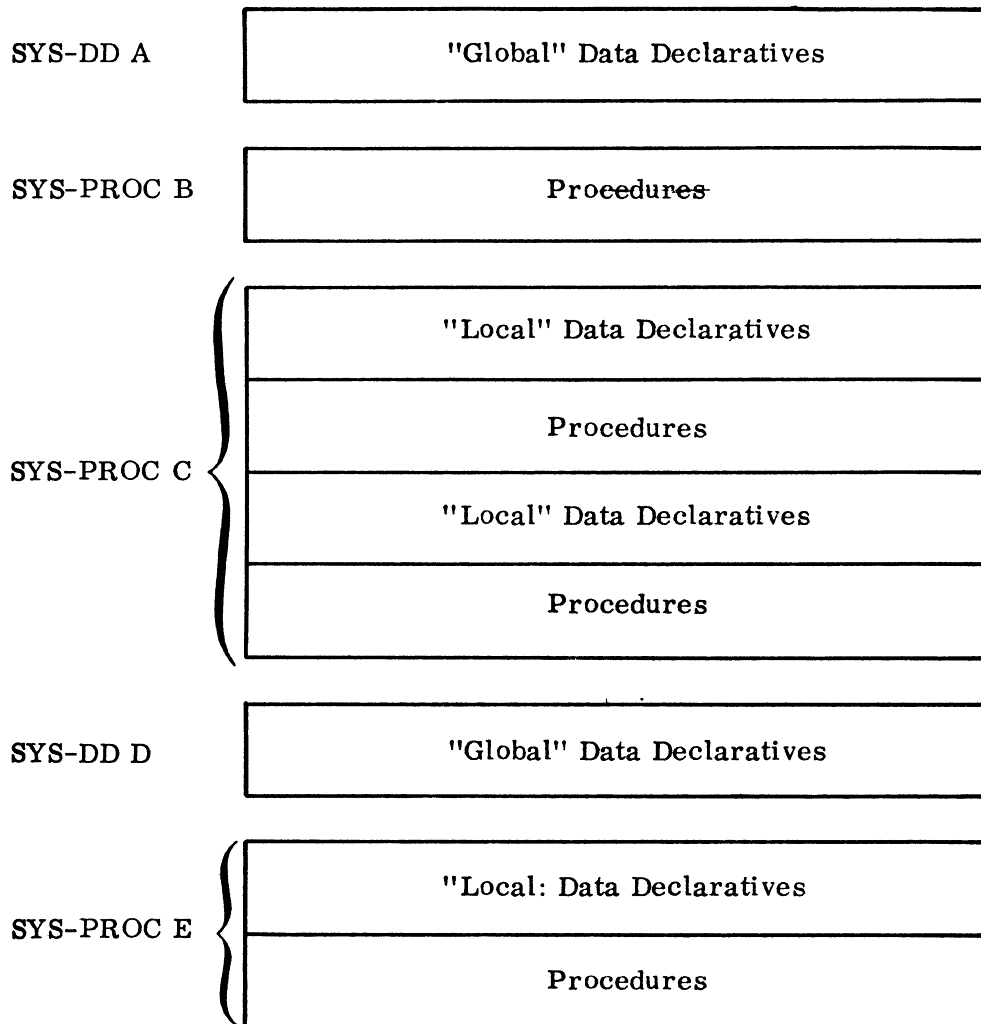


Figure 4-1. Representation of a Compile-Time System

reference declaratives during the compilations of D and E. These external references must be resolved and linked by the loader program when D and E are rejoined with A, B, and C for execution.

Procedures may be augmented by user-defined or compiler-intrinsic functions. Functions are called implicitly from CMS-2 dynamic statements which reference the function name in the same manner as a data unit.

DATA DECLARATIVES

The structure and size of data to be used in a program is defined by the programmer within the data design packages. The three major CMS-2 data types are variables, tables, and switches.

Variables contain a single quantity of data, irrespective of the target computer's word size. CMS-2 variables fall into six classes:

Integer	Signed or unsigned
Fixed Point	With scaling specified
Floating Point	
Boolean	(True or false states)
Status	Compiler-assigned values for user-defined mnemonics
Hollerith	Strings of character codes

A CMS-2 variable may be preset to a desired value within the definition statement. A shorthand notation permits simultaneous definition of multiple variables whose classifications are the same.

Tables hold ordered sets of information. The unit of data in a table is the item, which may include any number of computer words. CMS-2 tables may be one-dimensional (a "column" of items), or two- or three-dimensional arrays

(rows, columns, and planes of items). Items are subdivided into fields. Fields may be defined into the same type classifications described above for variables. The CMS-2 table structure also allows the programmer to

- Define a subset of adjacent items as a SUB-TABLE
- Allocate an ITEM AREA working storage area outside the table with the same field format as one item of the table
- Defer specification of the number ~~of items in~~ the table until load time
- Declare a LIKE-TABLE to automatically contain the same field structure as a previously-defined table
- Dynamically reallocate table data during program execution by use of indirect table addressing
- Pack field information across word boundaries or invoke a compiler algorithm to perform the packing

Switches contain a set of identifiers, or switch points, to facilitate program transfers and branches. The switch points represent program addresses of statement labels (S-SWITCH) or procedure names (P-SWITCH). Transfer of control to a particular switch point is usually determined by the value of a user-supplied index. In addition, an S-SWITCH may be declared as an Item-switch, with a specified constant defined with each switch point. Transfer is made to the switch point whose corresponding constant value matches an input value.

DYNAMIC STATEMENTS

CMS-2 dynamic statements specify processing operations or manipulate expressions. Algebraic expressions may include standard addition, subtraction, multiplication, and division operators, as well as exponentiation, mixed mode

values, and in-line redefinition of the scaling of fixed-point numbers. A true algebraic hierarchy of operation evaluation is used. Logical expressions use the operators EQ, NOT, LT, GT, LTEQ, and GTEQ. Boolean operators are AND, OR, and NOT. A single CMS-2 expression may include algebraic, logical, and Boolean operators.

Special operators are provided in CMS-2 to facilitate references to data structures and operations on them. These are:

BIT	To reference a string of bits in a data element
CHAR	To reference a character string
CORAD	To specify the core address
ABS	To obtain the absolute value of an expression
COMP	To complement a Boolean expression
POS	To position a magnetic tape file
LENGTH	To obtain an input/output file length
DISCAD	To specify an address on a disk
DRUMAD	To specify an address on a drum

The CMS-2 Statement operators allow the programmer to write his program in a machine-independent, easy to learn, problem-oriented language. Major CMS-2 operators are

SET	Performs all calculations or assigns a value to one or more data units. The assignment may be arithmetic, Hollerith, status, Boolean, or multiword.

SWAP — Exchanges the contents of two data units.

GOTO — Alters program flow or calls upon an S-SWITCH.

IF — Expresses a logical decision to provide conditional execution of one or more statements.

VARY — Establishes a program loop to repeat execution of a specified group of statements.

FIND — Searches a table for data ~~that~~ satisfies specified end conditions.

CMS-2 Input/Output Statements allow the program to communicate with various hardware devices while running in a non real-time environment under a monitor system. When CMS-2 I/O statements are used by the programmer, the compiler generates specific calls to Object Time Routines that must be loaded with the user's program. The Object Time Routines are designed to link with the monitor system and communicate with its I/O drivers. I/O declarative and statement features are briefly described below.

FILE — Defines the environment and pertinent information concerning an input or output operation, and reserves a buffer area for record transmission.

OPEN — Prepares an external device for I/O operations.

CLOSE — Deactivates a specified file and its external device, if appropriate.

INPUT — Directs an input operation from an external device to a FILE buffer area.

OUTPUT — Directs an output operation from a FILE buffer area to an external device.

FORMAT	Describes the desired conversion between external data blocks and internal data definitions.
ENCODE	Directs transformation of data elements into a common area, with conversion in accordance with a specified FORMAT.
DECODE	Directs unpacking of a common area and transmittal to data units as specified by a FORMAT declaration.
ENDFILE	Places an end-of-file mark on appropriate recording mediums.

COMPILE-TIME HEADER INFORMATION

Certain CMS-2 declarative statements specify controlling information to the compiler to direct the interpretation and code generation processes. These declarative statements are contained in the major header if the data concerns the entire compile time system. Control data pertaining only to one system element (SYS-DD or SYS-PROC) is placed in a minor header which immediately precedes the system element. Header statements and their functions are described here briefly.

MACHINE	A major header statement that specifies the target computer for which code is desired, such as the CP-642B, AN/UYK-7, or L-304.

OPTIONS — A major header statement that designates the compilation mode and listings desired. Some of these options are:

ABS — Object code is produced with program addresses allocated absolutely over the entire compile time system.

REL — Starting addresses for each system element are reset to zero, and relocatable object code is produced.

CMP — A compool output is produced for a SYS-DD. The compool consists of data definitions decoded into compiler format. Compools may be placed on a user library and retrieved during subsequent compiles. This procedure reduces compilation time significantly and speeds turnaround time.

SY — A symbolic listing is produced which lists the user's source statements together with octal and mnemonic representations of the machine code instructions that are generated.

	CR	An alphabetical cross-reference listing is produced that includes all data names and statement labels, their locations, and the locations of instructions that reference them.
	SA	A symbolic analysis listing summarizes all ~~data~~ definitions by declarative type and includes the data attributes.
	FC	A flowchart printout is generated from specific statements placed in the source program.

BASE — In an absolute compilation, the BASE value specifies the starting address of the program base, to which instructions and data are normally allocated.

DATAPOOL — Specifies that all data is to be allocated to a data base, separate and apart from the program base. Pooling of selected data definitions is provided by LOCDDPOOL and TABLEPOOL declarations.

EQUALS — The address allocation of a specific identifier may be established by the user at a fixed address or relative to another identifier.

NITEMS — The length (number of items) of a CMS-2 table is represented by an identifier to allow final length assignment to be made at program load time.

A complete description of these statements and other header statements (such as INDR-ALLOC, SYS-INDEX, MEANS, DEBUG, LOCALIZE, DEP, EXTERNAL, SPILL, MONITOR, and SOURCE) can be found in CMS-2 Users Reference Manual (M-5012).

PROGRAM CHECKOUT

CMS-2 debug statements may be placed in the source language of a user's program to facilitate rapid program checkout. These statements may reference any data units defined within the system. Machine code is generated by the compiler to call on object-time debug routines. The debug routines communicate with the monitor system during program execution to print the desired checkout data onto the system output.

Five program checkout statements are provided. Output code is generated only if the corresponding statements are enabled in the program header information. The object time routines are selectively activated at load time by a monitor control card. A programmer may then control and select the debug tools as needed.

DISPLAY	Causes the contents of machine registers and/or specified data units to be formatted and printed on the system output.
SNAP	The contents of a data unit are printed and stored. Subsequent executions cause a printout only when the data contents are modified.
RANGE	A high and low value are specified for a data unit. Each time the data is modified in the program, a message is printed if the value falls outside the range.

TRACE	A printout is generated for the execution of each CMS-2 statement between TRACE and END-TRACE boundaries.
PTRACE	Each CMS-2 procedure call encountered in the program being executed is identified by calling and called procedure names.

THE CMS-2 FLOWCHARTER

To obtain a printed CMS-2 flowchart output, ~~the user in~~cludes specific flowchart statements in his source program and selects the FC parameter on the OPTIONS header statement. The flowchart output contains two columns of logic symbols and associated labels, identification, references, narration, and comments. Eight CMS-2 operators control the flowchart output. These operators are ignored during code generation.

FCI	Imperative operator - causes the narrative to be placed in rectangular command symbol.
FCC	Procedure call - the narrative and called procedure name are printed within a hexagon symbol.
FCT	Transfer - causes exit from previous symbol to identifier of next specified symbol.
FCD	Decision - generates a two- or three-way branch from a diamond shaped symbol that contains a question narration.
FCP	Procedure entry point - identifies start of a procedure package, puts the procedure name in a small rectangular symbol.

FCS	Switch definitions - defines a statement or procedure switch referenced by flowchart statements but not included in the CMS-2 data designs.
FCE	End procedure or End Switch - delimits the definition of a flowchart procedure or switch.
FCM	Comment - gives additional information to clarify program flow. Comments are printed at the right hand side of the flowchart.

Section 5

SYSTEM OPERATION

Effective utilization of the capabilities of the CMS-2 system requires a familarization with the features and control statements of the various components. Operations of the monitor, librarian, loader, and tape utility programs are described briefly below. A complete description of these components may be found in Volume II of the CMS-2 Users Reference Manual (M-5012).

THE MS-2 MONITOR SYSTEM

MS-2 is an operating system designed to control the batch processing of CMS-2 system jobs on the CP-642 computer. The monitor controls execution of CMS-2 compilations, CMS-2 library operations, the absolute and relocatable CP-642 loaders, and the tape utility routines.

User's programs written in the CMS-2 language and compiled for the CP-642 computer can also be executed under control of MS-2. For these programs the following features of MS-2 are available:

Control of input/output to peripheral equipment

Debugging capabilities

Initialization of computer registers and keys

Communications to the computer operator

During the operation of the CMS-2 system in the CP-642 computer, the memory area is divided into three segments. The resident monitor area occupies a small portion of lower memory and contains control card processing and input/output routines of MS-2. The nonresident monitor area is utilized by other MS-2 routines called from the system tape only when

needed. The user's area is reserved for CMS-2 component programs (the compiler, librarian, loader) or execution of a user's CMS-2 program.

Control Card Operations

The programmer's instructions to the monitor system are conveyed on MS-2 control cards. Each MS-2 control card begins with a dollar sign ($) in column 1, followed by a control card identifier that specifies the operator or action desired. Other parameters follow, if needed, and are separated by commas.

MS-2 control cards are divided into five general categories:

1. Job definition
2. Operator communication
3. Processor calling
4. Debug statements
5. Miscellaneous

Figure 5-1 illustrates control card functions and the division of core into three segments.

Job definition control cards ($SEQUENCE, $PRIORITY, $JOB, $ENDJOB, $EOI) define the beginning and end of a user's job and provide accounting and scheduling information. Operator communication control cards ($TYPE, $HALT, $MTAPE, $UNLOAD) cause messages to be typed on the typewriter and in some cases, wait for a response. Processor calling control cards ($CMS-2, $LIBEXEC, $LOAD, $UTILITY) retrieve CMS-2 system components from the System Tape for execution. Processor calling cards are followed by card decks in the format required for the specific component. Control returns to the monitor when the component has completed its processing.

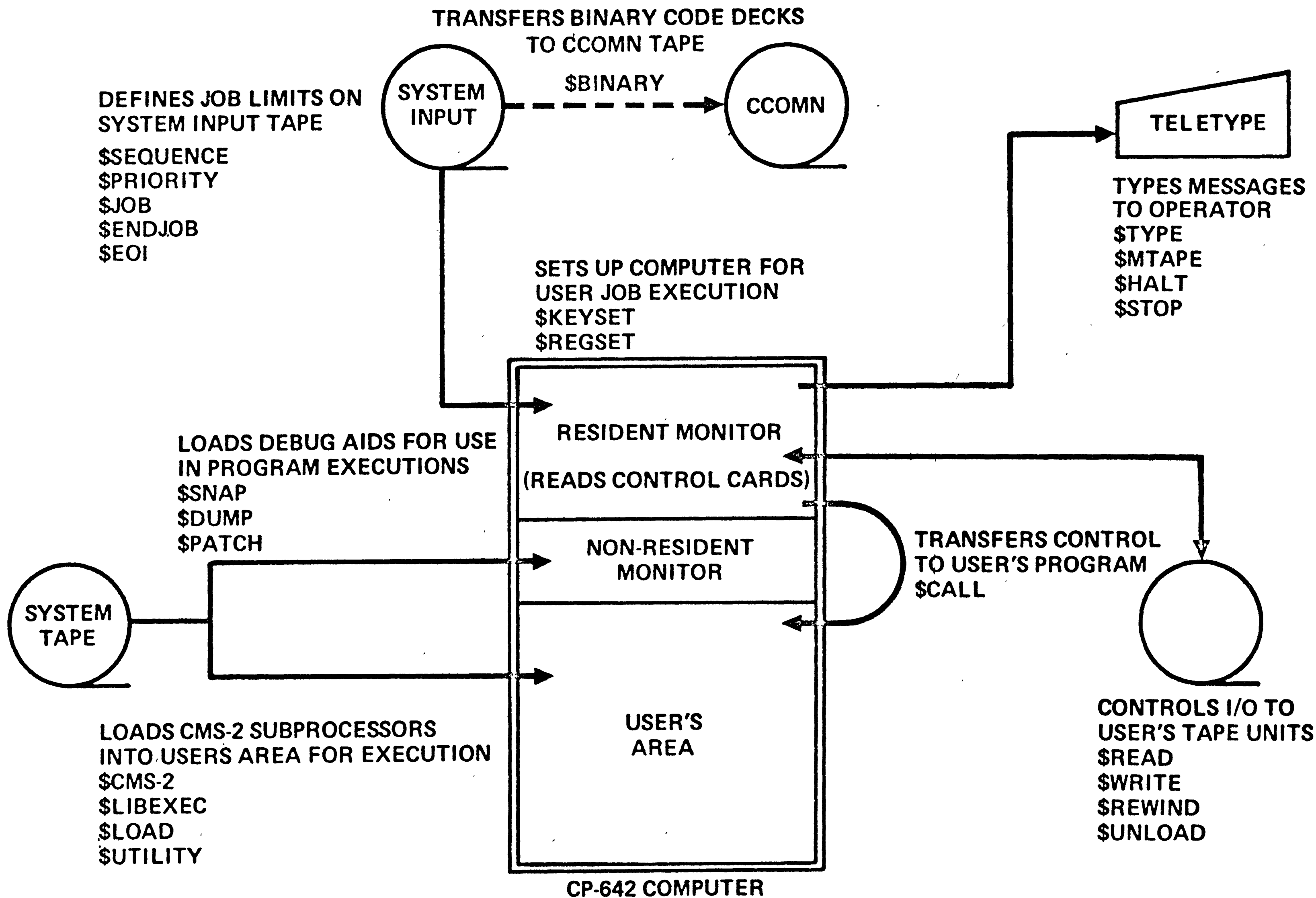


Figure 5-1. MS-2 Control Card Functions

Debug statement control cards ($DUMP, $SNAP, $PATCH) call debugging routines into the non-resident monitor area for use during execution of a user's CMS-2 program. The miscellaneous control cards provide a variety of other functions to the user. These cards include $BINARY, $CALL, $KEYSET, $REGSET, $READ, $WRITE, $REMARK, and $REWIND.

THE CMS-2 LIBRARIAN

The CMS-2 librarian can maintain program file information on magnetic tape in a format which provides for rapid and easy maintenance and retrieval. These libraries contain data used as input to the CMS-2 compiler or CP-642 loaders.

A CMS-2 library is a tape that contains system data designs (SYS-DDs), system procedures (SYS-PROCs) and header information. These elements may exist in any of three formats: CMS-2 source language written in a modified card-image form, relocatable object code binary card images produced by the CMS-2 compiler, or compools. Compools are tabular outputs of the CMS-2 compiler generated from the compilation of a SYS-DD. The compool may be retrieved as input to the compiler in lieu of the SYS-DD source statements. This eliminates the repeated recompilation of essentially stable and often used common system data designs.

The library maintenance routine (LIBEXEC) provides three basic functions: library creation, maintenance or updating, and listing. Only one of these functions is performed at a time. The librarian program is called with the MS-2 control card $LIBEXEC.

This card is followed by a set of control cards defining the operations to be performed. The following list contains major control cards that are used with library maintenance operations.

/PREPARE	Requests that a new CMS-2 library be created from source cards or an existing CMS-2 or CS-1 library
/EDIT	Requests that an existing CMS-2 library be updated and written onto a new tape
/LIST	Requests printout of specified library elements from an existing CMS-2 library
/TAPID	Identifies source of library data
/COPY	Specifies elements to be copied intact
/COPYC	Specifies elements to be copied and corrected
/I	Inserts new statement(s)
/D	Deletes statement(s)
/R	Replaces statement(s)
/RFILE	Starts the relocatable code file
/PRINT	Generates a listing of library elements
/ENDLIB	Specifies the end of the library run

Retrieval of library elements for subsequent compilations is accomplished by including appropriate control cards in the CMS-2 program source deck. These control cards and their basic functions are listed below.

LIBS	Identifies the library to be used
SEL-ELEM	Selects a specified element

SEL-SYS	Selects a specific group of elements
SEL-HEAD	Selects a header element
SEL-POOL	Selects a compool
CORRECT	Initiates corrections to an element being retrieved
/I	Inserts new statement(s)
/D	Deletes statement(s)
/R	Replaces statement(s)

CMS-2 LOADERS FOR THE CP-642

The CMS-2 system includes two loader programs for CP-642 machine code produced by the CMS-2 compiler. The absolute loader processes code compiled in the absolute mode. The relocatable loader loads code compiled in the relocatable mode, and performs address allocation and reference linkages to produce the executable program.

The CMS-2 compiler generates CP-642 object code in 80-column card format. For the ABS or REL parameter on the CMS-2 OPTIONS statement, card images are written onto the CCOMN output tape and may be input directly to the loader for immediate execution. REL code may also be placed on a CMS-2 user's library. For the parameters ABS(P) and REL(P), punched cards are produced. The parameters ABS(SV) and REL(SV) cause the card images to be written on the magnetic tape named COBJT, which is unloaded and saved at the end of job. The COBJT tape, punched cards, or REL code on a user's library may be input to the loader during a subsequent job.

Loader Control Cards

The loaders are requested when the MS-2 control card $LOAD is encountered in the job deck.

$LOAD	Requests the relocatable loader
$LOAD, A	Requests the absolute loader

The parameters D, S, R, T, P may be added individually or in any combination to activate the CMS-2 debug routines for DISPLAY, SNAP, RANGE, TRACE, and PTRACE, respectively. The parameter F may also be used to FORCE the load of object code that includes compiler errors.

The $LOAD card causes the appropriate loader to be called into memory, and MS-2 passes control to the loader. The loader then processes loader control cards in the job deck to direct the load operation. The six loader control cards are:

TAPE	To specify which input tape contains the binary object code to be loaded
LIBS	To designate a user's library which contains binary object code to be loaded
SELB	To specify which object code elements are to be selected from a tape or library for loading
BASE	To establish program and data base locations for relocatable code

TSD	To modify the length of variable-length tables in relocatable code
ENTRY	To designate the program entrance location for program execution

Loading Absolute CP-642 Code

The absolute loader accepts absolute object code input from binary cards on the system input, CCOMN tape, or an output tape (COBJT) from a previous job. Only the ENTRY and TAPE loader control cards are processed by the absolute loader; both are optional. The only output of the absolute loader is the object program loaded into memory and ready for execution. Figure 5-2 illustrates operation of the CP-642 absolute loader.

Relocatable object code may be input to the relocatable loader from four sources: binary card decks in the system input, CCOMN or saved tapes, or from a CMS-2 library. All six loader control cards may be used to direct the relocatable loader. At the completion of loading, the loader prints out a relocatable load map that includes the absolute address assigned to all global identifiers. Figure 5-3 illustrates operation of the relocatable loader.

THE MS-2 TAPE UTILITY PACKAGE

The MS-2 monitor system contains a group of magnetic tape handling routines which are requested by the MS-2 control card $UTILITY. The routines can perform such tasks as constructing, duplicating, comparing, listing, and reformatting data files on magnetic tape. These tasks are performed under the direction of a set of tape utility control cards.

CARD SEQUENCE

$LOAD, A

ENTRY

(BINARY DECK)

TAPE, CCOMN

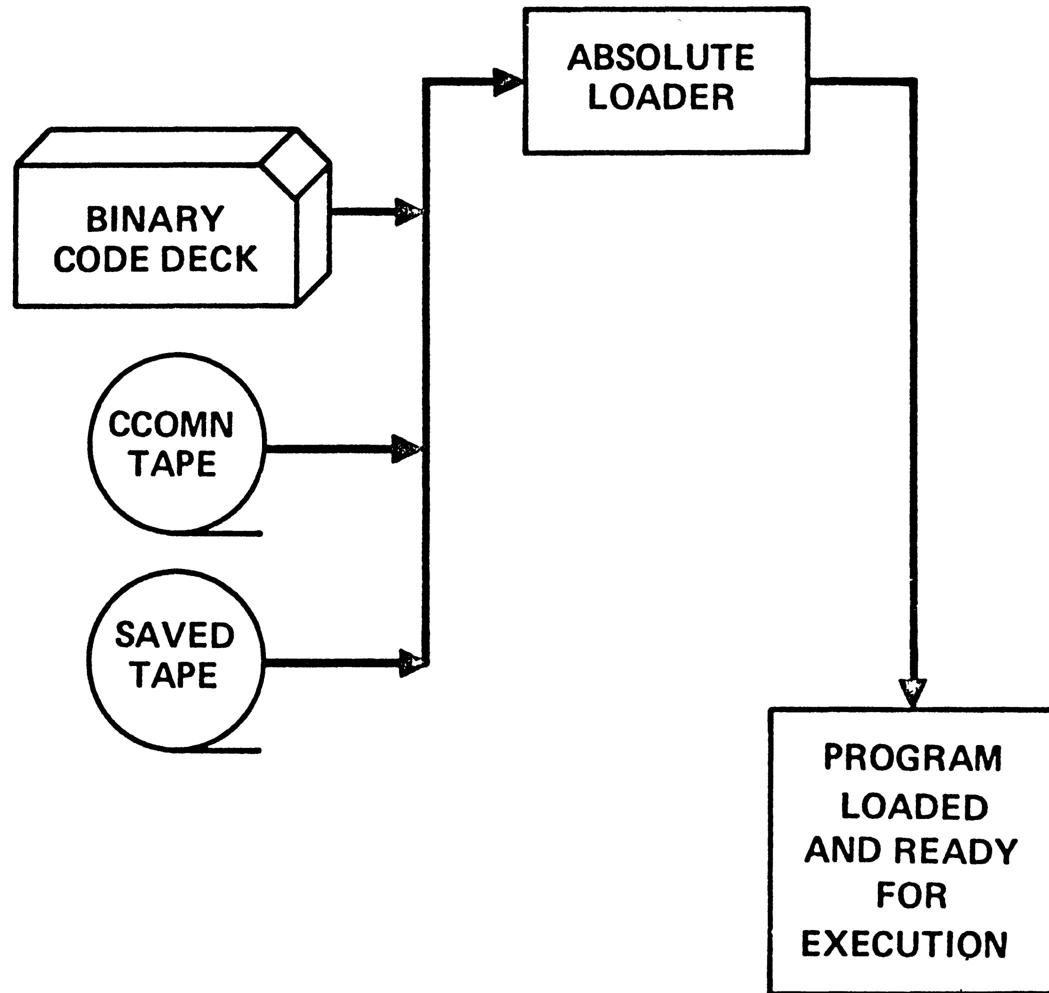
TAPE, EXTERNAL-ID

$DUMP

$PATCH

$SNAP

$CALL



CSC-STD70-009

Figure 5-2. The CP-642 Absolute Loader

CARD SEQUENCE

$LOAD
ENTRY
BASE
TSD
(BINARY DECK)

TAPE, CCOMN
BASE
SELB

TAPE, EXTERNAL-ID
BASE
SELB

LIBS, NAME (ID)
BASE
SELB

$DUMP
$PATCH
$SNAP

$CALL

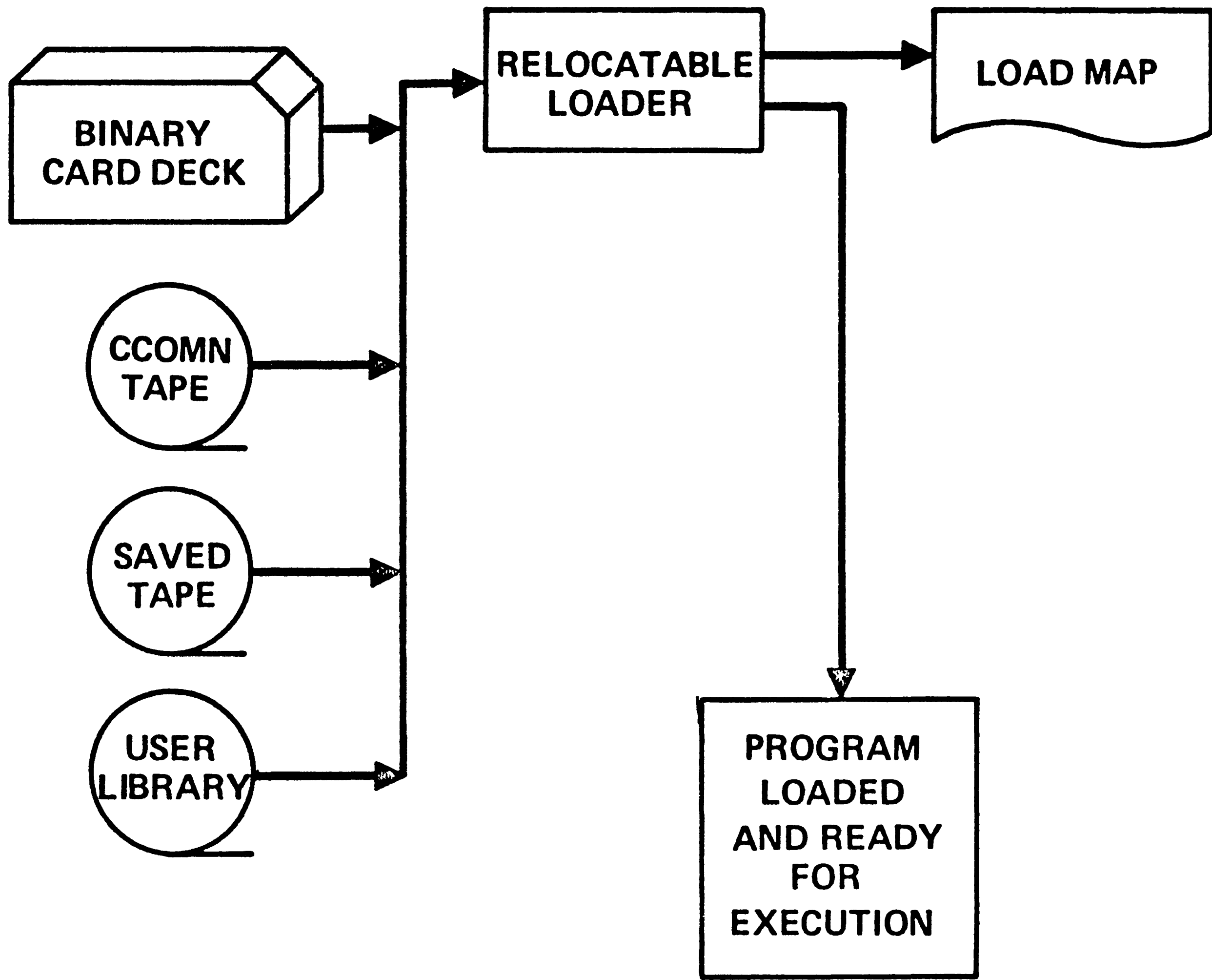


Figure 5-3. The CP-642 Relocatable Loader

The 12 tape utility control card operators are classified as follows:

Major Functions	Tape Positioning	Messages to Operator
FORMAT TAPE	SKIP	TYPE
DUPLICATE	BACKSPACE	HALT
COMPARE	REWIND	MOUNT TAPE
LIST	UNLOAD TAPE	
END FILE		

PANEL #1

SUBJECT: High Level Programming Language Compatibility

CHAIRMAN: Dr. Edward Bersoff
Logicon Corporation

PANELISTS: Mr. Vint Cerf
University of California at Los Angeles

Capt. Bruce Engelbach, U.S.A.F.
Space and Missle Systems Organization

Mr. Robert E. Nimensky
Systems Development Corporation

Mr. John P. O'Brien
Computer Sciences Corporation

Mr. Lynn Shirley
Informatics

Introduction

Entner: Before we begin our panel session, I'd like to steal a few cycles to mention a point which I think is important. Most of the people here today are software oriented and do understand the full meaning of the word "compiler". I have to say that I'm not one of these people. I don't understand software that well. I consider myself more hardware oriented than software oriented, and so I would like to discuss what I believe the word "compiler" means and give you an indication of the direction in which the AADC program is moving. Perhaps, in this manner, we can address these views, to an extent, within the panel sessions.

First off, I consider a compiler to be a software device which translates languages--a device which translates high level language into object code for a specific computer. Not a terribly intelligent device, necessarily. As part of our development program we hope to develop something we call a Program Synthesizer. I believe that some of the people here today who, as I, are more hardware oriented than software oriented are thinking in terms of what we call a Synthesizer when they think of a compiler. Essentially, when I say Synthesizer, what I mean is a device which has as an input mission requirements, hardware system definition, and let's say, objectives and constraints. Out of this device comes a definition of the specific computer hardware one would need to meet the requirements and the object code to run on the specific computer. Now, that, I believe, is quite different from the generally accepted concept of a compiler. I think an important point made this morning and yesterday is that we would like to see something that permits the system engineer to, in a very gross sense, define his requirements; define his objectives and come up with the object code for a specific computer. I think this is more in line with our thinking. Another interesting area which comes up when we address a mechanism which translates a Specific Operational Requirement (SOR) into a program, and let's say for the moment, the output appears in an intermediate language, it is not necessarily a program which has to be handled by people. If it goes directly from some sort of Synthesizer to a compiler, the task of writing the program has been taken out of the hands of the engineer and the programmer and has been handed over to a machine. The point is, how does automating the programming process using, for instance, the Algorithm Bank which I discussed yesterday--how does that modify requirements for high order languages?

With that, let me introduce our first panel chaired by Dr. Edward H. Bersoff of Logicon.

Panel 1: Software Commonality

Bersoff: The first thing I would like to do is introduce the panel. Starting at one end is Capt. Bruce Engelbach, who works for the Guidance Division of the Space Guidance Branch for the Deputy of Technology of the Space and Missile Systems Organization (SAMSO) of the Air Force. He has been active in the area of advanced development of aerospace computing systems since 1967. His prime interest is in the production of real time software. He's been monitoring the design and development of SPL/J6 and the SPLIT Compiler for the Air Force. Bob Nimensky is with the System Development Corporation of Santa Monica. He's in the Space and Range Department and is the author of SPL. He is currently working on the Meta Compiler. John O'Brien got his Bachelor's and Master's at Rutgers. Did graduate studies in physics and computer science. He spent four years at Bell Labs in Murry Hill working in solid state physics and low temperature physics. For the last three and one half years, he's been with the Computer Sciences Corporation and has worked on a Navy air traffic control program and, for the last two and a half years, he's been working on CMS-2. Currently, he's the Project Manager in charge of development and delivery of the CMS-2 system for FCPCPAC. Lynn Shirley is with Informatics. He participated in the iniital development of JOVIAL. He developed compilers for special purpose languages and COBOL. He created a dialect of PL-1 and a compiler for it. He's a former member of USASI Committee X3426E for JCVIAL prestandardization. He's currently a member of the Air Force committee. Vint Cerf has a BS in Math from Stanford, an MS in Computer Science from UCLA. He is now working on his PhD in Computer Science at UCLA. He worked at North American Aviation, IBM, where he worked on the QUICKTRAN time sharing system, Jacobi System's MINITS Time Sharing System. At the UCLA Computer Science Department, he is working on the ARPA Network Project. Multiprocessing architecture and operating system design study and the Meta 5 Compiler Computer System. His present interest is memory architecture and storage allocation algorithms for multiprocessing systems. I went to school at New York University. From there I went to NASA's Electronic Research Center where I participated in the development of CLASP. For the past six months, I've been with Logicon.

About two years ago, I sat in on a panel which was essentially the same (as this one)--Capt. Engelbach was on it--we discussed languages then, at the time, the question was not which language should we use for airborne programming; but can we use a (high level) language at all? And not do we need fixed point in our machine but can we get floating point? It seems that in the past two years things have changed considerably. I know this conversation is going to go off in many directions... I want to address compatibility first. So, I would like to ask the panel is it important that the AADC language be compatible with any others? Why? And if so, which one? Let's start with our man from SPL.

Nimensky: There are several assumptions one has to make. I know most of the military systems users are the ones who originally brought up most of the problems concerning compatibility. SPL has a compatibility problem with CLASP. We spent many, many man months of effort trying to get CLASP and SPL so that we'd only have one language for spaceborne computers. I think that (attempt) has basically failed. As we said this morning, we have

FORTRAN which is probably the most universally used language in the world today, and no two FORTRAN's are the same...it just doesn't work. There are many different versions of FORTRAN. Every manufacturer puts on his little "trinket" which he thinks is good for his system. Basically, I think compatibility doesn't work at all.

O'Brien: I think it's a very real problem, especially in the military with large systems. One of the problems you face is that a contractor and a group of people will develop a system for their customer, the Navy, Air Force, whoever it may be. Then the system is turned over to the customer. Now, if the customer is going to maintain that system, he has to know the language in which that system is written to be able to do some meaningful work--making changes and keeping that system running. If we don't have some sort of compatibility, look at the problem our customer has.

Bersoff: Compatibility with what for the AADC?

O'Brien: The AADC is a Navy project, right? It'll have Navy programmers maintaining that system when it's turned over to the Navy. Where are you going to train these programmers. You have a training problem. If we could adopt one standard language, it would simplify that training problem. Navy officers could get transferred from one assignment to another assignment, hopefully, without the retraining cycle.

Bersoff: Let me ask this: Is there a...I'd like to get an admission that there is no real technical reason to use CMS-2, or SPL or CLASP.

O'Brien: It's true.

Bersoff: It's more of a management decision.

O'Brien: It's a management decision. There was a technical reason... well, I guess there was a management reason for using CMS-2 in the existing NTDS system environment. NTDS has many, many programs written in the old CS-1 language. For that system to be rewritten in a new language would have just been out of the question. You'd have to throw away all the work that had been done over a period of time...all the money that was spent for all these programs would have been scrapped in a start-from-scratch effort. CMS-2 attempts to be compatible with CS-1 so that upward compatibility...and it isn't 100%. It's going to be a lot of work converting CS-1 programs to CMS-2 programs, but there is that attempt, and it's going to cut down the total cost. To go to another language would be impossible for NTDS.

Bersoff: OK, now let's say that within five to ten years Capt. Engelbach wants to buy an AADC for his system, and his people learn SPL, or perhaps NASA decides to buy one for some launch vehicle, and they know CLASP. Bruce, what do you think the problems are?

Engelbach: Well, I personally believe the problem is not one of constraining a language to be upwardly compatible, because, as we mentioned here, just the secretarial problems of attempting to keep one language document

consistent with the other isn't meaningful. I think the real problem which the Navy should recognize, and that the Air Force is also guilty of not recognizing, is that they don't want upward compatibility. They simply want a way in which they can economically utilize the programs they currently have now. Allowing for growth within the system, while the system contains the computer and all the software. So, as we're presently thinking of it, the upward compatibility, re-useability of the software is more of one compiler...say it's the newest one...in some manner or other being able to swallow the old program. Now, this does two things. It means you can, in some fashion, use your old programs until that point in time when their utility goes below some threshold and you can say, well, let's throw them away. And, at the same time, you can use that same compiler, or compiling system to create new programs to take care of the increasing requirements that the system has to meet. And, at the same time, you're not constraining, or building in inconsistencies...roadblocks into the new language. The new language is the one you're going to live with from now for ten years, whereas this compatibility problem is going to be wiped out in a few years because the old programs, the old systems, are no longer with us.

Bersoff: If you do your programming in SPL, and you'd like them to run on the AADC at some time.

Engelbach: I think...there's no reason in the world we couldn't build an SPL compiler to function within the hardware of AADC, whatever that hardware might be at some point in time. And if you had a lot of software that was applicable for my mission, written in CMS-2, I don't see any reason why I couldn't use both...the old software in the AADC hardware, and use SPL to construct any new software I need. Admittedly, it's not the cleanest way...we'd like to say one or the other, but it's the most economical way I can think of right now.

Floor: I'd like to ask Mr. Nimensky a question. Since he's developed his Meta Compiler, does this not remove the compatibility problem? Shouldn't it be easy to build a subset translater that would be able to handle your problems?

Nimensky: Well...it depends upon what assumptions you're working under. If we're talking about the Meta Compiler technique, and for one where it's relatively inexpensive to build compilers, then it's more suitable to design a language very definitely tailored to your problem, rather than use some general purpose language which really is all things to all people, but (nothing) to anybody in particular. It doesn't solve everybody's problem the way it should. If we can get compilers to be built fast and cheaply, then it's better to design your own language, because you know your problem best. It's best to have a language you can think in. If you can write your programs in the same language that you think about them, that's what I call an ideal programming language.

Bersoff: I can't agree with that, because somebody else will think about the same problem in a different way and want a different programming language. The idea is that the syntax should be the same for everybody

using the programming language so that there wouldn't be this variation of trying to read somebody else's program.

Nimensky: I didn't mean thinking about the problem. I'm talking about my problem oriented language for compilers, where I'm just thinking about the syntax. This is a language designed for thinking about syntax. I'm not thinking about all the other problems that a general purpose language (addresses). My generator language allows me to think just about the problem of producing code. This is what I meant by a language tailored to solve my particular problem.

Bersoff: Somebodyelse building a compiler might have a different syntax language and a different code generating language.

Nimensky: That's fine.

Bersoff: Except when he goes to work for SDC.

Nimensky: There's still no problem.

Bersoff: But he has to convert what he knows to the way you do things.

Nimensky: Take a for-instance. Number one, we talked about CMS-2/CS-1 compatibility. Now, any language which is larger...if JOVIAL is larger than FORTRAN, I can write a translator with my Meta Compiler which will translate a FORTRAN program into JOVIAL. If CMS-2 is bigger than CS-1, I can write a translator which will translate all the CS-1 programs into CMS-2. Conversely, I can take all my JOVIAL programs and map them into SPL. So, the clear question is, should I take JOVIAL and bastardize it so that it becomes difficult to add new features, or should I write a much more powerful language, and have a one time translation into my more powerful language, and from then on use it, or should I go along saying, "be standard, and everybody has to write in this language" and continue making the language worse and worse and worse. The more things you add on to a bad language...if your language has problems, it's very difficult.

Bersoff: In the discussion of CMS-2, I noticed the statement DIRECT, followed by $, followed by direct code. Couldn't there be in CMS-2 a statement, "SPL $", which would call in the SPL compiler, which would be part of the whole system, so that someone writing in SPL could use CMS-2?

O'Brien: That's very possible.

Cerf: Personally, I think that the right way to do (it), if you're going to start out designing a new language, which is what (has) happened for two or three iterations, anyway, that the design should be very carefully done. SPL is, as near as I can tell, pretty careful in the definition of the syntax of the language, so that it is extensible. I fully agree that if you start with a bad language it gets worse as you add more verbs to the language, but one of the strongest reasons for starting with a new one at this point is that most of the old languages that are around have very poor syntax. They can't be expanded very well; FORTRAN is a notable

example. PL-1 is rather a large language. It's syntax can be modified. It's been carefully examined. If anyone has seen the IBM Vienna documents, which are about that (several inches) thick, they describe the semantics of the language. But that's too much capability, I think, for the kind of programming we're going to need to do. So, the strongest thing to do, is start with a clean language which has a well defined syntax that's easily modifiable, and continue to live with (it). The AADC concept is supposed to span something like ten years, and, as nearly as I can tell, the... range of architectures that can be built with the modular AADC units is not going to be very different. The range is all in terms of capability and not in terms of structure. The instruction sets are pretty much the same for all the various spectra. So, one language should suffice.

Shirley: I think you mentioned really what part of the key point of this is--(it was) the word semantics. I'm kind of surprised that I hadn't heard it before, because the syntax structure only tells you how to follow a form to write something down so that something else can decipher what you wrote down, but the real key to any language, of course, is what it means when you write down a particular structure, or the syntax. That great fact is one, that I say, prevents, to the greatest extent, the kind of thing that was talked about: of being able to provide SPL to CLASP translators, or JOVIAL to SPL...that one may not be as difficult. In fact, there's a problem within the same languages themselves. To try and give you one very simple example...I'm the JOVIAL language, for instance, there is the case that you have both an integer...in fixed point language you may have an integer item which is operated on within fixed point hardware, and you can have an A-type item, which is really a mixed value, and what it means is, I want to add an integer item to a mixed item. Turns out it has never been clearly specified. Various compiler writers have written different things. In one case, people have chosen it to mean the result should be in integer and in another case they have chosen it to mean the result should be a mixed quantity. So you can have the example that you can add 3 and 1.5 and in one case you get the answer 4 and in the other case you get the answer 4.5. That kind of thing actually exists throughout almost any programming language. There's a tremendous area of variability that you can have in the semantics for whatever structures you choose to put into the language. And you'll always create a tremendous problem in compatibility, if compatibility means, "How do I translate programs from one language system to another language system?"

Bersoff: Maybe there's a way to get around compatibility by asking: Is there a need to program AADC in a high level language, given that its instruction set is so rich, or, on the other side, why can't the AADC handle high level language statements?

Floor: To start off with a personal opinion, if the machine is going to be as complicated as it is, and it's going to have the Synthesizer, I don't see any good reason to write in assembly languages. All you're doing is defeating the purpose of the Synthesizer, and defeating the purpose of having the richness of the machine by having John Doe off the street attempt to program it.

Bersoff: If you have an algorithm library you can program all your tasks in assembly language separately, and have the Synthesizer put it together. You don't, necessarily, need the compiler as part of the synthesizer.

Floor: Most of the people who program the machine aren't going to be PhD's.

Bersoff: I don't know how to program, if you're refering to me.

Floor: Not really. I'm just saying it's going to take a very intelligent person to be able to program efficiently (for) a machine with the kind of architecture and the kind of capabilities this machine may have.

Nimensky: You have a good example (in) the IBM 360. You have a bunch of users who are used to the 1401 computer, which is a very easy machine to program. It has a very small instruction set and, in business applications, has worked wonderously for years. Then, all of a sudden, they come out with the 360, and those programmers just cannot hack that machine. They had to go to COBOL because they could not understand the instruction (set). (The 360) has the richest instruction set of just about any machine. It's just so complicated, that the average programmer cannot code in that machine language.

Floor (Eva Lee): This brings up one of my pet peeves...we want to have the best engineer on the electrical system, the best engineer on the propulsion system and one (uses intelligence) in building the computer and then we turn around and say we can't have intelligent people programming it. I think that one gets in the field of avionics, and when putting a computer in a plane or spaceship or what have you, where there's a man in the loop involved, it's every bit as important that the person who is programming that (computer) have some scientific and avionics background as the man who builds the propulsion system and all the rest of them. I believe this is one of the things that contributes to the cost of avionic software. We still have this idea of building this million dollar machine and hiring the guy from behind the counter of the grocery store to program it. This is a fallacy, because you can take a few well-trained people and come up with something much better than you can by taking a roomful of coders, 1401 types or whatever you want to call them. They don't belong programming an avionics computer. They don't fit the picture.

O'Brien: I agree with you 100%, but in defense of programmers, let me say that they don't all come from behind the grocery counter. I think it's been demonstrated, not only in military systems, but commercially that given a good programmer, one of the best people you can possibly find, he'll be more productive using a good compiler than he will be if he has to use assembly language.

Lee: He may be a good programmer, but you still can't take a History or English major with a degree and put him into avionics programming. You cannot program an avionics computer without knowing something about the engineering systems... at least you cannot do it economically.

Cerf: Nobody's going to argue about that; the point is, what language do you program in? You certainly get a lot more done if you program in a high level language. The machine doesn't make the kind of stupid, little mistakes that are easy to make in assembly language, just because you don't have to keep track of 40,000 lines of code that you would in assembly language. (As) for the understanding of program...Mr. Nimensky brought that up... a 500 statement compiler vs a 35,000 instruction compiler (is) so much easier to understand. And I can regenerate that compiler if there are errors in it in a lot less time. So, the bright guy who knows about avionics wins all the way if he gets to work in a high level language.

Nimensky: Don't knock English Literature...that's my major...when you get down to it, programming is language.

Shirley: I don't know whether we all want to sit up here and try to sell this point that has already been sold...the thing I'd be interested in is a show of hands of how many people in the audience think we should only have an assembler for machine language. Maybe we'll find out there's no point in discussing it any further.

Bersoff: Is there anyone who strongly feels we should program in assembly language only? That you can solve the airborne problem without assembly language.

Floor: Right now? No.

Bersoff: I'd like to hear a reason why.

Floor (Alan Deerfield): I think you can solve it without an assembly language tomorrow. Today, with the state that your languages are in, you cannot solve it (with resorting to assembly language programming).

Bersoff: I guess we'd all agree to that. Now let's assume that SPL, CLASP, and CMS-2 all have working compilers, is there reason to assume one language would do the job better?

Floor: What problems do you envision using their languages for? We've heard a wide spectrum of languages. For example, I'd like to ask Capt. Engelbach what he thinks. He's the guy who's going to suffer with these languages.

Engelbach: I'd like to dispute your statement and, I guess, the general concensus of the audience, in that every day, literally every day, we do the aerospace problem...this is producing launch tapes for the TITAN III vehicle and for Atlas vehicles. We do it in FORTRAN. Now,...the problem is producing real time control and calculations to gain information necessary to provide that real time control. We do it in assembly language or in machine language on the 1824, which is a ridiculous, antiquated, little machine. But we also do the identical problem on the (CDC) 6600 in FORTRAN IV. Now, I ask you...I can solve the problem if given the right tool, and I tell you, if I have a 6600, I can do it in FORTRAN IV.

Bersoff: Do you respond to hardware interrupts on the 6600, or software interrupts?

Engelbach: I can do both. The reason I say I can do both is because I have a bastardized "chipawah" operating system on the 6600. You didn't restrict me to standard commercial software. The question here, and more germain, is given a class of aerospace computers...pair that with a higher order language... can I still solve the problem? Not only as most people in this room wanted it to be solved in a nice, very sophisticated engineering manner, but can I also do it within the cost and the dollars; and I say, yes I can.

Bersoff: You said a good high level language, but you didn't say CLASP, or SPL or CMS-2.

Engelbach: SPL has five implementation subsets, and the smallest implementation subset is CLASP. If we want to have a language argument we can get down and discuss whether it's ".S" or "S.". Okay, so there are trivial problems in the compatibility between CLASP and SPL. So, I say, either of those two languages. There is also an unidentified IBM subset of PL-1. It's called PL-1/M for militarized. Taking that subject, and doing the augmentation that that subset specifies and you can use PL-1. I'm not saying there's anything unique about the language. And I dare make very clear as to the language I'm talking about. There's no difference. I can use any of those... when you add on to that, the economic factors which I'm sure my associates in the Navy are more aware of than most of us, you not only, when you talk about language, have to say, "Can I get a compiler for it? Can I get an operating system that can match the compiler output? Can I live with it?" Then I have to say, there aren't many languages that can do it. PL-1 cannot. FORTRAN cannot. SPL can.

Floor: Do these things run on the machines you already have? In general, I think the thing that you're overlooking is that we're all taxpayers...we're all writing different compilers to run on different machines, presumably with flexible rear ends that can feed out to any computer, and yet, we're duplicating this cost over and over and over. There's the basic reason for compatibility. (Let's) decide what it is we want, what machines it must run on for you to be able to use it so that everybody doesn't have to go out and buy a new machine, or if you're going to get a new machine, try to look around and see that everyone is going to get something similar.

Bersoff: Who's going to be able to enforce the compatibility between the languages?

Floor: I think we're talking U. S. Government money, in general.

Bersoff: Exactly, and yet it seems that the Government hasn't forced compatibility on the contractor. That is...CMS-2 was started around 1967, about the time SPL was begun, about a year before CLASP was begun. Now, why?

Floor: Because of other economic factors. There are a whole range of them. Most of them are economic. I really think that very few of them are the kinds of things that you're discussing. The letter by letter, bit by bit compatibility.

Engelbach: The problems I face on a management level on compatibility have been made by two things. I don't want to point fingers at individuals because most individuals are very cooperative. It's been on a level of organization that says, "It wasn't built here. I have no confidence in it. So, don't bother me with it." The second half of this question is, if I acknowledge that somebody else has a good step forward in solving the problem, then I stand a very high probability of losing my next year's funding. In which case, not only does my major effort, which this language might be, but all my secondary and smaller efforts go down the tubes with it. What I'd rather do, either by default, or acknowledgement of the situation...is duplicate the major effort to preserve my secondary efforts. I don't want to connote minor...secondary. For example, in the Air Force today, we have a hardware project which was to build a prototype...a breadboard....of the next generation hardware. Well, they said, without examining it on the level necessary to who the differences and advantages of it, "This appears to be a duplication of a NASA effort." Therefore, zero funds...it's lamentable. The major part of that program was not a duplication. It was a supplementary effort, and, at some point in time, the two efforts could have been merged to come up with a system far better than we have now. I think these are two reasons why we in the Air Force, NASA, and Navy have not been able to get together and say, "Let's go with one." A few exceptions: COBOL, and that was a dictate of high enough up. And, NASA, with Ed Bersoff and I, on a personal relationship, were able for a period of time...have a very decent commonality between the NASA language and the Air Force language. My personal opinion is, that if I knew enough people, that could personally go out and talk to then on a first name basis, I feel we could get a heck of a lot further than we are now.

Bersoff: A large part of the difficulty is finding your counterpart in the various Government agencies who are doing something similar to what you're doing...to know who to work with is a major problem.

Nimensky: Let's look at the hardware, though. How many computers does the Navy have? How many different types of computers does the Navy have? All computers are basically the same. They all do the same thing. I can probably solve any problem on any one of them. So we say, if someone comes out with a new supercomputer, if you want to have it, you'll buy it. If I come out with a new superlanguage, you should treat it the same way. If it's a better language, let's use it. You treat hardware that way, but you don't say, "Oh, he's got a new supercomputer, let's force him to be compatible with IBM and so on."

Floor (Bruce Wald): I'd like to ask Mr. O'Brien (this). I assume that many of the system monitoring programs you're writing now, you are writing in CMS-2.

O'Brien: Correct.

Wald: Do you permit your programmers to use direct code, and if so, why?

O'Brien: We do, if they have a need to use it.

Wald: So, in your organization, which should be the expert on higher level languages, you can't write the monitor without breaking into assembly language.

O'Brien: That's true in many cases.

Wald: Is that a defect in CMS-2, or is there something wrong with higher level languages in general.

O'Brien: I don't think it's either. If you want to do a given job, you have to pick the right tool to do it...one of the basic philosophies of CMS-2 is not to inhibit the programmer from getting at any of the features of the computer he's programming for. When we don't provide him the tool to address a feature of the hardware, he can always use...machine code. The capability is there and it's needed. (That's not to) say the higher level language has defects. We talk about apples and oranges...they're two different things.

Wald: ...which the computer is capable of doing, which would be awkward or impossible to express in your high level language.

O'Brien: This is true, and this is usually in the area of executive, interrupt processing, that type of thing.

Floor: Isn't that exactly what we're talking about, though, in avionics and aerospace (applications). If you will, you identified the problem, and although I don't want to come on any stronger, you just said high level languages can't (do the job).

O'Brien: You take any large system. You have an executive, real time interfaces, interrupt processing, but the bulk of your programming can be done in a higher order language...(it is) a very effective tool...the executive and I/O drivers are particular to the hardware anf for those particular applications you're going to have to get into the hardware and use hardware instructions...that may be 2% of your entire problem.

Floor: In any case, you've just answered the question why in some cases you really have to resort to assembly language.

O'Brien: Certainly.

Floor: If all hardware were the same, we would never have to go out of high level language. We could write some macros to handle that one interface with that one piece of hardware. But, since they're all different, you have to go to the hardware level, which means the bit level, which means the machine code or assembly level.

Floor: The implication is that you can't write the real time controls in higher level languages.

Bersoff: That's not true.

Floor (McGonagle): You've got to. We have a living example in the B5500, where the whole thing is written in a subset of ALGOL. But, there is a strong point made here that you have to go to the hardware. The transition from the language to the hardware is made for code generators. It is not necessarily made by escaping to machine language. We have had the same

problem within the Company (Burroughs) of stopping people from writing in Polish notation. It's not an easy thing to do, but it can be done, and you can stop it. Now, note when we went from the 5500 to the 6500, we extended the language of ALGOL to take care of those problems. We made the mistake in the 5500 of providing an escape to machine language. We'll never do it again.

Floor: The hardware I'm talking about is not the computer hardware, it's the peripherals, the interface between the machine and the peripherals. The peripherals are all different. You need different disk packs to get certain control software.

Cerf: There's no problem. You can generate the bit patterns. That should not be a problem at all.

McGonagle: You can generate bit patterns. I'm generating micro-logic control to a printer with no electronic interface...and a disk pack control. It can be done.

Cerf: Was it very hard to do?

McGonagle: It took about a year to design the language to the point where we can now design an interface...

Wald: He's not from a charitable organization. It must be cheaper in the long run or he wouldn't be doing it.

McGonagle: And it's a small company that can't afford large masses of people.

Floor: Take the biggest reason why most people, based on looking at CMS-2, I can't see any reason for dropping down to the bit level on first glance. The only reason I can see why they would do it is from habit. I've been theoretical programming for CS-1 for thirteen months now; I'm writing my first program in high level language in CMS-2 because I found it gives me the capability to do something in a higher level language that I had to do in assembly code before.

Nimensky: You do have things like there are certain instuctions that are nice on the 360 and that you could put in a particular high level language and it's more equipment to do, say, a conversion or some test and compare with that machine instruction...if you're very interested in getting the one instruction to do a job, that a reason for using machine code. Like, on the 1824, I think, through our code generator, by using SPL they can get an instruction in the machine. There is absolutely no reason to direct code. In fact, we have a hardware operant feature...but we don't even use it because there's no need to address the accumulators or anything else.

Bersoff: I've been asked to ask the language people if any of the languages we're talking about can provide the capabilities required for all the possible AADC configurations? I would assume you're asking about the multiprocessor, in particular.

Nimensky: I think SPL has all the potentials. For parallel processing, it has a DO Statement for multiprocessing. You can segment your program. It has all the ability to get at any machine-type instruction (such as) we've been talking about, without having to go to direct code. It has hardware operants, so if any special machine features are used, the programmer can directly address them.

Bersoff: How would CMS-2 handle parallel operations? Would it be the compiler's job to recognize parallelism?

O'Brien: Right now, the existing CMS-2 will not, but we're in the process of writing CMS-2 for a contract with Univac that will operate on the AN/UYK 7 for the UYK 7. That will be a multiprocessor machine. We will be including capabilities in the language to handle that problem. Though, I don't see any problem for the AADC, once we know exactly what the AADC is and what's desired, we can write the system to handle the problem.

Floor: Are you talking about multiple, independent data streams, or a parallel data stream?

Bersoff: The AADC is organized in a lot of different ways. There is a parallel processor...the Matrix-Parallel Processor...then there is the multiprocessor portion, which might take a long program, segment it, and operate on the two sections in parallel. And, then you might operate on two different programs in two processors, all that could be going on at the same time. So, one of the jobs of the compiler might be to break up a large program into parallel parts, so that they can be operated on together.

McGonagle: You'll have to re-do that whole conception of scope. We did a study for Fort Monmouth on the detection of parallelism in program structure... there's a great deal of difficulty in this...everyone points to the DO Loop, (NOISE) it's not easy to detect. In fact, it's a very small percentage of your parallelism.

Bersoff: Shouldn't the programmer be able to recognize parallelism in his program.

McGonagle: What we did in the 825 was to allow the programmer to create the parallel path.

Bersoff: That's what SPL has with their "Parallel" and "Join" operations.

McGonagle: We had a problem with it, in that we found most programmers tended not to use it. The difficulty is not in using the parallel path, but in creating the monitor functions which adequately describe the relationships between parallel processors. Apparently, there's very little research being done anywhere in the country on parallel process structure.

Cerf: Just to answer that question, UCLA is doing a lot of work in that particular problem. There have been studies for the last six years describing models of programs...trying to describe what goes on when problems are running in parallel. Two aims of the studies have been to take sequential programming

languages and extract from them their parallel paths. That has only been partially successful because a lot of the parallelism is hidden by the nature of the language. It's implicit that if you don't say otherwise that the next statement is the one to be executed. Partly because of that, and partly because of data dependencies that are hidden you don't get all of the parallelism out through an analysis of sequential program. So, although we worked with FORTRAN, and Burroughs has worked in ALGOL, and some people are trying to work in PL-1 to create graphs or models of these parallel programs, that hasn't been very successful. So, we're in the middle of trying to design a parallel processing language, and the main element that is in that language that is missing from all others that we know about is that the programmer specifies three different things about his program, rather than two. In the current sequential languages he describes data structures and he describes computations. You get a little bit on control over how things get executed through the use of GO TO Statements and such things as that. In the case of a parallel processor language, you have to specify a third thing, which is the structure of the graph...or the order and dependencies of the various tasks to be executed. So, a third sublanguage has been added to something like FORTRAN, although we don't insist that computations be described in FORTRAN. The third language is a control sublanguage and the programmer actually tells us which things can be done in parallel and which things are dependent and must be done sequentially. Taking that information...the information about all the space that's needed by each task and so on, it's possible to, apriori, decide how to schedule that particular system on a multiprocessor system.

Floor: In kind of large systems, this raises another problem in that you tend to look at a small component of that large system at any point in time. When compiling a program, and the program is going to go into a library, and using pieces out of that library, and we don't always have enough description of the pieces that we're pulling. It's more than just a process that we're writing right now, it's that process as it fits into the total structure of the total system. It's that process plus a hundred or two hundred others. And the information on all of those must be known in order to determine the order of scheduling. It's not a simple thing.

Nimensky: Besides the do and join, you need the lock and unlock to protect the memory.

Bersoff: I think the conclusion we can reach initially is that, from a management point of view, I don't think there's any question that CMS-2 should be the language used to program the AADC. The question then becomes what should be done to change it? To modify CMS-2...to increase the capability we talked about...uh, let me hear your objection before we go on.

Cerf: Well, I don't think CMS-2 does the job completely because it seems to leave out all the information about handling interrupts and so on...enable, disable and a few other things seem to be needed in CMS-2.

Bersoff: Exactly. That's what I was driving at.

Cerf: Second, you need to have considerable more control over the scheduling of tasks if we're going to write programs which are suitable for a multiprocessor configuration. I don t necessarily advocate a whole control sublanguage, although that happens to be my opinion at the moment. So, I think that's missing. And, finally, I think we need to be able to write something like an operating system in the language, whatever it is, because if we want to have re-structurable computers, things that degrade nicely and understand what's going on about them, we re going to need to write some sort of operating system which runs along with this little program which controls firing and radar and all this sort of stuff.

Shirley: I'd like to go back to the point Capt. Engelbach made before. If you want to talk about the technical facility of the language, we have identified that there are a substantial number of languages which you can beef up to provide the kind of technical facility that has just been mentioned. And so the point of the question, "what particular language it should be?" then has to turn on the points made before, which are transferrability of personalities, ease of training of personnel, availability of compilers, what the costs of developing these new compilers possibly are? Those are the questions it turns on and not really the technical facility that exists in the language today.

Bersoff: So, it gets back to not being a technical question as to which language to use for the AADC, but a management question. Now, if my assumption's right that CMS-2 will be used in some form, what does that do to language compatibility? We're back where we began, I guess.

Engelbach: If CMS-2 were used, it would be the one that we're talking about today, it would be a beefed up one; so you might as well call it CMS-2 Prime. Your compatibility with CMS-2 as it stands today is zero. I should say it is whatever you would have with any other language. They're procedure oriented languages so that anybody who is familiar to a certain extent with JOVIAL, PL-1, ALGOL, SPL, they'll have the same familiarity of going from that background into CMS-2'. If you built CMS-2', you would be starting from the position NASA was in a year ago and the Air Force was in two and a half years ago, of constructing all of the system generating software. The things that prepare you for building the operational systems, the compilers, the intelligence, the people who built these compilers. For instance, Logicon has built a CLASP compiler for NASA. Okay, so you have right there within the community of software people a group that understands some of the problem, not all, but some of the problems inherent in building CLASP compilers. You also have one example to play with. The same way with SPL--you already have several examples. You have a body of people who have been through the mill. That's what you lose if you choose CMS-2'. What you gain, regrettably I have to admit that it's a pretty large gain, complete immunity from the upper echelons. You don't have to fight headquarters because you're choosing SPL. That's an Air Force language and that's not the thing you do if you're with NASA or when you're in the Navy, or in the Army, or in the Marine Corps, or anyplace else. You take CMS-2 because that's what everybody upstairs knows about. If you want to take something else, then you've got a massive education problem. I speak from brutal experience, because I did it.

Floor (Capt. R. Dunning): I don't think I quite agree with you in the Navy. I feel that in my position at the Programming Center (FCPCPAC), I could probably sell any compiler to the Navy, because my responsibility is compilers for, at least, a portion of the Navy. To my bosses I could sell that. Which leads to my next question, do you own SPLIT and can you give it to us?

Engelbach: On January 15, I can give it to you. I own it...what I mean is that the Air Force has contracted for a deliverable item entitled SPLIT, along with four code generators.

Dunning: You're about the political aspects of it. There would be some problem in selling...but, there isn't the problem with the Navy as there appears to be with the Air Force.

Engelbach: You're very much more fortunate than we are...let our hair down here a little. If you wanted to take the technology of building compilers, one example would be the SPLIT. There are several others. If you wanted to just pick it up, and if we could all vote on a new name, just give any arbitrary name that is not connected with anything else, we'd take it. Now, if that would circumvent some of the political problems that I have, that you have, that other people may have, then I say fine. I'm not tied to the name.

Bersoff: It turns out, in fact, that the three languages are pretty nearly identical to begin with.

Engelbach: They all have a common ancestor. ALGOL 58.

Bersoff: Right now we call SPL and CLASP SPL/CLASP. Maybe we could call (them) SPL/CLASP/CMS-2?

Floor: You know, this might be a real step forward, to just call them SPL 1, 2, 3, 4, and so forth.

Engelbach: Maybe we should be like IBM and take out a trade mark on all of PL-1, 1 to 100...that way, nobody else can call their language PL-1 or PL-2 or PL-3, all the way to 100.

Floor: Certainly if you took your Meta Compiler and constructed a language of whatever you wanted and called it CMS-2', it would make it easier to sell than if you called it the next language, or whatever you wanted to call it.

Bersoff: But, there are three different names now, so which one do you choose?

Nimensky: A rose is a rose...if you call SPL CMS-2', it's still SPL and it's still CMS-2'.

Engelbach: And, if you could get one document, and, if the Navy published it, let them put their title on it, and all we'd do for the Air Force is take your title page off and put our title page on and then we would at least

circumvent the problem Ed and I have been buried up to our eyeballs in, (of) "Okay, friends, is it a typographical or is it a printing error, or did we really mean something when we said, dot S or S dot?" So, once there are masters, and everyone puts on his cover sheet...

Floor: There is a real reason for this, of course, and that is...program maintenance. You get several different compilers (and) you do have some trouble with your program maintenance. Provided this is reduced, however, using the Meta Compiler technique...may I hear some comment on that.

Engelbach: Well, you derive using a production tool, it's just like automating and producing hardware...uniformity of products. I didn't say standardization. Uniformity across products and from product to product, which is not achievable using hand tooling methods...If we use the same tooling, then the uniformity between your product and mine is much higher than if we arbitrarily gave a contract to SDC and said, "build us a compiler," or (gave) another contract to CSC and said, "build us a compiler,"...Hand tooling can be used for variations in the final product.

Bersoff: I think what you need is a syntax analyzer that will recognize a set of syntax and produce code for any machine from there on.

Nimensky: Talking about maintenance, in the last two years we've had one individual on the syntax analyzer; and that's for correcting errors and bringing it from what we call Mark II up to Mark IV. We're constantly increasing it, correcting errors, and there's only one man who has made all the corrections and maintained the syntax analyzer. I say...one individual could maintain three or four code generators for different machines.

O'Brien: I would say that that's the experience we've had, too, in CMS-2.

Cerf: What happens if you have to move the Meta Compiler to another machine?

Nimensky: When we mentioned SPLIT...that is, by definition, what SPLIT is. SPLIT is making the implementation tool of SPL machine independent and transferrable. Our first step is to move it from the 360 to the 6600 and we will be able to operate on any machine we write a code generator for. We built an SPL code generator for the 1108.

Cerf: Is the Meta Compiler an interpretive system, or does it always compile?... You're redoing the code generator when you make this move?

Nimensky: We write a code generator to run on a 360 host machine, on which we can write a code generator for any machine using our Meta tool.

Cerf: What's the advantage of that over the conventional bootstrapping technique of moving it through a higher level language?

Engelbach: You're actually moving the tool that constructs the compilers, as well as the compiler itself. You're not bootstrapping the compiler, you're bootstrapping the Meta Compiler.

Shirley: You made a point which I think is very important to all of us, which was that the chances the compiler comes out somewhat different in this situation has been proved. In other words, it is not as likely to come up different as it has in the past...Eventhough you're using the same tools, it's not clear if you're going to really be right...that you're still not going to get a different compiler at that point.

Engelbach: Let me put it this way, if you had a code generator for a given machine, and one section of the Government, me, I will have a code generator for the 6600 Scope 3.2. If the Navy had been able to reconcile itself to using the Meta tool, then I would be very surprised if they just didn't take my code generator and use it rather than duplicating the code generator. It exists. Just plain use it...In that sense, not only would we have uniformity, it would be identical...The uniformity of the tool not only helps us in this case, but if they use the tool and build a code generator for the ANIUYK 7 or the AADC, and if I should pick up the hardware, either of those machines, I would of course pick up their code generator because, of course, I'm not going to do it. The uniformity I gain is, if we're both using the same tool, the front ends are going to be the same. The syntax analyzers are going to be the same. The semantics that you brought up earlier are going to have a high probability of being more similar because I now can constrain not only the interpretation of the semantics of SPL or CMS-2' into a binary form for a particular piece of hardware, but I also have a framework in which to perform that translation process. That framework would be the code generator language of the SPLIT tool...if you build a code generator for the 6600, when you are building a code generator for a machine of a very similar nature, the 6400, the 6000 Series, the similarity not only in the syntax, the front end, but the semantics, the part the code generator produces, will be...let's say, there would be a higher probability of their being similar. The interpretations would be made (in) the same (manner), enhancing, in the long run, the problem of portability.

Bersoff: I think we all agree that we should have compatibility. I think we ought to go back and do it. Thank you.

THIS PAGE INTENTIONALLY LEFT BLANK

Panel #2

Subject: Computer/Compiler Standardization

Chairman: Dr. Bruce Wald
Naval Research Laboratory

Panelists: Mr. Alan Deerfield
Raytheon Corporation

Mr. Vi Henderson
Logicon Corporation

Mr. J. David McGonagle
Burroughs Corporation

Mr. Robert Samtmann
Naval Air Development Center

Introduction

Entner: What can be done to standardize on elements of a computer to, in turn, standardize on a compiler's object code generator? This brings us to a point which was raised earlier today -- in the case of CMS-2, a unique object code generator is required for each computer to be serviced. What can be done to reduce this requirement for uniqueness? This, in a sense, is the subject of our next panel session on Computer/ Compiler Interface Requirements.

Wald: The title of this session has changed from "Standardization" to "Requirements." I don't know what to read into that... I'm going to deviate from the procedure of the last panel. I'm going to ask each of the members, in turn, after I make my opening remarks to state their name and qualifications. The reason for that is so they can include or hide the fact that they are English majors...Seriously, though, these computer languages are languages, and I would suggest that you might want to consider the aesthetic component. If the listings are ugly to look at, there is probably something wrong with the language...Now, I'll start by stating three disqualifications of myself to be on this panel. The first is that I am not an avionics expert. I wrote my first program in absolute hexadecimal in 1954. The only higher level language I've used extensively is NELIAC, which is another dialect of ALGOL 58. The programmers who worked under my direction always escaped into machine language when they had tricky things to do. My second disqualification is that I am not Jim Ward. My name is Bruce Wald. Jim, unfortunately, had a conflict this afternoon, although he left me some material, which I will quote. The third disqualification is like they say on "Meet the Press," I am going to try to bring out a story. I am going to try to focus debate or start some arguments, so the prejudices I'm about to state are not necessarily those of CBS, the Department of the Navy or even myself. So, let me state my prejudices, then I'll retire and let the members of the panel interact...The first one is that I'm rather disappointed. We seem to have lost sight of the reason for this conference, and the problems that were stated in the invitations to the conference. To put it crudely, all of us in this industry have been guilty of playing a game ...a game that we might entitle, "Proliferation for Fun and Profit," or "Job Security for High Priests of Software." I won't comment on the profit part since balance sheets have been going up and down, but DOD is finding this ride less and less fun. In 1966, and these figures were collected by Jim Ward by a methodology which understates the problem, a hundred and three different models of computers were delivered to DOD

as components in weapon systems. I am not counting the commercial computers and I am not counting the computers delivered to the "Spooks." In 1967, one hundred and thirteen different models were delivered. Of course, there's some overlays, but in both those years there are over a hundred different models of computers being delivered, and each of them had some support software -- at least an assembler and a simulator. Well, if we can't do something about this proliferation problem, we are going to get standardization by fiat. It may be from the Department of Defense, it may be from the Bureau of the Budget, it may be from the Government Accounting Office, but those people who pay our bills are getting very tired of taking a ride. Now, the real problem, as far as the financial aspect, is that this software, which is so expensive to produce, is not reuseable. In 1960, Maurice Halstead published a book entitled, "Machine Independent Computer Software." It was sort of a do-it-yourself manual on how to build a NELIAC compiler. For those of you who are not familiar with NELIAC compilers, it's a dialect of ALGOL 58, transition table technique used; self-compilation time about 30 seconds, compile and go -- not the most elegant code in the world, but you can make them better. That was ten years ago, but I think we'll agree that we don't have machine independence yet. We can't take an applications program written on a 7090 and move it to a 6600. We have trouble taking an applications program written on a USQ-17 and moving it to a USQ-20, and I'm sure we won't be able to move it to a UYK-7. I suspect that there are two identical USQ-20's on the same ship, programmed by two different organizations reporting to the same Admiral or, at least, to the same CNO, and an applications program can reside in one of those computers and can't reside in another, identical USQ-20. A couple of weeks ago we had occasion to get a metacompiler operating on our local machine; and this metacompiler, partially self-compiling, was running on a 3600 and we had to move it to (another) 3600. That job took two or three unnecessary weeks because in one case the operating system used tape for logical peripherals, and in the other case, it used drum for logical peripherals ... I don't think I was too happy with the language presentations I heard today and yesterday. You know, syntax crackers cost $1.98, or that's what they tell us. If they only cost $1.98, we ought to be running some controlled scientific experiments about what makes a good language for avionics for real time applications. I haven't seen the scientific experiment to tell me whether blanks should be ignored or blanks should be used as a delimiter. I haven't seen the experiment to say whether it's better to use LET and SET or use an equal sign. Apparently, people, because of history or personal opinion have just gone ahead and done it. I don't know whether it's better to describe a matrix

in something as concise as Iverson, or to spell it out in painful detail. Where is the evidence? We've heard about global and local optimization; now, I'd like you to focus on a different kind of global and local optimization, and that is Program Management. The fellow who is responsible for delivering a system has an optimization job to do. He wants to get the best system at the least cost within his time constraints. Nobody forces him to look at the global situation. He may solve, in an exemplary fashion, the problems of his particular system, but because of logistic incompatibilities with the rest of the Navy, or the other services, or those one hundred and sixty computer types, he's not done a good job of global optimization. Unless that man starts thinking globally, he's going to have standardization, by imperial fiat, forced down his throat ... We heard alot about metacompilers, and their good, automatic techniques for syntax cracking, but for code generators. Code generators are written by hand. Now, if there were a standard metalanguage in which one could describe the addressing structure and operation codes of the machine, and then the system would produce a compiler automatically, then we'd have a metacompiler. Right now, all we have are syntax analyzers ... Ron talked about a somewhat different metacompiler, or System Synthesizer Something which you would approach with a problem. Now, with what language would you approach Ron's Synthesizer. You certainly wouldn't approach it in CMS-2. You wouldn't approach it in SPL. Those languages describe a solution to a problem, not the problem itself. What about the poor gentleman who is trying to get Shortstop to work. How would he approach the problem? What language would he use to describe his problem to the System Synthesizer? Bear in mind that we are all servants, except for a few of us in the academic community, we don't exist for our own sake. The hardware people exist so that the hardware can solve problems. The software people exist so that people with military problems can use the hardware to solve problems. Finally, I'd like you to remember what Nimensky said about why compilers weren't used more. He gave three reasons. He said, first, the languages were not appropriate. Second, that the compilers were too expensive and, third, that what the compilers produced was too inefficient ... Now, there are some hardware people around who say that you software people have had ten years to deliver machine independent, transferable programs and you haven't done it and, maybe, now it's their turn. Suppose they deliver to you a FORTRAN machine. We would have to extend the language a little to explicitly declare the data types, to explicitly declare the semantics, but it would be a FORTRAN machine. You wouldn't know the addressing structure, you wouldn't know the internal word length, no compiler would be necessary, there would be no machine language; would this machine answer Nimensky's objections? Can we build it? If we can build it, what would

you system programmers do for a living? ... With that statement of prejudices, I will turn to the panel, ask them to introduce themselves, and speak for five minutes, if they wish, on their view of the problem, state their qualifications, and then maybe we'll get an argument going.

Samtmann: I'm Bob Samtmann from the Naval Air Development Center. I'm an engineer. I've been associated with the computer subsystem development for the F-111 over the past five years. While my work has been associated primarily with hardware development, I've picked up an interest in software by osmosis. Just recently I've been assigned to the F-14 program computer subsystem development. Also, recently, I've been getting active on the AADC program where I've been working with Systems Consultants Corporation on the instruction repertoire development. I will be taking an active lead in the compiler development -- whatever it will be ... I came to this meeting hoping I would gain a better insight into what path we should take ... not being too knowledgeable in the area of compilers ... I must admit, I really haven't gathered that much from this meeting. I've heard alot of discussion of whether we should go CMS-2 or some other way. Once again, it might be a matter of salesmanship, as was mentioned earlier, should we choose something other than CMS-2. I've had some real world experience in incompatibility between hardware and software. I could fill you in (on them) in about an hour ... There is definitely a problem, as everybody knows, in compatibility of languages, in compatibility of compilers running on host machines. Problems in efficiency ... Right now, we'll probably have to go back and do some additional thinking ... probe the problem some more ... Thats' about all I can say right now.

Henderson: I'm Vi Henderson of Logicon. I've been working in the aerospace field for approximately fifteen years; mainly with Air Force and NASA applications. I consider myself a software engineer. That is, I sympathize with the applications side... The first thing I would like to do is echo what Bruce has said, that we have lost sight, somewhat, of the whole problem. I've sat in language conferences a number of times and, typically, the one thing that is missing is attention to the application area. I've heard the comment that system programmers have become system programmers because they're expert programmers. I tend to think that they've become system programmers because they have forgotten, or have not addressed themselves to becoming expert in the application area. They haven't learned the guidance and control problem. They haven't learned the dynamics of a system, and that sort of thing. Therefore, they leave that field and work in an area they can handle technically. We've talked a great, and yet, it's ironical to me, the problem of coding

for an avionic system is relatively low cost, if we just talk about the coding itself. The major cost lie in getting to good specification and, second, taking the raw code and checking it out, testing it, validating it. We seem to continuously ignore those two sides of the problem. I consider them to be far more significant than a programming language problem. We need programming languages, but we should not forget that programming languages do not solve problems in themselves. They're only a means, a tool, an aid to solving the problem. Similarly, the compiler problem... I wish we could get rid of compilers. I heard Bruce, yesterday, say let's open up, take a new view, and take a language that can be executed directly by the machine. We continually look at the front end and the back end. Somehow or another, I think, to make progress, we have to look in between... this intermediate language which separates the code generator from the syntax analyzer. I believe we ought to get the engineer, the programmer, and everybody else together and try to define an intermediate language in which we can bring the computer and the source language together. That should be addressed, in my opinion... Ron raised the point about the synthesizer... No one has really addressed that. I know the Air Force is also looking at that problem and is undertaking an effort. There are two things to consider with respect to the synthesizer. The Air Force's side is more mission oriented and as we see the AADC development, we're talking about a synthesizer that is applied to a particular computer system, or a family of computers. We are not planning at this point, as I see it, designing a synthesizer to meet a particular mission requirement or set of missions. I believe that the synthesizer can only evolve... It will be a long, long time before it becomes an on-line tool. The development cycle has been, for software, a long tortuous road; particularly, the check-out side of it. For those applications, such as the Apollo... some very high cost system where you can't affort to buy the system to begin with... you get the tail-wagging-the-dog situation, where the amount of check out, redundancy, resources required to really test the software becomes enormous. We don't pay enough attention to that part of the problem, in building languages, building the application programs, and disciplining the whole management process.

Deerfield: I'm Alan Deerfield from the Raytheon Company. I'm a consulting scientist. I've been in the computer business for about twenty years. I'm a designer of computers. I'm not a software man. I think, to some extent, I understand software people. I'm very unhappy with you, just as I'm very unhappy with most computer designers. I suspect a software man would be unhappy with another type of compiler designer and other software people. I have the same kind of prejudices against my own types as I have against, I think, this group. I'd like to get a little antagonistic in a friendly way, so that you might feel a little freer

to throw one at me during the course of the conversations. I won't get angry. I probably won't be able to answer your questions either. I'd like to go on record with a couple of my views. Most of what Bruce Wald mentioned I agree with, although I'm sure if we got down to interpreting, our formal agreement would probably be different. Yes, I believe I can make a computer, for example, without very many problems, that would handle FORTRAN directly. I don't see any need for that type of computer. I don't see that it's a really difficult job, even. On the other side of the fence, though, I'm in favor of compilers. I'm not opposed to them. I think you people have concentrated on the wrong thing. I think syntax is necessary, but I could care less what type of syntax you use. Once you've defined your syntax and can handle it, I think that's great. The purpose of a language is not syntax; it's getting purpose and meaning across. For my own, I'd rather see a language that was strictly mathematical. As an engineer, as a person with mathematical problems, I'd be very happy if I could write my problems down and let the machine solve them. As far as JOVIAL, FORTRAN, or any of those things, I don't really have that kind of an opinion. Frankly, I did enjoy the comments on metacompilers, but principally, I enjoyed them because perhaps what follows the metacompiler I can eliminate. I like the idea of stopping at a metacompiler with a very small structure. But, I don't think you people should be spending your time on syntax. I think that it's necessary, but I don't think it's worth quibbling about. There are two other reasons why I think compilers are necessary in the military, or, at least, problems with them that I think should be attacked. The first is efficiency of the language. Efficiency, in my mind, comes down to how much memory do I need to carry with me -- and also speed. How fast does the product which you produce opérate in my computer?... If it takes me ten words of memory, for example, to express a subroutine in final machine language for something like a square root, and if I recognize that those ten words are always in the machine, then I'm paying ten words of physical hardware cost, which says I might compare that against a hardwired square root routine. There is a legitimate hardware, not software, tradeoff of your using ten words to program that square root for me. Even if you're as fast. Consequently, if I build a computer that has something like a sine or cosine in it, I trade my hardware cost against the hardware cost necessary for me to carry that sine routine in core, or in memory. I expect, for example, that if you have a compiler that has a sine or cosine or any other form of instruction that I build a computer to handle directly, that you ought to be able to modify your language so that you can accept my machine language instruction (in place of the software subroutine). Today, for example, I can take all your algebraic statements, and I can build a computer that will handle

your algebra directly. The thing I object to, then, is the fact that you spend an awfully lot of time developing algebraic handling. Let me just briefly elaborate. Suppose, for example, that (going step by step) I go from one accumulator to two accumulators. One reason for going to two accumulators is to eliminate unnecessary load and store orders. If I build such a machine, I expect the compiler designer to go in and modify his compiler to take advantage of those two accumulators and get rid of load and store orders that I don't need. When you do an analysis, and tell me that 50% load/store orders are used, and perhaps half of those are unnecessary, I'm carrying one heck of a lot of memory on-board I don't need, and that comes out in cost. If your compiler does not use my two accumulators, or my eight accumulators, or my sixteen accumulators, something is wrong. Also, if I can handle your algebra entirely, I expect you to drop it out of your compiler. I expect you to simply translate the algebraic statement into my machine language. I don't expect you to apply a method of saying, for example, I'm going to go into the innermost parenthesis and reorder my language. I like the machine to handle the algrbraic statement in the exact form the user wrote in. I'd like the printouts and the programs to be in a form he understands. The rearranged products of a compiler aren't very useful to me. I don't think they're very useful to the user. So I sit back and I say, one of the problems is that the best machine designers in this country never work with the best compiler designers in this country. I find that there isn't a company in the world that will really intimately, and I know they work together, but really, intimately take their software people, the compiler designers, for example, and have them, intimately, work with their designers. This is probably one of the worst things that's true about this business. Not just the separation, but even to the extent of getting you to work with us. It's almost impossible to get your interests up. And, it works both ways. It's almost impossible to get my hardware people interested in working with you. So, I sit back and I say this, every time I see the fact that you have methods for solving problems, and I don't care what those methods are, they become obsolete when I design a new machine. If I look at a machine, and the AADC, for example, is one... If I can't design a machine to be a 1975 - 1980 machine simply because your compilers won't use the features, I'm wasting my time. For you to settle upon any particular compiler right now, I think is nonsense. For you to base your entire existence upon extending things that were obsolete when you started, I think is to avoid the issue of why we're doing an AADC. Further than that, I guess you can explore my views as the discussion progresses, but basically I think I've gone on record as really saying, I don't think you're solving my user's problem at all. The user, to me, is the man who has the mission and not the programmer.

McGonagle: My name is Dave McGonagle, and I'm an english major, and I come from Burroughs Corporation, where hardware and software types exist on the same design team. The last two machines to come out of Burroughs have had, amazingly enough, a software man as a Program Manager. We do design machines with compilers in mind... As far as background is concerned, incidentally, I started in this business in 1951 at Wright Field, on CPC's, if there's anyone around who can remember them, and passed through the university computer environment as an applications programmer, and passed from there to Westinghouse with SOLOMON, and eventually to parallel processing and then wound up at Burroughs. I've known Bruce for a number of years. We've lived together, you might say, trying to get a multiprocessing operating system to work. Currently, I've just completed an assignment to design an avionic multiprocessor implementation in LSI, and we're now going to go ahead and build it. But, Mr. Nimensky said earlier about the transferability from the 360 to the SPECTRA... I'd like to remind you that the RCA people forgot something. You don't transfer software unless you also transfer the operating system interface. Therefore, 360 programs do not run on the SPECTRA 70. No one can guarantee OS 360 interface, unless they're willing to run OS 360... A problem raised here... the FORTRAN machine... A FORTRAN machine couldn't begin to solve the problem, because FORTRAN, itself, as a language, would not be satisfactory. We've built, in the last eight years, machines specifically designed for two different languages. The 5500, in which the machine language was reversed Polish, essentially the intermediate language put out by practically everybody's compiler. It was geared toward ALGOL. We've come out with a 3500, which is geared toward COBOL. I'm sure, if the Navy bought enough of them, we would come out with a different machine geared to CMS-2, or another one geared to SPL. They wouldn't be the same machine. They don't come out the same... Trouble with idiotic things, like COBOL's requirement that you don't store the result of an arithmetic operation until after you know there's no overflow. This causes you to design an adder that goes from left to right instead of right to left. Clever. Saves you a little time. Saves some memory space. Lots of little things like this happen. Reversed Polish... for that we implemented an automatic index register operation called a "stack." If someone asked me if the 5500 had an index register, I'd have to say, "Yes, it does, but it's dedicated in its use as a stack pointer." Do we do subscripting? Yes, we do, but we do subscripting from the top of the stack. Why? Because we looked at our compilers and found that 90% of the time we just finished computing the subscript at the time we wanted to use it. So the place where it already was was in the accumulator... Therefore, languages do impact on the machine design, and I think each language will get its own machine. Now, the time has come

when we can give you a different machine for each language. At this point I think we're talking about the same general idea. With the advent of LSI, the control logic for a machine will be stored in microprogram memory of one kind or another. Two things are going to happen: You're going to get the compatibility Bruce was leaning towards by emulating someone else's machine; in which you will emulate a machine completely, including the control mode instructions. You will also... emulate a COBOL machine in a ALGOL machine. You dynamically move from one machine description to another. Your machine for, essentially, direct execution of algorithmic statements, I expect, by the end of the year to have a demonstrable piece of hardware which is a direct execution APL machine. APL, very strongly, is an algorithmic language. It has some other features which interest me a great deal that I think begin to move in the directions I think hardware ought to be going. All we've done for twenty years is to reproduce the adding machine. We haven't done another stinking thing. If I go back and look at the CPC, its addressing structure was almost exactly the addressing structure of a 360. And, yet, many times we talk about arrays, we talk about lists, we talk about tree structures... We don't talk about addressing to a single cell in memory. Who, in their hardware, has built addressing algorithms for list structures. Who, in their hardware, has built addressing algorithms for trees. Yet those of you who are interested in information retreaval systems had better get interested in that problem. The AADC machine only incidentally is going to work in the F-14, just as incidentally is the airborne multiprocessor for Wright Field going to work in a bomber, or whatever is around right now. The problem comes up for an airborne command and control system, where your're talking about putting a big system in the air. This isn't peanuts anymore. You've got a space station thats out there and going to stay there for life. Are you going to put a programmer up there to write in SPL? Or does it just happen to be an engineer who would like to write in engineering english? APL happens to come much closer to that. Now, I'm not trying to sell APL. All I'm saying is that Ken Iverson made a long step toward making a programming lánguage which human beings could use. He did something else, he said that human beings should be interactive with the computer. If I don't understand an APL operator, I can find out by default, by trying... As long as I can program, I can now go to an ALGOL, or any other terminal without getting a syntax error. I take people out of our product management group, who, God knows, don't know anything about programming, and take them down to a terminal, and set them down... they can do useful work. Think about that in terms of your space applications. For every programmer you put up there, you have to put two... redundancy. There are a lot of things going forth... I would have the guts to ask the

company to go this way, i.e.(microprogramming), it just happened LSI came along and the hardware people said, "That's the way we'd like to do it," and so I sat back and said, "Great, that's the way I'd like to have it." Now, I can begin to design a machine the way it ought to be designed. We have yet in this Country... this is probably a stronger statement than it ought to be... to have intellectual honesty in the computer world. Whether it's my company which has a thing on one-pass compilers... JOVIAL vs. CMS... I don't care what it is... There is no honest, intellectual effort made to find out what is going on inside a computer. For years people argued that you couldn't measure the instruction utilization of a computer because you could only measure them statically. Well, just out of curiosity we took some static measurements. Would you believe last week I got some dynamic measurements off a B6500, and they're within 1% of my static measurements? Sid Fernback, who's been preaching he must have an extremely fast multiply... Would you believe, multiply is less than 6% of his problem? This is the guy who made us build the STRETCH and the LARK. This is the guy who said I must have a 20 nsec multiply to do my problem. Baloney! Six instructions in the 5500 make up 85% of all the instructions executed. Six instructions can be coded with three bits. Find me a machine with a three hit op code. Any address space... 50% of the hits, I'll guarantee you, are wasted 50% of the time. In a 360, thats 25% of your core which contains programs... the same thing is true of a B3500. Thank God it's not true of the 5500 because we used a smaller address space. Not because we understood anything more of what we were doing. There's a real need, a plea, that we stop taking the Government's money, or our company's money, and doing what we think we ought to be doing by the seat-of-the-pants. It's about time we started to honestly ask questions. What is computing all about? Now, there are at least three people here who have been honest enough to say, "We don't know." And, we're designing the computers. Why don't we know? Let's take a look at the last proposal I answered. Thou shalt execute 250 thousand instructions/second. What in the hell does that mean? Can anyone tell me? We go back to the original statement that there will be ten adds per multiply, and lo and behold, we go back to the latest RFP, and that's what it says, ten adds per multiply. That was made by vonNeuman. And, we don't know any more? Two hundred and fifty thousand short adds per second. By the same token, if I took my microprogrammed machine and make it look like a single address machine and meet their 250 K ops/sec requirement, which is what I did. I microcoded the thing to look like a 360, because that's what they understood; and if it took ten megacycle logic to make that speed, that's what I did; and if it took a half mic memory to meet that speed, that's what I did. By the same token, I'm executing a million instructions per second

per processor! If I can harness it... This is no longer funny. This is bucks you're throwing away... You guys, when you write these specs, have patience with people who have questions to ask. Because, some of us have questions concerning the organizations of the computers we're building. The ILLIAC IV came into existence because of people like Fernbach and the Weather Bureau. And, lo and behold, I think that they are finding out that neither they, nor the STAR, nor any of these machines are really going to do them that much good... There was talk this morning about detecting parallelism. In the ILLIAC IV... Now, Illinois will probably argue this one... from my SOLOMON experience I carefully avoided the ILLIAC... I had enough at Westinghouse... From my SOLOMON experience, if they can keep 20% of those PE's going at a time, I'll be floored, because I don't believe it. There is room. There's work to be done in languages, and it isn't the improvement of CMS-2. It might be a little more honest in a discussion of languages to call it CS-2, and compare monitors with monitors and languages with languages. I like the fact that you've incorporated the recognition that a programmer works with a total system, but there's a hell of a lot missing from your system, because until a programmer works interactively, he never really has use of your library. He can't really build a procedure by asking what he did, or what someone else did in front of him. He's got to go back to a folder and find out. The problems which came up when we bid TACFIRE. We were going to use JOVIAL. I had been a great hands-off man with JOVIAL. I didn't like it. And, I suddenly realized I was going to have 120 applications programmers on my hands. I had 300,000 lines of code to write in 12 months. I suddenly realized I was going to have to build a system which detected the procedure that had been declared but not yet written. That remembered the subroutine that you thought you were going to write, that was only four lines long, that you hadn't done. All this belongs in that debugging package as well. I think you should seriously consider the addition of simulators. Being a software man turned hardware designer, I brought a little different background into this thing. I lived through the problem of writing large systems. I had the great privilege of spending a week with Prof. Deichster about two months ago. When he first got off on his kick about programming languages without having GOTO in them, my immediate reaction was, my God, I can't live with this! And, then I got to thinking, what do GOTO statements do for me? Number one, they create a path in the program which I cannot (absolutely) debug all the cases for. Because, invariably, they're conditional. The thought of that, and the thought of being the pilot of the F-15 up there on terrain following radar scared the living hell out of me. I think it might be cheaper, even if it cost more, to buy the memory without GOTOs. However, I'm not saying it will. Bruce and I learned a lesson together. Programs

should be small. They should be logical. And, any program with more than 50 ALGOL - like statements in them are too damn long. I have been preaching this. Alan Baxter when on his 5500 at Virginia last year decided to find out what the world of students looked like. Lo and behold, the average segment in his system was 30 statements long. So, this is not unreasonable. There's a university doing all their programming, and the average segment is only 30 instructions long. Turns out data segments aren't much longer. And, this turns out not to be too different from those figures IBM has leaked out about their paging problems. That under 200 instructions were being executed on the model 67 between page calls. And it took 700 instructions to satisfy the page call. That's a snicker, but that's a triviality, which can be fixed. What can't be fixed is the ignoring of that 200 instruction limit. What can't be fixed is that people want to hold on to old things like GOTO's and JUMP's. It would have been very little trouble for you in CS-2 to have made the programs reentrant. I think it would also be worth your while to take a look if you really want multiple entry and exit points. All these things add to the problems of debugging, checking out and validating the modules. Now, Deichster with three other people wrote a multiprogramming operating system and check it out. Now, other than key punching errors, it ran the first time it was on the hardware. It is impossible for there to be any unexpected event in that system. Logically inconsistant for it to be. There is no program module larger than a single logical event, and every module is debugged logically. Now, we can build hardware that will do this... I think that is about what I have to say.

Wald: Surely, there must be an argument.

Deerfield: They're overwhelmed.

Wald: Will Dave then...I hate to give you the floor again, but you said something which seemed a little inconsistant. You said that different languages would imply different machines... The ALGOL machine, the COBOL machine, and so forth... but you also said they would have different bells and whistles; but you also said the compilers and compiler writers aren't taking advantage of the bells and whistles you already have on the machine. Now, these stupid compiler writers are going to have to become microprogrammers. Are they going to be any smarter?

McGonagle: Where did I say that, Bruce? The small number of instructions that are used?

Wald: Well, perhaps I made a mistake taking notes, but my notes say, "Compiler writers aren't taking advantage of machines."

McGonagle: I would not say compiler writers are not taking advantage of machines. When I discussed my comments on the small number of instructions which constitute the bulk of instructions executed, that's a function of the machine organization we have, and not of the compiler.

Wald: In the last issue of the monthly news magazine of the IEEE Computer Group, there was a challenge. It said, "I challenge you to give me an example of a compiler you have written that takes advantage of sophisticated instructions in your machine." Instructions like fancy index register control instructions, or repeat instructions of wide scope. Do you have any comment on that?

McGonagle: Maybe that ties in with the question of why am I in love with APL? APL narrows down the scope for me, considerably. APL lets me talk about A+B into C and not care if it's an array or a simple variable. Now, how does that affect me. Number one, borrowing from JOVIAL, it seems to me that the COMPOOL descriptions that we have should not be dropped at compile time. We are binding with those descriptions at compile time in a form that binding should not take place. We are trying to use those descriptions to generate code. Keep in mind, also, that in the systems we build, 90% of the memory accesses are through an indirect address that we happen to call a descriptor. If you look at an indirect address in anybody's machine, it's sixteen or "M" odd bits that get used, and the rest are empty. Some bright programmer may use them to generate a constant, in which case, as Bruce will remember, I come along with the operating system and clobber it. But, the rest of that word is available to me. That word can tell me whether I have an array or a simple variable. What does this mean? It means that when I go down that address path to fetch, at execution time, I pick up a word which I have to look at anyway to get the indirect address. When I look at it, I detect that it's an array. At that point I can substitute array multiply for simple variable multiply. The scope of my instruction, from the programming point of view, is one instruction. The compiler has not been complicated. From a hardware execution standpoint, I now execute that matrix multiply at microprogram speed. Once I build the absolute addresses for the three operands, I can retain them because I know I will not take an interrupt, or if I do, it will be at my leisure. So, I don't have to go to the address building mechanism every time. I go to it once, and I roll through there. I can now do a matrix multiply, effectively, at memory access speeds. What else does this give me? It

lets me write an algorithm. I debug it on simple variants, and know it will be executed correctly on arrays. Suppose the two arrays are of dimensions so that they don't... Question?

Floor (Cerf): I'm sorry, it turns out that multiplication is not commutative in arrays, so one has to be careful about that debugging technique.

McGonagle: Yes. Using the APL rules for default conditions, I effectively get defaults when I work in array spaces or complex arithmetic, which is comparable to an adder overflow. And I can give it back to you in the same way. It's an invalid operation. There are tricks such as the non-commutative matrix multiply which you should be aware of. But, generally, you can write the same program and have it work on complex arithmetic, double precision floating point or any type of data you wanted to, and it wouldn't change your program at all.

Cerf: I dare you to sort a bunch of complex numbers... I just want people to be careful when they make those generalizations.

McGonagle: Right. I screamed this morning about someone else's generalizations. I guess what I'm trying to say is that by transferring the scope of definition, by taking advantage in the hardware of all the information we have, we can take advantage of this to increase throughput and simplify programming text.

Wald: Let me break in, Dave, and ask a practical question or two. Suppose we standardized on an intermediate language. Some people like Polish strings with operator prefixed, some like infixes and some like trees. But, suppose we standardize at this point and a new machine appears and it has some microprogramming capability; at least 360 capability, if not complete emulation capabilities. Now, we have a little race. The race is to build code generators for that machine or to microprogram it for the intermediate language. Would some people in the audience want to bid that job, would want to tell me about the pro's and con's?

Floor: May I ask for a clarification? Do you mean that you're going to take this intermediate language and execute it directly on the machine compared against an existing compiler with a syntax cracker. All you'd have left to do is build a code generator...

Wald: The syntax is already cracked. We've gone from a Problem Oriented Language to something which, by a miracle, we've agreed on is a standard

intermediate language and we changed its name so there'd be no inter-service rivalry. Now, we have the option of letting Dave microprogram it so that his machine will execute this intermediate language, or building the second half of the compiler to translate it into a form closer to the structure of the machine. What are the pro's and con's?

Floor (Engelbach): I think I'd like to extend Dave's argument along this line a little further. There's been a lot of work done by the ALGOL Committee in Europe, which has really been to solve the problem that were facing rather to (sneeze) the hardware that already exists to help. Dave mentioned carrying all this information. He used the word descriptors. There's an article written on the implementation of ALGOL 60 which was never "conquertized" by making hardware of it, but it contains in it a couple of concepts, one which I suppose, you directly execute the data declarations. In this sense, it enables you only to declare only that working space that you need at the time your talking about it. It also allows you to do mode arithmetic without a lot of fancy preprocessing.

McGonagle: We do that now, Bruce.

Engelbach: I understand you do that, Dave, on the 5500. What I'm saying is let's extend that concept... We have really scribed a problem in a language, (followed by) a process of mulching this to an intermediate form... Then you give me the option of creating a code generator which is another piece of software, or do I want to microprogram in the intermediate form. I will counter by saying, I would prefer to eliminate the intermediate form and microprogram in the original source. Now, I'm sure Mr. Deerfield will pound on top of my head, but the object here is to solve the problem. The fewer preprocessing steps that I have to go through, the more efficient the total system is going to be. To bring in the example of the SPL Implementation Tool, which Dave seemed to like: the smaller the amount of code, the fewer the decision paths I have to look at, the higher the probability that I will have error free software. I'll never be (writing) that FFT, but I pity the guy that is, because that software can be very complex. With that in mind, I say let's put everything in the hardware, because I can make the hardware tester that turns itself into that system generator. And, if I make the software simple enough, then I stand a chance of coming near to error free software. I would say, let's not even bother with the intermediate language, go ahead and write the program store in the original source, be it CMS-2, or SPL or ALGOL. Like the 5500, which comes pretty close

to (executing) ALGOL.

McGonagle: Actually, our execution language is reverse Polish. This is the point... you have to go through a first pass in order to get rid of the noise... There's a lot of noise in an input language.

Engelbach: That's for the people to understand the language, but I have a fancy loader... So, I'm not compiling. I'm simply checking my source statements on whatever media -- a card, tape -- and I'm going to eat them. And this idiot loader is not going to do anything that is not reversible. In other words, I can take a program in source form, strip out the buzz words, put it in memory and execute it directly. As a converse, I can take that same program from the same memory, come back through the reverse process and reproduce the source, losing no meaning, neither syntax nor semantics.

Wald: You'll have to lose some buzz words. Also, there are some people who would argue...

Engelbach: I can have a whole set of buzz words which I don't lose the meaning (of). I may change two buzz words, but if they're buzz words, I can substitute the same meaning.

Floor: I'd like to counter your argument with a question. I believe it was in 1948 that the first, hardwired digital computer came out. Ever since that time we've gone to more general purpose computer. Perhaps, the reason why we've gone to general purpose computers is because one, nobody has ever been able to decide upon a language and two, decide upon how they were going to solve the problem and three, once they discover the problem it seems to continually change. Therefore, this is why we have the creation of compilers and general purpose computers. I'm curious as to how you're going to microprogram a continually changing system... CMS-2 may not be the best, but at least it's being modularized to accommodate change. How are you going to do this in the hardware? Change is really the name of the game.

Engelbach: It gets back to saying, the only thing to remain the same yesterday, today and tomorrow is change. Therefore, I must make available to me a facility which gives me the most powerful tool with which to make that change. As you've mentioned, the trend is away from hardwiring, where you cannot change anything, into general purpose software, where things can be changed a little easier... The trend now is toward

microprogramming. I can build a very powerful tool. It's a microprogram store. Generally, we're talking two thousand words of microprogram, which is a heck of a lot less program logic to change than is an operating system like OS 360, which takes up a quarter of a million bytes.

Floor (Bersoff): It bothers me that the people on the left side of the panel have alluded to the fact that software people want to perpetuate themselves, and that all that great hardware is there, but they don't want to use it. Just six months ago I was envolved in specifying the NASA space shuttle data management system. We had contractor people come into us who didn't want to give us a hardware floating point capability. We've been dying for that capability. Why hasn't it been offered? It's not that we don't want to use it, it's that no one is giving it to us. I think that's been the problem with aerospace for the past ten years. We can't get a hardware floating point multiply.

McGonagle: I can't comment about the last ten years, because this is our first venture into avionics.

Deerfield: I'd like to comment... It's been at least ten years since my discussions with Jim Ward and the people at Wright Field, when we identified the fact that anybody who didn't build a computer with floating point had rocks in his head. I'll tell you why I think it's not being built. In fact, I used to make a statement that any one of the computer companies who built these nothing-type machines, which I think most of these little computers are, are nothing-type machines. In fact, I think that people who design new machines when the old ones can certainly do most of the jobs are wasting their own time as well as everybody else's money. Companies seem to think that if they can get contracts, they ought to design a new machine... Two reasons are true why people don't make floating point computers. One is, it used to be, and it's not true anymore, that people used to think that floating point cost a lot of money to put into the hardware. And, there have been notable examples of companies that aren't really computer design companies, attempting to design floating point algorithms, and finding out they knew nothing about it, and it grew like topsy. They ended up, very often, saying it's too expensive. The truth is that floating point isn't all that complicated. The other thing turns out to be, and I used to say that any one of these little companies that built a computer with a floating point set would suddenly find the Air Force and the Navy and everyone else suddenly jumping up and down, saying, "I'll buy it!" Even without software and compilers. It turns out that the branches of the military have simply not specified floating point for their machines. They've been cost conscious. Only, they've

put the cost in the wrong place. People have argued, for example, that a fixed point word is smaller than a required floating point word. It turns out not to be true if you do an analysis, because you can get as much precision and accuracy in a floating point word of the same size as you can with a fixed point number. But, people have argued that the amount of memory and other things. Subsequently, contracts have been let without this spec in them. I think that's the only reason you don't have it, because you don't ask for it. Or, if you ask for it, you ask for it sort-of-like, it would be nice, if. I think all you have to do is ask for it, and everybody will break their back to give it to you.

Bersoff: When I asked for it, I was looked at as if I had two heads.

Deerfield: Well, that's true. I'll concede that I can make an argument to prove that in any given application you don't need it. But, that doesn't say you shouldn't have it... just because you don't need it.

Bersoff: What do you mean by "don't need?"

Deerfield: I can show you that a judicious combination of scaling and a lot of other ____________ will handle your problem. And, probably handle it efficiently... There was another reason why you don't get it. People used to think that the speed of a machine was slower to do floating point. It's obviously true that floating point multiplies and divides are faster than fixed point, simply because the speed is conditioned upon the number of bits in the mantissa. The floating point adds and subtracts takes a long time. You have to take an add order, for example, and compare the exponents, and you have to shift right and shift left. You have to normalize your instructions. It takes a lot longer, except for one thing. If your numbers happen to be already normalized, and if you don't insist on normalizing every answer you get, which is generally a mistake to do, what you discover is that in the cases where the numbers are, in fact, in the same exponent range, just as they are in fixed point, that the problem runs just as fast in the computer, and, consequently, it's a misanalysis. On the other hand, my own argument is that anyone who doesn't have floating point is really ridiculous. As a matter of fact, the question which was raised of whether we need fixed point... what you ought to have, but you wouldn't want to waste the bits, is a fixed point systems, which is precisely the size of your fixed point mantissa. Where numbers are entered as fixed point numbers, and let the machine treat them as floating point numbers by effectively having the agreed upon range in which you would normally be functioning as an exponent. So, that in the event of overflow, instead of having an overflow indicator, let the machine

act as though it were floating point in the first place. There won't be any problem at all.

Wald: Is anyone experienced with the G20?

Floor: That machine was built and sold commercially five years ago and didn't have any outstanding success.

McGonagle: It was built in America eight years ago and didn't have any outstanding financial success until a year and a half ago. It's called the B 5500.

Wald: Is that the way it treats fixed point?

McGonagle: That's why it does mixed arithmetic without any problem.

Deerfield: The only reason you're not getting these things is because you just don't ask for them.

Engelbach: I have to agree with you, but we're chasing the chicken and egg. Characteristically speaking, the Government -- at the risk of throwing mud on myself and a few of my associates -- doesn't know what the hell it's doing. It relies tremendously on consultant corporations like MITRE and Aerospace Corporation, and these people really don't have our problems, they don't have the profit motives that people like yourselves have. So, we get a hybrid. They're very conservative because they don't want to create a ruckus. They don't want to make a big mistake. The Air Force doesn't know what it's doing, so we come out with inane statements like, "If we had something that looked like...," or "if we had something that performed like a floating point machine, but we really don't want it, because we know its going to cost more money and be bigger, etc." What does it really mean? It means your people on your side are looking at that statement of work and are calculating down to the eyeball how much it's going to produce to that specification. And you say, the Air Force obviously doesn't want floating point so we're not even going to bid it as an option. Now, here's the Air Force on the other side, saying, "If industry doesn't propose floating point they must really think it's not worth taking the time to propose." Now, which comes first? I have to admit that occasionally, the Air Force lucks out and hires a good consultant and we get something done, but (normally) we have our run-of-the-mill luck and we don't get a good consultant.

McGonagle: You have consultants and we have product management. One is as bad as the other.

Wald: Before we relegate compilers/code generators to the past decade, would someone support the argument that isn't it better to translate those semantics once in the compiler than to force the microprogram interpreter to do it every time?

Deerfield: At the risk of talking too much... The two of you have said things which I really disagree with. I would like to make another brief comment here. First, I am not in favor of microprogramming. I think I've been through that for too many years.

Wald: You've put those childish things behind you?

Deerfield: I'd like to go on record that I have, although that's not quite true. There are some cases where I still feel it's the right way to go. I'm not in favor of complicating the computer hardware, either. I'm not in favor of eliminating compilers. What I'm saying is that in the hardware, we ought to put the mechanisms which are common to all your compilers. All the ones which you no longer even talk about. When I say, for example, handle the algebra, I simply mean... every one of your compilers has a routine which it goes through which does such things as locating the center of your parenthesis. I think these methods should be put in the hardware. In this manner, you still have the syntax and you still have all the variability and you can still write all your compiler programs and vary them. I think the hardware ought to take the burden. The other thing I want to say is the cost of hardware today, now that we're going to LSI, the gate cost is so low, that we're really not talking... The cost is now interconnection, rather than the number of gates on a chip. The cost of gates has gone so low that to not do these things, I think, is to miss the boat. So, again, when I say I'm against microprogramming, its for the reason that with the high speed scratch pads and task memories that we have, that I can write you a macroprogram which is just about as fast as your micro. A macro subroutine, if it competes with my micro-subroutines, is probably not worth translating from my machine language into my micro-language, and I think, we're getting too many levels of language here.

Floor: It seems to me what you're saying is that the difference between microprogramming and macroprogramming is basically a matter of memory speed and, perhaps, (turn around time).

Deerfield: I think what I'm really trying to tell you is that when microprogramming came in, one of the things it did was attempt to order the control unit of a computer so that one could subject it to ordered, step-by-step programming. Once you can get to every control line of the computer internally, which is what the control memory did for you, and order it, and make these lines evailable to a high speed control or task memory, you've erased the distinction, I think.

Wald: May I redirect you to another thought. And that is, we seemed to come to the conclusion a little earlier that there were some things that a programmer did that you had to escape to machine language to perform. Are those part of the problem the programmer is trying to solve, or are they artifacts of the machines that we can get rid of?

Deerfield: Are they problems that the user is trying to solve. He is neither the artifact designer who throws in gimmicks, features, bells and whistles, nor is he the programmer... Dave made the comment that he's never seen a machine which did list structuring within the machine. That's not quite true. We've recently built a multiprocessor that does some very fancy list searching within the hardware, which is structured after lists which are really part of the software problem. One of the interesting things was we looked at the customer's problem, and it happens to be a weapon delivery type problem. But, one of the things we looked out was that he went through tables of data that had been accumulated, and he did this process, which really took only a few instructions, but he did it some ten thousand times every major cycle of his machine, resulting in the fact that a considerable portion of machine time was occupied (unnecessarily). What we did was look at the specific problem, and we said, "What is it, as a user, you really want to do?" Let us make a list search specifically dedicated to your problem. It's not a general purpose listing or search, and we built in the bells and the whistles. As a matter of fact, it cut that particular problem down fantastically. And, then we built the JOVIAL. We asked the JOVIAL people to include this whistle because it's very important to our customer. The JOVIAL people, in this particular case, left it out. Hopefully, they provide a technique, though, whereby we can revert back to machine language, at least for the one function, which is one of the reasons why I'm adamant about the fact that occasionally you design something into your computer for your customer. I think your compiler ought to be able to use it... I've designed lots of bells and whistles into my computers which I thought were great, only nobody ever used them. I don't expect the compiler people to use them either.

Floor (Nimensky): I'd like to make several comments, before I forget, about what Dave said about GOTO statements. The entire meta system

is built without a single GOTO. Secondly, I think the problem is that industry does not meet more often like this, where we can sit down and hash these things out. Everybody is doing everything secretly and everyone is trying to one up everyone else, since they know better. I think the biggest problem is there is no free and open exchange. This floating point problem where everybody makes crazy assumptions. Floating point is more expensive than fixed point. Everyone knows all the answers. Nobody talks about it. When we have conferences, people take out their slides, talk and go away, instead of putting everything on the table and finding out what is the problem. Engineers very rarely talk to software people and software people rarely talk to engineers, except in these rare cases. What can you expect?... As far as getting to microprogramming, the software people look at it as an extension... Who is it who should design the software system for the computer? The engineer or the software man? This is where that problem resides... The engineer who builds the computer doesn't know how the software person wants to use it. He may want to program a sine routine. If there are ten programmers, they might want to program the sine ten different ways. Ten different users could have these ten different routines microprogrammed the way they want to. Just like one guy at IBM took the EULER language, took the ROS memory and built a list processor. He made the 360 look like a list machine.

McGonagle: He made it look like a 5500.

Nimensky: Microprogramming is certainly more flexible. It allows you to build a hard machine to use whatever language you want. That is, why not let the compiler writer design the instruction set he needs. If he only needs six instructions, why bother with the other 125 that AADC has if they are not needed.

Wald: How about the data declarations? Do you think there should be a translation process in mapping the memory or should we interpret the data declaration each time we pick up an operand?

Nimensky: That depends upon what the problem is. Any type of list processing, LIST, EULER, all these, you can mix data up in any way... You never know what it is. They have to be self-identifying. Either it has to be self-identifying by some pointer or some bit size or whatever. Again, in a microprogram would be lovely if we could microprogram the word length, the address fields and all these things. So we can structure the machine. In other words, give us a piece of hardware that programs the hardware and you can build the instruction set, you can build the addressing scheme

to suit your problem. Now, you've got the ideal match between the problem and the machine. This allows the user to write in a language in which he can think. This is the biggest problem. A programming language should approximate the language I'm solving the problem in. If I program a chess game, I want to use a language that looks like a chess board. If I'm programming matrices, I'll add matrix notation.

Wald: Vint, you look worried.

Floor (Cerf): I agree. I think you can write a program and let the compiler write the machine description you need. Then you walk away with a special purpose machine which solves your problem eventually. But, my impression is that, for airborne computer, there isn't enough room to store away a different computer for every function that has to be provided. So I wind up building a more general purpose machine.

Wald: Well, it's not out of the question in AADC. As you load a routine you load the microprogram and change the nature of the machine as it's executing that routine.

McGonagle: As part of a process description, there should come a definition of the machine... Part of the (task) state vector should be definition of the machine upon which it executes.

Wald: Let me sketch something on the blackboard. (See Figure 1.) We talked about problem oriented languages. And, we talked about procedure languages. And, this is, per chance, the language of ECM. This is CMS-2. Then we talk about some sort of intermediate languages. I guess the only reason we talk about intermediate languages is because we know very well how to build translators. There's a lot of academic interest in this process, therefore we talk about this one that's composed by human beings and translated into this form which is particularly easy to translate further or, in the case of the 5500, execute. But there's a potential translation process here. Then we have another translation process in which, somehow, a machine description has to be fed in. We have no standard language for this machine description. These are the code generators we're talking about. And, then we have the machine language. I'll put this in quotation marks. I don't know what machine language is for our microprogrammed machine, but let's take as a working definition, that form in which the program lives in mainstore. Then, finally, we have some hardware. We may very well have a microprogram, which is involved in some sort of algorithmic translation of the machine language into the control signals to operate the hardware. And, the name of this panel is Computer/Compiler Compatibility Requirements or Standardization, which of these steps along the way are candidates for standardization? Does anyone want to make

a recommendation?

Floor: Can you write an objective for why you want standardization or compatability?

Wald: Yes. Program reusability. Reusability of both our systems programs and our applications programs.

Floor: Just to make life easier for the programmer.

Wald: No. Just so the military can get its job done with zero declining budget.

Floor: And programmer trainability?

Wald: That would be nice, too.

Floor: Didn't you just compound our problem? In putting a problem oriented language up there, you immediately drew the arrows to a procedure oriented language. There is no reason for that to happen. We don't have a problem oriented language yet.

Wald: That's true. And, I do want to offer the option of eliminating as many of these translators as possible.

Floor: Let's go from the problem oriented language to the hardware.

Wald: Will either Raytheon or Burroughs give us a firm fixed price for the job?

Deerfield: One of your problems is that you're calling language things which are nothing more than your nouns and verbs. For example, when you say machine language, you're missing the point. I think you're missing the point when you say six instructions are all you need, because we're not talking about the instruction set at all. I think the point is that the nouns and the verbs aren't the thing at all. Up here, when you define your procedure, you've got a real language. When you said yesterday that you first have to recognize your "if" and then check if a boolean follows. You do a whole flock of things. I think that's a large percentage of your program. Those are the things that have nothing to do with whether it's an add, or a function like sine. I'm talking about the hardware literally picking up the procedures. Then, I think you'll have a hardware language. The hardware itself has its own built-in procedures

for trivialities. But, unless you talk about a special purpose computer, which builds in the procedure to solve the beam steering equations, or some specific function which the problem oriented user would like to solve. He would like to enter an item like compute fast Fourier transform, for example. Down at the computer end, there is a language, but it's on trivialities. I'm saying, nows the time when we can build into a computer the language which doesn't deal with how do you do a sine or cosine, but the language of how you proceed really to compile... The method which you program a solution today may be, in one case, the only way we know how to solve it, and, indeed, we will have to program at the microprogram level. On the other hand, the method you use to solve the problem may not be the way we want to build it into the machine. I think my example turns out to be the one I've given before, which says, basically, you write an algebraic statement and, now, you have a procedure. By following certain rules locate and rearrange the sequence of events under which you solve the algebraic problem in the final analysis.

Floor: In the AADC, can you eliminate from the intermediate language on down? It seems like the obvious thing to do.

Wald: I would suspect that would be the general direction the AADC would move in. Bear in mind there's the additional requirement in the AADC, that the form that is resident in main store be a very concise form. Because of the bussing problems to and size of Task Memory, that might be the controlling factor. There may be more time available within the PE to interpret than there is time on the bus to get non-concise versions into Task Memory.

Cerf: May I suggest that it's not clear yet that you need a Task Memory. I think we want to show that we want a local memory attached to the Processing Elements. The second thing is, there must be a certain amount of uniformity which is to be imposed upon the architectures of the machines because of the modularity of the LSI. So, at some joint there has to be compatability. I'm not sure where that goes on the diagram up there. If anyone on the panel cares to comment, I'd appreciate it.

Wald: Would, for the sake of argument, could we think about having the place at which uniformity applies as the intermediate language with data type declarations?

Cerf: I'm not sure how to respond to that.

Wald: I tossed a coin and came up with a place. Now, let's see what's wrong with it.

Floor: Some of the criticism of that has been that you have to have a much wider __________ because of the necessary descriptive information that has to be carried. Also, the cost in the logic to decipher the compatability of descriptors for a particular operation jacks the cost up to the point where it is not as economical as using conventional techniques.

Wald: The data descriptions can be concise. You could have defaults; you could have the length of the code inversely proportional to the probability that that data descriptor is used.

Floor: But, it's nevertheless something greater than the value. There has to be something there.

Floor: Isn't a lot of that cost for just the size of the bus and execution time?

Wald: I'd almost want to buck that one to Ron Entner.

Entner: I wouldn't be terribly concerned with the gate cost at this point, if that's where the problem is. Our major cost item will be the Main Store Memory. To that end, I would like to see whatever we do lead in the direction which would reduce the total amount of Main Store Memory required. That's one of the reasons for using the Task Memories. They're simply a convienent (random access) storage location which permits us to use the BORAM memory, which is at least an order-of-magnitude less expensive than a conventional main memory.

McGonagle: Our feeling is that the cost of gates in todays world is relatively cheap. I can buy from TI, now, enough parts to build five processors than what it used to cost me to build one AN/GYK. But, my total manufacturing costs for the AN/GYK was twice, going on three times, what it cost me to buy the LSI parts to build five machines. Now, microprogram store is still fairly expensive. But, by using overlay of the microprogram store, I can keep that cost fairly small. Again, using default conditions, 256 words/16 bit microprogram store will contain most of the interpretation and most of the instructions for 90% of any machine definition for emulation purposes. That means I haven't paid any great penalty for the majority of the cases. Thirty-two bits of information allows me a 16 bit address and 16 descriptor bits, which will tell me whether you've got one, two, four. . .

I give myself three bits for describing item field size. I can tell whether something is complex, simple, arrays, double precision, integer or floating. Whether or not it should be subscript. Whether or not there are more descriptors envolved. Access information. Lock and unlock. This is something which most of us tend to leave out. We've learned from bitter experience that, particularly as you go to parallel processing, it becomes absolutely essential to be able to insert monitor information into the access path to get at any piece of data in that memory, because you can't control that access path. So, I don't feel I'm paying a great price. I think I got this in the 16 bits that are already there free anyway.

Wald: I wonder if anyone would care to predict, just to see how cloudy the crystal ball is, when AADC finally exists, if we will be standardizing at problem languages, at languages like CMS-2, at intermediate languages, or if we're going to wipe out the whole translation process. I'll go down the panel.

Samtmann: If I had to take my pick right now... I had the opportunity to review several reports pertaining to SPL, CLASP and CMS-2. Once again, my experience is very limited ... Once again, it seems to be the general concensus that CMS-2 will do the job. Right now it looks like a modified CMS-2.

Henderson: Ideally, we hope to ultimately see the whole translation process disappear. At least, by the user. The user wants to solve a problem in the most effective way possible. At this point in time, all system type software compilers and translators have just inhibited us. Reality suggests that there will be some compromise. Whatever the procedure oriented language chosen, that I don't think is really important. The important thing is to minimize the distance between that point and the hardware, itself. I would hope the AADC would make some major contribution in this area.

Deerfield: In terms of problem oriented languages, I don't think we're going to have enough people to get together to really be able to define a problem oriented language solution. Consequently, as much as I think everyone would really like that. Being a realist, I don't think you'll have it at all. In terms of a procedure oriented language, it's my prediction you won't have any of the languages you presently have at all, because I think that the AADC computer system will be significantly different from any other computer system ever built before by the time it gets finished, that you will, in fact, want to write a new procedure oriented language. At least that's my fond hope and prediction. ...On the final end of it, I think that the solutions will be partly carried down the line. Namely, I think there will be some measure of microprogramming, some measure of variations from the procedure oriented language.

McGonagle: I find myself pretty much in agreement with Al. I'd like to hope that standardization, if there is any, will take place, in actuality, at the intermediate language level. I strongly suspect that in the next few years, as studies progress, gotten beyond the point of having to be in agreement as to 128 instructions, or whatever the magic number might be today. I think, instead, there will evolve an intermediate language which is convienent for the compiler writers and which is a reasonable language into which to escape for those people who feel they must escape into machine language, and that this will be considered the machine language of the AADC.

Wald: Thank You.

APPENDIX A

CMS-2 COMPILER DESIGN

Systems Consultants, Inc.

Washington, D.C. 20007

6 August 1970

NOTE

This paper is an excerpt from a Systems Consultants, Inc. report entitled, Interim Report on the Feasibility of Implementing the CMS-2 Compiler on the Advanced Avionic Digital Computer, prepared under Naval Air Development Center Contract No. N62269-70-C-0274.

3.0 CMS-2 Compiler Design

The purpose of this section is to provide insight into the concepts and methods employed by the CMS-2 compiler program for translating the high level CMS-2 compiler language, called source language, into executable computer instructions called the target or object code. The following paragraphs include a general discussion of the CMS-2 compiler structure and the major processing sections that comprise the compiler design. Each major processing section is further discussed in detail defining the overall methodology and compilation techniques employed as well as, identifying the inherent program design of the processing components of each major processing section.

3.1 General CMS-2 Processing

In order to better understand the compilation process as performed by the CMS-2 compiler, it is advantageous to first understand the basic structure of source programs to be compiled. A source program is defined for this report as the logical sequence of source statements necessary to solve a problem. The basic structure of a source program, as illustrated by Figure 3-1 can be comprised of up to three levels. In terms of hierarchy the highest level is the "System" which is made up of one or more elements. The elements represent the second level called "System Procedures", wherein the system procedures are comprised of the third level called "procedures". This tri-level source structure is further characterized by "global" and "local" elements. The global elements are those elements that are referenceable by the

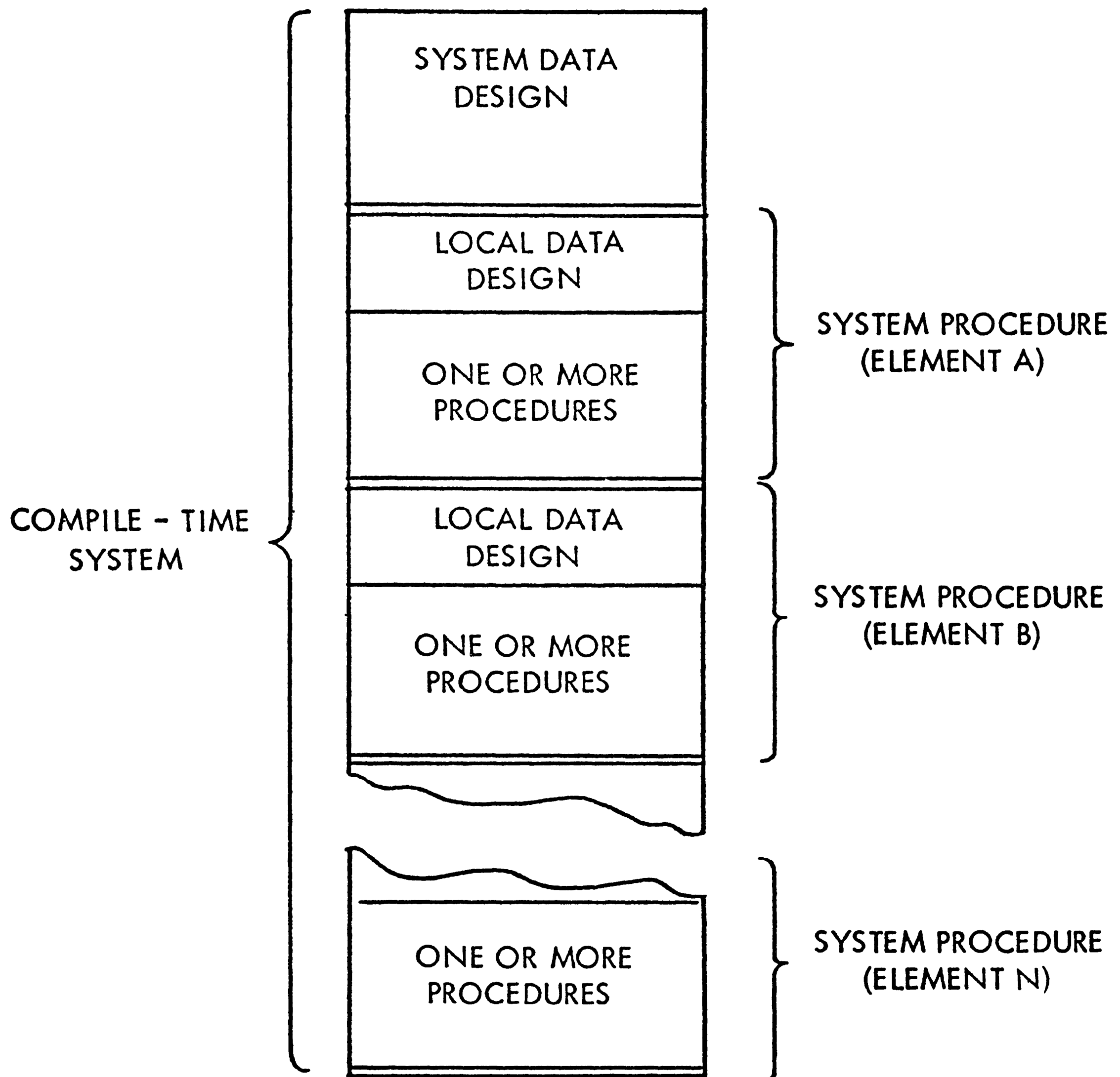


FIGURE 3-1 COMPILE - TIME SYSTEM

entire system, such as the "System Data Design" and "System Procedures." However, the global elements are made up of functional units, such as the "Local Data Designs" that are not global to the system, but are referenceable only within the element in which they are contained. The non-global elements then are called local elements.

The CMS-2 compiler program that translates the above defined source programs into executable object code, processes the source string of language (source programs) one statement at a time in a contiguous manner. The basic design of the current CMS-2 compiler as illustrated by Figure 3-2 consists of a Master Controller that acts as the upper level executive, and two phases that perform the actual compilation process. Before further discussion it should be noted that the CMS-2 compiler was originally designed to be a three phase operation, but Phase 2 that was to perform code optimization has not been implemented and therefore is not a topic of this report.

Phase 1, shown in Figure 3-2, is controlled by the Master Controller and consists of a Director that acts as the Phase 1 executive, statement processors that perform the syntactic analysis function of Phase 1, and utility routines which support the statement processors by performing specialized operations. The Phase 1 operation accepts the source program and performs a syntactic analysis that produces the machine independent intermediate language (IL). The IL form of the program being compiled consists of the sequence of basic computer operations necessary to represent

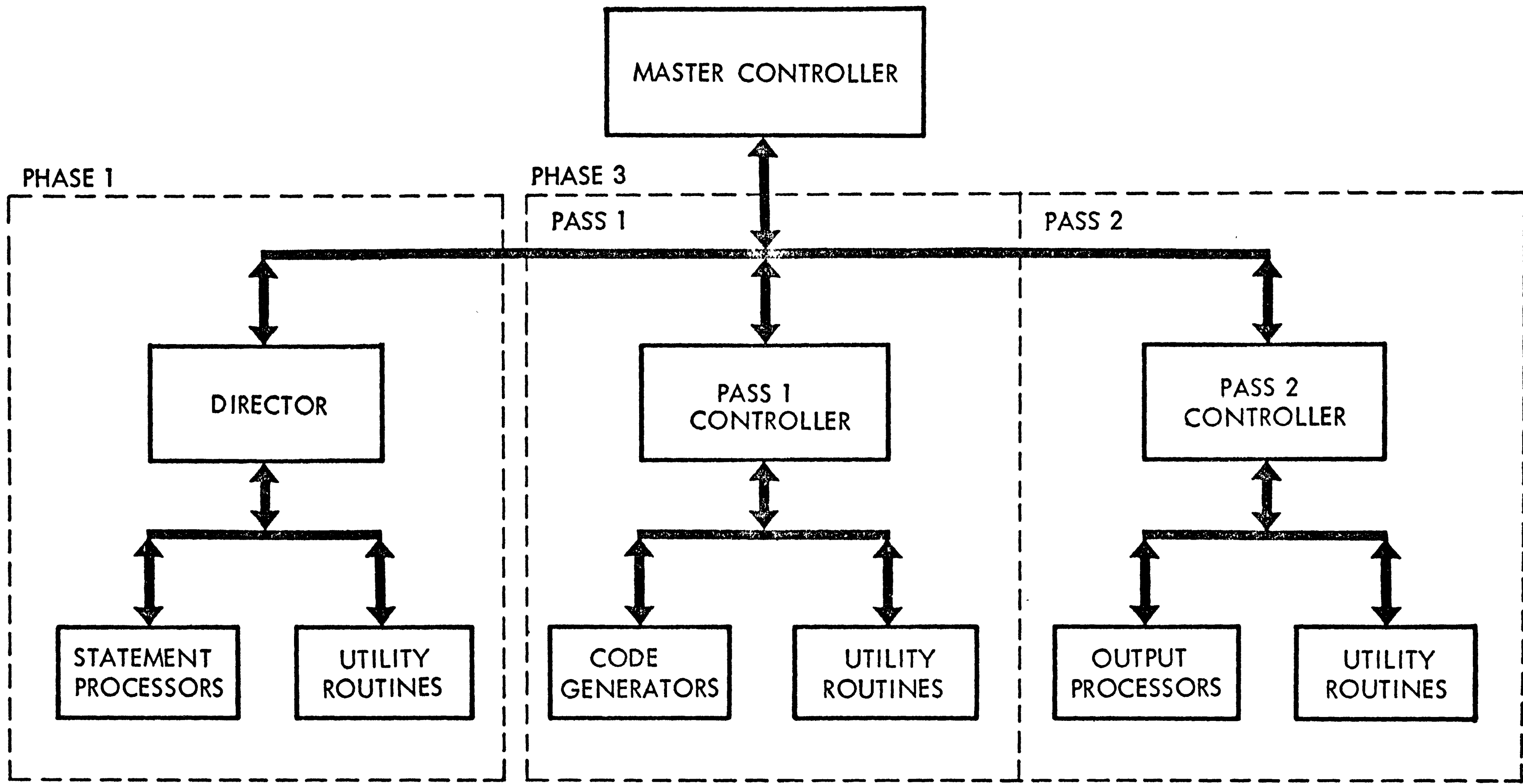


FIGURE 3-2 COMPILER STRUCTURE

the high level logic of the source program in a form best suited for object code generation performed by Phase 3.

Phase 3 (See Figure 3-2) of the compiler is a two pass operation, wherein Pass 1 consists of a Controller, code generators and utility routines. The Pass 1 Controller acts as the executive controlling the various code generators that produce the actual object code from the IL generated by Phase 1. Each of the code generators is supported by the Pass 1 utility routines. Pass 2 of Phase 3, which produces the final object and hard copy outputs from the compilation process, consists of a Controller, output generators and utility routines. The Pass 2 Controller coordinates the output operations by initiating the appropriate output generators which format the final outputs and transfer the output data to the appropriate peripheral devices. The utility routines of Pass 2 support the processing of the output generators.

The inputs and outputs of each section of the compiler described above are illustrated by Figure 3-3. In this figure the source input to Phase 1 of the compilation process consists of source programs from card decks, system and user libraries and common pool (COMPOOL) processing. The outputs of the Phase 1 process are the IL form of the program being compiled (magnetic tape), local Constant and Identifier Symbol Tables (CIST) and minor System Communications Tables (SCOT) (magnetic tape), and global CIST and major SCOT (computer memory). These Phase 1 outputs are the inputs to the Phase 3, Pass 1 operation which produces the final object code generation for the program being compiled. The outputs from the Phase 3 process are the final

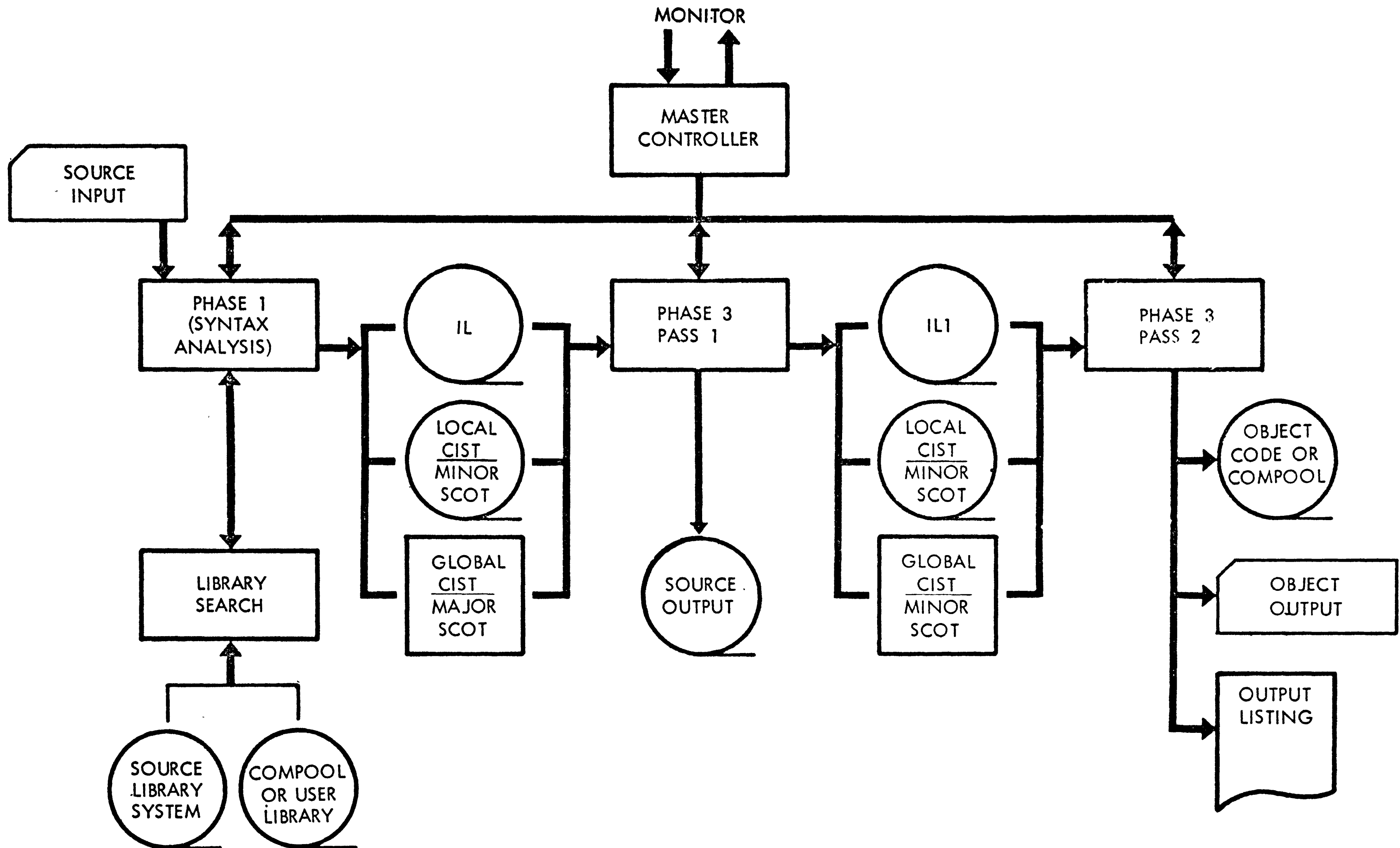


FIGURE 3-3 CMS-2 COMPILER PROCESSING

intermediate language (IL1) tape which contains the generated object code, the local CIST and minor SCOT tapes with final allocation of constants and identifiers for the target computer, and the global CIST and major SCOT tables. The Pass 1 outputs become the inputs for the final output generation performed by Pass 2. The outputs of Pass 2 are final object and hardcopy outputs as well as, COMPOOL definitions for use in future compilations.

3.2 CMS-2 Compiler Components and Processing

CMS-2 is an algebraic compiler that utilizes a multi-table-driven technique to perform the syntactical analysis and partical code generation of the source input. A multitable-driven technique involves the use of internal compiler tables that contain the source language primitives and operator hierarachies that are used to verify the input source statement for correctness and control statement parsing. There are various methods of parsing used to deconstruct the source statements in a manner such that its syntactic correctness is verified according to the grammatical rules of the language. CMS-2 uses the common parsing algorithm called, "Reverse Polish Parsing" This particular algorithm not only verifies the syntactic correctness of a source statement, it also arranges the source statement into a string that allows orderly code generation.

The following subsections further discuss the three major sections of the CMS-2 compiler (Master Controller, Phase 1, and Phase 3). The discussions include defining the relationship between various components and associated sub-components, wherein the

memory requirements for all sub-components are tabulated (where possible) for that version of CMS-2 running on and generating code for the CP642B computer.

3.2.1 Master Controller

The Master Controller performs two distinct functions as the general upper level executive of the CMS-2 compiler. The first is that of interfacing with the "Resident Monitor Program (MS-II)" that provides control of all batch processing operations and the loading of the Master Controller when a compilation task is encountered. The second function of the Master Controller is that of compiler phase coordination generally discussed in sub-section 3.1. The following paragraphs provide a more explicit discussion of the Master Controller and its functional coordination of the two phase compilation process.

The MS-II program loads the Master Controller for coordination of the compiling process upon encountering a compilation task. At this time the Master Controller, as illustrated in Figure 3-4, performs all pre-compiler setups, including the selection of magnetic tape units to be used by the compiler and the loading and initialization of the Phase 1 section. After the initialization is complete, the Master Controller passes control to the Director of Phase 1 for syntactic analysis and intermediate language generation which is further discussed in Paragraph 3.2.2 of this report.

When the Phase 1 processing is completed and control is returned to the Master Controller, Phase 3, Pass 1 is loaded and initialized. The Phase 3 initialization includes the preservation

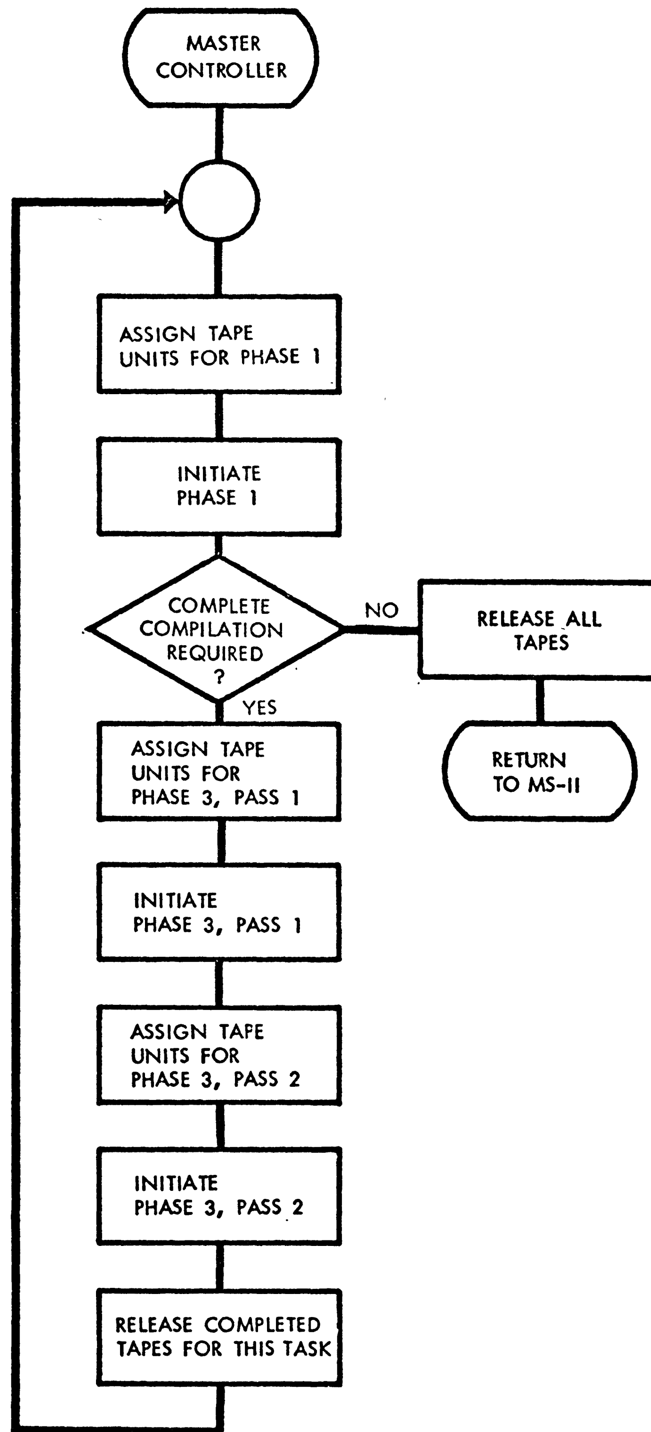


FIGURE 3-4 MASTER CONTROLLER PROCESSING

of the system data resident in computer memory and on magnetic tape for Phase 3 generation of the final intermediate language (IL1) tapes. When the initialization process is complete, the Master Controller passes control to the Phase 3, Pass 1 Controller.

Following the completion of Pass 1, the Master Controller loads and initiates Phase 3, Pass 2 which generates final outputs (source and object) for the compiled program. After the outputting process is complete, the Master Controller checks for a subsequent compilation process following the one just completed. If there is another program to compile, the Master Controller begins the compilation sequence again. If there are no further compilations to be performed, the Master Controller returns control to MS-II for continued batch processing.

3.2.2 Phase 1 Syntactic Analysis

The Phase 1 process performs the syntactic analysis of the source input language for the program being compiled, and generates the intermediate language (IL) of the program based upon the syntactic analysis. The syntactic analysis of the source input string is accomplished using a multitable-driven technique which implies that both the language primitives, such as, the "SET", "IF", and "FIND" statements, as well as the grammatical rules are defined by compiler tables. For example, as the source string is input a syntactic term is extracted and classified according to the language definition tables. The corresponding IL item is then generated depending upon this classification. The scanning of the source string is in a left-to-right manner with the source language statements all having the same general

form of *Verb - Operand*. In this form, the verb determines that portion of the language to be processed, and the operand supplies the necessary parameters to support the operation indicated by the verb.

The Phase 1 process is accomplished by three major components: the Director for Phase 1 Control; a set of Statement Processors for syntactic analysis and parsing to the IL; and Utility routines which perform specific operations common to two or more statement processors. The following paragraphs define each Phase 1 component as well as the data base and IL for the Phase 1 process.

3.2.2.1 Phase 1 Control

The control routine of Phase 1, called the Director, coordinates the process of the syntactic analysis of the source input. The Director begins with the initial scanning of each statement from the source input string. The scanning process is accomplished by a utility routine which extracts a single syntactic term from the input string. As the Director receives the first syntactic term of a source statement, it attempts to identifify the term using a table of syntactic language primitives containing the verbs (such as, SET, IF, and FIND) and operators (such as mathematical or relational operators) of the language. If the syntactic unit is identified as a language verb, the Director switches control to the appropriate statement processor which completes parsing and IL generation of that statement. If the Director is unable to identify the syntactic term as a language primitive, a special directive statement, identifier,

or a direct coded instruction (mnemonic or machine language), then an error has occurred and a utility routine is called to register a syntax error on the IL for that statement and further processing of that statement is aborted. This process is continued until the Director encounters the source program terminate statement, at which time the Director "wraps up" the Phase 1 operation and returns control to the Master Controller program.

It is noted that the previously mentioned statement processors which perform all parsing are actually a part of the Director program in the current CMS-2 design. However, most statement processors are independent of all others and therefore, for purposes of design analysis, the statement processors are discussed as a major component of the compiler in the following paragraphs.

3.2.2.2 Phase 1 Statement Processors

The statement processors of Phase 1 are called by the Director to complete parsing and IL generation for the source statement being processed. The statement processor called by the Director is determined by the verb of the statement which the Director identifies during preliminary analysis. There is a statement processor for each verb in the CMS-2 or CS-1 languages, each of which performs specific parsing of a source statement. Table 3-1 lists the Phase 1 statement processors of the current CMS-2 Compiler showing the mnemonic name, and the text name of each, as well as the associated utilities. Explicit memory requirements for the statement processors are not available for this report and are not shown in Table 3-1.

STATEMENT PROCESSORS →

UTILITY ROUTINES ↓

Note: Sizes for the statement processors are not available.

TABLE 3-1 STATEMENT PROCESSORS/UTILITIES CROSS CORRELATION

After a statement processor receives control from the Director it begins syntactic analysis of the operands of the input source statement. This analysis is accomplished by extracting syntactic terms in a left-to-right scan. As each syntactic term of the operand is extracted from the source string, the statement processor attempts identification of the term based upon the range of grammatical choices possible dictated by any previously successful term identification for that statement. For example, if a simple assignment statement verb is recognized by the Director as the language primitive "SET", the SET statement processor is given control. The next syntactic term expected is either a defined data receptacle or an index; if it cannot be identified as such, a statement error has occurred. An error message is then generated to the IL and processing of that statement is aborted. If the syntactic term is identified as a data receptacle, then the appropriate IL entry is generated, and the next syntactic term is extracted. Because a data receptacle or index has been identified, the next expected syntactic term may be another data receptacle, index, or the language primitive "TO". This type of statement analysis, based upon syntactic dependencies, is continued until statement parsing is completed, i.e., an end of statement is encountered, or a syntax error is located; at which time control is returned to the Director.

All statement processors rely upon sub-routine calls to utility routines that perform specialized operations such as number conversions, error processing, source string scanning, and

special parsing. The following paragraphs describe the role of the utility routines for Phase 1 and relates each utility routine to the statement processors requiring the use of the routine.

3.2.2.3 Phase 1 Utilities

The utility routines of Phase 1 are used in common by two or more statement processors to perform specific operations in support of syntactic analysis and statement parsing. Table 3-1 provides a list of the statement processors and the utility routines required by each processor. The size of the utility routines are attached to provide the total number of instructions required by the current CMS-2 compiler to process each possible statement of the CMS-2 language into the IL form of the language.

3.2.2.4 Phase 1 Data Base

The Phase 1 data base provides the syntactic analysis tables as well as storage areas for saving information about the program being compiled. The saved information generally consists of the defined names and format parameters of all declared data designs, statement labels, and formal sub-routine definitions of the CMS-2 language called procedures or functions. Table 3-2 provides a list of the data structures for the data base used in processing the source input.

In addition to the previously mentioned data designs, Phase 1 also generates three other data tables that are maintained on magnetic tape. These tables are: non-global Constant and Identifier Symbol Table (local CIST), minor Systems Communications Table (minor SCOT) and the Phase 1 generated IL form of the source program. These tables are defined as follows:

MNEMONIC	TEXT NAME	MEM. REQ.
CIST*	Constant-and-Identifier Symbol table	10,000
DIMIT*	Dimension table	180
FPRAM*	Formal Parameters	180
GLTBL*	Generated Label table	30
HOLOV*	Hollerith Overflow table	180
MJSCOT*	Major Systems Communications table	N/A
ABMT1	Abort Message 1	9
ABMT2	Abort Message 2	10
CARD	Hollerith Card Image	16
CLRT	Clear and Set Codes	10
CNVRT	8090 to CMS-2 Internal Code Conversion	64
CPLTB	Unhashed CIST Codes	15
CRKT1	Data Unit Codes	37
CRKT2	Fractional Modifiers	10
CS2OT	Object-time Routines	58
CSWTG	Global CSWITCH Parameters	40
CSWTL	Local CSWITCH Parameters	40
DATCT	Data CIST Codes (Sub-table of LMTRX)	-
DBT	Debug Parameters	7
DBVAL	Debug Values (Sub-table of LMTRX)	-
DELIM	CMS-2 Delimiters	156
DELMF	Format Delimiters (Sub-table of LMTRX)	-
DELMS	Special Characters	10

* Common to all phases of the Compiler

TABLE 3-2 Phase 1 Data Base
Sheet 1 of 4

MNEMONIC	TEXT NAME	MEM. REQ.
DISH	Directory Assisted CIST Pointers	1,800
DIXB	System Index Registers (Sub-table of LMTRX)	-
DMCHT	Machine Types	63
DOT1	Options Parameters	9
DOT2	Print Options	13
DSTRG	Binary to Decimal (Sub-table of DSTRT)	-
DSTRT	Working Storage Area for Number Conversion	N/A
DTXXT	TXXT Codes (Sub-table of LMTRX)	-
DUSWK	Data Unit Sortable Code	2
FILTB	File Hardware	7
FINT	Table Element CIST Codes	15
FLACT	File Actions	6
FRMDS	Format Descriptions (Sub-table of LMTRX)	-
FRMTB	Format TXXT Codes (Sub-table of LMTRX)	-
GOTB	GoTo CIST Codes (Sub-table of LMTRX)	-
ILBUF	Intermediate Language Buffer	100
ILBUFC	(Sub-table of ILBUF)	-
IMPBF	Field/Variable Mode Buffer	4
IMPCC	IMP CIST Codes	26
IMPMC	Mode CIST Codes	26
INCHR	Input Characters	64
IVAL	MFORM Table Limits	2
LMTRX	LEVEL Matrix	55

TABLE 3-2 Phase 1 Data Base
Sheet 2 of 4

MNEMONIC	TEXT NAME	MEM. REQ.
LOOPS	Vary Loop Pointers	11
LSTTB	Status CIST Codes	4
LVLPT	LEVEL Paren table	6
MATRX	CS-1 Delimiters (Sub-table of LMTRX)	-
MFORM	MODE Format	16
MODT	Implied or User's Mode	4
NMBR	Converted Decimal Number Storage	2
NMBRT	Converted Decimal Number Storage	2
PREGS	Display Registers Format	13
PRMT	Primitive table	18
PRNVL	Mixed Expression Paren Value	24
PRTB1	Director Parent table	12
RANGC	Range Exceeded Format	12
RELOG	Relational Logical Operators	23
RQMJH	Required Major Headers	6
RTOP	IL Operator Codes	24
SCB3T	Procedure Linkage CIST Codes	6
SCNTP	Numeric Work Area	4
SCOUT	Syntactic Term Storage	8
SETB	Ranges on SETs	25
SPCND	Special Conditions	16
STRNG	Unpacked Card Images	160
SUBTB	Subsript CIST Codes	14

TABLE 3-2 Phase 1 Data Base
Sheet 3 of 4

MNEMONIC	TEXT NAME	MEM. REQ.
SYST	System Identifiers	12
TBCD	Table Varying CIST Codes	4
TBLDS	Table Delimiters	3
TBLPK	Packing Descriptors	3
TBOVT	Table Overflow Names	4
TEMP	Temporary Numeric Working Area	4
TONTO	Overlay Table SCOUT	-
TOP	Operator Push-Down Stack	100
TREG	Machine Register Codes	25
TRGT	Mixed Expression Target	48
VARTB	VARY Operator Values	25
WDLST	Word or Less CIST Codes	34
WDTP	Word Type	1
WOTPD	Word Type Indicator	1
WRK	Hashing Work Area	1

TABLE 3-2 Phase 1 Data Base

Sheet 4 of 4

- Non-global CIST and Minor SCOT. The non-global or local definitions of data structures, sub-routine names, and statement labels of each program element being compiled are placed in files on magnetic tape. These files are referenced during subsequent passes of the compiler as the associated program element is processed; i.e., as each subsequent program element is processed the non-global data for the previously processed element is overlaid by the data necessary for processing the current element. This technique reduces computer memory required by the compiler by providing only that data necessary to process the current program element.
- Intermediate Language (IL). The IL is generated on magnetic tape by the Phase 1 statement processors. The IL generated for any source program represents the program in a parsed machine independent format that allows any CMS-2 Phase 3 generator, for a specific computer, to generate the associated language for that computer.

  The IL consists of control words and operand words. The control words are subdivided into fields used to designate the class of computer operation desired such as store, branch, Boolean test, and logical test. Also, control words give the mode of interpretation of the operand words following the control word, and pointers to items of the compiler identifier tables defining selected

operand references of the parsed source statement. The operand words primarily consist of operand interpretation cues supporting the control words of the IL. The combination of control and operand words of the IL describe the basic machine independent computer operations to be performed and allow the object generators of Phase 3 to select the actual machine instructions necessary to perform the indicated operation. It is this combination of machine instructions that represents the object language form of the compiled source program.

An example of the IL generation is as follows:

Source Statement:

SET A TO B+C $

IL generation

1. Add - control word
2. CIST pointer to B - operand word, first operand
3. CIST pointer to C - operand word, second operand
4. Store - control word, previous result as second operand.
5. CIST pointer to A - operand word, first operand

The IL shown above represents the parsed form of the source statement. The first operation performed is the addition of variables B and C. The next operation is the storage of the result of the addition in variable A. The general operations, ADD and STORE indicated in the IL, are then resolved by the Phase

3 process into the actual machine instructions necessary to perform the operation on the target computer

3.2.3 Phase 3, Pass 1 Object Code Selection

Phase 3 for the CMS-2 compiler generates the final machine code of the program being compiled. The current CMS-2 compiler has several Phase 3 code generators, i.e., one for each type of computer on which CMS-2 language programs are used. When a source program is submitted to the CMS-2 compiler, the target or object computer is named. The appropriate Phase 3 generator for that computer is then used during the compile. The code generator (Phase 3) referenced in the balance of this section is for the UNIVAC CP642B computer.

Phase 3 for the CP642B computer is a two-pass operation. The first pass generates partial object code instructions and performs all allocations of the object code. The second pass then completes the object instructions and then generates the selected outputs. Figure 3-5 illustrates the processing flow of Phase 3.

Phase 3, Pass 1 produces partial object code generation of the program being compiled from the IL form of the program and definitive data collected by the Phase 1 syntactic analysis. The inputs to the Phase 3, Pass 1 process are as follows:

- Local Constant and Identifier Symbol Table (CIST) and minor Systems Communications Table (SCOT) for each element of the program being compiled (magnetic tape).

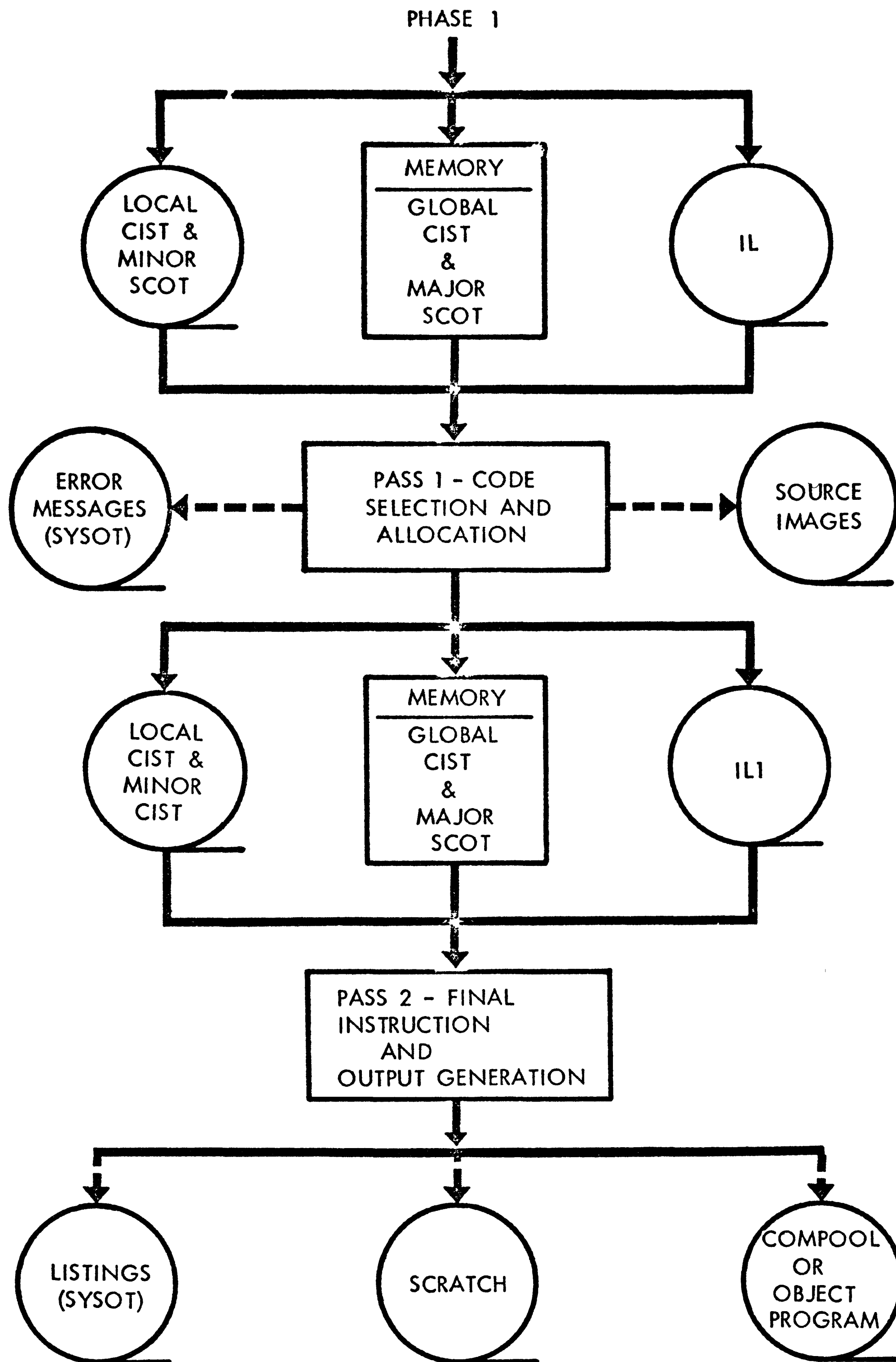


FIGURE 3-5 PHASE 3 PROCESSING

- Global CIST and major SCOT for all elements of the program being compiled (computer memory).
- Intermediate Language (IL) form of all elements of the program being compiled (magnetic tape).

The outputs from Phase 3, Pass 1 are as follows:

- Local CIST and minor SCOT for each element of the program being compiled (magnetic tape).
- Global CIST and major SCOT for all elements of the program being compiled (computer memory).
- Final Intermediate Language (IL1) for all elements of the program being compiled (magnetic tape).
- System output containing generated error messages (magnetic tape).
- Source images of the program being compiled (magnetic tape).

The CIST data generated by Phase 3, Pass 1 is the same as that received from Phase 1 with the appropriate allocation for all constants and identifiers now attached. Further discussion of Phase 3, Pass 1 data and IL1 is given in Subsection 3.2.3.4.

The Phase 3, Pass 1 (CP642B version) of the CMS-2 compiler consists of four major components: Pass 1 Control, code processors, utility routines, and the Phase 3 data base. The following paragraphs describe each of the above components of Phase 3, Pass 1.

3.2.3.1 Pass 1 Control

The Phase 3, Pass 1 Control routine is entered upon completion of Phase 1 processing. The control routine then initializes Phase 3, Pass 1 processing by loading the local CIST

and minor SCOT tables into computer memory. The IL tape is read and the appropriate processor is called to generate the partial machine codes and address allocations for the IL item. If source output of the program being compiled was requested, the card image corresponding to the IL item is dumped on a magnetic tape. As each IL item is processed the partial machine instructions are placed on the IL1 tape and another IL item is extracted. When the end of the current element is reached, the completed local CIST and minor SCOT tables are dumped to magnetic tape, and the local CIST and minor SCOT for the next element are placed in computer memeory for the processing of the IL items corresponding to the element. The Pass 1 process continues until the end of the IL tape is reached at which time control is returned to the Master Controller of CMS-2. Figure 3-6 illustrates the functional flow of the Phase 3, Pass 1 processing described above.

3.2.3.2 Pass 1 Code Processors

The Pass 1 code processors are called by the control routine to process specific IL items. There are three basic classes of code processors defined as follows:

- Declarative Code processors - which process IL items corresponding to the declaration statements of the CMS-2 language.
- Imperative Statement processors - which process IL items corresponding to the imperative statements of the CMS-2 language.
- Machine Code processors - which process IL items corresponding to direct (machine code), or mnemonic

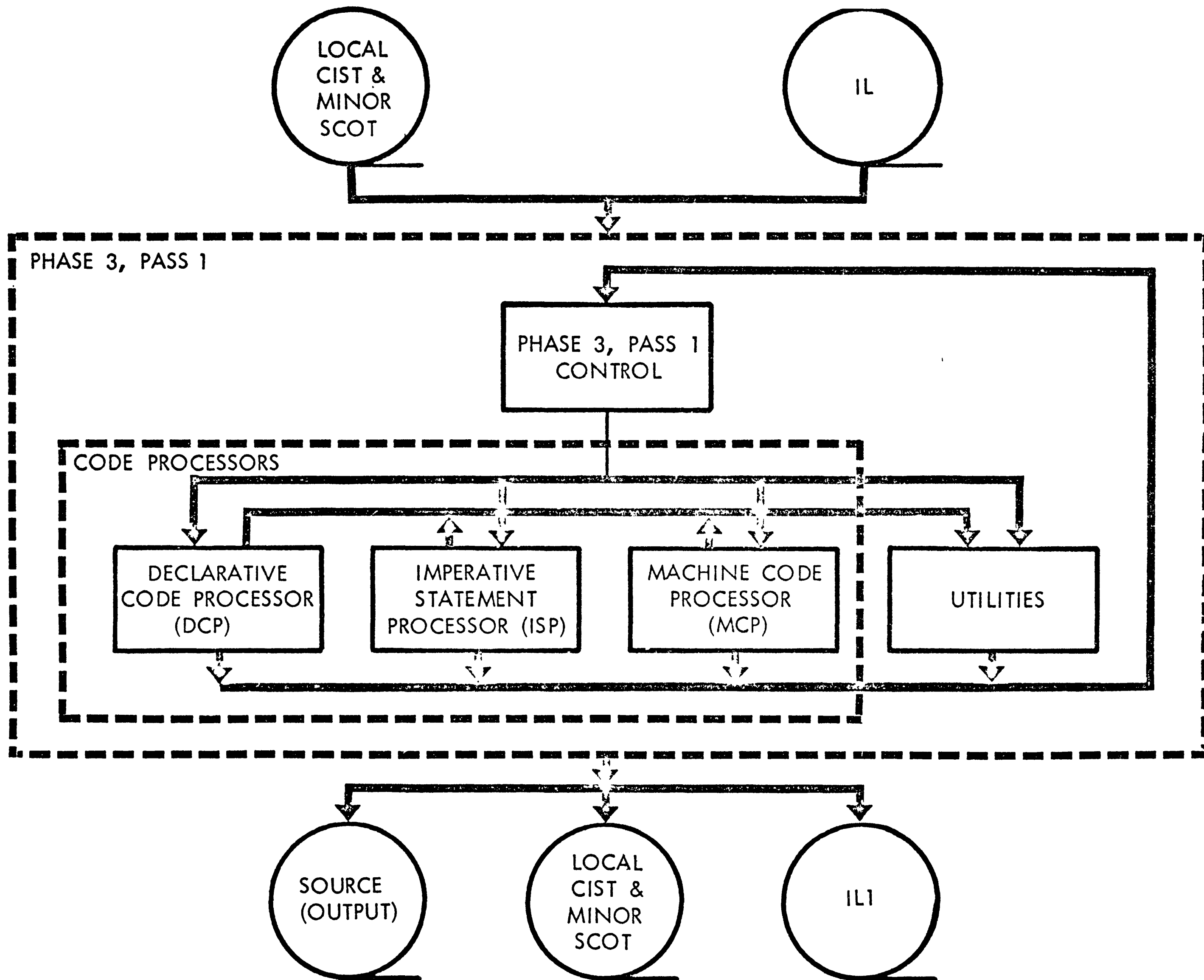


FIGURE 3-6 PHASE 3, PASS 1 PROCESSING

statements which appeared in the source program being compiled.

Each of the above mentioned processors have several associated routines which in turn utilize the provided utility routines for Phase 3, Pass 1. Table 3-3 provides a list of the Phase 3, Pass 1 code processors and the size of each. In addition, the utility routines used by each processor are also given. Where memory requirements were not available, it is denoted in the table by N/A.

3.2.3.3 Pass 1 Utilities

The Phase 3 utility routines support the code processor operations. The utility routines perform such operations as extracting IL items, formatting and writing the IL1, table searches, and miscellaneous computations. Table 3-3 lists the code processors and the required utility routines to support the code processing operation.

3.2.3.4 Pass 1 Data Base

The data base for the Phase 3 process is basically the same as for Phase 1. The Phase 3 process utilizes the data to complete the allocation process and for final object code generation. The primary data tables are the previously described CIST and SCOT tables for storage of constant and identifier symbol storage and system communication respectively. As the constants and symbols in CIST are encountered during the Phase 3, Pass 1 process the appropriate final allocation of the constant or identifier is calculated and placed in the CIST item. Table 3-4 lists the major data structures referenced during the Phase 3, Pass 1 process.

PHASE 3, PASS 1 CODE GENERATORS → / UTILITY ROUTINES ↓	TEXT - 43 (TEXT PROCESSOR)	PIERR - 32 (PHASE 1 ERROR)	P3AEND-8 (END P3, PASS 1)	DCP - 426 (DECLARATION PROCESSOR)	DCP74 - N/A (FXPT CONSTANT PRESET)	DCP75 - N/A (FLPT CONSTANT PRESET)	DCP76 - N/A (HOLLERITH PRESET)	MCP - 1092 (MACHINE CODE PROCESSOR)	PREMCP - 8 (PRE-MACHINE CODE PROCESSOR)	PROC TALK-11 (PROCEDURE TALK)	INDXP - 69 (INDEX PROCESSOR)	LAB - 33 (LABEL PROCESSOR)	IRG - 13 (RELATIONAL PROCESSOR)	IAG - 60 (ARITHMETIC GENERATOR)	ISG-443 (STORE GENERATOR)	IBG - 36 (BOOLEAN GENERATOR)	ITG - 218 (TEST GENERATOR)	GO TO PROC - 45 (GO TO PROCESSOR)	SWPSWPRC - 123 (SWITCH PROCESSOR)	STVARY - 14 (VARY GENERATOR)	ENDVARY - 130 (END VARY GENERATOR)	YRESUMG - 22 (RESUME GENERATOR)	VTESTG - (VARY TEST GENERATOR)	IMG-23 (SUBPROGRAM CALL GENERATOR)	STOPPROC - 10 (STOP PROCESSOR)	IOG - 111 (OBJECT GENERATOR)
CDOUT - 60	X																									
WRIL1 - 163	X	X	X	X	X	X	X	X	X	X		X														
CDEND - 14	X		X							X																
PCSCR - 29	X		X							X																
SORCEIO - 5	X		X							X																
PRERR - 123		X																								
DPAREA - 16	X		X							X																
GLAB - 47	X		X							X																
CSCRIO - 5	X		X							X																
CRCRCIST - 30	X		X							X																
BASEINIT - 19	X		X							X																
ABSDINIT - N/A	X		X							X																
ABSPINIT - 53	X		X							X																
RELINIT - 9	X		X							X																
DUMPQUE - 38			X						X	X		X												X		
PRECON - 39			X							X																
SCANCIST - 77			X							X																
PRECONILL - 36			X							X																
WRILIEND - 17			X							X																
GCL - 21			X					X		X			X	X	X	X	X	X	X		X		X			
DCPOOL - 45				X																						
DCPSIZ - 55				X																						
REORG - 8				X						X																X
WILIORG - 7				X	X	X	X		X			X														
GETIL - 46					X	X	X			X	X		X	X		X	X	X	X		X		X			X
JDESIG - 23								X																		
KYB DESIG - 215								X																		
MONOUNDO - 143								X																		
PKERR - 30								X																		
TRUBN - 21								X																		
TRUCN - 170								X																		
QUEBRK - 12									X	X		X	X	X	X	X	X	X	X	X	X	X	X	X		X
BINOCT - 26								X																		
SEARCHT - 56								X																		
PACK - 71								X																		
LOCAT - 62								X																		
PSTART - 52										X																
PEND - 167										X																
PEXIT - 104										X																
PRPACKL - 13										X																
PRPACKC - 6										X																
GETLAB - 15										X					X								X			X
PACKIL1 - 13										X																
PRRESET1 - 8										X																
PRACKL - 13										X																
GETMKEY - 23										X								X								
CKCON - 29										X								X							X	
IPKILQUE - 4										X			X	X	X	X	X	X	X		X	X	X	X		X
PRORG - 5										X*																
GETCISTB - 25											X		X	X	X	X	X	X	X		X		X			X
STRCISTB - 24											X						X	X					X			X
RJCONST - 97												X	X		X	X	X									X
DELQUE - 25												X		X	X	X	X		X		X					
AQERASE - 15									X	X		X	X	X	X	X	X	X	X	X	X	X	X	X		X
ISPINIT - 29													X	X		X	X		X							
SOCP - 643													X	X	X	X	X		X		X		X			X
SHIFTA - 28														X	X											
IANOPS - 50														X												
IADD - 115														X												
IAGPRE - 123														X												
IAGINCR - 63														X												
IAGBAR-29														X												
GETOP - 106													X	X	X	X	X		X		X		X			

NOTE: N/A MEANS NOT AVAILABLE

TABLE 3-3

PHASE 3, PASS 1 CODE GENERATORS/UTILITIES CROSS CORRELATION

SHEET 1 OF 3

PHASE 3, PASS 1 CODE GENERATORS → / UTILITY ROUTINES ↓	TEXT - 43 (TEXT PROCESSOR)	PIERR - 32 (PHASE 1 ERROR)	P3AEND-8 (END P3, PASS 1)	DCP - 426 (DECLARATION PROCESSOR)	DCP74 - N/A (FXPT CONSTANT PRESET)	DCP75 - N/A (FLPT CONSTANT PRESET)	DCP76 - N/A (HOLLERITH PRESET)	MCP - 1092 (MACHINE CODE PROCESSOR)	PREMCP - 8 (PRE-MACHINE CODE PROCESSOR)	PROC TALK-11 (PROCEDURE TALK)	INDXP - 69 (INDEX PROCESSOR)	LAB - 33 (LABEL PROCESSOR)	IRG - 13 (RELATIONAL PROCESSOR)	IAG - 60 (ARITHMETIC GENERATOR)	ISG-443 (STORE GENERATOR)	IBG - 36 (BOOLEAN GENERATOR)	ITG - 218 (TEST GENERATOR)	GO TO PROC - 45 (GO TO PROCESSOR)	SWPSWPRC - 123 (SWITCH PROCESSOR)	STVARY - 14 (VARY GENERATOR)	ENDVARY - 130 (END VARY GENERATOR)	YRESUMG - 22 (RESUME GENERATOR)	VTESTG - (VARY TEST GENERATOR)	IMG-23 (SUBPROGRAM CALL GENERATOR)	STOPROC - 10 (STOP PROCESSOR)	IOG - 111 ([illegible] GENERATOR)
POCTGCL - 19														X	X	X										
FFL2FX - 17														X												
FFX2FL - 7			X			X				X				X												
RJSTRING - 18																		X								
GETEMPWD - 7														X	X											
ISTRK - 63														X	X											
GETREG - 11														X	X											
READIL4 - 7					X	X					X			X												
READENDOP - 25																										
WRIL1OUT - 43	X	X	X	X	X	X	X	X		X		X														X
PWAITLAB - 58	X	X	X	X	X	X	X	X		X		X														X
UPDATE - 13	X	X	X	X	X	X	X	X		X		X														X
FIL1 - 58	X	X	X	X	X	X	X	X		X		X														X
WRIL1BC - 16	X	X	X	X	X	X	X	X		X		X														X
IPKILQS - 401										X			X	X	X	X	X	X	X	X	X	X	X	X		X
IPKILQ - 84										X			X	X	X	X	X	X	X	X	X	X	X	X		X
CKSUBC - 39										X			X	X	X	X	X	X	X	X	X	X	X	X		X
GETSUBR - 35										X			X	X	X	X	X	X	X	X	X	X	X	X		X
SBREGI - 15										X			X	X	X	X	X	X	X	X	X	X	X	X		Z
ICKRPT - 37										X			X	X	X	X	X	X	X	X	X	X	X	X		X
SQAC - 11	X	X								X			X	X	X	X	X	X	X	X	X	X	X	X		X
CHNGAQB - 321										X			X	X	X	X	X	X	X	X	X	X	X	X		X
AQMOVE - 20										X			X	X	X	X	X	X	X	X	X	X	X	X		X
AQCLEAR - 16										X			X	X	X	X	X	X	X	X	X	X	X	X		X
CLOBAQ - 21										X			X	X	X	X	X	X	X	X	X	X	X	X		X
PKAQSUB - 39										X			X	X	X	X	X	X	X	X	X	X	X	X		X
SUBCLOB - 37										X			X	X	X	X	X	X	X	X	X	X	X	X		X
QPROCSEG - 15			X						X	X		X												X		
COMPAQW - 17													X	X	X	X		X								
COMPAQB - 41													X	X	X	X		X								
CKAQSUB - 51													X	X	X	X		X								
QXFRCRD - 11			X						X	X		X												X		
QXFRINST - 32			X						X	X		X												X		
QUPBEGPNT - 61			X						X	X		X												X		
QMOVE - 19			X						X	X		X												X		
STRCISTA - 24													X	X	X		X	X								
GCKCON - 69										X								X							X	
ALIGN - 29														X	X	X			X		X					
PKFUNCT - 19													X	X	X	X	X		X		X		X			
IAGOPC - 6														X												
ISUBT-127														X												
SOCPMOD - 17													X	X	X	X	X		X		X		X			X
SOCPS - 18													X	X	X	X	X		X		X		X			X
SWMG - 28													X	X	X	X	X		X		X		X			X
SOCPBC - 74													X	X	X	X	X		X		X		X			X
SUBG - 213													X	X	X	X	X		X		X		X			X
PACKSUB - 32													X	X	X	X	X		X		X		X			X
SOCPIAS - 23													X	X	X	X	X		X		X		X			X
GETOPAD - 12													X	X	X											
BCMODC - 11														X	X											
BCMODF - 6														X	X											
BCMODOP - 43														X	X											
BCMODS - 158														X	X											
BCMTEMP - 7														X	X											
ISTRAD - 27														X	X	X			X		X					

NOTE: N/A MEANS NOT AVAILABLE

TABLE 3-3
PHASE 3, PASS 1 CODE GENERATORS/UTILITIES CROSS CORRELATION
SHEET 2 OF 3

PHASE 3, PASS 1 CODE GENERATORS → / UTILITY ROUTINES ↓	TEXT - 43 (TEXT PROCESSOR)	PIERR - 32 (PHASE 1 ERROR)	P3AEND-8 (END P3, PASS 1)	DCP - 426 (DECLARATION PROCESSOR)	DCP74 - N/A (FXPT CONSTANT PRESET)	DCP75 - N/A (FLPT CONSTANT PRESET)	DCP76 - N/A (HOLLERITH PRESET)	MCP - 1092 (MACHINE CODE PROCESSOR)	PREMCP - 8 (PRE-MACHINE CODE PROCESSOR)	PROC TALK-11 (PROCEDURE TALK)	INDXP - 69 (INDEX PROCESSOR)	LAB - 33 (LABEL PROCESSOR)	IRG - 13 (RELATIONAL PROCESSOR)	IAG - 80 (ARITHMETIC GENERATOR)	ISG-443 (STORE GENERATOR)	IBG - 36 (BOOLEAN GENERATOR)	ITG - 218 (TEST GENERATOR)	GO TO PROC - 45 (GO TO PROCESSOR)	SWPSWPPC - 123 (SWITCH PROCESSOR)	STVARY - 14 (VARY GENERATOR)	ENDVARY - 130 (END VARY GENERATOR)	YRESUMG - 22 (RESUME GENERATOR)	VTESTG - (VARY TEST GENERATOR)	IMG-23 (SUBPROGRAM CALL GENERATOR)	STOPROC - 10 (STOP PROCESSOR)	IOG - 111 (OB... GENERAT...)
IMUL - 49														X												
IAG2P - 37														X												
IDIV - 87														X												
ISTRAD - 27														X												
IAGTEMP - 28														X												
ICPL - 32														X												
IEXP - 76														X												
IABS - 77														X												
IFLOAT - 81														X												
CONOPM - 72														X												
COMAQ - 66														X	X	X	X		X							
IAOPS - 50														X	X	X										
IFPCPL - 32														X												
GETFPM- 10														X												
IFPABS - 36														X												
ERASEOP - 7													X	X	X											
BCMOD - 57														X	X											
GETEMP - 26														X	X											
ALIGNC - 39														X	X	X										
POCTXC - 7														X	X	X										
SAVEA - 6														X												
SAVEB - 11														X												
ISGPRE - 51															X											
STRALIGN - 26															X											
STROP - 51													X	X	X		X	X								
ISGRPL - 117															X											
GETSKDES - 7													X		X											
CLROP - 41													X		X											
ISGMW - 54															X											
MWLOOP - 320													X		X											
ISGETMP - 8															X											
ISGTMPR - 6															X											
RELJP - 36															X		X	X					X			X
ISPINIT - 29													X	X	X	X	X									
COMPOPS - 27															X	X										
POCTXPC - 8															X	X										
RESA - 6															X	X										
RESB - 11															X	X										
ISGBARS - 73															X											
FKVRBL - 39											X		X	X	X	X	X	X	X	X	X	X	X	X		X
GETCON - 4													X	X	X		X	X								
ALIGNH - 9															X											
IBGAND - 30																X										
IBGOR - 36																X										
IBGCP - 19																X										
IBGREL - 212																X										
IBGOPC - N/A																X										
PKCILT - 48													X	X		X	X		X		X		X			X
MKFUNCT - 31													X	X	X	X	X		X		X		X			
ITGV - 99																	X									
TSPROC - N/A													X	X	X		X	X								
BASICTSS - 76														X	X		X									
GETCISTA - 25										X			X	X	X	X	X	X	X	X	X	X	X			X
IFILESV - 29																										X
IDUSTL - 51																										X
IOFORMAT - 65																										X
IOGNAME - 127																										X
ITDUSTL - 59																										X
IIDUSTL - 116																										X
IFDUSTL - 95																										X
IVDUSTL - 44																										X

NOTE: N/A MEANS NOT AVAILABLE

TABLE 3-3
PHASE 3, PASS 1 CODE GENERATORS/UTILITIES CROSS CORRELATION
SHEET 3 OF 3

MNEMONIC	TEXT NAME	MEM. REQ.
CIST*	Constant-and-Identifier Symbol table	10,000
DIMIT*	Dimension table	180
FPRAM*	Formal Parameters table	180
GLTBL*	Generated Label table	30
HOLOV*	Hollerith Overflow table	180
MJSCOT*	Major Systems Communications table	N/A
WIL1T	Write IL1 Input table	4
CILT	Current IL table	-
POCT	Parsed Operand Control table	120
ILQUE	IL1 Code Outputs	-
TWST	Temporary Word Status table	-
AQTBL	Accumulator Status table	54
BRTBL	Index Register Status table	42
TOPT	Optimization Pointer table	-
RELJT	Relative Jump table	-
CCONJ	Converse J	-
SUBGT	Subscript Generator table	-
STRT	Store table	-
ITGT	Test Generator table	-
INDXT	Index table	-
CSITP	Table Parameter table	-
VTAB	Vary table	-
CUIR	Compiler Use table	-

* Common to all phases of the Compiler

TABLE 3-4 Phase 3, Pass 1 Data Base

The Phase 3, Pass 1 process also generates the final IL machine dependent form of the program being compiled. This IL called IL1 consists of data strings representing card images, statement labels, machine instructions or words of data. There are several formats for the IL1 items, but basically each provides the output processors of Phase 3 with the information necessary to produce the final outputs.

3.2.4 Phase 3, Pass 2 Output Generation

The Phase 3, Pass 2 of the CMS-2 compiler is the output phase. The processing receives as input the previously described local and global CIST tables, major and minor SCOT tables and the IL1 form of the program being compiled. From these inputs Pass 2 processing formats various object code (machine executable instructions), and program compilation listings. The actual outputs produced by Pass 2 are selected using a source header card read and processed by Phase 1. Figure 3-7 illustrates the structure of Phase 3, Pass 2.

Phase 3, Pass 2 is composed of a control routine, output processors, and utility routines. The following paragraphs describe the processing of each of the components.

3.2.4.1 Pass 2 Control

The Phase 3, Pass 2 Control routine receives control from the Master Controller upon completion of Phase 3, Pass 1 processing. The control routine then calls the individual output processors to produce the selected outputs for the program being compiled. Upon completion of all output generation, control is returned to the Master Controller for post-compile housekeeping.

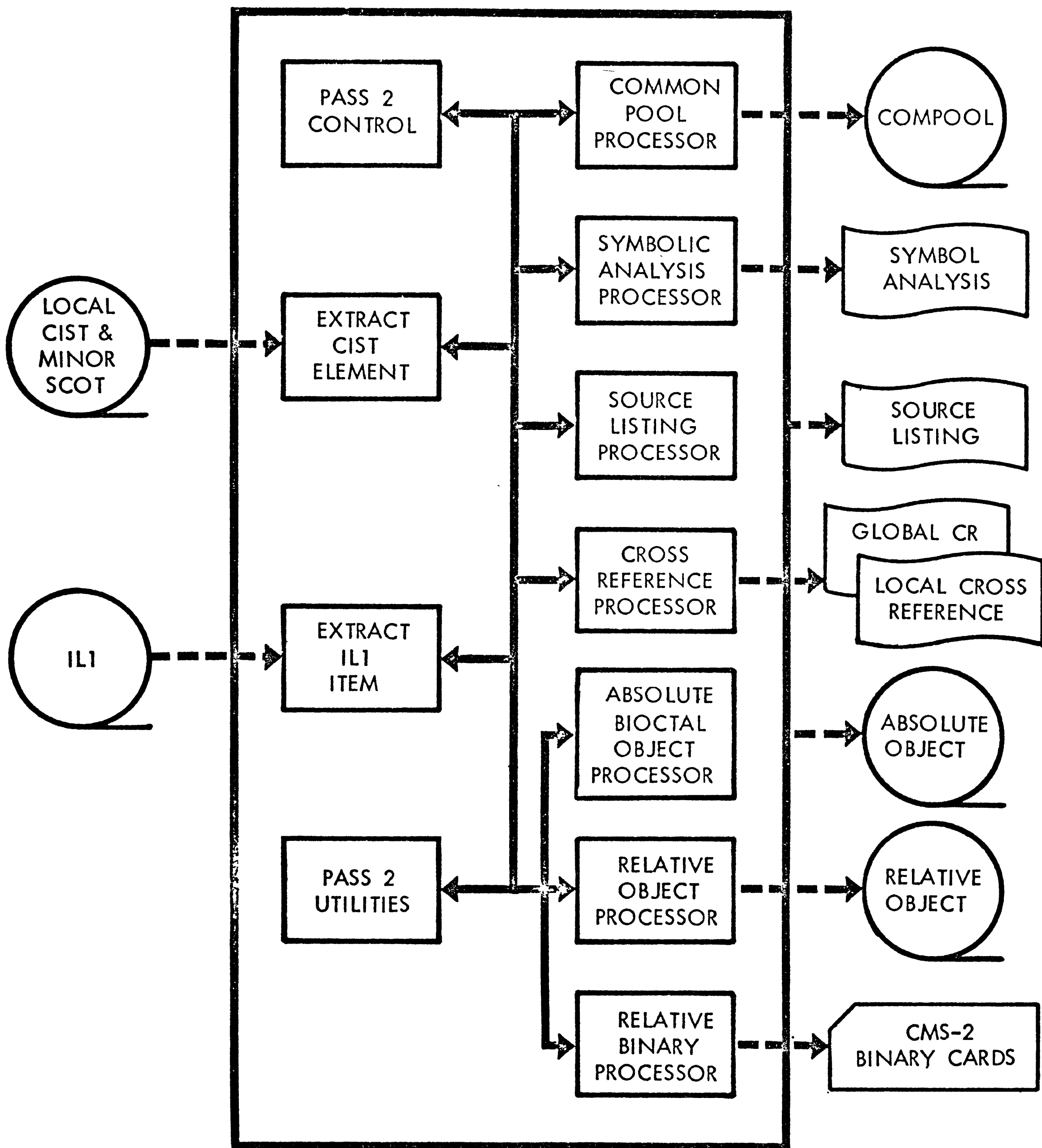


FIGURE 3-7 PHASE 3, PASS 2 PROCESSING

3.2.4.2 Pass 2 Output Processors

The Phase 3, Pass 2 output processors generate the hard-copy and object (machine instructions) output for the CMS-2 compiler. The output processor for each selected output is initiated by the Pass 2 Control routine. Once control is received, each processor runs to completion using the generated data from the other phases of the compiler to produce the desired output. Each processor is dependent on the utility routines described in the following paragraphs for specialized computation. Table 3-5 lists each processor of Phase 3, Pass 2 identifying the utility routines necessary in support of the specific output process. Explicit memory requirements for the Pass 2 output processor are not available for this report and therefore are not included in the table.

3.2.4.3 Pass 2 Utilities

The Phase 3, Pass 2 utility routines support the output processors by performing operations such as extracting IL1 items, extracting CIST items, output to magnetic tape, and code conversions. Table 3-5 identifies the utilities required by each output processor. The memory requirements for the utilities of Pass 2 are not listed in Table 3-5 because they were unavailable for this report.

3.2.4.4 Pass 2 Data Base

The Phase 3, Pass 2 data base consists of tables containing the accumulated data from the previous compiler passes. The data includes the completed global and local CIST, major and minor SCOT, and the IL1 from Phase 3, Pass 1. Table 3-6 lists

OUTPUT GENERATORS

UTILITY ROUTINES

Utility Routine	GETP3C - P3CSCR TAPE I.C.	GETIL1 - IL1SCR TAPE INPUT C	P3BPML - COMPOOL OPTION GENERATOR	P3BSA - SYMBOL ANALYSIS LISTING	P3BSY - SY AND SOURCE LISTING GENERATOR	P3BCR - CROSS REFERENCE LISTING	P3BREB - BINARY CARD OUTPUT	P3BCABS - CS-1 COMPATIBLE ABS. OUTPUT	P3BREL - CS-1 COMPATIBLE RELATIVE OUTPUT	P3BALLOCAT - ALLOCATOR	ALPHABETIZE - ALPHABETIZE LABELS
PYERR - PARITY ERROR CHECK	X										
PAIROF - PARITY ERROR CHECK		X									
COBJ - MAP TAPE OUTPUT CONTROLLER			X	X	X	X	X				
LABELCONV - INTERNAL TO 8090 CODE CONVERT			X	X	X	X	X				
SPLASH - OUTPUT CIST FOR COMPOOL			X								
P3BHEADER - LIST MAJOR HEADER				X	X	X					
PTAPR - PRINT BUFFER PRESETTER				X							
FILESPR - FILES PROCESSOR				X							
PTA - PRINT ONE SA BUFFER				X							
PTDNC - PRINT DECIMAL NUMBER				X							
PTHED - SA CLASS HEADER PRINTER				X							
PTSHC - PRINT SHORT MESSAGE				X							
FORMATSPR - FORMATS PROCESSOR				X							
FROMTWO - POSITION CHARACTERS				X							
TABLESPR - TABLES PROCESSOR				X							
PTDIM - TABLE DIMENSION				X							
PTIND - MAJOR INDEX PRINTER				X							
STRING - CHAR STRING MOVING				X	X	X	X				
SWITCHESPR - SWITCHES PROCESSOR				X							
AREA - FORMAL PARAMETER LOCATOR				X							
AREAPR - PARAMETER PRESETTER				X							
VARIABLEPR - VARIABLE PROCESSOR				X							
PROCFUNCPR - PROC/FUNC PROCESSOR				X							
LOCINDEXPR - LOC-INDEX PROCESSOR				X							
ADDPOS - LISTING ADDRESS POSITIONER					X						
BINOCT - BINARY TO OCTAL 8090 CODE				X	X	X					
FORMATTER - CARD IMAGE PRINTOUT FORMAT					X						
HOLCONV - HOLLERITH CODE CONVERSION					X		X	X	X		
IL1OPADR - OPERAND ADDRESS EVALUATOR					X	X	X	X	X	X	
IL1OPSCAN - OPERAND ADDRESS LOCATOR					X	X	X	X	X	X	
REMNEGEN - REVERSE MNEMONIC GEN.					X						
YOPPOS - POSITION OPERAND					X						
TEMPOV - OVER FLOW MESSAGE PRINTER						X					
CISTEQFND - EQUALS LABEL FINDFR						X					
BOPOUT - BINARY OUTPUT							X				
CLEARTAB1 - CLEAR TABLE							X				
DACOUT - DAC CARD FORMATTER							X				
EDCPROC - EDC CARD FORMATTER							X				
ERCOUT - ERC CARD FORMATTER							X				
EXTERNAL - EXTERNAL REFERENCE CHECKS							X				
GETNXT - GET NEXT CIST ENTRY							X				
INSTC - PARAMETER PRESETTER							X				
RELOCPROC - RELOCATION BIT POSITIONER							X				
TSSUB - TSF CARD FORMATTER							X				
P3BINST - OUTPUT INSTR. IMAGE GEN.							X	X	X		
EXTERNPROC - ERC CARD PROCESSOR							X				
P3BCOMPC - CS-1 COMPATIBLE HEADER & EOF								X	X		
P3BCS1OUT - CS-1 OUTPUT CONTROLLER								X	X		
RELCHARSET - RELOCATION CODE POSITIONER									X		
GENERLAB - GENERATED OPERAND ADDRESS					X	X	X	X	X	X	
ALPHABETER - INTERMEDIATE CIST ALPHABETIZER										X	X
CNVRTSRT - CONVERT TO SORTABLE CODE											X
CNVRFSRT - CONVERT FROM SORTABLE CODE											X

Note: Memory requirements for generators and utilities are unavailable.

TABLE 3-5 PHASE 3, PASS 2 OUTPUT GENERATORS/UTILITIES CROSS CORRELATION

MNEMONIC	TEXT NAME	MEM. REQ.
CIST*	Constant-and-Identifier Symbol table	10,000
DIMIT*	Dimension table	180
FPRAM*	Formal Parameter table	180
GLTBL*	Generated Label table	30
HOLOV*	Hollerith Overflow table	180
MJSCOT*	Major Systems Communications table	N/A
DELTA	CIST Codes for SA Class	-
CHR	Character table	-
ARET		-
FILTB	File table	-
NAMEQ		-
FLCNQ		-
USESQ		-
EMBRO		-
CS1TA		-

* Common to all phases of the Compiler

TABLE 3-6 Phase 3, Pass 2 Data Base

the tables referenced by each output processor during the associated generation. The memory requirement for the Pass 2 tables was unavailable for this report.

3.3 Summary

The CMS-2 compiler is a multitable-driven compiler that uses internal tables to control syntactic analysis and parsing of the source input statements. The parsing algorithm used in the compiler is "Reverse Polish String Notation" which allows machine instructions to be generated in the correct order. This means that the computer operations necessary to evaluate complex expressions from a source statement are generated in the correct logical sequence according to standard algebraic rules.

The CMS-2 compiler is a two phase compiler, where the first phase (Phase 1) performs syntactic analysis and parsing of the source input producing an intermediate language (IL). The second phase (Phase 3) performs actual machine code generation and produces the compilation outputs. (Phase 2 of the compiler is a proposed optimization pass that has not been implemented).

The tables presented in this section indicate the component breakdown of the CMS-2 compiler and show the interdependent relationships existing between the major operations (processors) and supporting operations (utilities) and data designs. In the following section these tables are used to show if and how the CMS-2 compiler design can be implemented on the AADC.

APPENDIX B

LIST OF ATTENDEES

1. P. Andrews - NAVSHIPS
2. D. D. Achterberg - Hughes Aircraft
3. R. Balestra - NAVAIR
4. R. Y. Beesburg - NADC
5. E. Bersoff - Logicon
6. B. Blair - FCPCP
7. P. Brady - NADC
8. C. H. Bremer - NAVCOSSACT
9. C. D. Caposell - NAVAIR
10. V. Cerf - UCLA
11. M. Cove - SYSCON
12. LCDR J. L. Grandall, Jr. - NAVORD
13. A. J. Deerfield - Raytheon
14. CDR R. M. Dove, Jr. - OPNAV
15. CAPT R. A. Dunning - FCPCPAC
16. L. D. Egan - Logicon
17. M. Ellis - PRD
18. R. B. Engelbach - SAMSO
19. R. S. Entner - NAVAIR
20. R. G. Estell - FCPCPAC
21. R. S. Firey - NAVMAT
22. LT J. R. Foster - AFAL
23. N. Frost - ASNAG

24. G. G. Gallagher - SAMSO
25. R. Gauthier - RLG Associates
26. A. Glista - NAVAIR
27. D. Haratz - USAECON
28. J. Henderson - Logicon
29. R. Hong - Grumman Aerospace Corp.
30. J. Ihnqt - NRL
31. R. Jenkins - SYSCON
32. R. Kazerman - ITT
33. S. G. Kennedy - NAVMISCEN
34. E. Kitterman - NAVCOSSACT
35. J. J. Lavoie - CSC
36. E. Lee - NASA/MSC
37. F. J. Lueking - NAVAIR
38. C. F. Mattes - NADC
39. LT(jg) L.C.G. Miller - FCPCLANT
40. J. D. McGonagle - Burroughs
41. R. E. Nimensky - SDC
42. J. P. O'Brien - CSC
43. Y. Patt - ECOM
44. C. B. Peters - Informatics
45. CAPT D. C. Peterson - USAF/AFSC
46. R. Peretti - Grumman Aerospace Corporation

LIST OF ATTENDEES - continued

47. F. R. Reinert - NADC

48. G. A. Rischall - Hughes Aircraft Co.

49. R. L. Samtmann - NADC

50. B. H. Scheff - PRD

51. B. Shay - NRL

52. L. Shirley - Informatics

53. C. F. Showalter - NAVAIR

54. J. F. Smith - Honeywell

55. W. Smith - NRL

56. J. Stiles - PRD

57. K. P. Thompson - NRL

58. K. L. Thurber - Honeywell

59. J. Trimble - ONR

60. B. Wald - NRL

61. J. A. Ward - NAVORD

62. A. Wenk - Westinghouse

63. B. A. Zempolich - NAVAIR

NOTES