PROCEEDINGS of the FIRST SPACEBORNE COMPUTER SOFTWARE WORKSHOP

20–22 SEPTEMBER 1966

SPONSORED BY THE AIR FORCE SPACE SYSTEMS DIVISION AND THE AEROSPACE CORPORATION

HELD AT THE AEROSPACE CORPORATION, EL SEGUNDO, CALIFORNIA

PROCEEDINGS PUBLISHED BY SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE., SANTA MONICA, CALIF.

PROCEEDINGS OF THE FIRST SPACEBORNE COMPUTER SOFTWARE WORKSHOP

Sponsored by the
Air Force Space Systems Division
and
The Aerospace Corporation

Held at the
Aerospace Corporation
El Segundo, California

on
20-22 September 1966

Proceedings published by the
System Development Corporation

THEME

The United States Space Program is investing an increasing proportion of its funds and reliance on the digital computer, both on the ground and on board the spacecraft. In the on-board case, this has been made possible by the rapid advances in miniaturization which have led to the achievement of the computing power of a medium-sized general purpose ground computer within the physical dimensions of a desk calculator. But what is the situation in the programming of these computers? Has the software technology made a corresponding advance to exploit the new hardware? What has it done to minimize errors, effort, time and cost? Is it helping to narrow the gap between the application problem and the computer? What potential has it acquired to influence the course of future hardware development?

The objectives of this workshop are: first, to encourage the exchange of experiences and ideas among spaceborne software specialists; second, to assess the state of spaceborne software in the context of software technology as a whole; and third, to initiate a search for definitions and guidelines for application in future spaceborne software development.

CONTENTS

*This document was not available at the time of publication of the proceedings. See note on referenced page.

SESSION 4 - LANGUAGE AND PROCESSOR CONSIDERATIONS FOR SPACEBORNE SOFTWARE

*This document was not available at the time of publication of the proceedings. See note on referenced page.

SUMMARY OF SESSION 1

by

Major M. A. Ikezawa
Air Force Space Systems Division

This session was intended to keynote the workshop with a set of observations by computer personalities from three institutions which have worked closely with Air Force problems. It was also intentional that the three speakers represent different subsets of the computer world.

The first speaker, Ladimer J. Andrews of the Aerospace Corporation, provided the original stimulus from which the idea of the workshop evolved. The comments he made were meant to be a re-creation of that original stimulus which he provided about two years ago.

The second speaker, Thomas B. Steel, Jr., of the System Development Corporation, provided a provocative critique of spaceborne programming from a broader software viewpoint. His well articulated observations elicited much comment during the workshop.

The last speaker, Dr. Barry W. Boehm of the RAND Corporation, brought out some implications of aerospace computation in general.

The session was opened by Col. D. V. Miller, Vice-Commander of the Air Force Space Systems Division.

(page 4 blank)

AERO/SPACE SOFTWARE IN PERSPECTIVE

by

L. J. ANDREWS

Aerospace Corporation

INTRODUCTION

Several years ago it became apparent that the software costs associated with aerospace equipment would soon approach the hardware costs. Order of magnitude decreases in component costs coupled with more demanding and sophisticated functional tasks has accelerated this tendency. What we would like to foresee as an output from this workshop is an exposition of current software problems, a dissemination of current practices, and inter-agency discussions leading to the means and steps required to either effect a common usage of aerospace software, or provide techniques for the expeditious generation and validation of aerospace software. Whichever of these two approaches is favored certain criteria of success are evident. Specifically, the actual cost of the generated and validated program should be lower than present methods allow, the time to produce the program and effect changes should be reduced, our confidence in the validity of the flight program should be increased, the procedures or techniques used should contribute to a growing body of knowledge, and the methods advocated should provide a framework for orderly progress in keeping with state-of-the-art advances.

APPROACH

While it is not my intention to presuppose an outcome from this workshop, an intriguing concept that satisfies the success criteria is that of an interservice library of programs oriented toward the specialized needs of flight computers for aircraft and space vehicles.

machine, with some difficulty, in a higher language that is very nearly machine independent. This latter feature represents a formidable task especially when the present specialization of aerospace computers is considered. But as in vehicle and mission independence so also is progress being made toward machine independence. Because of the tremendous strides being made in semiconductor technology, the austere functional capability and the sparsity of parts formerly required for reliability have been relaxed and near future flight machines are of a much more general purpose nature. But beyond extending the sophistication of airborne computers there are two trends in newer machine organizations that considerably aid the cause. Paradoxically these trends are in opposite directions. The first is a trend to effect airborne and commercial instruction set compatibility and the second is a trend to provide a problem-oriented machine instructions. Both of these trends closely couple to the business of this workshop; the former for the near term and the latter trend becoming of importance as the software goals, the possibilities, and the detailed paths of progress become better defined. For this reason we would like to anticipate that the results of this workshop, and others to follow, can have a profound influence on machine organizations of the future. To view this influence more pragmatically, a constant problem of the machine manufacturer is to define a spectrum of requirements upon which to base his next generation of airborne computers. Because the requirements for military systems are usually not well known to industry far in advance, are subject to redirection, (and is some cases vanish) the computer supplier has ample opportunity to make very costly mistakes.

The concept of a modular library of programs tends to desensitize the mission requirements from the machine organization to the benefit of both the supplier and the customer. The notion is that the excellent measure of machine

SOFTWARE STATE-OF-THE-ART

(A Summary of Mr. T. B. Steel's paper
by Mr. H. Ilger of the
System Development Corporation)

Software is more than just the computer program. It is the interfacing function between hardware and skinware -- the latter, of course, being the man referred to in man-machine relations. Software, then, means the programs, the procedures, and the arrangement and format of input data.

The principal object of most software work is the object program. In this respect, it is interesting to note a familiar comment that 85% of all NELIAC code produced consists of the MELIAC compilers.

The language of software is the language used to communicate with the machine. At first though, it would seem that our natural language would be ideal for this. We are a long way from this stage at present.

The early languages used in programming were symbolic assembly languages with a single, one-for-one representation of the binary machine code. This permitted the user to defer or ignore details of assignment of locations. As more automated techniques developed more clerical and bookkeeping tasks were performed for the programmer.

Next, languages became more like mathematical notation, such as FORTRAN. By 1958, compilers were available which produced somewhat poor code. Then came ALGOL and its derivatives -- MAD, JOVIAL, NELIAC and others.

There have been many developments in the past ten years in procedure languages. These are languages in which one describes the processing steps needed to solve problems. That is, you solve the problem logically. That may change, and is in the process of changing now, in a direction I will refer to a little later.

I would like to make some comments on language standardization. ALGOL has generated many dialects, as have all the languages. There has not been

and guidance. Space will come to look more like ground-based systems. In 5-10 years, we will have the equivalent of ground-based computers in space.

In conclusion:

- The main differences between ground and space are reliability and validation.
- The Information Processing community tolerates a wide amount of unreliability.
- The costs of getting all the errors out is so high, we can and do risk it.
- We need to get ways to check out systems.
- We need to be able to describe what we want to do.
- Automated checkout is needed.
- We need to be aware of the implications of failure.

(page 12 blank)

CURRENT TRENDS IN AEROSPACE COMPUTATION AND SOME IMPLICATIONS

Barry Boehm*

The RAND Corporation, Santa Monica, California

INTRODUCTION

In this talk, I will point out what appear to me to be some salient trends in aerospace computation, and indicate some possible windfalls or pitfalls which may await the alert or unwary spaceborne software practitioner.

One trend which I wish I could guarantee is that expressed in a recent Los Angeles Times article [1], citing salaries of $10,000 a year for beginning programmers and $25,000 a year for experienced ones, and quoting the president of Digitek: "The richest man on the earth in the year 2000 will be a programmer."† Although the figures appear to be somewhat inflated, there's a trend we'd all like to participate in!

---

*Any views expressed in this Paper are those of the author. They should not be interpreted as reflecting the views of The RAND Corporation or the official opinion or policy of any of its governmental or private research sponsors. Papers are reproduced by The RAND Corporation as a courtesy to members of its staff.

This Paper was presented at the SSD/Aerospace Workshop on Spaceborne Computer Software at Aerospace Corporation, El Segundo, California, 20 September 1966.

†In one sense, this quote may be true because almost everyone, rich or poor, may be "programmers" by that time. Even in its more straightforward interpretation, though, the statement is worth a minute's thought: programmers are often very close to fresh, critical information--a proximity which has been the key to the development of many famous fortunes.

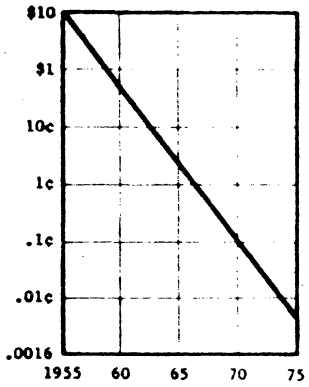


Figure 1

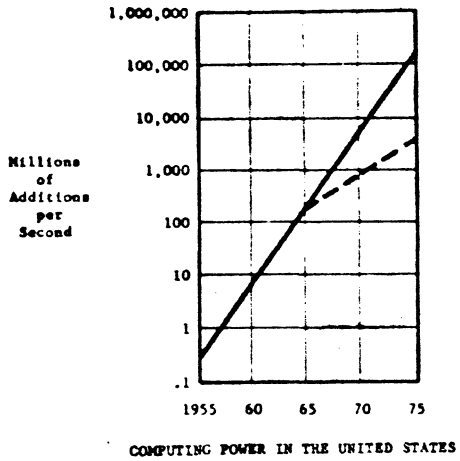


Figure 2

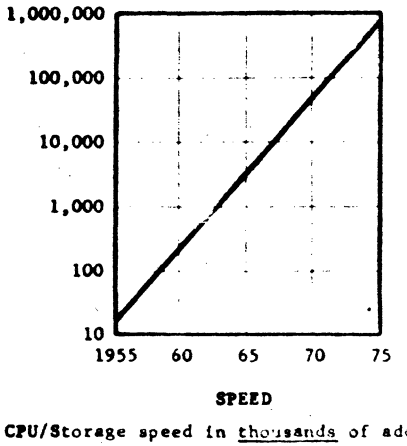


Figure 3

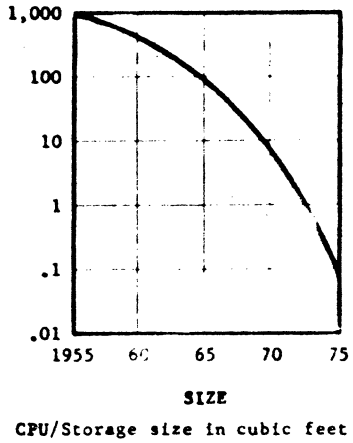


Figure 4

of these devices with each other and with CPU memories. New terminal devices employing electronic and photo-optical methods are being developed, providing an input-output capability which is not only faster and more reliable than electromechanical devices, but often also, as in the case of graphic input devices such as the light pen and RAND Tablet, a more natural way to communicate.

Also, new methods of organizing CPUs are maturing, particularly associative memories and multiprocessors. The Westinghouse "Solomon" computer will be capable of performing approximately 1000 operations simultaneously; Boeing's proposed Information Systems Simulator (ISS), possibly 100,000 simultaneous operations.

RANGE OF APPLICATIONS

A fundamental law of human nature is that people are insatiable: no matter how much they have, they always need a little more. This law has many forms (e.g., Parkinson's Second Law: "Expenditure rises to meet income"; Second law of Communications Economics: "Demand eventually exceeds channel capacity") and is at least as old as Eve and the Garden of Eden.

It is certainly true so far for computing power. Every increase in computer capability is matched by an increase in the number, size, and complexity of problems people need to have solved. Today's computer is being used to optimize processes, such as boost trajectories and chemical reactions, which its predecessor of a few years ago could barely simulate. And more complex processes in biology and meteorology which were infeasible

software remains essentially a handicraft industry. Programs are produced, one at a time and with great personal care, like Harris tweeds or fine musical instruments; some indeed have considerable aesthetic appeal.* The concept of interchangeable parts is little used; indeed, there are very few programming standards upon which to base interchangeability. Some significant progress has been made on assemblers and compilers, but even here provisions for the inevitable debugging phase are primitive. Conversion of programs from one machine to another can be extremely dreary and time-consuming work. Programming documentation is spotty: many programs are useless if their author isn't around to explain them, and most operating systems operate at a fraction of their power because people can't penetrate the murky documentation that surrounds them. Is it any wonder that software is scrambling to keep up?

IMPLICATIONS: PROGRAMMING STANDARDS

In the current software situation, then, it is evident that the individual, firm, or country which finds more efficient ways of producing software will be in a position of considerable advantage. Thus, there is a strong need for more natural problem-oriented languages, programming languages and operating systems with more helpful debugging features, acceptable programming standards, and more understandable documentation.

---

*Save your old hand-coded computer programs. Some connoisseur in the year 2066 will pay a fortune for them.

and leave the earth at a distance of exactly one astronomical unit from the sun.

Some Guidelines

Here are some guidelines which I have found most useful in the programming process [8]:

1) Wherever possible, use machine-independent programming languages;
2) Encourage logical simplicity over slight gains in program efficiency;
3) Develop programs in modular form;
4) Document with frequent examples;
5) Anticipate the direction of extensions to programs, and provide a clean, well-defined interface for incorporating them into a program.

Reference 9 contains a number of further useful guidelines.

Documentation

Some automated aids to documentation are becoming available, such as Raytheon's program analyzer and the NOTS flowchart producer, but people are still the key to good documentation. Anyone who has attempted to plow through IBM's five-foot shelf of System 360 documentation is aware that quantity is no substitute for quality. I would like to suggest a rule which I have found fairly successful. This is the

> Golden Rule of Documentation: Document unto others as you would have others document unto you.

Think about it. How often do you use a double standard for documentation?

decided to make the computer less accessible to engineers. The management found a marked tendency for engineers to use old designs and their extrapolations because computer programs were available to analyze them, rather than inventing new designs. The computer software system, often in very subtle ways, can stimulate mediocrity rather than creativity.

What Can We Do About It?

There are no panaceas, but there is one discipline being developed which can shed light on such problems. This is systems analysis, best described in Ref. 10, but difficult to summarize because it is less a body of standard techniques than a state of mind. The systems analysis approach commits the analyst to a careful definition and continuous re-examination of his project's goals, to ensure that the problem he solves is the appropriate one. It also involves him in a continuous confirmation of the relevance of his efforts to the achievement of his goals, to guarantee that his solutions solve the problems he wants them to solve.

The key words are continuous and relevance. It is all too easy to abrogate one's responsibility for maintaining relevance, somewhere along the line, and to find brilliant solutions to the wrong problems. It is especially easy for computer software specialists to do so, and especially dangerous, as the software system's limitations quickly become project limitations--often in a way that the user, who is taking a lot on faith, doesn't fully recognize.

REFERENCES

1. Sederberg, Arelo, "Software: Hard Knock for Computer Industry," Los Angeles Times, Sec. 1, pp. 1-3, September 18, 1966.

2. Armer, Paul, "Computer Aspects of Technological Change, Automation, and Economic Progress," in Technology and the American Economy, Appendix Vol. I, The Outlook for Technological Change and Employment, U.S. Government Printing Office, February 1966, pp. I-205 through I-232; also, The RAND Corporation, P-3478, November 1966.

3. Ware, W. H., Future Computer Technology and Its Impact, The RAND Corporation, P-3279, March 1966.

4. Rajchman, J. A., "New Trends in Computer Memories," Electronic Design, Vol. 12, No. 1, January 6, 1964, pp. 53-59.

5. Adams Associates, Inc., Computer Characteristics Quarterly, July 1966.

6. Hobbs, L. C., "Impact of Hardware in the 1970's," Datamation, Vol. 12, No. 3, March 1966, pp. 36-44.

7. Baker, R.M.L., and M. W. Makemson, An Introduction to Astrodynamics, Academic Press, New York, 1960.

8. Boehm, B. W., "Development and Usage of the ROCKET Trajectory Program," Proceedings, ICRPG Working Group on Design Automation, Chemical Propulsion Information Agency Publication No. 92, September 1965; also, The RAND Corporation, P-3187, August 1965.

9. Bemer, R. W., "Economics of Programming Production," Datamation, Vol. 12, No. 9, September 1966, pp. 32-39.

10. Quade, E. S. (ed.), Analysis for Military Decisions, Rand McNally, Chicago, 1964.

SESSION 2

The State of Spaceborne Software

Chairman: Dr. Walter A. Sturm
Aerospace Corporation

(page 28 blank)

SUMMARY OF SESSION 2

by

Dr. Walter A. Sturm
Aerospace Corporation

The three case studies presented in this session included three basically different types of digital computers, and different time periods as well. All of the presentations were based on a general discussion of the systems' organizations from the viewpoint of software development. The first paper emphasized the development of the software itself, and the simulation tools; the second paper stressed the application of the management tools which were used to control the software development; the third paper described the problems associated with validating the flight software.

The fourth paper summarized the data collected by SDC in the course of their industry-wide survey. The main point was that spaceborne software development encompasses a striking assemblage of individual problems, each of which is familiar to the experienced programmer in one application or another.

(page 30 blank)

THE GEMINI COMPUTER SOFTWARE SYSTEM

by

P. P. MOONEY

IBM Federal Systems Division

INTRODUCTION

In March of 1962, IBM was awarded a contract for development of the Gemini digital computer and system integration of the Inertial Guidance System (IGS) which included the computer, an inertial platform, a keyboard and display unit, and an incremental velocity indicator. Sometime later in the program, an Auxiliary Tape Memory was added to the system.

The computer developed by IBM performs guidance and navigation calculations, based upon sensor inputs, for Ascent Guidance, Rendezvous, Orbit Navigation, Orbit Determination, Orbit Prediction, Touchdown Predict and Re-entry. It also performs calculations for astronaut display, receives telemetry commands from the ground and sends IGS telemetry information to the ground.

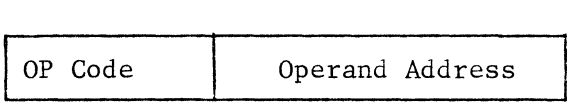
All of these computations are done in real time and therefore, the programming of the computer is a task which is complicated by the attendent problems associated with a real-time system. The following sections in this paper will discuss the Gemini computer briefly and the programming considerations emanating from the resultant design. Then, some of the programming problems which were encountered will be presented along with the solutions implemented to overcome them. Next, will be a discussion of the Gemini software tools which were developed and used very successfully in the course of developing the flight programs for 12 Gemini flights.

THE GEMINI COMPUTER

The Gemini Computer is a general purpose, binary, fixed-point computer, designed for missile or space vehicles. The memory is random access.

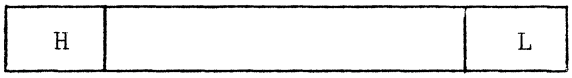
There are basically two types of words in the computer. Namely, an instruction word and a data word. Instructions are 13 bits in length as shown in the figure below:

Instruction Word



A data word is 26 bits long, occupying Syllables 0 and 1 of a word. All data is in two's compliment so there is no real-sign bit.

Data Word



Since all 4 of the operation code bits of the instruction are used there are 16 operation codes in the computer. They are:

ARITHMETIC	LOGICAL	BRANCH	I/O
CLA	SHF (SHIFT)	HOP	CLD (CLEAR DISCRETE)
ADD	AND	TRA	PRO (PROCESS I/O)
SUB		TMI	
RSU (REVERSE SUBTRACT)		TNZ	
MPY			
DIV			
SPQ (STORE PRODUCT/ QUOTIENT)			
STO			

In order to solve this problem, the ATM was developed. It is a magnetic tape with the following characteristics:

length	-	525 feet
speed	-	1½ inches/second
capacity	-	approximately 100,000 13-bit words

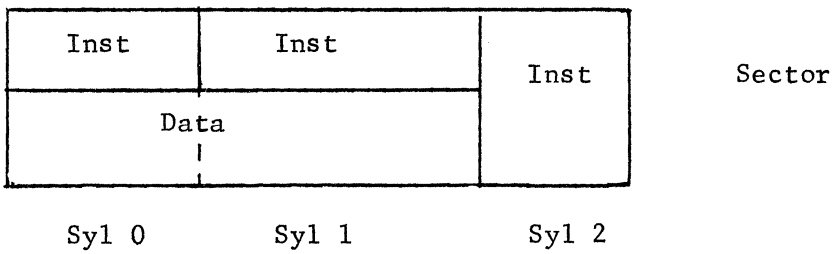
Since the ATM was incorporated in the middle of the project, the basic ground rule for interfacing with the computer was minimum hardware changes to the computer. This placed the burden of reading from the tape upon the program. The tape and computer run asynchronously so that the critical timing was accomplished within the program. It was required to sense the information coming from the tape, and perform the necessary logic to store the data and return to look for more data before it passes under the read heads.

Since there are no record markers on the tape, programs on the tape are identified by a program word which proceeds each program. Again, it is up to the tape read subroutine to recognize the proper program number in order to begin reading information from the tape into memory.

The design of the ATM has significant influence on the programming of the Gemini Computer. Since Syllable 2 of the memory is Read Only, and cannot be written into with data from the tape, it was necessary to make the best use of this part of memory by putting a Hardcore program in this area. That is, it is necessary to define a set of fixed subroutines which do not change for the remainder of the project. The reason for this will be shown later. Therefore, all subroutines such as trigonometric, log, tape read, telemetry, and keyboard were judiciously chosen to be placed in the Hardcore. These are all subroutines which are non-mission dependent and will never change.

These instructions will be in various subroutines which will naturally be in several different sectors. Therefore, the Residual sector becomes full quite fast. In the following section, is a discussion on how the Assembly Program greatly helps to overcome the problem of sectorized memory, and how to effectively use the feature of the Residual sector. But, it also requires work on the programmer to make maximum use of the Residual memory. This is done by time sharing many of the locations in that sector. That is, by use of the Psuedo Operation, SYN, a core location in Residual sector can be used by several different modes or subroutines. It is up to the programmer to know which variables to time share. But, by very judicious choices and thorough testing of the program, this concept has been successfully employed and helped to maximize the use of available memory.

Also, relative to data access is a consideration of how to allot data and instructions in each sector. A typical sector layout is shown in the figure below:



Since Syllable 2 is a Read Only portion of memory, it almost always contains instructions. Only through use of a special mode (Halfword Mode) data can be read from Syllable 2. Syllable 0 and 1 can be either instructions or data as can be seen. By observing this typical layout, one can see that there are two considerations that the programmer must be concerned with. First, how many locations to allow for data. Unfortunately, this process could not be automated in the assembler because the sequential execution of instructions is within a syllable until a HOP instruction is encountered (otherwise the instruction address register will count up to 256 and reset

but allows absolute coding when desired by the programmer. In addition, it also provides the capability for several psuedo operations which significantly ease the task for the programmer.

It is designed such that the programmer give minimum attention to the problems inherent in a sectorized memory. In addition, extensive post-processor and pre-processor information is provided on the printout to aid in optimizing the source program as well as preventing unnecessary delays in searching through the listing during critical test and simulation periods. Another of the very useful features of the assembler is an Edit pass, which precedes each assembly. This allows the programmer to make changes to the source program through correction cards, and makes effective use of tape operation.

The unique feature of the Gemini assembly program that is the key to optimum memory usage, is the Automatic Storage Allocator which is accomplished between pass one and pass two of the assembly process. The design of this concept relieves the programmer from specifying which variables and constants must be put in the Residual sector, and which ones should be in the other sectors. It operates on a priority scheme which arrives at the most optimum use of the Residual memory.

CAPABILITIES OF THE ASSEMBLER

In order to give a deeper appreciation of the Gemini Assembler, it is well to present a more detailed discussion of the various capabilities which the assembler provides to the programmer. The capabilities which are in the resultant assembler were arrived at by implementing, where possible, as many of the useful features found in FMS, by creative thinking on the part of the assembler designers, and by requests and suggestions from the application programmers. Several of the ideas have been carried over into the design of the assembly programs for the IBM Saturn V computer. Similarly, it lacks some useful features such as MACRO's which would be very useful,

- A bit table by sector showing the locations in memory used by the program.

PSUEDO OPERATIONS

The Gemini Assembler provides several psuedo operations for the programmer.

- ID Card - Printed on each page of the printout
- Comments Cards - Information to be printed out at the place in the listing indicated by the sequence number. For instance:

ARC TANGENT SUBROUTINE

- Origin (ORG) - This code causes the succeeding instructions to begin at the sector, syllable and word coded in the location field of the card. (Similar to FMS.)
- Reserve (RES) - By use of this psuedo operation, the assembler is told how many locations within a syllable it may use for instructions before starting in the next syllable.
- Block Started by Symbol (BSS) - This specifies a block of storage which has the left-hand symbol of the BSS card as the symbolic name of the first word in the block. The location field of the BSS card tells how many locations to be reserved (similar to FMS).
- Equate (EQU) - This psuedo operation specifies the sector syllable and word to which the left-hand symbol of the psuedo operation is to be assigned. (Similar to FMS.)
- Decimal (DEC) and Octal (OCT) - These psuedo operations define the value to assigned to left-hand symbol. (Similar to FMS.)
- Synonomous (SYN) - The card specifies the names of variables and constants in the location field which are to be assigned the same memory location as the left-hand symbol of the card. (Similar to FMS.)
- Pointer (PTR) - This psuedo operation generates a 13-bit word containing the Gemini memory address of the variable given in the location field. This is very useful in setting up modified address instructions symbolically.

The next items to be assigned locations are those HOP constants which the programmer has defined. (Normally the assembler generates HOP constants. But the programmer can define his own for performing address modification.)

Then, the constant and literal lists are searched to assess their usage as determined during pass one. If they are used in more than one sector, they must be duplicated and assigned locations for the appropriate sectors. If a sector overflows an attempt is made to place the remaining constants or literals in Residual. Otherwise, the symbol becomes undefined. The variables that are time shared (SYN) are also processed in the ASA phase. It should be noted here, that this phase of the assembly process is done while tapes are being rewound from pass one, and therefore, adds no time to the assembly process.

Pass two of the assembly follows immediately behind the ASA phase. It performs the following tasks:

- Generates the necessary HOP instruction and HOP constants from program continuity. (These were identified in pass one.)
- Completes the core map for each instruction
- Encodes and writes out listing with error conditions
- Computes values for constants, HOP constants, literals, and places them in core map
- Writes out on tape the core map, the variable, left-hand symbol and constant names, and assigned address for use by the simulator.

SYSTEM DESCRIPTION AND OPERATION

The Gemini assembler system is a tape-oriented system, requiring two channels on the 7094 II. A self-loading system tape containing the assembler and punch program is the heart of the system. The output of the assembler is an eight-file tape, which is used as input to the simulator and punch programs.

resulting from that operation. It also checks for any violations of programming ground rules and overflow conditions. This gives the programmer diagnostic-type information not available if the program were to be debugged on the Gemini computer. The Simulator Program provides an accurate simulation of the functional operation of the Gemini computer and associated I/O equipment.

To greatly enhance the debugging and analysis task, the Gemini Simulator will also output any or all the following information for a given simulation run made on the 7090:

- Full trace - provides a printout of every executed instruction, the contents of the accumulator resulting from that operation, and any diagnostic-type information.
- Flow trace - prints out every executed TRA type instruction, thereby providing a program flow debugging feature with minimum printout.
- Store trace - prints out every executed STO instruction.
- Spot trace - gives a full trace over selected areas of the program thereby providing a selective debugging feature with minimum printout.
- Core dump - this gives an octal dump of all core locations at any given instant of execution selected by control cards.
- Symbolic dump - this prints out all memory locations which have been assigned a symbolic name together with the decimal value of the contents of that location.

The simulator is used in one of two ways. Either as a static simulator or as a dynamic simulator. As a static simulator it receives inputs on cards and computes the outputs for a test case. these results are then printed out for analysis.

- A flexible system capable of providing good turn around time from mission planning to program validation.
- Timely and effective documentation.

REFERENCES

1. Description of Gemini Digital Computer, IBM Technical Report No. 65-554-0089, 22 November 1965.

2. IBM 7090 DPS Gemini Assembler and Punch Program Reference Manual, IBM Technical Report No. 66-538-01, 4 January 1966.

3. The Gemini Simulator Reference Manual, IBM Technical Report No. 64-542-011B, 15 March 1965.

(page 48 blank)

Maneuvering Ballistic Re-Entry Vehicle

by

P. L. Phipps
UNIVAC Corporation

(This paper was not available at the time of publication of the proceedings.)

SOFTWARE ASPECTS OF THE TITAN III PROGRAM

by

FRANK R. TROEGER

Logicon, Inc.

A number of related launch vehicles go under the general designation of Titan-III. The particular program to which this paper refers is the inertially guided version of Titan-III, the Titan-IIIC. This vehicle consists of a two stage core adapted from the Titan-II ICBM, to which are strapped two large solid rocket motors, each capable of 1.2 million pounds of thrust. Atop the core is the third stage body, the so-called transtage, whose special features include two restartable main engines and a separate attitude control system for use during coasting periods.

Development of the Titan-IIIC was begun approximately four years ago. The objective of the program was to develop quickly and at reasonable cost a capability to inject into earth orbit a variety of military payloads. The key words of the program were to be versatility and quick reaction capability.

Early studies of likely T-III missions had determined that a self-contained inertial system was better suited to the guidance problem than a ground-based radio system. As key components of such a system and to be compatible with the broad objectives of the program, an all attitude inertial measurement unit and an easily programmed random-access computer were indicated. However, when it was shown that the already existing Titan-II guidance system could be adapted to be technically adequate for the Titan-III missions, its lower developmental costs and smaller schedule risks prevailed.

Except for the incorporation of an environmental control system and for some minor modifications to the logic of the computer, the Titan-III guidance system is essentially the same as that used in Titan-II. Its principal subsystems are a three-gimbal inertial measurement unit (IMU) built by the AC Electronics Division of General Motors, and a digital computer built by the Federal Systems Division of IBM.

analog or bilevel data. This implies that the programmer must ensure that words required to be telemetered appear in the accumulator at specified word times. The problem is aggravated in certain missions because of the need to compress bandwidth at altitudes in excess of 3000 miles, bringing with it a further 4 to 1 data reduction and telemetering of only 8 words of every 64 appearing in the accumulator.

4) In multiplication, the least bit of both operands is truncated off prior to initiation, thereby introducing a small bias in the product. Furthermore, because of the algorithm used, there is the characteristic that a multiplicand of zero may either form a product of zero or a LSB =1 depending upon the presence of a one in a particular bit of the multiplier.

There are a number of other hardware-imposed restrictions, that together with those that have been listed place a premium upon the programmer having had prior experience with this machine. As a rough estimate, it probably takes six months to break in a new programmer for this machine -- and it might be said that prior programming experience on large general purpose machines does not significantly alter this estimate.

It might be suggested that such an environment forms the ideal justification for a compiler. This possibility was looked into more than five years ago on the Titan-II program by both IBM and TRW. While it was difficult to prove, at that time it was estimated that the efficiency of compiled code would be less than 80% that of good hand coding. This factor plus the uncertain costs of developing such a compiler caused the idea to be dropped. On Titan-III the idea has from time to time been renewed, but the conclusion has been the same. The fact that much of the time even with hand coding there has been a shortage of storage has not added to the attractiveness of a compiler development.

There are a number of other hardware features which when coupled with operational requirements tend to make the programmer's job a little

- SERIAL, BINARY, COMPLETE-VALUE, FIXED POINT
- 6000 RPM DRUM MEMORY

  51 tracks - 9792 instructions

  12 tracks - 768 fixed constants

  3 tracks - 192 target constants

  3 tracks - 192 words data storage

  1 track - 8 word A revolver

  1 track - 3 word F revolver

  1 track - 2 word revolver - accumulator

  3 tracks - M, P, Q revolvers

  1 track - Accelerometer processor

- CONCURRENT ARITHMETIC - 25 BIT DATA WORD

add, subtract, transfer	6400 per second
multiply (24 bit accuracy)	533 1/3 per second
divide	128 per second

- INSTRUCTION REPERTOIRE

  5 arithmetic : +, -, x, ÷, ∧

  17 airborne output: 14 discretes, 3 dc analogs

  3 conditional branch

  7 transfer

  1 attitude data processor

  3 miscellaneous

  1 instruction modify

TABLE I

TITAN-III MGC CAPABILITIES

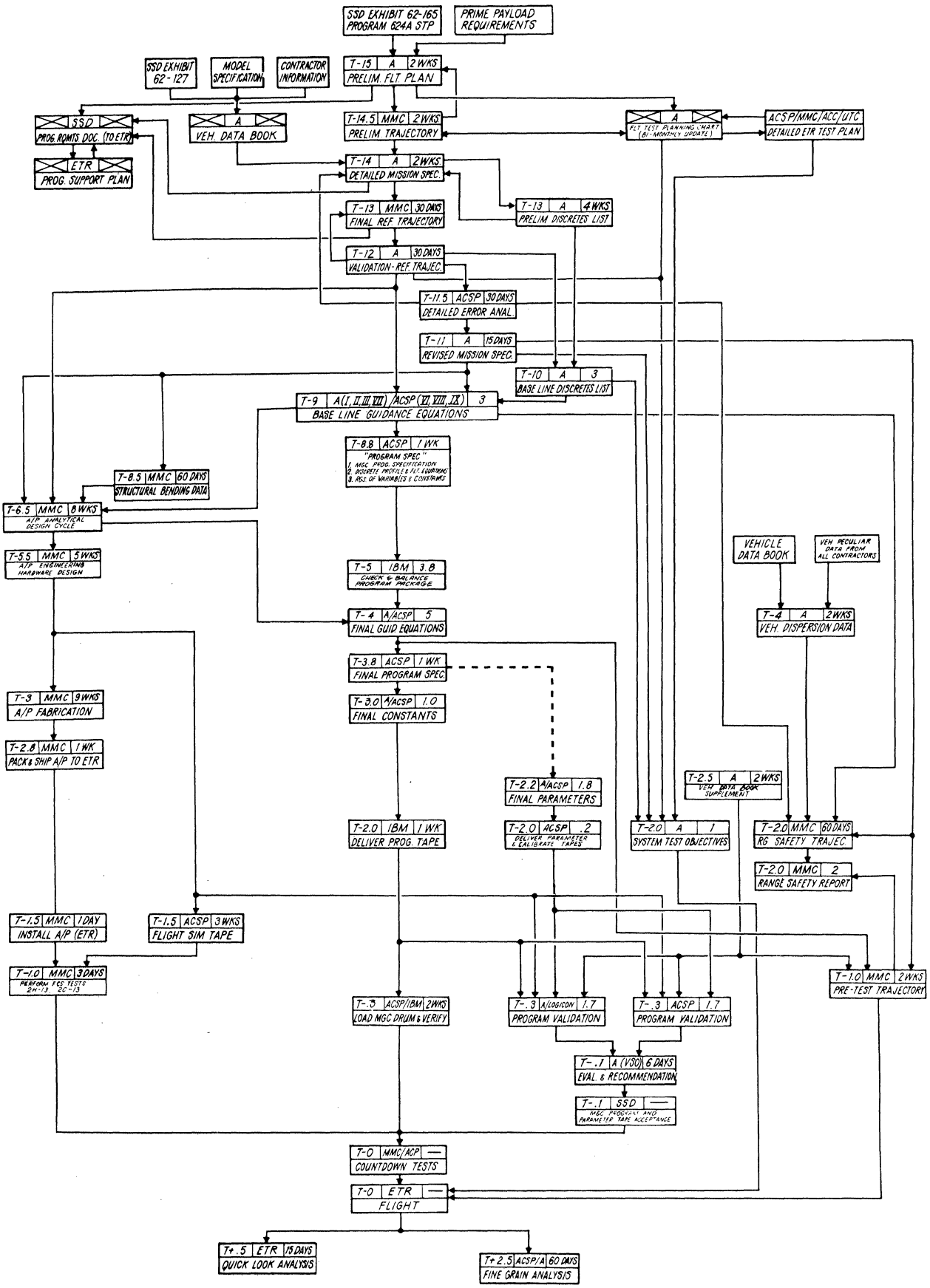


FIGURE 1. MISSION PROGRAMMING

In an attempt to minimize the impact of such changes the plan calls for the delivery of a "check and balance" program package, thereby permitting the equation-writer to assess the alternatives himself. This package also permits a check on the general progress of programming and is an opportunity to call out misunderstandings if any exist.

Validation

Validation is really the answer to the question: is the MGC software ready to fly? This prime question can be broken down into three distinct criteria which can then be tested separately:

1) Are all mission and systems requirements accurately reflected in the programming specification?
2) Are the requirements of the programming specification met by the MGC coding?
3) Are all mission and systems requirements satisfied by the MGC coding?

It should be noted that only the third criterion constitutes a completely necessary and sufficient basis upon which to fly. While unlikely, there may be subtle effects of MGC program mechanization which prevent criteria 1) and 2) from implying 3). The question might then be asked, why test to anything but criterion 3)? This is the "black box" approach to validation. In certain applications it may have merit; on T-III we have not used it. There are several reasons:

1) In its pure state nothing is known about the innards of the program. Consequently it is necessary to devise tests to uncover all possible failure modes. For missions of the duration and complexity of T-III, this would be a horrendous task.
2) A lesson learned by our hardware brothers years ago is that you can learn much more about how something will work - and how it will fail - by opening it up than you can by probing it from the outside. Double-whiskered diodes, solder balls, graphite particles are all good examples.

On T-III a two pronged approach is taken to validation. Both AC Electronics and Aerospace/Logicon have independent votes. Thus far there has not been disagreement in their final recommendation to SSD. The methods

of the equation is checked automatically. These programs are valuable tools in that they can handle relatively coarse coding. They are also relatively inexpensive to run.

Both approaches are valuable, and it is interesting to note that they tend to complement, not compete with one another. At any given point in a validation,about the same number of errors have been found by each. It is as though they were starting at opposite ends of the program, both working toward the middle.

At both AC Electronics and Aerospace/Logicon, closed-loop interpretive simulations form the test of criterion 3). By the time these tests are run, the program is well shaken down, and these runs constitute merely final confidence tests.

The ground programs are handled in a somewhat different manner. Simulations of the green-light countdown procedures are run by Aerospace/ Logicon to test the accuracy of the compensation for accelerometer and gyro terms and to test the proper use of these values after the transition into the flight program. AC Electronics relies primarily on lab tests of the programs on an MGC and its related ground equipment. Again, it has been found that the methods tend to complement each other.

If the flight be unsuccessful, the cause may or may not be tracked down. Regardless, the chances are that the failure will have software impact. There may be the attempt to build in a capability to detect and correct the failure. Or there may be the attempt to have the software better adapt to the particular failure condition so that for a given mission even if the prime objectives cannot be completely satisfied, there is the opportunity to accomplish certain secondary objectives.

I cannot stress too strongly this particular facet of spacecraft software design. In essence, this ability to apply some intelligence in adapting to the conditions encountered is something relatively new to unmanned systems. Credit for its incorporation into the T-III program goes largely to R. V. Erilane of Aerospace. Certainly there was little such capability applied to missile systems, derived rate for radar systems having both range and range-rate is one of the few instances of adaption in the face of malfunction.

Some people will argue that a set of guidance equations need only perform adequately in the presence of parameter deviations less than $\pm 3\,\sigma$ from nominal. Academically and statistically, this is a reasonable position. But, talk to a propulsion engineer about the standard deviation in differential thrust buildup of two engines, fired after some time in orbit. He may describe what he hopes will happen, but he may also interject that once in a while one engine may burp in coming on - but that really isn't a propulsion malfunction. After a couple of thousand in-orbit firings of at least a couple of hundred engines, he will have better data or maybe a better engine. Meanwhile though, it sure would help if guidance could tolerate engine burps. And yes, it would be nice if the engines do fail to come on, that we go through say, one more cycle; if that is unsuccessful, then try something else.

Note that here I am speaking of an adaptive ability that is quite different from that which has been more generally studied in the computer field, viz.

Case History

In the Titan-III program thus far there have been a number of flights for which software development and software validation have been the major elements in the critical path. Flight Plan II-1 was one such instance. Preparing for an early February launch, a series of short-multiply errors was found in the minor loop just a few days before Christmas. The balance of the airborne coding pivots on the coding of the minor loop. Thus a relatively minor error resulted in more than 900 instruction changes. These changes were coded within 10 days, and by resorting to validation running concurrent with debugging, the flight date was met.

A somewhat different problem arose in the case of Flight Plan VII A. Because of

a) the plane change (s) involved,

b) the high terminal accuracy required

c) the presence of IMU gimbal stops and

d) the need to point the telemetry antenna back at earth

this flight plan had always been considered quite complex. Difficulty in freezing mission and payload requirements plus the desire to minimize storage requirements led to late definition of the equations. Table III lists the dates on which equation changes were incorporated into the program specification. These dates are to be compared against the milestone dates listed in the mission programming. For all intents and purposes, change 3 formed the baseline equations; this at T-6 months compared to the T-9 date in the mission programming chart. The addition of a requirement to toast the solar cells of the payload (to keep them from being too cold to operate when the payload was released into orbit) was the largest single change from that point. Gwen a normal-sized program, even that change would not have been large, but IBM was already approaching 90% of total instruction capacity. Past this point coding efficiency generally drops off so badly that this last 10% is considered unusable space. Nonetheless IBM did manage to squeeze in the

DATE	CHANGE NUMBER
15Feb65	PRELIMINARY ISSUE OF BLK 1-4, 20
24Mar65	BL - INCOMPLETE
5Apr65	1 - MINOR ADDITIONS
9Apr65	2 - MODIFIED FIXED POINT NAVIGATION
11May65	3 - COMPLETE ISSUE OF BASE LINE
8Jun65	4 - NEW DRIFT MATRIX
16Jun65	5 - MODIFIED BLK 16, 7, 9, 18
21Jun65	6 - REISSUE OF FIG. 1
12Jul65	7 - CORRECTED BLK 7, ADDED INITIALIZATIONS
20Jul65	8 - REVISED TOASTING LOGIC
5Aug65	9 - REVISED FIG. 1, BLK 5, 6, 8, 19
17Aug65	10 - REISSUE OF COMPLETE DOCUMENT
27Aug65	11 - IBM LOGIC CHANGES
2Sep65	12 - IBM SCALING CHANGES
7Sep65	13 - REVISED BLOCK 9
29Oct65	14 - FINAL CONSTANTS, PARAMETERS
4Nov65	15 - CHANGED 13 PARAMETERS, 1 CONSTANT
16Nov65	16 - CHANGED 34 PARAMETERS
7Dec65	17 - CHANGED 35 PARAMETERS, 4 CONSTANTS
9Dec65	18 - MODIFY BLK 9, FIG. 1, 4 PARAMETERS
14Dec65	19 - CHANGED 9 PARAMETERS
21Dec65	LAUNCH

TABLE III

FLIGHT PLAN VII

• Mode Sort and Idle	118	
• Sequence Check	62	
• Trouble Test	82	
• Common Routine	20	
• GGE Routine	27	
• Timing Margin	24	
• Punch	208	*
• Count-down Steering	57	
• Ready	733	
• Target Track Sum Check	130	*
• Data Load	338	*
• Flight Equation Test	118	**
• Vehicle Simulation Test	130	**
• Hold	12	
	2059	

* Deleted for Flight Plan VII (1433)

** Modified to fit for Flight Plan VII

TABLE V

INSTRUCTION COUNT
GROUND PROGRAM

solenoid valves. Further plans are being made to incorporate the function of Stage O thrust vector control within the computer. Being new, the requirements of such innovations are likely to be difficult to pin down. Not only airborne requirements have to be established, but also the method of pre-launch checkout. Until such requirements can mature, they are going to be subject to change.

In addition to the problems of the new software functions, there is likely to be a new problem with the old functions, namely there will likely be a greater sense of freedom in requesting changes. Heretofore, the nature of the computer tended to inhibit requests for all but essential changes. In jest, it has frequently been said that what every mission planner really wants is the capability to write up his mission requirements immediately before launch and to have the computer convert these requirements into a flight program on the spot. The fact of the matter is that as we begin to develop both hardware and software capable of satisfying such desires, it becomes increasingly difficult to convince people that we are not already there.

In summary, it is not at all clear that the change to a new computer will necessarily bring with it any marked improvements in software schedules, cost, or quality. These improvements could be achieved by cutting back on the new functional requirements imposed on the MGC, but only at the expense of optimal system design, and only at the expense of the versatility and flexibility that are key to the Titan-III program.

A SUMMARY OF CURRENT SPACEBORNE SOFTWARE SYSTEMS

by

A. E. Tucker
System Development Corporation
Santa Monica, California

INTRODUCTION

This summary of current spaceborne software systems is based upon a state-of-the-art survey conducted by the System Development Corporation (SDC) between September of 1965 and March of 1966. This survey included a search of current literature plus personal contacts with 19 different organizations which were actively engaged in the spaceborne software field. The objectives of the survey were to determine the nature of the products being produced, how these products were being produced, and the problems being experienced in their production.

Preliminary results and conclusions from the survey were published as Volume IV of the working papers for this Workshop. Final results and conclusions are expected to be published in December and will include material from this Workshop where applicable.

The missile and space programs, for which software information was collected during the survey, are listed in Figure 1. The level of detail of the data collected on the software aspects of each of the programs listed was not the same. Time limitations did not allow a detailed study of every program. However, based upon the availability of information, specific missile and space programs were selected as baseline systems against which the data from other systems could be compared. The principal baseline system for the survey was the Titan III program.

Figure 2 presents the definition of a spaceborne software system used by SDC in conducting the survey. Qualifying statements concerning this definition are discussed in the working paper previously referenced and will not be presented here.

A SPACEBORNE SOFTWARE SYSTEM INCLUDES ALL ACTIVITIES INVOLVED IN PLANNING, DESIGNING, DEVELOPING, TESTING, VALIDATING, AND DOCUMENTING THE DIGITAL COMPUTER PROGRAMS THAT WILL BE USED BY THE DIGITAL COMPUTER ON BOARD THE SPACE VEHICLE.

Figure 2. Definition of a Spaceborne Software System

A simple explanation of this software system concept is as follows: From an environment which defines requirements, capabilities and resources, a development process is established to produce an end item. In producing the end item, problem areas exist due to both the environment and the production process.

The preliminary survey results and conclusions I will present today will be discussed in respect to the four basic areas of the conceptual software system. The order in which they will be presented will be: (1) the development process; (2) the end item; (3) the environment; and (4) problem areas.

THE DEVELOPMENT PROCESS

A. THE SIX DEVELOPMENT PHASES

Survey results indicated that the current spaceborne software development process is composed of six distinct phases. Figure 4 presents these six phases in their sequential order of performance. The activities performed in each phase can be briefly summarized as follows:

1. Mission Planning

Mission planning is the phase in which the mission requirements for the on-board data management system are established. In general, these requirements are for a total missile or space project irrespective of the number of flights within the project. Flight specific mission specifications for the on-board data management system are established on the basis of a project's general mission requirements and the flight specific objectives. Flight specific data management mission specifications normally include:

a. The objectives to be achieved;
b. The functions to be performed;
c. Data inputs--sources, rates and nature;
d. Outputs--rates and accuracy;
e. Descriptions of the nominal flight path and mission phases;
f. Allowable tolerances from nominal conditions.

Mission specifications are generally first published as preliminary documents in order to accommodate review by interfacing contractors. Final mission specifications may be issued as new documents or as modifications to the preliminary documents.

2. Formulation of Computer Program Requirements

The objective of this development phase is to define the digital computer program specifications. These specifications generally include the following:

a. All mathematical equations;
b. Functional units or blocks;
c. Operational sequence;
d. Initializing process;
e. I/O operations;
f. Timing operations;
g. Precision and scaling of all variables and constants;
h. Symbol definition;
i. A description of the total program.

Computer program specifications include both the flight program and all supporting ground programs.

3. Computer Program Design and Development

This development phase includes the activities of computer program design and design verification, coding, debugging and testing of written code and the establishment of performance and acceptance criteria for the written programs. The output of this activity is a set of digital computer programs which constitute the total software package necessary to achieve the on-board data processing requirements of a specific flight. The total set of programs include those necessary for in-flight functions plus those required for pre-launch functions such as calibration, self-checking, etc.

and Checkout is apparently considered by some organizations to be a continuation of the validation phase or as an implementation phase which follows the development.

The organizational structures employed in the development of spaceborne software were found to range from a single contractor having total responsibility for all phases, to a set of four different contractors each responsible for a specific phase, or phases, of the process. Survey results did not appear to be affected by the organizational structure being employed. No specific problem area or characteristic of the development process could be directly related to the organizational structure.

Figure 5 presents a composite view of the distribution of effort; i.e., time and resources, expended in the spaceborne software development and implementation process.

PHASE	PER CENT OF EFFORT (TIME AND RESOURCES)
1. MISSION PLANNING	10%
2. FORMULATION OF COMPUTER REQUIREMENTS	30%
3. COMPUTER PROGRAM DESIGN AND DEVELOPMENT	45%
4. COMPUTER PROGRAM VALIDATION	13%
5. & 6. PRE-LAUNCH CHECKOUT POST-LAUNCH EVALUATION	2%

Figure 5. Distribution of Effort in the Development and Implementation Process

The values given are averages from the survey data and do not represent any one specific system. These average values are in good agreement with estimated and predicted values that were available prior to conducting the survey.

type simulations employed. Digital simulation programs for the following items were generally found to exist for every missile or space program surveyed:

a. The flight vehicle and its dynamics;

b. A space operating environment including an earth model;

c. Flight vehicle hardware and systems which interface with the on-board data system;

d. The on-board computer.

The detail to which simulation programs were written appeared to be a function of the software development phase in which they were employed. In general, simulations utilized in the Mission Planning phase are less detailed than those used in the validation phase. It was evident during the survey that each contractor or subcontractor engaged in spaceborne software activities has developed his own particular set of simulation programs to meet his specific needs. This observation leads to the conclusion that considerable duplication of effort exists in the total process. This apparent duplication of effort results from the fact that the simulation tools produced, by any one contractor, are tailored for operation in that contractors large-scale data processing facility. In general these simulation programs will not operate in another facility without considerable modification. Thus, even though simulations of the same type are performed by various contractors (or different groups within a single contractor organization), each is usually independently developed. The duplication of effort is advantageous from a point of view providing a cross check on results, but also leads to problems when variations in results are due to differences in the simulation programs used.

which will affect software development are being followed in the development of these advanced machines. The first is that of compatability in which the instruction set of the flight computer will be identical to a sub-set of ground-based machines. Thus, the software support tools available for the large ground-based computers will, in general, be applicable for the spaceborne computers. The second concept is that of micro-programmed computers such as the Instruction Computer, currently under development by the RCA Corporation. Both of these concepts will be discussed in papers to be presented later in this Workshop.

3. Interleaved Activities

The third distinctive feature of the spaceborne software development process is that of interleaved or concurrent activities. This feature was very prominent throughout the survey. Concurrency in activities is required to meet imposed schedules and in order to incorporate changes which are constantly being introduced. To allow the interleaving of activities within the total development process, spaceborne software development is subjected to considerable subdivision in terms of functional blocks and units.

The three distinctive features of the development process just discussed represent areas for which substantial survey data was obtained and indicate areas where significant problems are encountered. These problems will be discussed later.

THE END ITEM

Figure 7 presents a listing of the distinctive features of the end item produced by the development process.

C. SCIENTIFIC DATA PROCESSING AND COMPUTATION

In general, most of the functions currently being assigned to the on-board computer, particularly for systems in the early state of development, are engineering type computations. The extent to which this type of data processing and computation exists was found to be directly related to the evolutionary state of the particular missile or space program. Following the initial development stage, the type of functions assigned to the on-board computer continuously progress toward more purely data processing functions while retaining the initial scientific type functions. Examples of this are system and subsystem monitoring, signal conditioning, operational mode options, etc. The point to be made here is that while most current spaceborne software is initially oriented toward scientific type processing, the mix between this type of processing and pure data processing changes as more experience with the missile or space program is obtained.

D. MACHINE LANGUAGE

This characteristic relates to the code by which the end product is written. With few exceptions, the language in which today's spaceborne software is written is the actual machine language or a symbolic language which is very close to machine language and is assembled into machine code on a one-for-one instruction basis.

E. SINGLE PERFORMANCE

While most of those involved in spaceborne software planning and development stated that the on-board software for a particular missile or space program was originally to be developed to meet all operational requirements with only minor modifications and changes in constants, these same individuals stated that this objective is rarely met and that a new end item is required for each vehicle flight. While portions of a specific flight software package are used on subsequent flights, significant changes or new requirements demand a flight

1. SYSTEM ALIGNMENT, CALIBRATION
2. TARGET INSERTION, VERIFICATION (WEAPONS)
3. LAUNCH STATUS (GO -- NO GO)
4. NAVIGATION AND GUIDANCE
5. FLIGHT CONTROL
6. ARMING AND FUSING (WEAPONS)
7. SEPARATION ERROR CORRECTION
8. UPPER STAGE BURN
9. VEHICLE STATUS AND MISSION CONTROL
10. SENSOR CONTROL
11. POSITION PREDICTION
12. MAN-MACHINE COMMUNICATION
13. DE-BOOST SEQUENCING
14. LANDING POINT PREDICTION

Figure 8. Current Spaceborne Computer Functions

The environmental sources for on-board computer functions are the original project objectives, uncertainties in hardware design, and new requirements established on the basis of previous experience. Figure 9 presents a composite of the survey data percentages of the flight computer program which is used to perform specific functions.

1. THE ON-BOARD COMPUTER
2. THE GROUND-BASED COMPUTER
3. OTHER FLIGHT HARDWARE

Figure 10. Hardware Capabilities and Limitations

1. The On-Board Computer

The first of these is the on-board computer. Operational speed, methods of timing, memory size, and instruction sets were identified as the major on-board computer capabilities in respect to software. While the operating speeds and timing capabilities of most currently used on-board computers were reported as adequate to meet initial project requirements, these same factors were identified as the limiting factors in allowing the programmer to accommodate new requirements.

It is apparent that the operational speeds of on-board computers continue to increase with time. Sequentially addressed machines are the slower of the systems currently in existence with randomly addressed machines having speeds approaching those of ground-based systems. Sequentially addressed machines contain an inherent timing ability which in most cases is available to the programmer. The newer randomly addressed type computers, although containing a clock for operational use, do not generally provide the programmer with a capability to establish timed program cycles. The inability to establish adequate timing cycles is becoming a problem area as new functions are required to be packed within a given time span.

The size of the available on-board memory and its relationship to spaceborne software development is self-evident. The size of the memories being carried into space today has substantially increased

3. Other Flight Hardware

As indicated previously, many times the flight hardware with which the on-board data management system must interface is not completely defined at the time software development must start. When this condition exists, the software is developed on the basis of estimated constants and variable ranges. In many cases, substantial changes in the software are required when the actual hardware characteristics are established.

C. ENVIRONMENTAL RESOURCES

Environmental resources constitute the third area of the spaceborne software system environment. Resources can be divided into three types. These are shown in Figure 11.

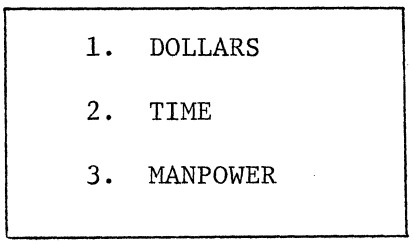


Figure 11. Environmental Resources

Dollar information per se was not collected during the survey. However, dollar values can be inferred from the time and manpower information collected.

Figure 12 presents time and manpower examples of specific end items produced for the missile or space programs named. The figure presents the calendar time versus the man months of effort used in the design, production, checkout, and validation of the end item. These activities account for approximately 58% of the total time and effort required in spaceborne software development.

Keeping clearly in mind that the values indicated by the figure are for a particular flight specific end item, an indication that the state of development of a missile or space program affects the software development can be seen. The Gemini and Titan III programs must be considered to be in a substantially different state of development than the LEM and Saturn V programs.

The availability of manpower to perform the programming for spaceborne software was indicated as becoming a problem area. The shortage of available personnel is evident from the number of classified advertisements requesting people for this type of activity. Those surveyed stated that even an experienced programmer requires six months of training before he is qualified for expert work. In addition, such an individual normally performs this type of programming for only a two-year period.

PROBLEM AREAS

Having discussed current spaceborne software systems in respect to the development process, the end item and the environment, let us turn our attention to the subject of problem areas.

Figure 13 presents seven significant problem areas which were identified on the basis of the survey results. All seven of the items listed should be recognized as related to one or more of the points previously discussed. However, I would like to briefly discuss them as they are listed.

1. CHANGING SPECIFICATIONS
2. AVAILABLE LEAD TIME
3. INSUFFICIENT MEMORY
4. PROGRAMMING INNOVATION REQUIRED TO ACCOMMODATE REQUIREMENTS
5. INCOMPLETE SPECIFICATIONS
6. LACK OF PROCESS DEFINITIONS AND CONTROLS
7. COMMUNICATIONS BETWEEN INTERFACING GROUPS

Figure 13. Spaceborne Software System Problem Areas

It appears that currently spaceborne software program specifications are established primarily by those oriented toward the functional hardware systems. Such specifications require considerable modification to be adequate for the design of the software.

F. LACK OF PROCESS DEFINITIONS AND CONTROLS

This problem area is concerned primarily with the management aspects of the process. The starting and end points of the six phases of the development process discussed are inadequately defined for managerial and contractual purposes. Considerable effort is currently being devoted to this problem area by attempting to define milestone procedures and activity definitions for all of the major tasks involved.

G. COMMUNICATIONS BETWEEN INTERFACING GROUPS

This problem area is one which is common to all development processes involving different groups. For current spaceborne software systems this problem area is considered to result primarily from the lack of appropriate documentation. In general, the documentation produced for those missile and space programs which are in their early stages of development is meager, sketchy, and incomplete. This situation does improve in the later stages of development, but still constitutes a major problem area.

CONCLUSION

An unspecified objective of the SDC survey of spaceborne software systems was to determine to what degree, if any, spaceborne software systems were unique. I believe you will agree that the individual items I have presented concerning the environment, the development process, the end item and problem areas are not unique and exist in one form or another in most software systems. However, it appears that the combination of the elements and problem areas of current spaceborne software systems, coupled with the limited physical constraints of the hardware for which it is developed, does represent a degree of uniqueness.

SESSION 3

Hardware/Software Interaction

Chairman: Leon S. Levy
IBM Federal Systems Division

SUMMARY OF SESSION 3

by

L. S. Levy
IBM, Owego, N. Y.

The objectives of the hardware/software interactions session were twofold:

1. Assess the SOA of hardware/software interaction;
2. Evaluate current computer trends as a basis for new software formulation.

The consensus was that in the past, constraints of weight and power were severe and resulted in the design of machines which were difficult to program. A major measure of relief for the programmer is provided by the elimination of sequential access memories from the central processor due to availability of random access memories with airborne physical characteristics. Sequential bulk memories will still have system application.

Environmental constraints which are still expected to influence component selection and machine architecture include reliability and radiation resistance. However, the dominant architectural factor in the future will be ease of software implementation.

Since machine designs will be much more strongly influenced by software characteristics, programmers and system analysts must assist in formulating these characteristics. Projected features of the next generation spaceborne computers are extensive multiprogramming, use of higher order programming languages, and on-line real time control.

In the future much more direct guidance of computer architectural requirements should proceed from computer programmers. However, it appears that programmers will have to be motivated to provide this guidance. A continuing, and more extensive, dialogue of hardware/software is needed with more emphasis on programming desiderata and hardware potentialities.

"HISTORIC PERSPECTIVE: MACHINES AND PROGRAMMING CHARACTERISTICS"

By

D. B. Brosius
Data Systems Division
AUTONETICS

processing or computational center and to treat the otner elements of the system as peripheral equipment which are adapted to the computer interface. Thus, the digital computer in the spaceborne system has been progressing from a nighly specialized functional element designed toward its unique functions in a specific application toward a nighly flexible processing center adaptable to a variety of applications. Progress along these lines has been mainly limited by the technology required to meet system constraints on physical size, weight, power consumption, reliability, and cost.

Guidance computers provide a clear illustration of the early phases of this trend. The Autonetics developed D17 computer had approximately 2,700 words of storage and functioned almost entirely as an element of the guidance and control system. For the D37 computer, the memory was expanded to approximately 8,000 words, the increased storage being largely required to accomodate expanded pre-launch functions particularly in the areas of communications, ground equipment monitoring and control, and generally expanded system flexibility.

Until rather recently, the vast majority of computers developed for spaceborne applications were characterized by serial logic, fixed point, two's complement arithmetic, rotating memories, and a rather limited instruction set. The most significant trend in the design of later machines has been the use of random access core memories and high speed parallel logic. And in keeping with the general trend of expanding capabilities, instruction sets have been expanded and features such as indexing and indirect addressing have become common. Input/Output has remained rather specialized although features such as wired interrupts allow a flexibility not previously available.

The following discussion of programming characteristics will be mainly addressed to the former class of computers but in general the impact of the more recent trends in computer design, while very significant to software techniques, is largely a matter of degree.

PROGRAMMING CHARACTERISTICS

Programs for spaceborne applications typically involve several distinct classes of program functions. In the area of ground, or pre-launch functions, the following broad categories are involved.

functions require special concern for program timing, but in general all program routines must be placed in a real-time framework. In many cases real-time requirements on particular functions may be severe enough to require particular sophistication on the programmers part in order to satisfy them. This is particularly true of the guidance and control functions where timing requirements are directly related to system accuracy and stability.

2) The program computations are tightly constrained by accuracy requirements. Accuracy constraints typically express themselves in terms of the necessity for careful consideration of fixed point scaling in program computations and the possible need for double precision accuracy in some computations. Additionally, the effect of truncation and round-off errors innerent in the particular mechanizations being used must be evaluated. Overall system accuracy constraints of course also generate program time constraints, i.e., iteration or solution rate requirements.

3) The programs are very closely tied to the electrical interface between the computer and other sub-systems. Because of the specialized design of spaceborne systems, the digital computer forms an integral part of the total system hardware. For this reason, the computer programs are typically concerned with such things as the required pulse width of input/output signals, control of external I/O multiplexors, rise time of I/O signals, etc. A portion of this concern is expressed merely in terms of further timing constraints, much of it however results in constraints on overall program structure; i.e., the order in which functions are performed or the rate of function execution.

4) A further program characteristic arises from the specialized nature of the spaceborne system; namely, while the general nature of programs for different applications may be quite similar functionally, the specific programming techniques and organization

Coupled with the storage optimization problem are the critical timing contraints indicated earlier. Generally speaking, one optimizes storage requirements at the expense of program executions time and vice versa. Hence, in many cases the programmer is faced with the dilemma of resolving simultaneously a time optimization and a storage optimization program.

This type of concern is particularly great on rotating memory machines because of the intimate relationship between storage allocation and program timing.

Two examples of more specific constraints on program structure and organization resulting from requirements of a particular system are transient recovery and program anti-skip requirements.

A transient recovery constraint is required for some weapon system applications. This type of consideration requires program structuring to adjust program timing for time which is "lost" during transient conditions; more specifically it requires a time compensation factor in the guidance computations which is a function of computer down time due to transient conditions.

"Anti-skip" requirements refer to a requirement for insuring that critical program events, such as launch, or warhead arming do not occur unless the prerequesite sequence of prior events has been executed correctly. This type of requirement is an extension of the considerations generally classed as flight safety checks, the distinction mainly being that "anti-skip" is particularly designed to protect against a computer failure which would cause jump or skip in the program execution sequence.

SOFTWARE DEVELOPMENT

Having indicated the general characteristics of the machines and the programming problems one further area remains for discussion. This area involves the techniques concerned with the actual development of the software.

Generally speaking, there are four distinct functions involved in the development of a program:

1. Program Definition - the study and analysis involved in determining the program's functions and overall structure,

Assembly, as indicated in the preceding discussion, largely refers to a machine level assembler. In many cases, the assembly process may be restricted to a strict translation requiring absolute memory allocation by the programmer. Consistent with the other elements of the system, the assembler is quite often of a highly special purpose nature, producing data listings which are of unique value for the particular system and programming problem.

In addition to language translation, a certain amount of program error checking is done during the assembly process. This is generally limited to detection of formatting and coding errors although many assemblers provide partial program execution time tracing.

Checkout and verification of the program is accomplished using various combinations of simulation tools and operational hardware. Initial checkout is usually accomplished using a functional simulation at the computer level. Various open and closed loop sub-system simulation levels may then be utilized for particular sub-programs. The total ground phase of the application is generally verified using operational hardware and exercising the entire range of operational functions. The airborne phase is typically verified using digital or digital/analog closed loop simulation. This type of simulation may range from a fully software implemented digital simulation to a hybrid simulation utilizing a specially designed hardware site.

The need for a high degree of program confidence, particularly in the flight phase, dictates a rather detailed, meticulous verification process which adds significantly to program development time especially for program revisions which in themselves may represent rather minor changes.

SUMMARY

In conclusion, both the machines and the programs developed in the past have been highly special purpose in nature and highly specialized in design. The software development has therefore represented a particularly specialized effort. However, the distinct trend toward more capable, general purpose computers for spaceborne applications promises to partially generalize the standardize the programming effort in addition to making possible the use of higher level software development aids. And, in fact, it is becoming necessary that the trend toward more advanced software techniques be accelerated in order to meet the growing demands on spaceborne software.

SELECTION OF AN ORDER CODE DURING MACHINE DESIGN

by

R. K. Draving
UNIVAC Defense Systems Division

There are other unique constraints that compound the design problems of an aerospace computer. These occur from the necessity for having a "fail-safe" device. One of these constraints is often a possible requirement for redundancy, based purely upon the limited MTBF of a nonredundant unit. Also, "fail-safe" presents a requirement for nondestructive readout memory systems to protect a machine program and critical constants from transients. This is obviously important because a transient which changes a single bit within a several thousand-word program can destroy the performance of the computer and possibly the system which it controls.

2. THEORETICAL DESIGN PLAN

Although rare in actual practice, one can construct a procedure based upon empirical evidence which will formalize a design plan for a computer.[1] The basic elements of such a plan are illustrated in Figure 1. Each functional requirement of a space system can be translated into a series of operational requirements for the controlling computer. In the case illustrated the functional requirements may be those of a typical booster and satellite guidance system. It is the purpose of the design analysis to translate these requirements into actual hardware characteristics and to determine what hardware tradeoffs may be made while satisfying the operational requirements. One can take each of the functional specifications and, by using a postulated and relatively unlimited general-purpose computer vocabulary and utilizing indexing as needed, assign to each function a sequence of instructions. By knowing certain characteristics of each function certain probabilities can be associated with decision functions as they arise and a set of quantities can be determined which represents the required operational characteristics of the computer. These are:

1. Number of instructions executed
2. Instruction distribution
3. Input/output utilization
4. Storage requirements

A multitude of design tradeoffs may be taken at this stage of the design. Each approach must be evaluated on the basis of practical implementation. However, they all lead to a number of computer parameters. One very important parameter is the order code to be implemented. The order code is dependent in some measure upon other computer parameters. It is the purpose of this paper to illustrate some of the factors and some of the findings that have been associated with the order code itself.

3. TAILORING A REPERTOIRE

Order code selection requires a consideration of many factors. Some of these are presented in the following listings:

Instruction Word-Length

Perhaps one of the most important single factors in determining the instruction word-length is the type of addressing to be utilized. There may be full addressing capability, extension registers, and indexing. The operand word-length is also an important consideration since there is a need for a modular relationship between the instructions and operands to obtain efficient use of memory. The modularity of operands and instructions assumes a random access electrically-alterable memory.

Operand Word-Length

Operand word-length is greatly affected by the problem to be solved. In general, an operand word-length is more than sufficient for a typical instruction word. There are, however, factors which can influence an order code. For example, it is not usually efficient to design the operand word-length of a computer about the worst-case precision requirements. Often it is more efficient to provide for double operand instructions to provide greater precision.

Memory Size and Type

An important tradeoff can usually be made relating memory cost to the cost of providing more capable instructions within the central processor. This is of particular importance for complex electrically-alterable memories such as NDRO thin-film. It is important to have a relatively extensive general-purpose central processor to obtain the greatest bit efficiency within the memory. On the other hand, use of a low cost memory (e.g., a fixed core rope) can provide an excellent tradeoff in terms of increased memory size while reducing the complexity of the central processor. It should be noted, however, that there are software implications inherent in the latter tradeoff.

System Application

Obviously the application of the order code to the system or problem to be solved is of key importance. This, in fact, is the primary reason for tailoring. Implicit, however,

second assumes a basic set of instructions which is a subset of the total population set which could just solve the given problem. Figure 2 shows the perturbation to a control system problem typical of a ballistic missile in which the basic set of instructions is perturbed by additional commands from the total population set. When an instruction from the total population is used, a finite program savings resulted. Three system cases were considered; one computer per program, 10 computers per program, and 100 computers per program. It can be seen that the greater the number of computers per program the less cost savings that accrue. If the savings of bits of memory are taken into account we get a different result. (See Figure 3.) For the above comparisons some relatively simple cost parameters were assumed as shown. The reader is cautioned not to use the figures as any absolute approach. However, the techniques employed combined with an appropriate criterion can be used as a tool in assessing the value of a particular instruction. The precise vocabularies employed as a basic set and as the total population, as well as the other assumptions, are presented in reference 2.

4. INDEXING

The use of index registers can have a startling effect upon the efficiency of memory storage.[3] Table 1 tabulates the size and running time of common matrix and vector manipulations utilized for typical space missions. A general-purpose vocabulary was used; however, in one case three index registers were assumed and in the other case no index registers were assumed. Although these subroutines are highly iterative in nature they do show an amazing change of total storage requirements made possible by the simple capability of indexing as shown in the summary given in Table 2. It should be noted, however, that in nonparallel computing systems such as a serial-by-bit machine in which the indexing of operand and operand addresses requires times comparable with short arithmetic operations, the additional time required for indexing becomes a very significant factor in the design tradeoff.

5. SUBROUTINE CAPABILITY

Table 3 presents the program storage requirements that would be necessary if no subroutine capability were available to perform typical vector and matrix operations of a control problem. Again, this illustrates typical mathematical subroutines characteristic of space missions.

INSTRUCTION	I_S	IND H_C	EST F_U	$F_U \cdot I_S$	$F_U \cdot I_S \cdot I_C$	MEMORY SAVINGS 1/1	EFFECTIVE HDWE COST 1/1	TOTAL SAVINGS 1/1
If (A) - (Y) > 0 SKIP	6	255	95	570	2850	1026	-771	3621

MEMORY SAVINGS 1011	EFFECTIVE HARDWARE COST 10/1	TOTAL SAVINGS 10/1	MEMORY SAVINGS 100/1	EFFECTIVE HARDWARE COST 100/1	TOTAL SAVINGS 100/1
10260	-7710	10560	102600	-77100	79950

TABLE 2

	I_S	IND H_C	CAP H_C	EST F_U	$F_U \cdot I_S$	$F_U \cdot I_S \cdot I_C$	MEMORY Savings 1/1	EFFECTIVE HDWE Cost 1/1	TOTAL Savings 1/1	MEMORY Savings 10/1	EFFECTIVE HARDWARE Cost 10/1	TOTAL Savings 10/1	MEMORY Savings 100/1	EFFECTIVE HARDWARE Cost 100/1	TOTAL Savings 100/1
If (A) - (Y) > 0 Skip	6	255		95	570	2850	1026	-771	3621	10260	-7710	10560	102600	-77100	79950
(Y)→A	3	80		45	135	675	243	-163	884	2430	-1630	2330	24300	-16300	16700
Y→A	1			200	200	1000	360			3600			36000		
-Y→A	1	130		50	50	250	90	-1040	4250	900	-10400	13650	9000	-104000	103500
(A) + (Y)→A	1			200	200	1000	360			3600			36000		
(A) - (Y)→A	1			200	200	1000	360			3600			36000		
Y→Q	1			20	20	100	36			360			3600		
-Y→Q	1	130		20	20	100	36	-122	822	360	-1220	1920	3600	-12200	12000
(Q) + (Y)→Q	1			50	50	250	90			900			9000		
(Q) - (Y)→Q	1			50	50	250	90			900			9000		
Y→BP^I; I=0, 1,2,3	3	95		200	600	3000	1080	-985	3985	10800	-9850	12850	108000	-98500	101500
If (A) > 0 Jump	2	180		175	350	1750	540	-360	2110	5400	-3600	5350	54000	-36000	37750
B^D + 1→$B^D \geq 0$ Jump	1	180		175	175	875	315	-135	1010	3150	-1350	2225	31500	-13500	14375
B^D + 1→$B^D < 0$ Jump	6	155		20	120	600	216	-61	661	2160	-610	1210	21600	-6100	6700
Y + (P) + (B^E)→F	4	170		100	400	2000	720	-550	2550	7200	-5500	7500	72000	-55000	57000
w_i→Y; i=s,1,f	3	70		40	120	600	216	-50	650	2160	-1460	2060	21600	-5000	5600
$(\overline{Q})$→Q 2's comp.	2	35		45	90	450	162	-127	577	1620	-1270	1720	16200	-12700	13150
$(\overline{AQ})$→AQ 2's comp.	4	115		25	100	500	180	-65	565	1800	-650	1150	18000	-6500	7000
$\sqrt{(A)}$→A	505	1500		1	50	250	90	1410	-1160						
|(A)|→A	2	35		50	100	500	180	-145	645	1800	-1450	1950	18000	-14500	15500
Floating Point $\sqrt{(A)}$→A	605	1850		1	60	300	108	1742	-1442						
(Q_n) = 1 Set Flag	4			55	220	1100	396			3960			39600		
(Q_n) = 0 Set Flag	3	185		55	165	825	297	-1201	5751	2970	-12010	16560	29700	-120100	124650
If (Q_n) = 1 Skip; Test Flag	3			75	225	1125	405			2970			29700		
If (Q_n) = 0 Skip; Test Flag	4			75	300	1500	540			3960			39600		
(A) + (Y)→Y	3	100		145	435	2175	783	-683	2858	7830	-6830	9005	78300	-68300	70475
(A) - (Y)→Y	3	100		75	225	1125	405	-305	1430	4050	-3050	4175	40500	-30500	31625
(Y) - (A)→Y	4	135		90	360	1800	648	-275	1923	3600	-8250	3825	36000	-27500	23500
(Y) + 1→Y	4	115		160	640	3200	1152	-1037	4237	11520	-10370	13688	115200	-103700	104900
(Y).+ 1→Y	6	95		130	468	2340	842	-687	3527	8420	-6870	10388	84200	-68700	81000

Figure 3. 36 Bit (Memory Savings Included)

Table 2. Trade-off Memory Storage for Indexing
In Matrix and Vector Operations

STORAGE		
• NO INDEXING OR INSTRUCTION MODIFICATION		
SETUP AND EXIT INSTRUCTIONS		14,580
SUBROUTINE INSTRUCTIONS		496
	TOTAL	15,076
• INDEXING (EFFECTIVE OPERAND ADDRESSING MODIFICATION)		
SETUP (CALLING SEQUENCES ONLY) INSTRUCTIONS		2,707
SUBROUTINE INSTRUCTIONS		326
	TOTAL	3,033
MEMORY SAVINGS THROUGH USE OF INDEXING		12,043

6. WORD LENGTH VERSUS BIT EFFICIENCY

A sample study was conducted by UNIVAC in programming a guidance simulation problem[4] for representative computers having different word-lengths.[5] The following machines were used:

18-bits - UNIVAC 1218 Computer

24-bits - A 24-bit general-purpose computer

30-bits - UNIVAC 1206 Computer

36-bits - UNIVAC 1108 Computer

Although several options were available for writing the program on each machine (compiler versus assembler, fixed-point versus floating), the one chosen for comparison presents each machine in its most favorable application. Figure 4 compares the number of computational instructions needed to program a problem for various machines. There is a continual reduction in the number of instructions as the more capable machine is employed.

Figure 4 shows the bits required for the program by each computer. This graph is an interesting demonstration of the efficiency of memory storage based upon the instruction word-length. In contrast to Figure 4, Figure 5 shows an increasing number of bits required for the larger instruction word-length computers for accomplishing the same problem. Thus, the shorter word-length has a tendency for more efficient bit usage. It is noted that the 1108 tends to reverse this trend, due somewhat to its large instruction vocabulary and multiple accumulators. The trend, however, is reversed primarily because of the powerful floating-point commands employed.

7. DESIGN TRENDS

The discussion to this point has been to show some design techniques utilized in the past and to demonstrate the basic power of such fundamental concepts as indexing, subroutine capability and floating-point arithmetic. We have, however, recently observed a significant trend in new computer designs. Two major aerospace computer manufacturers (viz., UNIVAC and IBM) are developing functional copies of proven surface computers for airborne applications. The newest UNIVAC computer for avionics applications (similar to those of complex spaceborne missions) is the UNIVAC 1830A. This machine is a direct derivative of the UNIVAC 1206/1230 series of ruggedized ground computers. We are also aware that IBM is developing functional copies of their

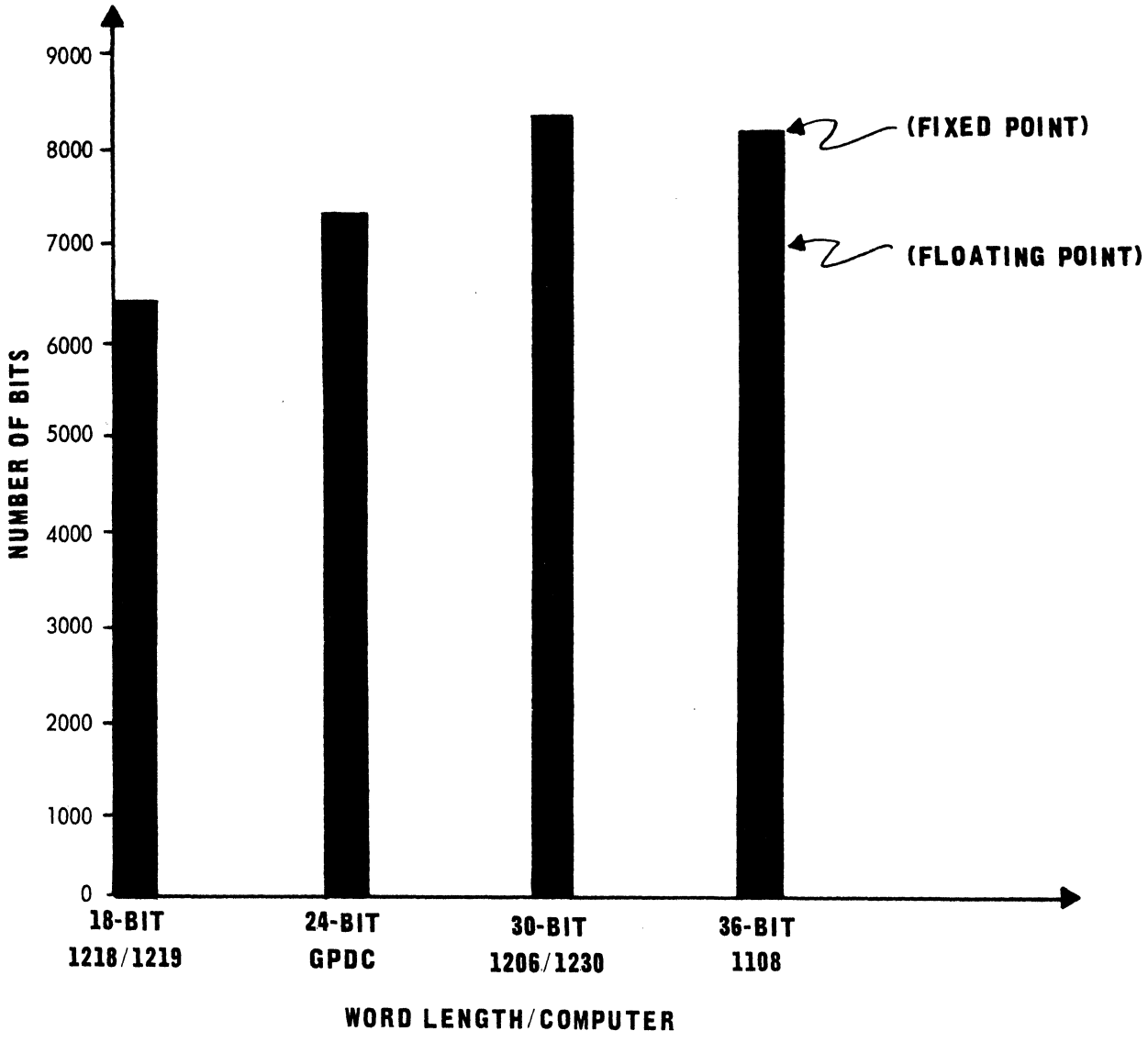


Figure 5. Comparison of Bits Required

REFERENCES

Reference 1 — Design Techniques for a Real-Time Digital Control Computer, by R. K. Draving, A. Kaplan, UNIVAC Division Sperry Rand Corporation, and L. G. King, Bendix Systems Division, National Electronics Convention, Dayton, Ohio; 1962.

Reference 2 — Report for Subtask 15.1 of CCM 23 to Contract No. 64(694)-621, dated 16 June 1966, by D. A. Jacobson, UNIVAC Division Sperry Rand Corporation.

Reference 3 — Detailed Organization Characteristics for Standardized Space Guidance System Study (SSGS), STL Subcontract 3780S-SC, 13 April 1964, by B. J. Jansen, C. L. Firm, UNIVAC Division Sperry Rand Corporation.

Reference 4 — A Simplified Guidance Simulation Specification, G. A. Champine dated 10 March 1965, UNIVAC internal document.

Reference 5 — Computer Comparison using a Benchmark Program, 14 September 1965, by W. L. Smith, UNIVAC Division Sperry Rand Corporation.

FACTORS OF THE MACHINE DESIGN THAT INFLUENCE PROGRAMMING

by

Lindley S. Wilson
Aerospace Corporation
El Segundo, California

can be thought of as composed of three minor channels corresponding to the three segments of each instruction cell in the channel. The channel then is specifically indicated but selection of one of the 64 peripheral positions is a function of time and is determined by which one happens to be under the active read head at the time the instruction is given.

Two constant channels are accessible to all instruction locations, but others are each accessible only to the instructions in one or two bands. To make the constants more accessible time-wise but complicating the programmer's task again, each "data track" channel has multiple read heads and their spacing is not the same on each channel. Frequently used constants are often stored in many places to make them more accessible. Even so, all too frequently, delay instructions must be used to hold the machine idle until a desired constant swings round under an available read head. To ease the access problem, but complicating addressing still further, there are 5 two-word, 1 three-word, and 1 eight-word revolvers* or buffer registers. Some of these also have multiple read heads accessing the long portion of the channel outside the buffer proper. Thus, a piece of information may be accessed by any one of several instructions in a channel. Note that an instruction referencing a given word on the three-word (or F) revolver on one drum revolution will reference the following F revolver word on the next drum revolution. I might also mention that the accumulator or S revolver is two words long so that the primary read head references one accumulator word at even word times and the other at odd word times. To make things functionally easier but complicating them also, there are two auxiliary read heads picking up accumulator words nine and sixteen word times

---

* A simple revolver consists of a complete channel on the drum with a write head, and a read head which normally reads information coming under it and transfers it to the write head a few words upstream forming a cycle. The cycle can be interrupted at any word time to enter new information through the write head after which the cycling operation resumes. The old information remains available as it passes across auxiliary read heads on the long trip around to replacement under the write head.

indexed command may be slower than an unindexed command. Thus, there is a tradeoff between time lost and space saved to be considered. For example, if storage considerations were paramount, a matrix multiplication might use double indexing, whereas, if time were more important, single indexing or no indexing might be used. In other applications, like table look up where the argument and its relative address can be made identical, indexing is very effective in saving running time, storage and programming effort. I'll mention one more addressing problem that of setting the return link for an interrupt or from a subroutine. Nearly all machines under consideration make automatic provision for this but in different degrees of sophistication. The drum computer mentioned earlier has no automatic link back provision and subroutine returns have to be preset each time before subroutine entry. This is probably because the design of this machine is so inimical to closed subroutines. Many machines automatically set the return link in temporary storage whence the subroutines must immediately store it in a safe place for reentry. Others automatically store the link in a register and link back by an indirect jump. Another scheme requiring more storage but avoiding conflict in the case of nested subroutines automatically stores the link in the destination address and sends control to the next sequential address and so on; there are many variations.

Timing Constraints

I call them timing problems though some of them might also be classed as addressing, storage or speed constraints.

First there are timing constraints imposed by certain instructions. Several machines have come out with a scheme which allows multiplication and division which often require several add times for execution to be shared with shorter instructions. In one version, the multiplier (or dividend) is initially in the accumulator which then becomes available for short instructions. Several add times later, the string of short instructions must

enough so that no signals are lost represent another real time constraint. With a slow machine, this could mean a large portion of running time. Auxiliary hardware which automatically samples and accumulates sensor outputs can greatly increase machine capacity.

Eccentric Instructions

Incomplete instructions or instructions with operating exceptions can also increase the programming burden and bring about storage and/or time penalties. Divison sometimes occurs with the quotient automatically rounded and no remainder. If it is desired to know when a quotient exceeds a certain limit, this lack could force an additional multiplication and comparison. This would also be true when the remainder rather than the quotient was desired as is occasionally the case. Multiplication yielding only the most significant half of the product, is open to the same criticism. These deficiencies also become important when double precision arithmetic is required. The shift operation can be performed by multiplication or division, and thus, is sometimes omitted. This occurs more often in machines where multiplication or division can be time shared (as mentioned above) so that the time penalty involved depends upon the application. On one computer, which has time sharing, there is also a shift command which is limited to 1 or 2 shifts. On a test routine involving only arithmetic operations with tight scaling requirements, 54 out of the total required 283 instructions would have been eliminated by a more adequate shift command. This storage could also have been saved by using multiply commands but only by taking a severe time penalty, since that particular routine was not amenable to time sharing. A drum computer in current use has no shift commands -- division by zero or multiplication being used. A dangerous feature was introduced here in that multiplication on this computer is in error by one bit if (considering the multiplier an integer) the 2^4 bit of the

inhibit its execution by a bit in the instruction code. Key registers may be automatically stored and a return link unique to the interrupt set up. Other special instruction might be floating point operations, square root, trig functions and radix conversion. If large amounts of data are to be handled, block instructions and list search oriented instructions might be useful.

Reliability

If reliability is insufficient, part of the running time may be used up in diagnostic tests. When power is lost, discretes and temporary storage may have to be restored when power comes on and if possible, the program must be patched. Special programs may be required to provide for all contingencies during input and output.

Compatible Backup Hardware

One of the most important hardware constraints is lack of a large compatible computer to do simulation, checkout, assembly and compiling operations. If the instruction set of the space-borne computer is a subset of the large computer, simulation should be very easy and the running speeds should be very much faster than if interpretive simulation is necessary.

SUMMARY AND CONCLUSIONS

Hardware constraints are very real and can cause an appreciable penalty in programming effort. However, they must be considered in the light of intended applications and compensating software. With a large backup computer system and compensating software, most of the constraints can be eliminated.

However, if storage and/or running time is tight, it may be necessary to do very elegant and complicated coding which is beyond the capability of present software languages. At this point, hardware constraints on programming can be significant.

ADVANCED HARDWARE CHARACTERISTICS OF AEROSPACE COMPUTERS*

By

D. L. Meginnity
Manager
Data Processing Systems Department
TRW Systems

*This is not a formal definitive paper on the subject matter but an approximate recreation of the brief talk given by the author to stimulate discussion at the "Hardware/Software Interaction" session of the "Spaceborne Computer Software Workshop" sponsored by AFSSD and Aerospace Corporation, 20 - 22 September 1966.

of the basic scientific (data gathering) nature of the probes in this category, some form of degraded performance of subsystems will probably be tolerated. Therefore the concept of graceful degradation of performance will also become a computer requirement as a partial substitute or complement of extreme MTBF requirements. Sophisticated applications of redundancy and self-repair techniques will undoubtedly be used to obtain the needed MTBF's.

EXTREME ENVIRONMENTS

There are two extreme environments worth noting for future aerospace computers. The first is nuclear radiation hardening; resistance to both high radiation rates and high integrated doses are important. The requirements are generated by both military applications and general scientific space probes. The other extreme environment will be for space hardware to operate over extreme temperature environments (e.g., -150°C to 200°C). In large part, of course, these temperature extremes would be conditioned by other subsystems. However, there will be increasing requirements for all spacecraft subsystems to operate over substantially larger temperature ranges.

PHYSICAL CONSTRAINTS

There are no new physical constraints for aerospace computers but size, weight, and power consumption will continue to be critical factors in the feasibility of applying computers to future missions. Significant reduction in these physical parameters will be required before complex computers may be used for unmanned spacecraft.

FUNCTIONAL CHARACTERISTICS

The functional characteristics of future aerospace computers will continue to span the spectrum from the very simple special purpose processor to machines which in capability are only constrained by other parameters (size, cost, reliability, etc.). This latter end of the spectrum will continue to expand until computers equivalent to the

FEATURE / APPLICATION	SIZE/WEIGHT	POWER	RADIATION RESISTANCE	RELIABILITY	FUNCTIONAL CAPABILITY	LOW COST
GP BOOSTER GUIDANCE				X		X
UNMANNED SPACECRAFT	X X	X X	X	X X	X	
MANNED SPACECRAFT	X	X		X X	X X	
STRATEGIC MISSILE	X		X X	X	X	X
TACTICAL MISSILE	X X					X X
AIRCRAFT					X X	X X

X = SOMEWHAT IMPORTANT

X X = EXTREMELY IMPORTANT

TABLE I

IMPORTANCE OF FEATURES TO DIFFERENT MISSIONS

Thin film hybrid circuits using bi-polar semiconductors are large and considerably more expensive than any of the above alternatives but the better controlled passive elements facilitate design of high efficiency - high output power circuits in a class impractical for monolithic contruction. Examples where hybrid circuits may be required are in the memory electronics and for computer input-output applications. These techniques are readily available for current design applications.

Thin film active circuits are in a relatively early stage of development at a few research centers. The primary advantages expected of circuits employing this technology are unusually favorable ratios of power to speed and size. The current development problems are inability to consistently produce films of sufficient uniformity and a poor understanding of experienced failure modes. This technology will most probably not be sufficiently advanced to allow system application for 3 to 5 years.

Several companies are currently expending considerable research and development efforts in the area of magnetic logic. This logic circuit family is being developed specifically to yield immunity to radiation several orders of magnitude more intense than can be tolerated by the best semiconductor circuits currently available. Because of the elimination of transistors from these circuits, significant reliability advantages are also predicted. The primary disadvantage is the relatively slow speed of these devices, less than 1/10 the speed of standard DTL. Further, these circuits are not readily available today and require some further development. However, because of the unusually high interest for military applications, magnetic logic circuits will probably be available in the next few years.

MEMORY SYSTEMS

At the present moment Ferrite-cores still represent the major storage element used in high-speed direct access storage systems. In spaceborne computer applications, conventional 3-D organized coincident current memories are being designed presently for 2 μs cycle time operation and capacities of 250K bits per module. This type of memory system

as noted. It should be noted that the basic component technology will afford a wide variety of computer designs for various missions; the computers presented in Figures I through IV were chosen to illustrate some approximate limits of the possible design parameters.

- TYPE: FLEXIBLE MODULAR GP WITH HIGH INPUT/OUTPUT CAPABILITY FOR ELINT, FIRE CONTROL ETC., DATA PROCESSING
- SPEED: 4 μSEC ADD, 20 μSEC MULTIPLY
- MEMORY: 8K - 32K MODULAR PLUS BULK STORAGE
- WEIGHT: 30 - 50 LB
- POWER: 100 W
- MTBF: 1 YEAR
- COST: $20K - $30K
- AVAILABILITY: 1970

FIGURE II

AIRCRAFT LOW COST COMPUTER

- TYPE: FLEXIBLE MODULAR GP MULTIPROCESSOR WITH EXTREMELY CAPABLE INPUT/OUTPUT FOR DATA MANAGEMENT AND MISSION MANAGEMENT
- SPEED: 0.5 μSEC ADD, 3 μSEC MULTIPLY
- MEMORY: 50K - 100K MODULAR PLUS BULK STORAGE
- WEIGHT: 70 - 90 LB
- POWER: 200 W
- MTBF: 10 YEARS (DEGRADED TO 1/4 MAXIMUM CAPACITY)
- COST: $400K - $500K
- AVAILABILITY: 1975

FIGURE IV

MANNED SPACECRAFT COMPUTER

A COMPATIBILITY SOLUTION - 4PI

By R. B. Talmadge
IBM

Apart from their intended usage, spaceborne systems are distinguished from ground systems by the fact that the operational environment, that is, the environment in which the mission programs are executed, is implemented on a computer which is physically distinct from the computer upon which the support functions are implemented. The physical separation has created special difficulties in system communication and has somewhat retarded the use of sophisticated programming techniques commonly found in other systems. There is therefore now a strong movement to try to simplify the job of system design by specifying use of spaceborne computers which are compatible with standard commercial systems. This movement has resulted, for example, in a spaceborne computer for the MOL program (an IBM 4 PI computer) which is compatible to a System/360 machine. At least one other manufacturer has produced a spaceborne computer which is compatible to a ground based machine.

What is to be discussed here today is not the 4 PI computer itself, but some of the rationale for compatibility, its use in overall system design now (as exemplified by MOL), and what role it might play in the future. But first, since compatibility is a relative term, let us be more specific: what we are talking about is a spaceborne computer which is in all essential programming aspects identical to a standard commercial computer. That is, word sizes, data formats, and instruction formats are identical; and instruction set variation, if it occurs, is strictly non-conflicting. The word commercial is important here, for one purpose of compatibility is to permit utilization of as much as possible of a standard, commercially supplied system. The standard system represents a considerable investment in time

7. A Simulation environment in which programs can be run under conditions as nearly identical as possible to the operational environment.

8. An Executive supervisor which co-ordinates activity between, and exercises control over, the total set of functions.

These parts are not all distinguishable in existing systems. For almost invariably the designers have made the tacit assumption that the operational environment is identical to that in which the system processors work. The system processors therefore not only produce code for operation in the given computer, but the code uses linkages and communication conventions designed for the executive supervisor. Hence, the distinction between simulation environment, operational environment, and the executive supervisor disappears.

Current systems in which the executive controls the first five functions (which are support functions in any system) are generally not able to support program execution within the time constraints required for spaceborne applications. Recovery of an existing system therefore dictates two possible courses of action. First, one could re-design the executive supervisor to support spaceborne applications, and then modify the existing system to conform to this design. Second, one could design a spaceborne specific operational environment, implement it, and also insert routines under the existing executive to provide the facilities necessary to integrate this environment within the system. Compatibility makes the first course possible, but it is neither practical nor palatable. It is not practical because modification of the entire system is required; it is not palatable because it defeats the purpose of recovering a standard system. Clearly, such action would be distinctly inferior to an all new system design in which the support computer was the same as the spaceborne computer.

For these purposes compatibility is a mixed blessing. On the one hand, the fact that the programs can execute directly is almost sufficient reason to let them do so, thereby speeding up the process of simulation. However, this requires that the simulation programs be written in such a way as to overlay portions of the executive (and the spaceborne programs) in order to gain control at the proper time. The simulation routines are therefore sensitive to changes of program location. If, on the other hand, the spaceborne computer were not compatible, then a full interpretive routine would have to be implemented in order to carry out instruction execution. Such an interpretive program is not location sensitive, and in many other ways actually simplifies implementation of the tasks involved. Furthermore, in such a 'soft' simulation environment, the additional execution time is of little consequence. In this area, therefore, compatibility affects the possible course of action, but it is far from obvious in what manner it should best be used, or if the effect is beneficial or detrimental.

It is in the area of program preparation that the effect of compatibility is directly felt. If we consider only languages concerned with computation, together with their associated language processors, then compatibility should enable us to start with the output of these processors in preparing programs for execution in the operational environment. The objective here is to use the existing language processors, as well as all existing system functions, in conjunction with a new preparation processor. The new processor will accept as input program modules generated by the language processors, together with statements by the programmer specifying operational environment information, and will produce as output programs in the form required for spaceborne operation. If this can be done, then the amount of effort required is certainly smaller than that which would be required if one had to modify the language processors. But more important, improvements to the processors, or the development of additional processors, can be fitted into the system without the necessity for any additional effort.

The crucial question for the operational environment is whether it should be distinguished from the support environment. The simplest answer (as we have seen, the usual answer) to this question is no. But consider some characteristics of only a few examples of usage.

1. The Spaceborne System: data acquisition and real-time control, absolute response times, many special devices.

2. Program preparation: large volumes of data accessed from files, user oriented, no special response required.

3. Conversational Mode: user oriented in strictest sense, moderate file access, variable response times.

4. Partial differential equations: almost no external communication, raw computing power all important.

To suppose a single operational environment suitable for all these is analogous to supposing a single structure is suitable as suspension bridge, hotel and family residence. It can be done, but the structure is not likely to be satisfactory in any function.

It is a measure of the immaturity of the computing world that so much time (and money) is spent in seeking a single solution to problems which can only have many solutions. Standardization is confused with uniqueness, and use of a general purpose computer with general purpose usage. With a slight change in attitude it would be possible to produce what is perhaps the most important unwritten book on computing, call it "The Programming System Design Handbook." One cannot predict its contents, but if we follow our previous analogy it will function much like the Structural Engineer's Handbook in making standardized techniques, conventions, and materials available for design of a specific system. Compatibility can then be achieved at the functional level, where its effect is significant, rather than at the instruction set level, where its advantage is localized or its effect actually detrimental.

A MACHINE ORGANIZATION SOLUTION

THE VARIABLE INSTRUCTION COMPUTER

By

A. L. SPENCE

Radio Corporation of America

As a result of the variable instruction concept and its implementation we have achieved varying degrees of versatility, emulation and reliability.

VIC's Physical Characteristics

The Variable Instruction Computer consists of two physical units. Figure 1 shows a sketch of the Central Processor Unit which occupies a volume of 1.6 cu. ft. and weighs 65 lbs. Figure 2 is a sketch of the Main Memory Unit which occupies a volume 1.4 cu. ft. and weighs 55 lbs. with 8196 36 bit words of core.

VIC Organization

Figure 3 shows the VIC organization which is as follows:

The main memory contains two memory modules of 4096 38 bit words of coincident current magnetic cores. It is expandable to a total of 32768 words with four 4096 modules per box. The cycle time of the main memory is 3 usec and an access time of 600 nsec. The main memory dissipates 160 watts for the two modules and increases in increments of 20 watts for each memory module added. There is a separate power supply for each memory module.

The high speed memory consists of two modules, each containing 256 38 bit words. The memory element consists of two magnetic cores per bit, linear select. A high speed memory address register, a high speed memory local register for data and local control logic are present in each module.

The control module is made up of two Order Registers, two Variable Registers and a Controller to sequence, code and direct the actions specified by these registers. The order register contains the macro instruction being processed. The operation code portion of the macro instruction addresses the location in high speed memory where the appropriate micro instructions are located. The contents of the specified high speed memory location, which consists of three micro instructions, are transferred into the variable register where a sequence counter steps through the processing of the three micros unless instructed to jump or terminate by an end bit.

The arithmetic and logic module consists of a shift control device, a function control device, an iteration counter, a one step shifter, the arithmetic and logic circuitry, and six registers W, X, Y, W_i, X_i, Y_i. The function control device controls the process of the arithmetic and logic circuitry. The iteration counter is generally used in the iterative mode in which it is desired to repeat an algorithm formed in an iterative instruction such as MULTIPLY. The one step shifter performs up to a 36 bit shift as specified by the shift control device. The six registers provide temporary storage data during

are called the C1 field which specifies the arithmetic function to be performed by the micro subroutine. It is this field which allows the individual to vary the function of the micro subroutine to perform for example, one's complement or two's complement addition. The C2 field contains the address of the scratch pad location which the micro subroutine will use during the execution of the macro function, i.e., accumulator or index registers. The last fifteen bits of the Program Word contain the main memory address field. In our future development of VIC we are examining the possibilities of providing a variable Program Word format which would provide us with capability to perform emulation at binary object program level. The Variable Word - this is the word stored in control memory - contains three twelve bit micro instructions which with one or more variable word form a micro subroutine.

VIC Instruction Flow

The instruction flow for VIC is shown in Figure 6. The macro instructions are stored in main memory locations and brought into the Order Register where the decoder takes the operation code field of the macro instructions and extracts from control memory the first variable of the micro subroutine addressed by the operation code. This first variable, consisting of three micro instructions, is transferred into the Variable Register where the decoder picks out the first micro instruction and performs the logic sequence or arithmetic operation which the micro instruction represents. The decoding of the micro instructions is continued until an end bit is encountered. At this time the next macro instruction is brought into the Order Register and the whole cycle is repeated.

Example of Macro Instructions Micro Programmed

The programming activity to date has been concentrated on micro programming the macro functions of the IBM 7090 computer. Figure 8 demonstrates the high degree of success we have had in programming one micro subroutine to perform the functions of several macro instructions. For example, the three micro instruction subroutines which perform the ADD macro instruction can also perform the IBM 7090 functions of

SUBTRACT (SUB)
CLEAR AND ADD (CLA)
LOAD QUOTIENT (LDQ)
AND ACCUMULATOR (ANA)
OR ACCUMULATOR (ORA)

Other functions which can be obtained from this micro subroutine by simply modifying the C1 function field of the macro instruction are

Set accumulator to zero
Set all ones in the accumulator
Add one to the accumulator
Subtract one from the accumulator

The ability to utilize one micro subroutine to perform several macro functions maximizes the use of control memory locations thereby providing a more powerful instruction repetoire with a smaller high speed memory.

Summary

With the present VIC we are able to emulate the functions of non I/O instruction of fixed word length machines. In our continuing development of the variable instruction concept we are striving to achieve higher degrees of emulation.

In the area of versatility we are endeavoring to incorporate the necessary logic in VIC which will allow us to perform variable length word arithmetic operations.

In the area of reliability via microprogramming, and here is meant the avoidance of computer malfunctioning components via micro subroutine substitution,

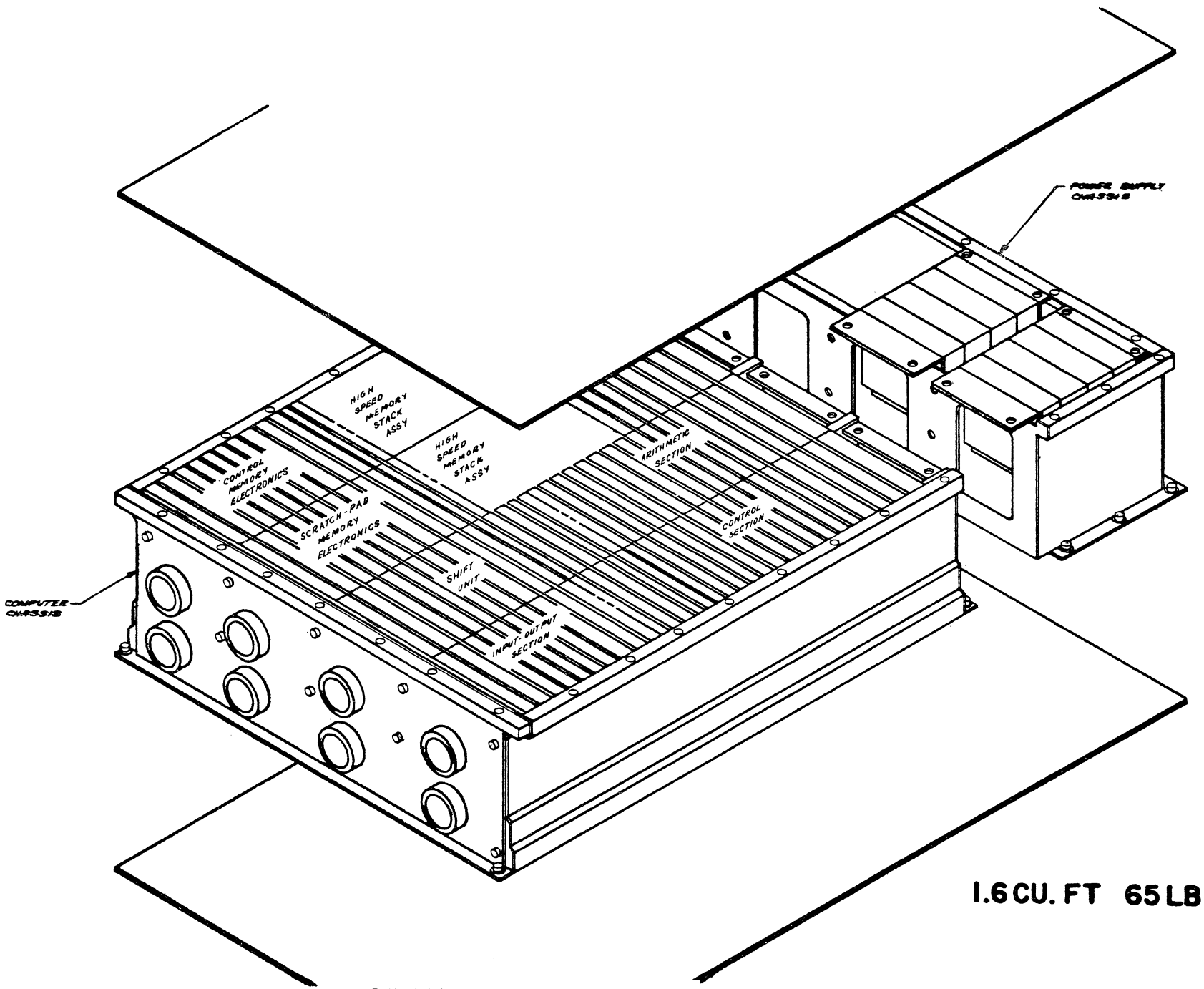


Figure 1 - CENTRAL PROCESSOR UNIT

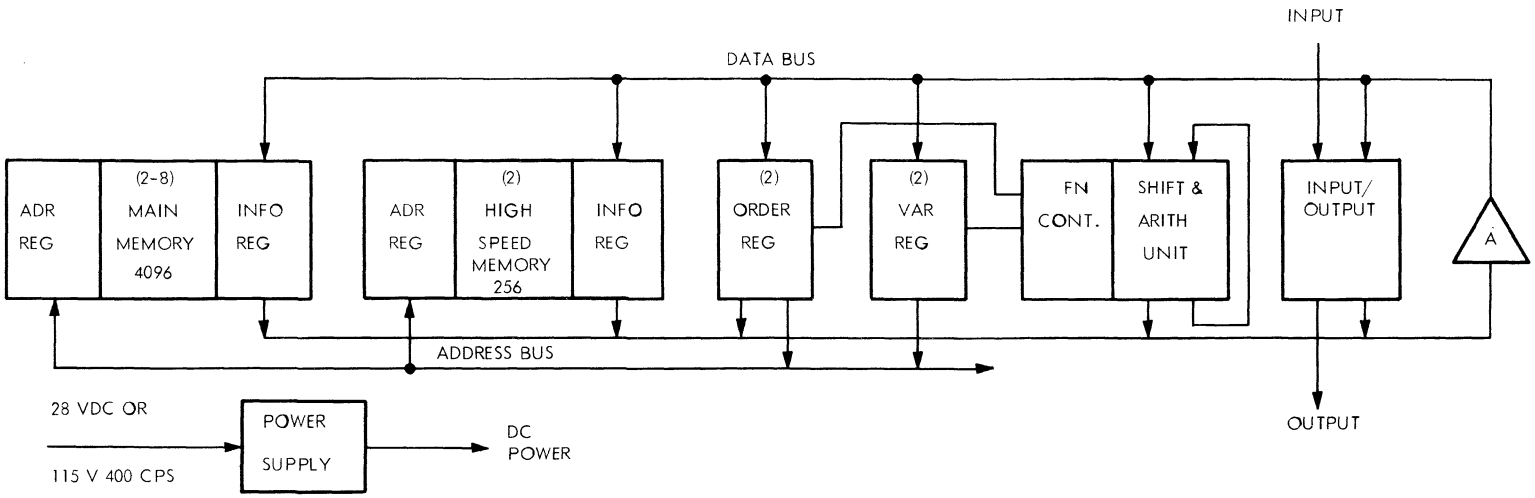


Figure 3 - ORGANIZATION

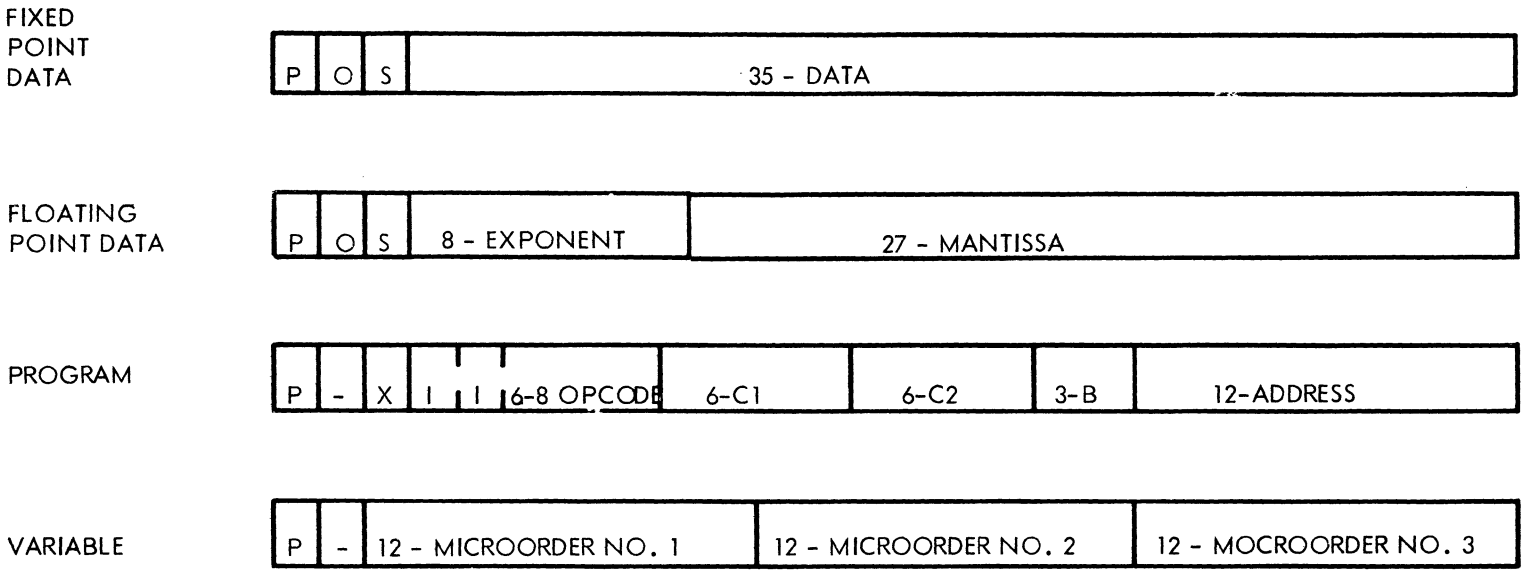


Figure 5 - WORD FORMATS

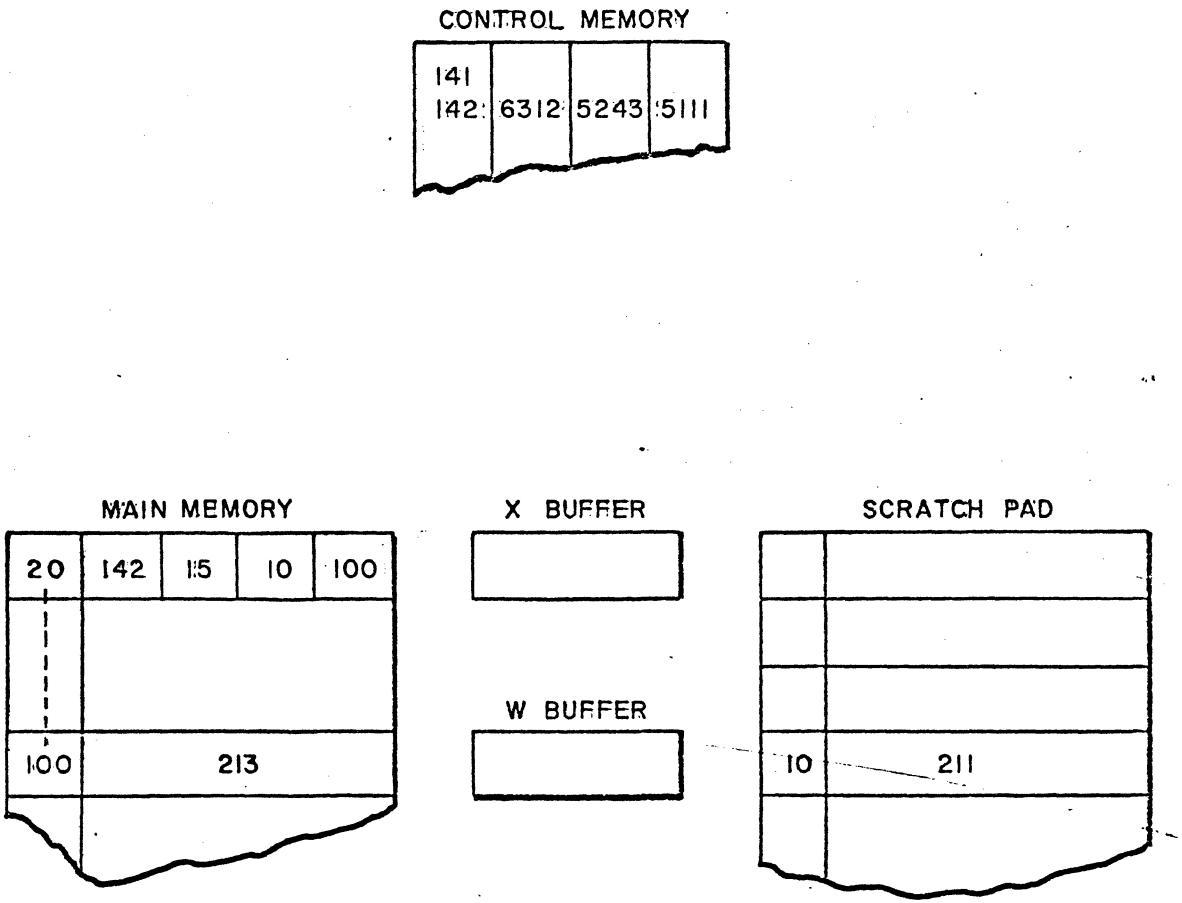


Figure 7 - MICRO SUBROUTING EXAMPLE

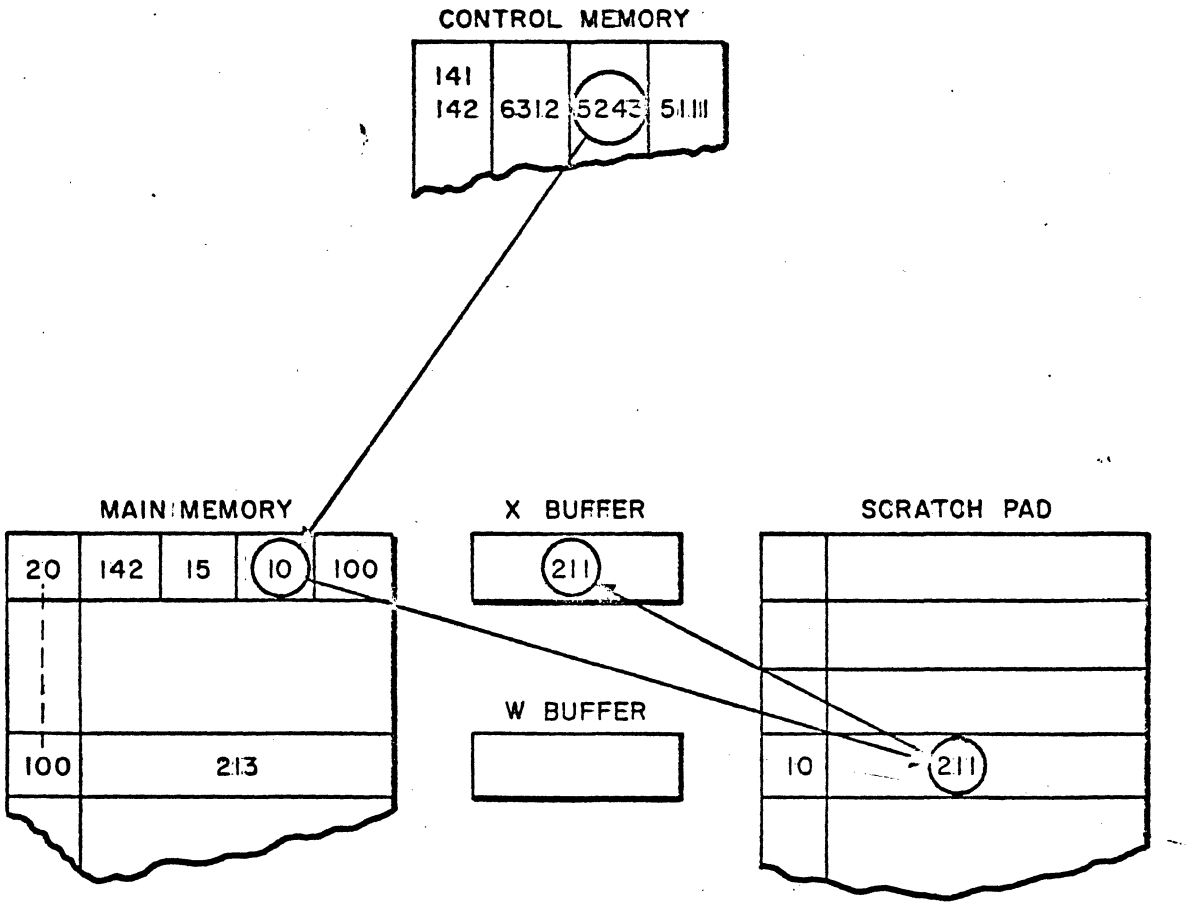


Figure 9 - MICRO SUBROUTINE EXAMPLE

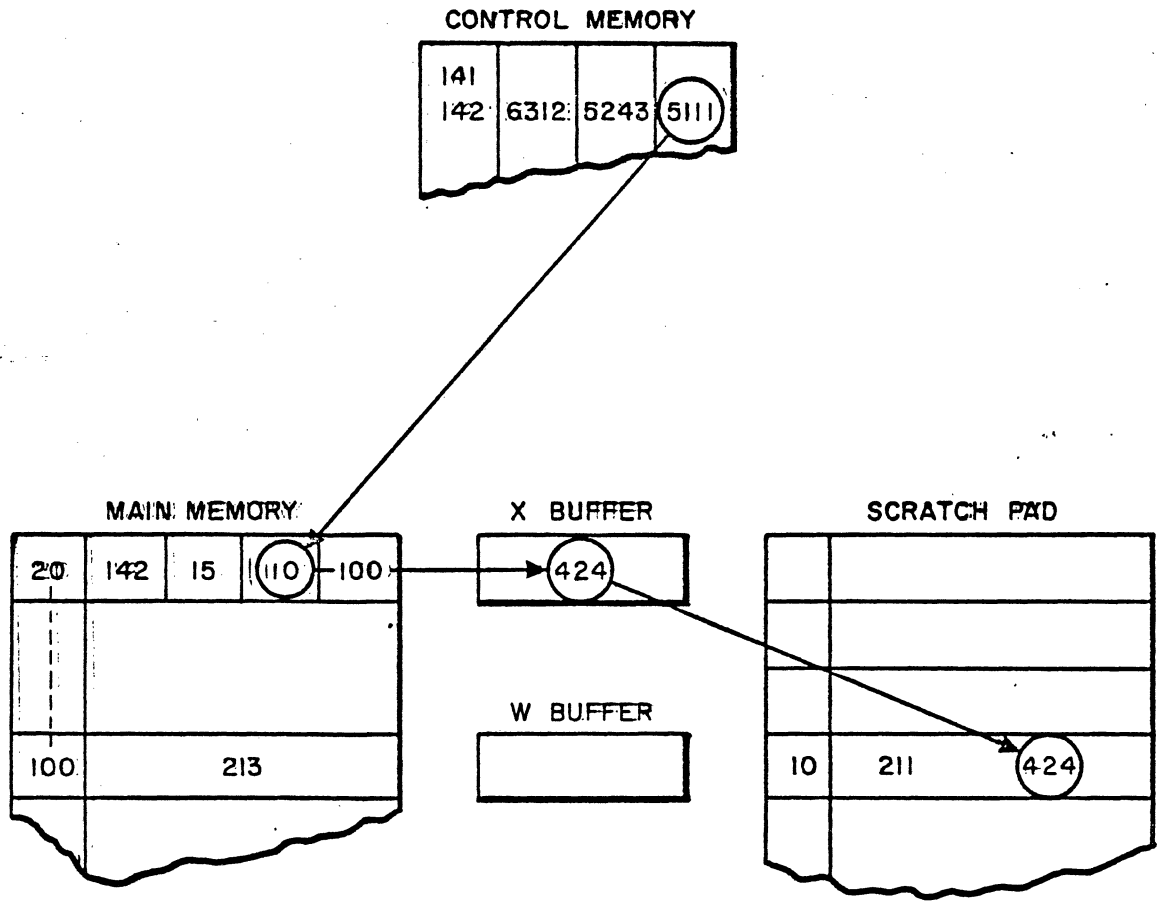


Figure 11- MICRO SUBROUTINE EXAMPLE

SESSION 4

Language and Processor Considerations

for Spaceborne Software

Chairman: Ralph B. Conn
Aerospace Corporation
San Bernardino Operations

(page 186 blank)

SUMMARY OF SESSION 4

Ralph B. Conn, Aerospace Corporation, San Bernardino Operations

The session on Language and Processor Considerations for Spaceborne Software was particularly successful in generating discussion. One hour and fifteen minutes were allotted to discussion during the session. This time was too short, so additional time was allocated resulting in a total discussion time of over two hours. The following text presents in summary form the discussion pertinent to each of the papers plus the chairman's summary of the general conclusions.* Individual questions and answers are not presented since the discussions were not recorded verbatim.

NELIAC was selected because of the familiarity of General Precision programmers with this language. The amount of NELIAC-generated code in the final program was decreased because an increase in storage efficiency was required. Hand coding is estimated to be about 20% more efficient in utilization of storage space, and about 10% more efficient in execution time. The NELIAC compiler was produced very inexpensively using a boot-strap technique. Code efficiency was not considered an important consideration. NELIAC programs were prepared using a specially selected set of 48 characters.

An auxiliary tape memory is in the porposal stage for the Apollo computer. Half of the Apollo computer memory can handle interpretive programs, the rest being devoted to utility programs, subroutines, input/output, etc. The interpretive method used in the Apollo computer minimizes the need for support tools. Both in-house program validation and validation as required by NASA are performed. The proposed automatic documentation is expected to help NASA by eliminating unneeded material. The astronaut cannot alter the program in flight. There is about a four-month lead time to build and install an Apollo computer program memory. Three months are devoted to the building of the memory and one month to installation. Changes may be made during the three-month manufacturing period if the memory portion to be changed has not been assembled.

* Use of discussion notes taken by M. I. Halpern and H. J. Ilger is gratefully acknowledged. However, the session chairman assumes the responsibility for the veracity of the discussion summary.

enough information about the status of development and acceptance of PL-1 in the software community to make it an acceptable choice. The opinion was expressed several times (by industry members of the audience) that the necessity for the use of a common language should be approached very carefully. It was suggested that the decision to use or not to use be left to the project manager. There seemed to be fairly general agreement that use of a standard language to state equations, algorithms, and program logic would be a desirable first step.

The specification of a common language is only a part of the total spaceborne software development problem. The opinion was expressed that the coding of the equations is only a very small part of the spaceborne software problem. Another very important part concerns the managing of the development of spaceborne software systems. The common language aspect of the problem is perhaps the most tractable and has received the most emphasis by the SDC study. Technical direction by the Air Force/Aerospace was a big factor in approaching the language aspect first.

Air Force representatives said that a standardized spaceborne programming language is desirable for application to aerospace problems, one has been needed for a long time, and work should start immediately on the development of a standard language. There did not seem to be general agreement among those present as to whether or not one standard language could cover all problems.

The existence of a syntax-directed, semantic-directed translator, written in a widely accepted language, would be a strong step toward making implementation of a common language practical.

Language Features of the Apollo Guidance Computer

by

T. L. Lawton

MIT Instrumentation Lab.

(Although this paper has been received, it has not been cleared for open publication and, therefore, could not be included in the proceedings. Open publication of this paper has been requested.)

(page 192 blank)

Application of NELIAC to Aerospace Programming

Dr. G. Graham Murray
Kearfott Products - San Marcos Division
General Precision, Inc.

This is a brief report on the experience of the Kearfott Products Division in applying the NELIAC compiler to an inertial navigation problem. The culmination of this effort was a series of system flight tests at Holloman Air Force Base, under the USAF's Mark II Comparative Evaluation, the purpose of which was to gather data on various inertial navigation sets. It was stipulated that the test program would be on a "first come -- first served" basis with a deadline of 1 May 1965, after which no system would be allowed to enter the test program.

In December of 1964, Bell Aerosystems selected the Kearfott L 90-1 integrated circuit computer for integration with the Hipernas III platform. Both companies desired that the combined system be entered in the Holloman evaluation. But only four short months -- January through April -- were available for the development of interface hardware, ground support equipment, checkout, and delivery. The deadline of 1 May was to be absolute. So far one particular in this narrative has been omitted: It was also required that a complete operational computer program be prepared and delivered with the hardware.

On December 21, the Kearfott part of the project was initiated at a kick-off meeting at the San Marcos facility in San Diego County. As subcontractor to Bell, Kearfott was to refurbish the L 90-1 because of wear and tear due to an extensive cross-country tour by truck and trailer, to build an input/output unit, to modify the existing ground support equipment, and to prepare the computer programs.

It was first necessary to halt the tour, which had reached Cleveland in a heavy snow storm, and give instructions to have the computer brought back. The truck and trailer, containing the ground support equipment, spare parts, and other specialized test gear, were driven day and night. By early evening on December 24, the truck and trailer, with their contents intact, arrived at San Marcos. Everything required was now at hand for the extensive hardware modifications and new equipment design.

Such sudden events and changes in project direction are not unique, but rather typical in the aerospace industry. There is no need to

Figure 3 shows the block diagram of the computer. Because of the black-and-white reproduction it is impossible to trace the red lines which represent parallel transfer of data. However, a general statement can be made: All data in and out of the memory is in parallel. Other parallel paths are between the address register and the instruction counter, from the F (Field) Register to the Address Register, and from the Instruction Register to the I/O Address Register.

A unique feature of the L 90-1 is the Sigmator, represented by the R and S glass delay line registers. The Sigmator may be regarded as a separate input/output incremental machine functioning independently of and simultaneously with the central arithmetic unit. The Sigmator contains its own program furnished from the main memory. This program may be changed to fit a particular mission and can be altered on order of the central arithmetic unit. The central computer communicates with the Sigmator in much the same way it communicates with main memory.

The following paragraphs briefly describe the functions of the Sigmator:

Asynchronous Pulse Inputs: Positive and negative pulses may be summed by the Sigmator without intervention by the central computer. This type of operation has been used in the past with such diverse systems as star tracker, camera control, Doppler, etc. Maximum pulse rate is 11 Kc (higher with special logic). The number of channels accommodated is dependent on system requirements.

Pulse Outputs: Pulse rates from 0-11 Kc (higher with special logic) may be generated by the Sigmator for use in such equipment as inertial platforms (pulse torquing). The central computer is involved only to the extent of furnishing one word of information to the Sigmator when a rate change is required. The number of channels is dependent on system requirements.

Integration: Real time integration is performed "off-line" by the Sigmator with no intervention by the central computer. For example, the velocity in a particular channel (computed by summing velocity pulse inputs) may be integrated to determine position. Also integrations may be initiated by the main computer by furnishing two words, integrand and initial value, to the Sigmator. The number of individual integrations performed is dependent on system requirements.

In Figure 5 some of the important features of the language are presented. The ability to use machine coding when desired proved very valuable on the project under discussion. Also worthy of note are the Boolean tests and nesting of comparisons possible in NELIAC. A short example in the NELIAC language, illustrating these points is given in Figure 5.

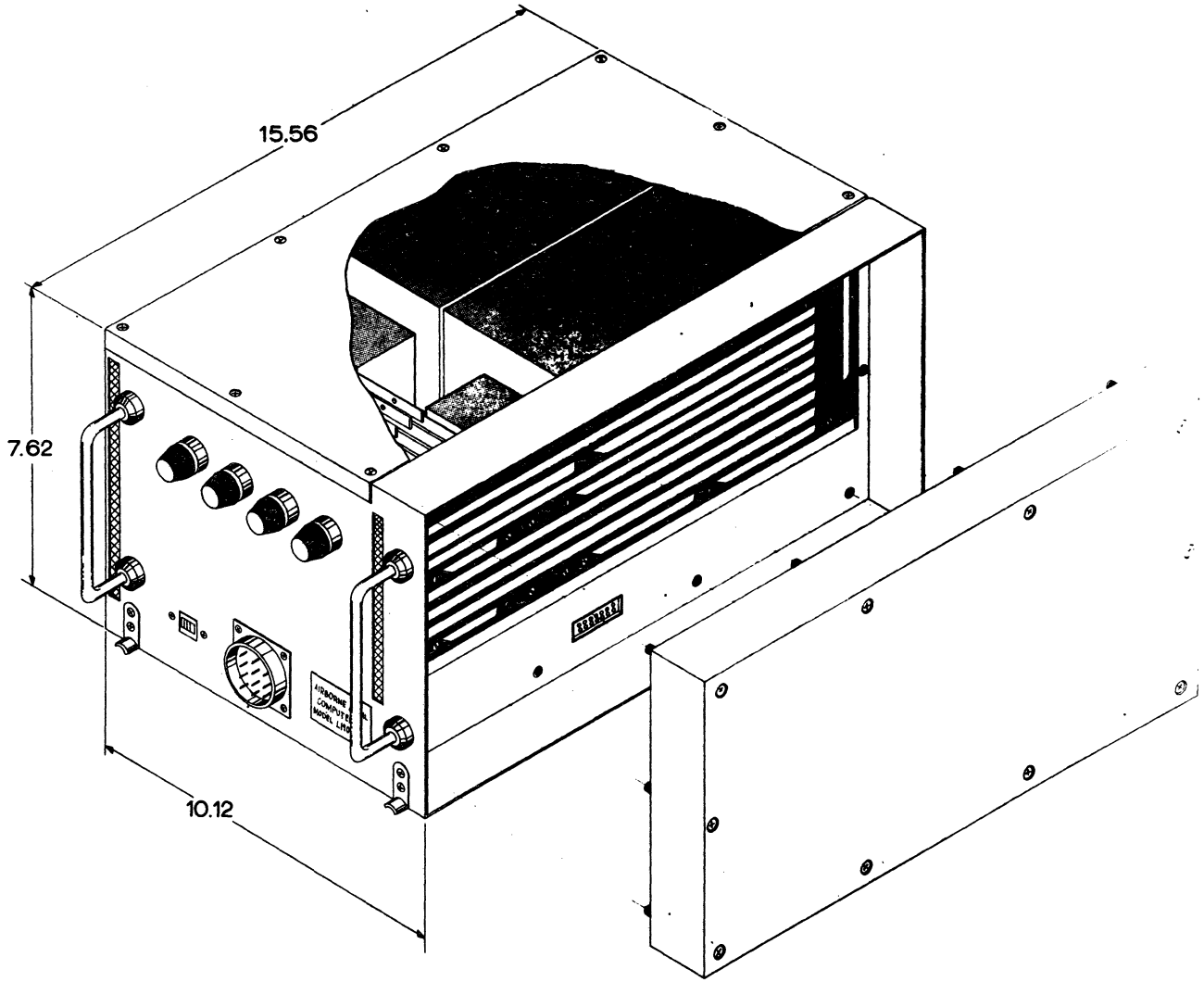
As can be seen from Figure 6, the Guidance Program required approximately 7,000 words. Before these programs could be prepared, it was necessary to completely redo the mathematical formulation, which had been prepared for a DDA type computer. Hence, the magnitude of the programming task was considerable. The Laboratory Calibration Program, loaded only as required, uses auto-collimator inputs to compensate for bias, scale-factor, and misalignment in the inertial instruments. This program alone was 3,000 words. On the other hand, the Operational Program was nearly 4,000 words in length.

The first part of the Operational Program, called Vertical Gyro Alignment and Calibration, consists of coarse level, coarse align (gyrocompassing) and torque set. Upon completion of the latter, data is gathered to permit computation of the vertical gyro mass unbalance. This routine requires 300 words and is called upon perhaps once a week.

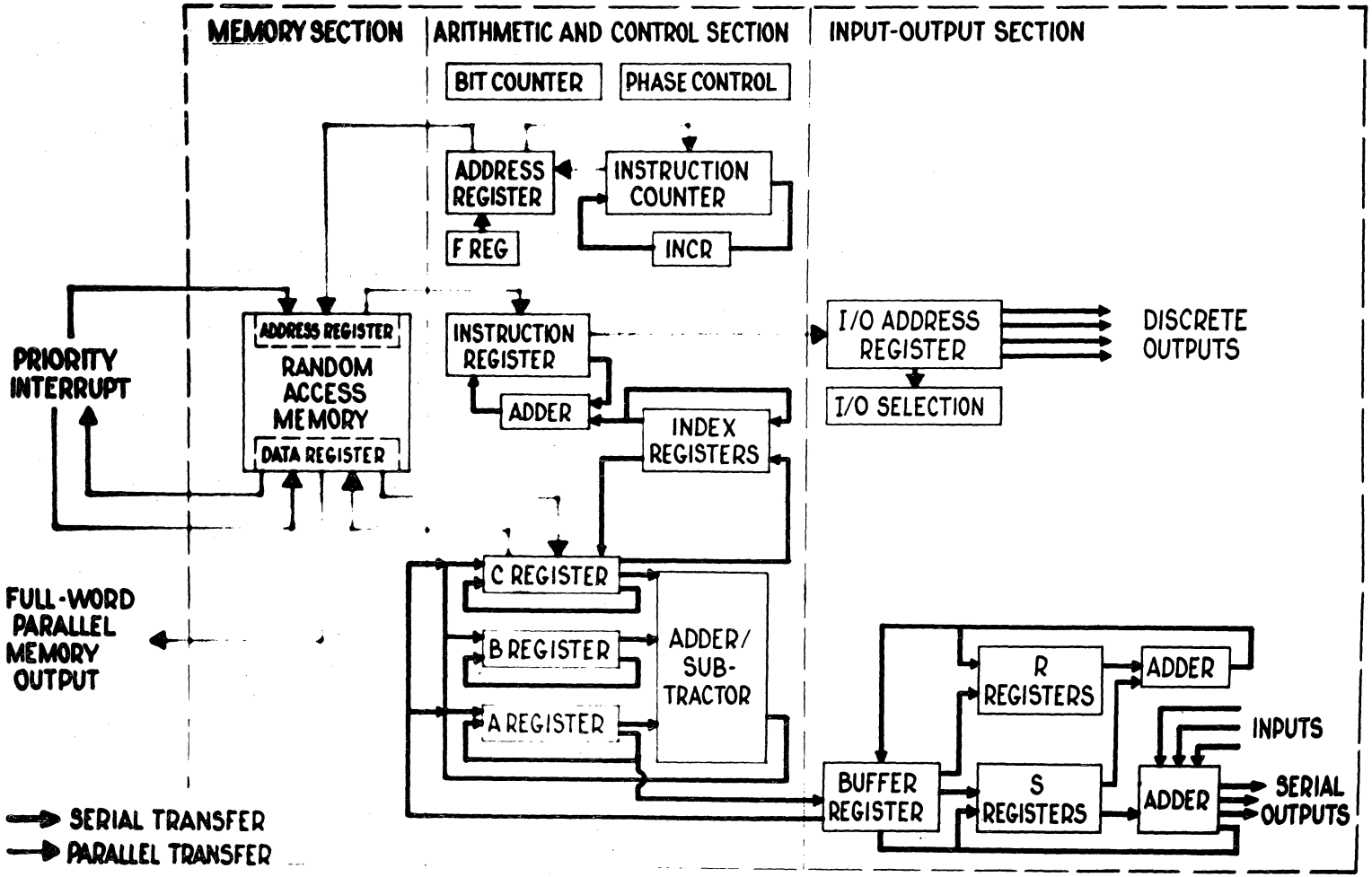
In the 2100 - word Platform Alignment Mode coarse leveling at maximum slew rates is followed by fine leveling based upon earth-rate and velocity meter inputs. Compensation for accelerometer errors and gravitational anomalies are applied to the velocity meter inputs. Torque signals are corrected for gyro errors. Finally, gyrocompassing and torque-set takes place.

1500 words are reserved for the Navigation routine. In this mode the platform is kept level with respect to the ellipsoidal earth model. To maintain the level condition, the computer will provide continual torque signals at earth-rate and, as the aircraft moves over the surface of the earth, the required rotation rate. This rate is computed from horizontal velocity components obtained by applying compensation for accelerometer errors and gravitational anomalies to the velocity meter inputs. Finally, latitude and longitude are computed using the navigational matrix, whose elements are updated by differential equations.

L 90-1



L 90-1 SIMPLIFIED BLOCK DIAGRAM



NELIAC COMPILER

COMPILES RAPIDLY

MACHINE CODING POSSIBLE

BOOLEAN TESTS

NESTING OF COMPARISONS

EXAMPLE:

```
IF X LSS .3 THEN .1=Y$
IF .3 LEQ X LSS .7 THEN
.24=Y$ .36=Y$$
```

CONSIDERATIONS IN SELECTING A SPACEBORNE PROGRAMMING LANGUAGE

T. C. Spillman, IBM Federal Systems Division

1.0 INTRODUCTION

This paper discusses some of the language features that will serve as the basic tools necessary to help solve the spaceborne computer software problem. This discussion is limited to features that enhance the problem-solving capability of the language, and does not include consideration of peripheral features, such as debugging aids. Although the peripheral features are important in considering a language from the actual programming point of view, they are not important in considering the capability of the language to solve application problems.

The results of this discussion indicate that no existing programming language offers a satisfactory solution to the spaceborne software problem. Although languages such as PL/1 and JOVIAL solve portions of the problem, neither language encompasses the total problem.

2.0 THE SPACEBORNE COMPUTER SOFTWARE PROBLEM

A point mentioned by the SDC Spaceborne Software Systems Study was that if the spaceborne software problem were sufficiently well defined, a problem-oriented language could be designed to attain a solution. However, the study also points out that the spaceborne software problem is not well defined. Therefore, a very flexible procedure-oriented language is necessary to help solve the problem in its present form; this language could eventually serve as the basis for future problem-oriented languages. This suggests that a spaceborne programming language should be capable of creating its own compiler. Although compiler writing is not necessarily a spaceborne-language application problem, there are obvious advantages to the spaceborne community to having the spaceborne language compiler written in that language. Also, any language that is not capable of creating a compiler is probably not sufficiently flexible or efficient for the spaceborne software problem.

Although the spaceborne software problem cannot be defined precisely, an attempt can be made to characterize it sufficiently so as to determine some of the language features needed to solve the problem. The following areas should be considered when selecting a spaceborne programming language:

a. The language should have a powerful scientific computational capability.

3.2.1 RANDOM ACCESS DEVICES. When creating a random-access file, the programmer needs a means of requesting a specified amount of space on a particular type of device. This space should optionally be assigned in contiguous blocks on one device, or in noncontiguous blocks on one or more devices of the same device type. The positioning among the noncontiguous blocks introduces some problems when writing files in a nonsequential manner. However, the noncontiguous block concept is very valuable in certain environments, since it removes the necessity to reorder the data on the device as device space becomes fragmented. Permanent files that use randomizing techniques to transform key fields into the location of records within the file are probably best stored in a contiguous area. Temporary files that are accessed sequentially and all files whose records are chained together should be stored in system-assigned noncontiguous areas.

The I/O statements that reference the files must allow the programmer to access the file directly, thus specifying the location of the record within the file; they must allow the programmer to access the file sequentially, thus effectively referencing the device as if it were a tape file; and they must allow the programmer to chain records of a file together and to search through this chain to retrieve a particular record. The language notation used to accomplish these facilities should be symbolic, consistent with the philosophy of reducing system overhead and maximizing the flexibility of device usage.

3.3 Maximizing Hardware Usage

Maximizing the use of the hardware can be very critical in the spaceborne environment, since hardware minimization is often an application requirement. The language should be able to efficiently manipulate the contents of main memory, to specify asynchronous program execution, and to control hardware and software interrupts, including unsolicited external interrupts.

3.3.1 DYNAMIC DATA STORAGE ALLOCATION. The language should contain statements that allow the programmer to allocate and free data storage areas. Multiple allocations of the same area should optionally be mechanized by a push-down stack or association of an indirect address (pointer) with each of the multiple areas. A minimal number of restrictions should be placed upon references to these areas or contents of these areas. A contiguous space situation similar to that previously mentioned for I/O exists when core space is allocated dynamically. Although the control system may contain routines that reorder programs in core, the language should allow data storage allocation to be both contiguous with the allocating program and noncontiguous. Again, noncontiguous allocations may allow the system to operate more efficiently.

3.3.4.1 Compiler-Oriented Interrupt Conditions. A compiler-oriented interrupt condition is characterized by its method of detection. It is not associated with a hardware interrupt condition; rather it is detected by compiler-generated code. The interrupt action specification must be associated with the compilation of the statements that cause the interrupt. Thus, the specification must be on a statement by statement basis, or in logical statement groupings if more than one statement is to be affected.

3.3.4.2 Solicited I/O Interrupt Conditions. A solicited I/O interrupt may occur during or upon completion of an I/O operation. The action specification for such a condition should be associated with the I/O statement itself. Programmer control over this type of condition is important in that it allows the programmer to logically remove I/O from other processing, thus maximizing overlap.

3.3.4.3 Program Completion Interrupt Condition. The program completion interrupt condition is similar to the I/O completion condition and would be used with asynchronous program execution. The interrupt action specification should be associated with the statement that initiated the completed program. The initiating program will be interrupted, and the interrupt action will be taken.

3.3.4.4 Arithmetic Interrupt Conditions. An arithmetic interrupt condition is associated with the execution of an arithmetic operation. Detection of these conditions is effected by hardware interrupt and can be enabled or disabled under program control. The interrupt action specification should be associated with the enabling statements.

3.3.4.5 Unsolicited External Interrupt Conditions. Unsolicited interrupt conditions are associated with the occurrence of an external signal specifying that current program execution is to be suspended while other processing takes place. The programmer needs a method of specifying that a statement or group of statements will process the external signal. The programmer must also be able to enable and disable the external interrupt.

3.4 Object Code Optimization

Although the problem of object code optimization is mostly a compiler problem, it does have an influence on the language. The language should not cause "worst-case" code to be generated, e.g., if the language does not distinguish between input and output parameters in a subroutine linkage, some unnecessary or "worst-case" code may have to be generated. However, the language may contain features that cause large amounts of code to be generated if these features add problem-solving capability to the

STANDARDIZE THE SYSTEM, NOT THE LANGUAGE!

M. I. Halpern

Lockheed Palo Alto Research Laboratory,
Lockheed Missiles & Space Company

The spaceborne software problem, as the four-volume Preliminary Results of the Spaceborne Software System Study acknowledges (I, 19), contains no unique element; it is simply a collection of all the standard software problems, all presented simultaneously and at their most critical. What is novel in the situation before us is that the solution space within which we may move has not yet been cluttered up with faulty and fragmentary answers that we cannot afford to abandon. There are as yet in the spaceborne software picture few or none of the arbitrary constraints that make the general software problem so discouraging to contemplate; we have here in this special situation the precious gift of starting afresh. It is a little disturbing, therefore, to see how similar the current debates about spaceborne software are to those held in past years on the subject of command and control programming systems.[3] There is some reason to fear that we are moving toward a duplication of the situation that prevails in C&C programming: so heavy a commitment to a permaturely chosen standard language that no new one, however clearly superior, can hope to displace it. The C&C situation represents an honest and perhaps unavoidable mistake; to recreate that situation in spaceborne software, despite the lesson of that earlier experience, would be unforgivable.

What we have learned in the last few years is not that we chose the wrong language for C&C - JOVIAL may well be the best of those proposed - but that any such decision was needless, and that the debate leading to it was concerned with the wrong subject at the wrong time. We know now that

(1) A programming language, and the processor that translates and otherwise supports it, are two distinct and separable things

that would follow from the strictest standardization on a single language and compiler:

(1) There would be but one translator to design, construct, maintain, and document

(2) Programmers and program packages would be as freely interchangeable among various space-vehicle programming projects

(3) The creators of the processor would be free to concentrate their energies on what is properly their business, the internal improvement and extension of the processor itself

(4) Continuity would be established across machine varieties and generations, with upward compatibility assured.

How long will it be before we see a system of this description? About minus 2 years; it was in 1964 that the first specimen of this new genus processor assumed operational status,[4,5] and substantial experience has since been acquired in using it to implement various languages[7] and to run on and compile for a variety of machines. There were, it turns out, only one or two wholly new developments needed to implement this processor; for the most part it involved merely the putting together of a number of known features, so organized that their full exploitation was possible for the first time. It will be necessary in explaining this to review some fundamentals.

We are considering a processor that is to be capable of being introduced somehow to an indefinite number and variety of programming languages, all of which it must be able to translate from then on. What is required in the design of such a translator is made clear if we think of a programming language as having three dimensions or aspects:

defining a programming language. The macro instruction is not properly just another item in the machine-language coder's bag of tricks; it is a way - the best way, we contend - for the higher-level language designer to implement his language. It is in this role and only in it that macros are recommended in this and the writer's other papers on the subject. More will be said of their use after the other two programming-language dimensions have been dealt with briefly.

The notational dimension has been the subject of much rearch in the past few years; it is this aspect of language that the various metalanguages, meta-assemblers, metacompilers, and the like have been concerned with. It might have been expected, therefore, that this dimension, like the functional, would be suitably provided for by existing techniques. In fact, not one existing higher-level programming language can be described in its entirety by any formalism so far described, and it is just in this heavily worked-over field that original work has had to be done by the writer and his associates in order to realize a generalized processor. For the reader curious about this point, we suggest that the reason why the metalinguistic formalisms so far proposed have been of little practical consequence is that they have placed too high a premimum on mathematical elegance and linguistic suggestiveness. (Since this thesis is part of a lengthy argument that is to be published elsewhere, and is not essential to the present discussion, no support for it will be offered here.) If such considerations are renounced, and all our energies are given to the development of simple, explicit devices for describing real programming languages like FORTRAN and COBOL (as opposed to nonreal ones like arbitrary subsets of ALGOL), there results a notation-describing technique capable not only of describing them straight-forwardly, but even of describing a nontrivial subset of the natural language. Detailed exposition and illustration of this technique has been published.[5]

That group would continue in existence (perhaps not full time) as a maintenance and revision committee; their chief task would be to poll users of their language at reasonable intervals for comments and suggestions. They would not merely invite and await such feedback; they would go to the working programmer and actively solicit it. The user would be encouraged to devote some thought to the improvement of his language by his knowledge that, being a mere collection of macros, it could be quickly modified. (If the processor in use were a conventional compiler, few programmers would be naive enough to expect any results from their suggestions until long after the occasion that prompted them had vanished, and their suggestions would accordingly be few and half-hearted.) In this fashion, advantage would be taken of the experience of all programmers using the language, but not by giving each of them access to a macro-defining capability, and a vague charter to write his own language. It would be a guiding rule, of course, that no operational program could be rendered incorrect by any proposed change. This means that any modification of an existing operator would have to include the older version as a special case, or else some slight burden-a new operator name to remember, for example - would have to be imposed on users of the new version in order to protect older programs.

The very different way in which the SAGE project leaders tried to use macros was due to their committing, in common with most of the programming community, the genetic fallacy - the fallacy of supposing that to know the origin of a thing is to understand it. Macro-instructions did first appear as adjuncts to assembly-language programming, and for most software technicians have never succeeded in rising above these disgraceful antecedents. One of the many misapprehensions that follows from this view of the macro-instruction as merely a tiny step upward from the machine language is the notion that it is necessarily wasteful of space [Preliminary Results (III, 133)] and generally conducive to inefficient coding. In particular, the charge is

expand into coding that would, at excution time, compute the effective address of each such instruction in the program before permitting its excution. If the effective address turned out to be within the bounds set by the programmer, execution would be permited; if not, it would be ignored, and transfer would be made instead to an error routine specified by the user. The result would be the trapping of the first transfer instruction that attempted to pass control outside the range correct for it, and before it could wipe out the evidence permitting exact identification of the bug.

In memory protection, a very similar trick permits a great deal more selectiveness than any hardware scheme so far proposed. By redefining as macros all machine-language instructions destructive of memory, for example, a programmer can make any segment of memory "read only" for his program's purposes. He can be extremely precise in saying just what can be done, and what cannot, to any unit of memory. He can specify by this means, for example, that address fields alone between (say) locations 4025 and 4712 may not be touched; he is not forced to choose between leaving a memory segment totally unprotected and making it totally unalterable, and he is not forced to treat memory in fixed segments of 1024 words each.

Further extensions of this technique promise to give the user close control over timing problems as well: for this purpose, each machine-language instruction would be redefined as a macro that would expand, when used, into coding that would compute its execution time. The programmer would be able at any point to order the recording of the exact execution time elapsed since some specified datum point, either for the built-in "worst case" condition or for such data as he provided.[2] All these techniques for dealing with bugs, with memory protection, and with the timing problem are available only to the macro-user, and they are available to him on a very easy, informal basis. Provided he knows machine language, he can implement them himself, can insert

edition, Automatic Programming Information Bulletin No. 23 (October 1964), ed. R. H. Goodman, Brighton College of Technology, Brighton, England

3. -----,"XPOP: A Command and Control Programming System," Datamation (December 1964), 39 - 48

4. -----, "XPOP: A Metalanguage Without Metaphysics," Proceddings of the [1964] Fall Joint Computer Conference, pp. 57 - 68

5. -----, A Manual of the XPOP Programming System, 2nd ed., Lockheed Missiles & Space Company, Palo Alto Research Laboratory, Palo Alto, California

6. -----, "Computer Programming: The Debugging Epoch Opens," Computers and Automation (November 1965), 28 -31

7. M. Roger Stark, ALTEXT Multiple Purpose Language Manual, Lockheed Palo Alto Research Laboratory, Lockheed Missiles & Space Company, Palo Alto, California (revision of 27 September 1965)

(page 222 blank)

PHOENIX COMPILER LANGUAGE
and
SOFTWARE SYSTEM

A. J. Stone

Hughes Aircraft Company
Culver City, California

Presented at

SPACEBORNE COMPUTER SOFTWARE WORKSHOP
Sponsored by the Air Force Space System
Division and The Aerospace Corporation
September 20-22, 1966

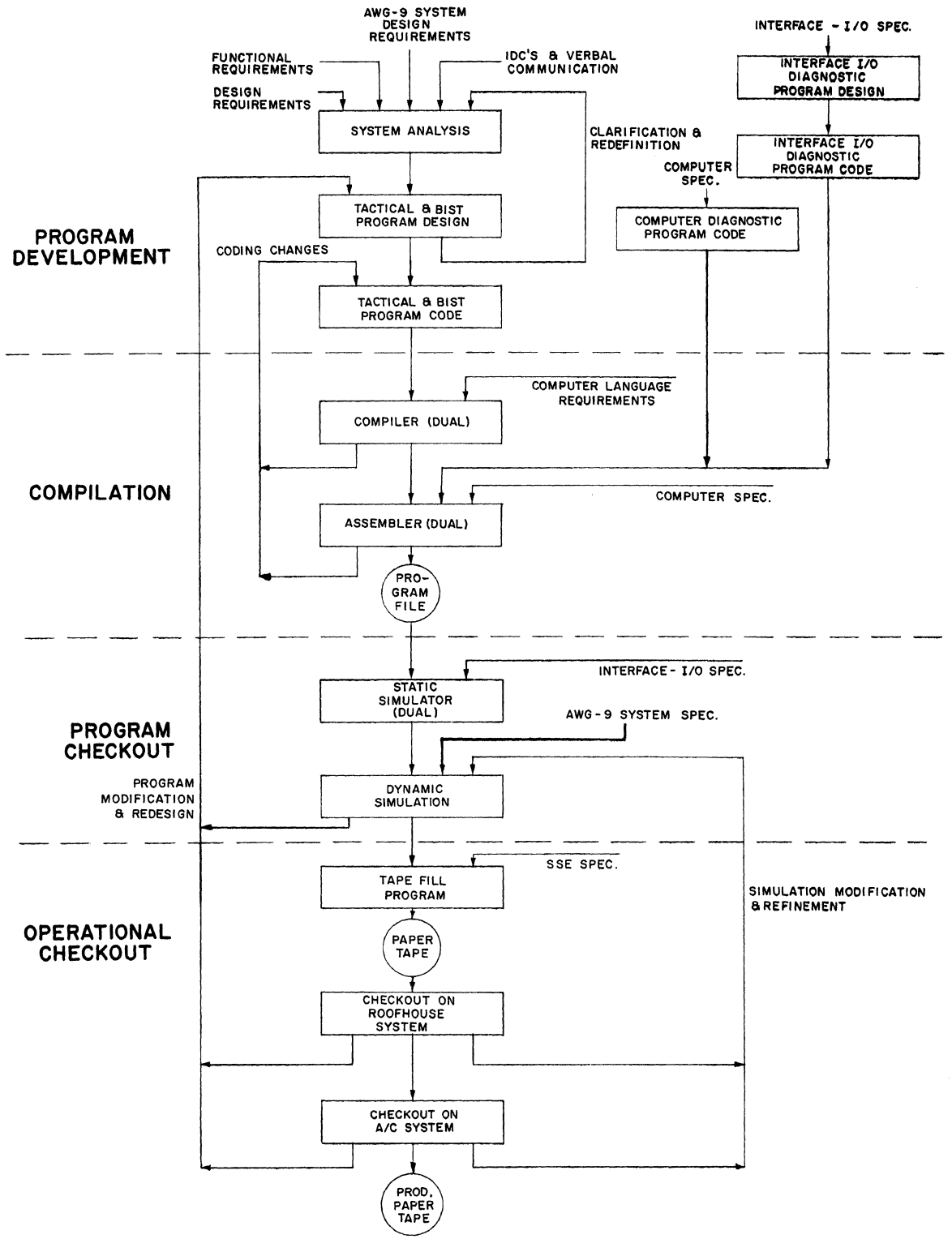


Figure 1 PHOENIX COMPUTER PROGRAM PROCESS

In one computer run, the integrated system of programs can perform any of the following processing functions upon any or all of the flight programs in the system files:

Symbolic editing (including adding a new program or changing existing ones)
Compilation
Assembly
Static Simulation
Dynamic Simulation
Generation of a fill-tape

In addition, control cards may contain symbolic references to program variables. For example, this allows one to specify simulation parameters symbolically without first inspecting the results of an assembly.

The entire software package is a dual system in the sense that it will handle either the UNIVAC or CDC PHOENIX computers, depending on control card option. The software will also deal appropriately with programs which reside in NDRO memory as well as programs which reside on an airborne magnetic tape and are executed from DRO memory.

A brief description of each software system element is presented below:

Control Program - Acts as a monitor and operating system for the rest of the software system, processes all control cards and calls the appropriate software programs into memory. (However the entire system itself operates under IBSYS in a multi-level overlay structure.)

Update-Edit - Permits symbolic changes and editing of the master program file stored on magnetic tape and creates a new updated master file.

Compiler - Translates programs written in a procedure-oriented, machine independent programming language (METAPLAN) into symbolic assembly code for either of the two PHOENIX object computers.

Meta-Assembler - Translates symbolic assembly language programs into binary (absolute or relocatable) code by 1. converting mnemonic operation codes to their binary equivalent, 2. generating and assigning binary addresses to symbolic locations, 3. allocating storage for instructions and data, and 4. converting decimal or octal constants into binary. Two versions of the Meta-Assembler process code for the two PHOENIX object computers.

METAPLAN LANGUAGE

METAPLAN is a procedure-oriented language with features designed for real-time and system programming. It is quite open-ended and easily tailored for specific requirements. The nucleus of the language was originally used by Programmatics Inc. for systems programming. Hughes and Programmatics designed a considerable body of language modifications and extensions for the requirements of the PHOENIX real-time application and jointly developed the entire software system.

Some properties of interest are:

1. Flexible address expressions.
2. Distinction between the name of a memory location, its contents, or its indirect contents. Numeric addresses can be used as well as symbolic.
3. Data Description features for describing "ITEMS and FIELDS" including attributes such as signedness, fixed-point fractional scaling, and field position and length within a computer word. This allows quite complex data structures to be described and operated upon, as well as bit and field manipulation.
4. Facilities for dealing with time constraints.
5. Automatic generation by the compiler of any necessary masking, shifting etc. to line up fields and/or binary points.
6. High object code efficiency in terms of both storage and execution time.
7. More programmer control over factors affecting generated code than in conventional languages.
8. Symbolic assembly language is a subset of METAPLAN and can be freely intermixed with it.

Other properties and characteristics of the language are as follows:

A METAPLAN program consists of a series of statements. A statement in general contains a command and argument and may be labeled. The complete set of METAPLAN commands is shown in Figure 3.

There are two types of statements: declarative and imperative.

Declarative statements describe data and parameters. Object computer differences are accomodated in the declarative portions of a program (e.g., a "word-length" parameter can be defined and used in the imperative code where appropriate). To change a METAPLAN program from one PHOENIX computer to the other, only the declarative, data description portion of the source program is changed.

ITERATION	FOR {ARG.} FROM {ARG.} THRU {ARG.} BY {ARG.} ⋮ STEP {ARG.}
	SEARCH {ARG.} FROM {ARG.} THRU {ARG.}
	IF STATEMENT

Figure 3 METAPLAN COMMANDS (continued)

METAPLAN COMPILER

A few properties of interest are worth noting regarding the implementation of the METAPLAN compiler. The compiler can be considered both syntax and semantics directed. The syntax of the source language is specified by syntax equations within the compiler. The semantics of each command or construct are defined by PROCedures. These are written in terms of a meta-language consisting of primitive PROCedures and directives. Changes to the source language and/or generated object code are accomplished by appropriately changing the syntax equations or the PROCedures.

While the PROCedures themselves could be compiled each time along with a source program, it is more efficient if they are compiled separately into binary and subsequently used in their binary form. However, it is conceivable that a user might define some special METAPLAN commands for his own use and compile these PROCedures along with his program.

CONCLUSIONS

The principal objectives which influenced the design of the PHOENIX Software have been met successfully.

1. Efficiency The compiler-generated object code is better than 90% efficient when compared to hand-generated machine code. This efficiency holds for both PHOENIX object computers and for both the number of memory locations used as well as execution time on the object machines. A 90% efficiency minimum was one of the primary design goals, since conventional compiler efficiencies would be entirely inadequate for the time and memory limitations of the application. The efficiencies were quite rigorously measured, based on the weighted average of three representative sample problems.

2. Flexibility The requirements of an airborne, real-time application demand language features and an inherent adaptability not found in most existing compiler-level programming languages. This type of application is characterized by changing and/or incomplete specifications of both the airborne hardware and flight program. Consequently, many of the detailed software requirements may not be evident in the early stages of such a project. Considerable tailoring and modification of METAPLAN and other parts of the software system took place without disrupting either the flight programming effort or the software development itself.

3. Machine Independence Except for differences in data description, only a single flight program has been written even though this program must operate on two dissimilar computers. The task of providing this dual object machine capability has been left to the software.

(page 234 blank)

EFFICIENCY CONSIDERATIONS IN PROBLEM-ORIENTED PROCESSOR DESIGN

Vilas D. Henderson and E L Smith
Logicon Inc.

ABSTRACT

The approach is taken that efficiency of any one component of the total spaceborne software system is affected by the efficiencies of all other components and that of the composite system. The functional requirements of current and future spaceborne software systems are reviewed, and a problem-oriented processor design concept is advanced to establish a basis for the formulation of software system efficiency criteria. These criteria are then developed for the composite software system, for the processor, and for the object programs it generates. Finally, tradeoffs are suggested between software system efficiency and cost.

1. INTRODUCTION

The theme of this paper centers on the problem of developing an efficient spaceborne software system. We firmly believe that a really good software system for spaceborne applications can only be created through a total systems approach. The viewpoint is taken that efficiency in the source language, the problem-oriented processor, and resultant object code are all interdependent. This viewpoint makes it difficult to discuss the efficiency of any part of the software system intelligently without some insight into the entire system. We have therefore gone to some length to present a processor concept upon which meaningful processor efficiency discussions can be made in the context of a total spaceborne software system.

Although little progress has been made in developing problem-oriented processors as aids for the development of spaceborne software, considerable attention is now focusing on this subject. Background and experience point to the difficulties to be encountered in attempting to design problem-oriented processors that will generate satisfactory spaceborne object code. Experience with present-generation processors for classes of advanced computers has caused a great deal of

complicated flight programs and a consequent increase in the amount of object code that must be validated. The increase in inflight computing capability is making it possible to absorb additional flight functions, such as those performed by digital autopilots, into the onboard computer, and this may require additional computational frequencies. Major additional peripheral processing functions will place greater demands on the computer's interrupt capability and its methodology for parallel data processing. It is not difficult to foresee a requirement for a significant multiprogramming capability to handle the need for multiple computational frequencies and both real-time and non-real-time computational problems.

Among the significant new requirements expected to confront the spaceborne software community, one that stands out is that for inflight programmability. This new requirement may introduce all kinds of factors heretofore of no concern: assuring the sacredness of permanent memory; providing an inflight programming language and an inflight processor; performing inflight validation; and incorporating auxiliary memory capabilities. A second foreseeable new requirement, for a real-time onboard malfunction detection and correction capability, will increase the complexity of onboard software because of extended communications with other subsystems, as well as greater onboard data-handling and decision-making needs. With man in the loop, the nature of any malfunction detection and correction software may be appreciably different from that necessary to support unmanned missions. We also anticipate that major non-real-time computational functions will be superimposed upon real-time functions. Some that are already apparent are trend analysis; inflight simulation and prediction; self-checking and self-validation; inflight compilation; and targeting.

The addition of these computational requirements can be of great consequence in the design of the problem-oriented processor and the specification of efficiency requirements. A thorough analysis will unveil many others of importance; we have just brought forth a few here to show the projected change in the character of preflight and inflight computing to be performed by the onboard computer.

We hope that this introductory discussion has served to point out the importance of deeply understanding the significance of a total spaceborne software system, especially in a functional sense, prior to language and processor design and standardization; in other words, of taking a total systems approach. In subsequent sections we discuss the nature of a problem-oriented processor which produces object code in accordance with efficiency criteria and system and environmental constraints. A particular processor design concept is presented for

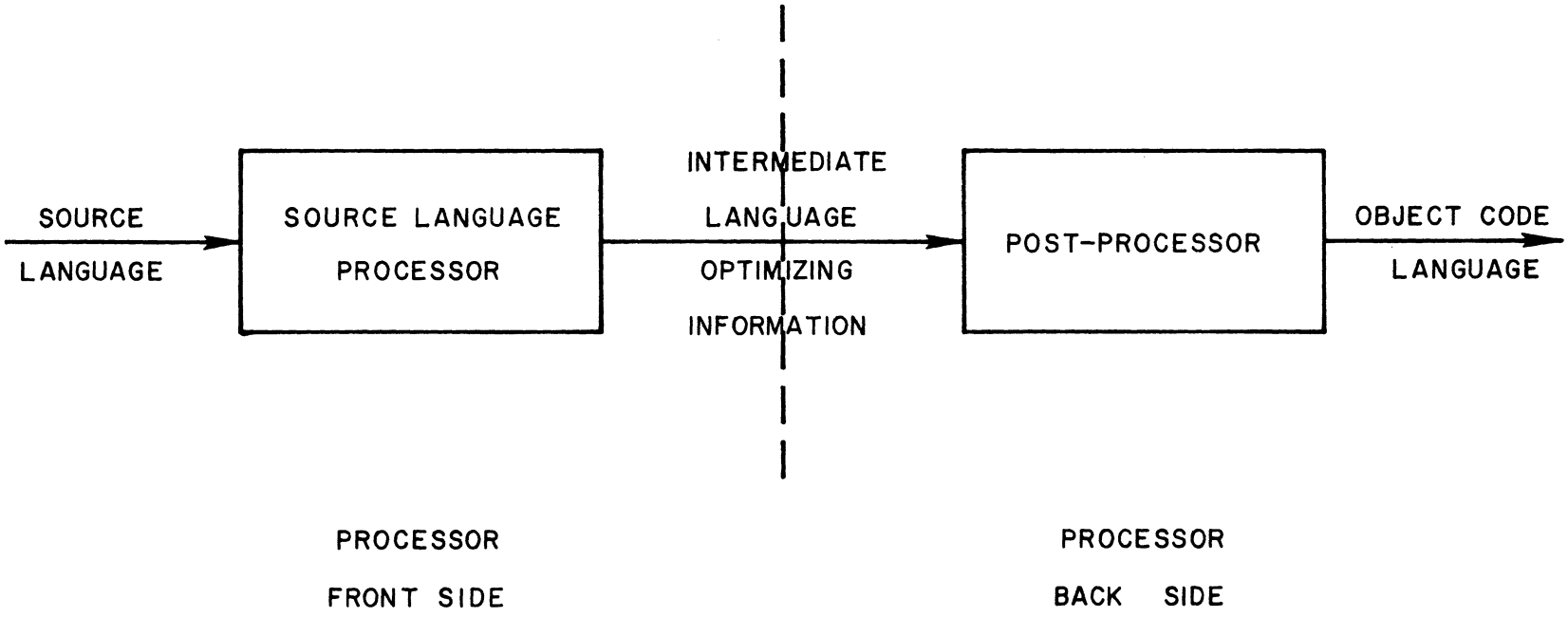


FIGURE 1. THE PROBLEM-ORIENTED PROCESSOR

is lost through translation to a machine-independent language yet is absolutely necessary for the generation of efficient object code? As it turns out, answers to the question are relatively independent of source language. The types of information required include:

1) Actual required size of any given variable, array, or table.

2) Data organization in an array or table and even dependency of a variable on its assigned arrangement in a word -- packed bits, ones or twos complement form, fixed-point scaling, etc.

3) Distinctions between address and normal arithmetic.

4) Bounds on coding sequences which may be considered as a single unit with respect to optimization.

5) Areas of code that may permit extended optimization operations because of external language restrictions.

6) Isolation of calling sequences so that they may be adapted to target machine capabilities.

7) Identification of operations on indices so that machine facilities of index registers and indirect modes may be utilized.

8) Identification of bit and decision sequences so that efficient hardware methods may be used.

It is clear that the ways of obtaining the information from the source language are dependent on the language, but the fact that the information generally can be determined regardless of the particular language is the important thing. This suggests that the development of an intermediate language augmented with an explicit set of optimization information would allow the convenience of simple translation and also would provide a direct source of information for optimization procedures.

Development of generalized specifications for a processor would then require several steps:

1) Definition of an intermediate low-level language through which translation passes.

2) Identification of information items which are useful in code optimization and which are normally available.

3) Definition of a data format which can easily be used by data collection and optimizing routines.

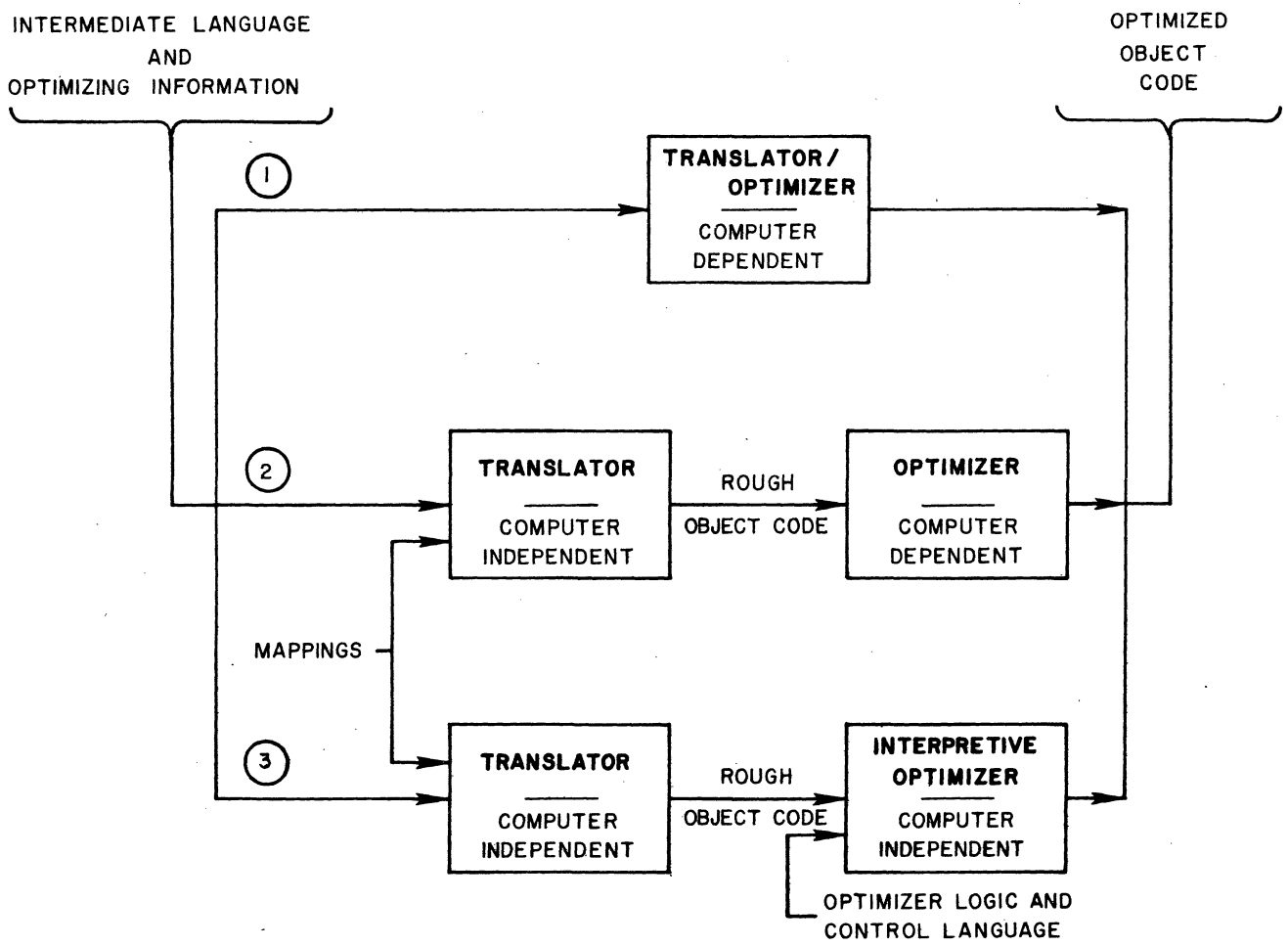


FIGURE 2. POST-PROCESSOR DESIGN CHOICES

version might have no improvements at all but simply be a direct translation. Observation of poor code would allow specification of algorithms to improve that particular type of coding stream. Each time another bad stream was observed, it would be possible to see what caused it and introduce another optimization algorithm. Further, such a procedure would allow direct control over the investment in optimization since each improvement would be a small project and would be useful without completion of others. Optimization effort could be terminated as soon as generated programs were running and satisfying operational constraints.

3. EFFICIENCY CRITERIA

Whether the spaceborne software system and it components are considered efficient obviously depends upon the criteria selected for stating and measuring their efficiency. In this paper we are concerned with the efficiency of the overall spaceborne software system; the efficiency of the target computer object programs as generated by the post-processor; and the efficiency of the problem-oriented processor's own internal workings, particularly the post-processor's.

3.1 Spaceborne Software System Efficiency

Let us again state the need to consider efficiency of the total spaceborne software system. Then in this context our subsequent discussion concerning particular efficiency criteria, and their influence upon the problem-oriented processor, becomes meaningful.

We define an efficient spaceborne software system as one that possesses a structure, organization, and content which permit change, growth, and adjustment to new requirements and constraints without unreasonable compromises in constituent efficiency criteria such as those discussed in Sections 3.2 and 3.3. Now, how does one quantitatively state or measure such characteristics in the system, and how does the specification of overall system efficiency affect the design of the language and the processor?

The answer to the first question must be given in terms of incremental costs and response time relative to some set of software performance standards that are achievable and generally accepted by the spaceborne community. Once these have been selected, improvement in the efficiency of program design can be compared against actual performance of individuals and companies engaged in the development and validation of spaceborne software.

The closely knit relationship among the object program efficiency criteria makes it difficult to treat each criterion (and its fulfillment) as though it were completely separate from the others. In this section we discuss the efficiency criteria from the point of various problem areas and the probable impact on the post-processor that will do the necessary optimization.

3.2.1 Execution Time of Designated Object Program Segments

The multilevel computational frequencies required in most spaceborne applications make spaceborne programs similar in character to those operating in the environment of a multiprogramming system. A distinct and demanding difference, though, is that the various subprograms executing at different computational frequencies must communicate with each other, and there is an absolute interrupt sequencing requirement. Further, multiprogramming operating systems are usually designed to sequence and execute programs in a way that most efficiently uses the total computer system, retaining control over priorities and memory, whereas the real-time constraints of a spaceborne software system do not permit such flexibility.

Any interrupt system for multilevel program execution has a certain amount of overhead associated with it. Depending upon the computer hardware, this overhead can be minimal or of real significance. We assume that, for the problem which this paper addresses, it is not possible to select the ideal computer for a given set of operational requirements -- although this is the best way to guarantee a high measure of efficiency. So the necessity of dealing with a large class of interrupt systems can affect the language, and certainly it affects the post-processor design, not only to meet timing requirements but also to achieve memory and register utilization efficiency.

Suppose a particular multicycle program exists in which there are three real-time computational cycles having frequencies of 100, 10, and 1 cycle per second and respective computing durations of n_1, n_2, and n_3 seconds. Obviously, the computation duration required for each of the 100 high-frequency cycles is critical. The condition

$$100\, n_1 + 10\, n_2 + n_3 + (\text{interrupt overhead}) < 1 \text{ second}$$

must be satisfied. There must be a reasonable relationship among n_1, n_2, and n_3; and in those cases where there is a marginal relationship, i.e., the computational requirements of one loop are very demanding

provided through either the front or the back side. A language such as FORTRAN may be inadequate as the front side source language, just because of the data compaction requirement. Hence it can be seen that the design decision whether to deal with this problem from the front side or the back side of the processor may have considerable impact upon the selection of a source language and the design of the total processor.

Register utilization normally is of no consequence to the programmer when using high-level languages. However, the manner in which registers are assigned by a processor affects the efficiency of the object code, in terms of both the number of instructions employed and the time required for execution. So the question arises whether the control and allocation of machine registers should be retained by the programmer or performed by the processor. Certainly the answer to this question affects the language and processor design.

Another important consideration when dealing with spaceborne computers concerns the allocation of code on the basis of whether a particular segment of memory is classified as nondestructive or destructive. The programmer must be able to control the allocation of code in memory, and for the most part be able to retain control over memory allocation directly or be able to specify the memory characteristics to the processor. Processor design can be greatly affected by this requirement, particularly the post-processor.

3.3 Post-Processor Efficiency

Although our primary emphasis is placed upon the efficiency aspects of the object program produced by the post-processor, some discussion is relevant concerning the efficiency of the post-processor software itself. The criteria we have selected for measuring and stating post-processor efficiency are total operating speed, total memory requirements, and growth potential.

3.3.1 Operating Speed

To require a post-processor to produce optimum object code is contradictory to any requirement for compiling object code rapidly. As long as the post-processor operates in a ground-based computer, we consider post-processor operating speed irrelevant. However, a future requirement for inflight programmability may cause a need for placing emphasis on the inflight processor's operating speed. Of course it is

generated with varying degrees of sophistication. On the one hand, a simple-minded straightforward translation to an operable code on the target computer could be made with no regard for efficiency in the object code. Alternatively, a very complex optimization process could be undertaken to produce object code as efficient as that produced by the most proficient programmer doing machine-language coding.

The simple extreme is only of academic interest in this discussion because the only pertinent efficiency consideration is that it result in an operable code obtained with the least effort, the least cost, and the smallest investment in a post-processor. Thus this extreme is efficient in the sense that there is no optimizing cost. For the most part, however, such code will be unacceptable for use operationally on the target computer. Nevertheless, there may be real advantages to using such code for program design, development, and checkout.

The real question pertinent to the other extreme is: What price should be paid to achieve satisfactory optimization? The cost can always be measured in dollars, but it is also appropriate to equate it to such things as manpower, schedules, and technology advance.

Intuitively it is difficult to see the practicality or even the feasibility of designing and implementing general algorithms for the generation of optimum object code for a variety of computers and for a variety of efficiency criteria. Certainly there is a need to study optimization techniques for classes of computers and to generate practical procedures for optimization covering classes of computers; but this is far out of the scope of this paper. Here we can only assess the optimization problems for various efficiency criteria in a nonanalytic way, paying particular attention to practical constraints and tradeoffs.

Another point to consider is that the definition and acceptance of a spaceborne software system which includes a problem-oriented language and processor mean that spaceborne computers must be sized accordingly. It is safe to say that no problem-oriented processor can produce object code as efficient as that produced by a good programmer in machine language. Up to now, computer sizing has generally assumed machine coding without a factor for inefficient processor-compiled code. The very selection of a computer for a particular application, then, must take processor efficiency into account. However, the efficiency of a processor is not fully known until it has been developed. So great care must be exercised in computer sizing to allow for object code generated by a problem-oriented processor which in all probability doesn't exist when the sizing takes place.

PRELIMINARY SDC RECOMMENDATIONS FOR A COMMON SPACEBORNE PROGRAMMING LANGUAGE

by

L. J. Carey and W. E. Meyer
System Development Corporation

I. INTRODUCTION

A recommendation for the adoption of a common, higher-order programming language for spaceborne software was made to the Air Force Systems Command, Space Systems Division in July 1966. This recommendation was based on a study of spaceborne software systems conducted by SDC in the fall of 1965 and the spring of 1966. This paper is an overview of work done to this date, August 1966, by the SDC Spaceborne Software Systems Study team to identify and specify the characteristics and capabilities of a spaceborne programming language.

The language study was initiated in July 1966. Personnel working on this project include: H. Ilger, A. Tucker, S. Manus, W. Meyer, and L. Carey. As I indicated previously this document is an overview of our investigations to date. We intend to describe:

- The goals for a Space Programming Language.
- The computer programming requirements in the spaceborne area for a programming language.
- A review of existing languages to determine if they can be utilized as is or as a base for a Space Programming Language.
- Our present conception of a Space Programming Language

The goals we have identified for the language are classical. They are particularly appropriate however to the spaceborne community because of the number of contractors, the program production methods, and the computers utilized in the spaceborne software development process.

A. REDUCED LEAD TIME

In order to facilitate reduced lead time, we expect to be able to provide better interface communications between the specification writer and the programmer for the operational computer program, between the engineer and the programmer on a single project, and also between the contractors on various projects, by adopting the higher-order language. We expect to be able to facilitate faster coding and program production by reducing the amount of scripting required by programmers, reducing the amount of attention to detail, and by providing a language which is conceptually closer to the problem statement.

B. REDUCED COST

We hope to be able to reduce the cost of spaceborne software development by facilitating the transferability of programs, especially support programs which would be useful to other contractors working on the same project, or similar projects. A common set of programming tools such as simulation tools, support tools, documentation tools, debug tools, etc., should be available in the common language. The language will facilitate the reuse of programs developed in the early stages of mission planning. Some of these programs can be used directly or with little modification in the operational computer. Lastly, through reduced production time, speedier preparation of programs and reduced manpower requirements the cost of computer program development is reduced.

in implementing language processors to produce code for an operational computer utilizing a compatible general purpose machine.

2. Data Processing Functions

The number of data processing functions, in the time period identified, is going to increase drastically. Information processing is just beginning to be developed in spaceborne application areas. New functions will largely be of the data manipulation type usually in support of space experiments and military missions.

3. Programming Personnel

We project the same types of personnel, engineers, and data processors will be programming the space applications in the future. There will be a larger number of professional data processors in spaceborne work. Engineers -- or engineers turned programmers -- will continue to dominate the problem formulation stages of spaceborne computer program development. Professional programmers will perform the bulk of the support and operational computer programming. We also foresee in the time period we have identified, some on-board computer programming by astronauts, largely mathematical calculations.

4. Opportune Time

We feel this is a particularly opportune time to review and to recommend a programming language. The factors which make this an opportune time are:

- A third generation of general-purpose computers are now available.
- A second generation of spaceborne computers are being developed.
- An increased number of data processing functions are being scheduled for spaceborne computers.

MISSION DEVELOPMENT PROGRAMMING	SUPPORT PROGRAMMING	SPACE COMPUTER PROGRAMMING
TASKS		
Mission Profile Development Equation Formulation Scientific Simulations	Computer Simulator Vehicle Simulator Programmer Support Tool	Prelaunch Checking Keyboard, Display Instruments Navigation, Guidance and Control Data Transmission Surveillance, Reconnaissance Weapons Reliability and Failure Support
PERSONNEL		
Scientists, Engineers	Programmers	Programmers
COMPUTER		
Large, General-Purpose, Ground	Large, General-Purpose, Ground	Small, Special Purpose, Spaceborne

Figure 1. Programming Areas

First of all, there is a programming area which we will refer to as "mission development programming" in the mission planning stage of the project. The programming tasks in this stage of the project consist largely of the development of a mission profile, equation formulation, and scientific simulations. Personnel utilizating data processing for these tasks are usually mathematicians, engineers and scientists. The computer used is a ground-based, large, general-purpose machine.

Decision making consists of choosing alternate formulas and/or selecting of optimum values or actions. This type of programming is not a large portion of spaceborne data processing, but it is found in all three areas of spaceborne programming.

Data manipulation is used here to designate the type of information processing required for sorting, searching, merging, or selecting data. This type of programming is required in the operational program to some extent, these functions will probably increase. Data manipulation is utilized to a large extent in support programming.

Symbol manipulation is used here to mean the manipulation of hollerith or encoded data. This is primarily required in support programming.

Program control is largely in the operational program for control of hardware actions. Programming of this type is also utilized in support programming for control of programs and/or input/output.

D. LANGUAGE ATTRIBUTES REQUIRED FOR PROGRAMMING

The following discussion deals with the identification of the language elements required in the various programming areas of the spaceborne software effort. The lists presented are preliminary ones and will be modified as the on-going work indicates additions to, or subtractions from, the lists. The language elements identified will form the basis for the selection criteria for choosing a common spaceborne programming language. The lists will not be discussed in detail but in general terms.

The language elements for the required programming tasks have been organized in three general areas--mission development, spaceborne computer programming, and support programming. Mission development as indicated earlier, refers to the programming required to develop the equations which

2. Spaceborne Computer Programming

Figure 4 shows the language elements required for spaceborne computer programming. These include most of the requirements of mission development plus extra elements required for the more special requirements of the spaceborne computer programming. These include fixed-point arithmetic, the ability to manipulate bits and strings of bits, the ability to program in machine language, positive programmer control of memory allocation, and a strong capability to state optimizing information with respect to both space and time.

Floating-Point Arithmetic	Machine Language
Fixed-Point Arithmetic	Bit Manipulation
Full Relational Set	Flexible Manipulation
Vector or Array Capabilities	Flexible I/O
Logical And, Or, Not	Optimization
Function Subroutines	Better Data Storage Control
Procedure Subroutines	Segmentation
Built-In Subroutines	Overlay
Flexible Loop - Do, While, If	Multi-processing
	Timing

Figure 4. Language Elements Required - Spaceborne Computer Programming

3. Support Programming

Figure 5 lists the language elements required for support programming. As can be seen, these include all elements required for mission development and spaceborne computer programming plus the ability to manipulate characters and strings of characters, list processing facilities, and the capability of specifying recursive subroutines.

1. Problem or Procedure Oriented
2. Oriented to Engineering Calculations
3. Useful for Writing Compilers, Simulators, etc.
4. Well known and Implemented Widely
5. General Purpose Capabilities
6. No Dialects or Preprocessors

Figure 6. Criteria for Review of Existing Languages

The first indicates that assembly languages, oriented to a specific machine, will not be considered. Only higher-order problem or procedure-oriented languages will be included.

The next two criteria are self-explanatory. The fourth is intended to insure that a reasonably large pool of programmers exist who know the language and further, through implementation, its difficulities have come to light and, perhaps, have been corrected.

The fifth criterion is intended to eliminate languages which are intended for so special an application that they are difficult to use for general purpose programming.

A. CANDIDATE LANGUAGES

On the basis of these criteria, an initial set of seven languages has been chosen for consideration and comparison. The list is not necessarily final; it may be expanded as additional information is obtained. The list is shown in Figure 7.

The latest available description of PL/1, "PL/1 Language Specifications," IBM Systems Reference Library, Form C28, 6571-3, July 1966.

Not all of the languages on the list satisfy all of the criteria; however, if a language is deficient in one area, it should excel in others, and this is true of all of the languages on the list. The basis for the evaluation of these languages are the language specifications in the references indicated.

B. LANGUAGE REQUIREMENTS AND CAPABILITIES

We have identified seven candidate programming languages as meeting our criteria for programming languages. We have reviewed these languages and attempted to compare their capabilities against our requirements. In referring to Figure 8, the candidate languages are listed across the top of the page. Language capabilities required for our application area are listed along the left-hand side in order of importance. The most important attribute appearing first. The capabilities are graded alphabetically from "A" to "F." "A" being the most superior grade and indicating the language contains substantially all of the capabilities we desire.

The ability of a language to express iterative operations to state "why," "do," and "if," looping capabilities. PL/1 appears to have all that is required. I might state that the other languages lack some dynamic looping control capability.

For code optimization, PL/1 is superior, however, none of the lanuages, including PL/1, do a very good job in this area. We will place special emphasis on this in our spaceborne language.

For decision making, sufficient logical and relational operators are available in all of the languages. We would hope to be able to provide considerable more flexibility of input/output language control in this area for the spaceborne programming language.

Multi-task and interrupt capabilities are relatively new features and as one might suspect, PL/1 has some innovations to facilitate usage of these features.

The readability of a language is largely dependent upon what kind of characters are available to that language and the allowable length of any labels or literal descriptions. PL/1, COBOL, and JOVIAL are all fairly good.

A cursory analysis of the technical considerations would lead one to the conclusion that PL/1 followed by JOVIAL are the superior technical languages. One other observation needs to be made regarding PL/1. PL/1 as specified in the SRL document has not been implemented as yet. There may be attractive features in the language which might be very expensive to implement and thus are not cost effective. Further, there are a number of PL/1 language capabilities which are <u>not</u> needed for an optimum Space Programming Language.

FORTRAN, primarily lacks data structuring capability, fixed point arithmetic capabilities, and some capability to manipulate symbolic data. JOVIAL lacks capabilities to handle some data structures. It also lacks language features to facilitate code optimization, input/output is weak, formatting and multi-processing capabilities don't exist. PL/1 lacks machine language, some VECTOR and matrix arithmetic capabilities, control of data packing, etc. None of the languages have some of the capabilities we would like to emphasize, such as code optimization and function oriented arithmetic and algebraic capabilities.

This is as far as we have gone in our present study. We have resolved the review of existing languages to two basic conclusions: 1)that none of the existing languages will fulfill our requirements satisfactorily, 2) that the FORTRAN, JOVIAL and PL/1 languages appear to be the best candidates for a base language for the Space Programming Language.

V. OUR PRESENT CONCEPTION OF A SPACEBORNE PROGRAMMING LANGUAGE

Implementation of the Space Programming Language should produce the following kinds of capabilities.

For problem formulation, we will have a subset of the SPL language which is easily learned and which can be compiled in a very short amount of time and has high utility for mathematical calculations. This language should be implemented on the general-purpose computer. Further, this subset is one which could also be used for on-board programming. This subset should also be highly useable in an interactive or time-sharing mode.

For support programming, we should have the full language including all capabilities of the problem formulation subset and the operational space programming subset. The support programming language should have capabilities

SUMMARY

Is a common POL for spaceborne programming worth implementing? We think we can give an unqualified "yes" to this question. A common POL should reduce lead time through better communication and program production techniques. It should reduce costs through the transferability of programs, reuse of programs, and communication of programming information. It should provide manpower flexibility through reduced amounts of training and in reduced lead time in the learning of a programming language.

How much do spaceborne programming requirements resemble general-purpose POL requirements? The difference is largely one of emphasis. There is greater emphasis made upon program quality and computer storage for spaceborne software Further, the development of the operational program in a simulated environme occurs considerably more often.

Is an application specific language like SPL(Space Programming Language) worth implementing? We certainly feel that it would be. We feel that a language largely based upon an existing, widely-implemented language, and having capabilities which are tailored to the problem area, would certainly be appropriate. We believe it would be useful in a large majority of the applications in the spaceborne areas. We also believe that it should evolve and continually be improved so that it remains a highly effective language.

Thank you.

SESSION 5

Spaceborne Software System Management

Chairman: Ron D. Knight
System Development
Corporation

(page 276 blank)

SUMMARY OF SESSION 5

Ron D. Knight
System Development Corporation

The papers presented in this session covered the range of management techniques applied to software development, with emphasis on the process of specifying software, controlling necessary changes, and documenting products. The topic of estimating software costs was given attention in the paper by Mr. LaBolle. A thorough description of the management of Project Gemini software provided an indication of the strict controls necessary to adhere to a rigid schedule and thereby help to assure a successful program.

The SDC study showed that the problems of spaceborne software development are different in degree, in some instances, but not in kind from those found in other application areas of the industry. This view was shared by most of the speakers. There was not, however, such unanimity with respect to the application of, for example, AFSCM 375 series across the board for spaceborne software. Objections included anticipated excessive costs, loss of management prerogatives, and possible jeopardy of schedules. There were many proponents of better management processes, but no general agreement on a particular system or technique was reached.

To summarize, the consensus is that good management practices are an obvious requirement to ensure timely software as well as hardware development. The extent to which the Air Force should impose a uniform system of management tools for software system definition, development, and acquisition was a subject of wide disagreement.

AN OVERVIEW OF CONFIGURATION MANAGEMENT IN THE AIR FORCE SATELLITE CONTROL FACILITY

by

J. B. Munson
System Development Corporation

When I was asked to speak on the SCF's method of configuration management I was not sure how to approach the topic....

The term configuration management seems to be very ambiguous in our culture despite many efforts to give it a very precise definition.

When I use the term I mean it to encompass the management procedures and the process involved in the development of any complex system and the maintenance of that system's integrity during evolutionary modification. That is how I define the term, if you ask me what the term connotes, I know most people immediately call to mind visions of paperwork, red tape, proper channels and lengthy delays. --This is unfortunate because procedures are supposed to serve a beneficial and enabling function--not interfere with the job. We feel we have achieved this enabling function in the SCF. We have been developing and tailoring our procedures constantly during the last six years. And, in fact, are currently undergoing a transition from our system to the AFSCM 375 concept. This will not be as big a job as it might otherwise have been since, as you will see, there are many similarities between our system and 375. We also share another feature with 375 - the concept that you use only those portions of the procedure needed to ensure adequate management control, or more simply put: the more complex the function being developed, the more extensive the control procedures.

For the purpose of this presentation, I will restrict my discussion to the means by which the Satellite Control Facility manages the process of computer program development.

For those of you who may be unfamiliar with the Satellite Control Facility (Figure 1) let me describe it in very broad terms as a general purpose, computer based, command control system. The nerve center of the facility is a central computer complex, and the associated command functions, which is located at the Satellite Test Center in Sunnyvale, California. This center is connected by an elaborate communications network to a number of tracking stations located throughout the world. The general purpose portions of the system provide acquisition data to the various stations, acquire the satellite as it passes over the station, collect telemetry data from the satellite (which is passed back and displayed in Sunnyvale in real time), transmits commands to the satellite and provides tracking data used to update the ephemeris for future acquisitions.

The computer complex which provides this capability is depicted in Figure 2.

Each tracking station has two Control Data 160A computers, one for handling the tracking and commanding data and the other for processing the raw telemetry data. These, in turn, are connected to a CDC 160A computer in Sunnyvale called a bird (or satellite) buffer, which can be automatically switched from station to station, enabling it to follow its "bird" as it orbits the earth. The ephemeris computation, acquisition prediction, command generation and other associated functions are performed on CDC 3600 computers in an off-line fashion. They use data fed them by the bird buffers and transmit acquisition and command data through the buffers to the stations.

I realize that this is a very cursory examination of our system; however, if I've gotten the point across that it is a very large, highly complex, interactive system I've succeeded in my purpose of setting the stage for the description of our software development activity.

One other point, though, before I begin that discussion. The SCF, due to its size and complexity, employs a number of separate software firms, approximately ten, in the development of the computer programs which make up the system. This gives rise to a fairly unique concept (for software) used by the SCF to aid in the development process. This is the existence of the role of computer program Integration Contractor, a responsibility of the System Development Corporation.

Figure 3 details the responsibility of the Integrating Contractor. In essence this chart says that he is responsible for quality control - he must see that the pieces being developed will work together; he puts them together and he validates that the software components work as a system.

This takes us, finally, to the process involved in the SCF computer program development cycle. As an aid in developing this discussion I've chosen to jump into the center of the cycle and start with the smallest discrete unit, the individual computer program (Figure 4).

The deliverable product consists of the discrete computer program and associated materials.

Each program is reviewed for the Air Force by the Integration Contractor to see that it conforms to its specification.

The design specification had been previously prepared by the Programming Contractor and reviewed for the Air Force by the general system engineer, the Aerospace Corporation, and the Integration Contractor. It contained both the detailed coding specifications and an acceptance test plan which had to be approved prior to the commencement of the coding itself.

COMPUTER PROGRAM INTEGRATION CONTRACTOR ROLE

- CONCEPTUALIZATION AND PLANNING
- INTERFACE ASSURANCE
- ASSOCIATE CONTRACTOR SUPPORT
- GENERAL PURPOSE PROGRAM DEVELOPMENT
- COMPONENT VERIFICATION
- SYSTEM VALIDATION
- CHANGE CONTROL

Obviously, in a system as large as ours many such programs are being developed at any given time and the dynamic and expanding nature of our environment has required the consolidation of changes into packages or models. Each model contains a well structured set of modifications to provide a step function increase in system capabilities. To give you an idea of the pace of our system -

Model 7 is operational
Model 8 is in installation
Model 9 is in system test
Model 10 is in development
Model 11 is in design
And the collection of requirements for Model 12 is in progress.

This may not seem quite as exciting as it is unless I further explain that the total system consists of approximately 1.5 million instructions for the various computers and from 10% to 50% of them may be changed in any given model.

How, then, is this process controlled?

First, requirements for changes to the system (Figure 5) are constantly being evaluated by the engineering side of the SCF These requirements come from many sources - to name a few:

A. Satellite Program offices who have flight requirements.
B. The Satellite Control Facility operations people who have been using the system and require changes.
C. And the engineering area itself which wants to anticipate general purpose needs of their expanding support mission.

Next, these requirements are sorted and scheduled by Aerospace and the Air Force - with the help of the Integrating Contractor - in meetings of a configuration control board.

For complex or large changes and new procurements, Aerospace will prepare design criteria for transmission of the statement of the requirement to the programming contractor. For most modifications to the system, though, this step is by-passed and requirements or changes are relayed to the appropriate contractor through contractual change channels. In any event, the Programming Contractor responds to the requirement with a document - referred to as the implementation concept - which provides his formulation of the design required for the functional change. This document may imply changes in many discrete computer programs; however, it is written at the system level and is used to evaluate the contractor's plan of attack. This document is reviewed by Aerospace and the Air Force and must be approved prior to any further effort.

Following review and approval of the implementation plan - and there may be many of these, from many contractors, for a given model - the Integration Contractor must integrate these plans into the current system configuration and provide for the programming contractors an interface specification which supplies him with the details of how his area must be organized to fit into the system. This includes such items as central data definitions, allocations of core storage, standards and conventions for interfacing with the executive system, communication conventions for interfacing with other functions, computers or special purpose digital equipment and utilization of system constants or tables. Upon receipt of the approved interface specification the contractor can finish the detailed program design and proceed with program production.

To illustrate the magnitude of this delivery and review process SDC received, for Model 8.0, in excess of 60 new or modified computer programs -- and this was for the 3600 general purpose system only, the 160A portions had been virtually re-written.

To show how we accomplish this mammoth integration and validation job I have to add a couple of boxes to the flow chart (Figure 6). These two products are prepared by the Integrating Contractor during the development process. The first - the validation plan - provides in precise detail the Integrating Contractor's plan for testing the software system to validate and demonstrate that it meets the system requirements. This document is closely reviewed by the requirements and engineering personnel to see that in fact it validates that the system provides the intended capabilities and by the operations people to see that it demonstrates the system's operability to meet their satellite support mission.

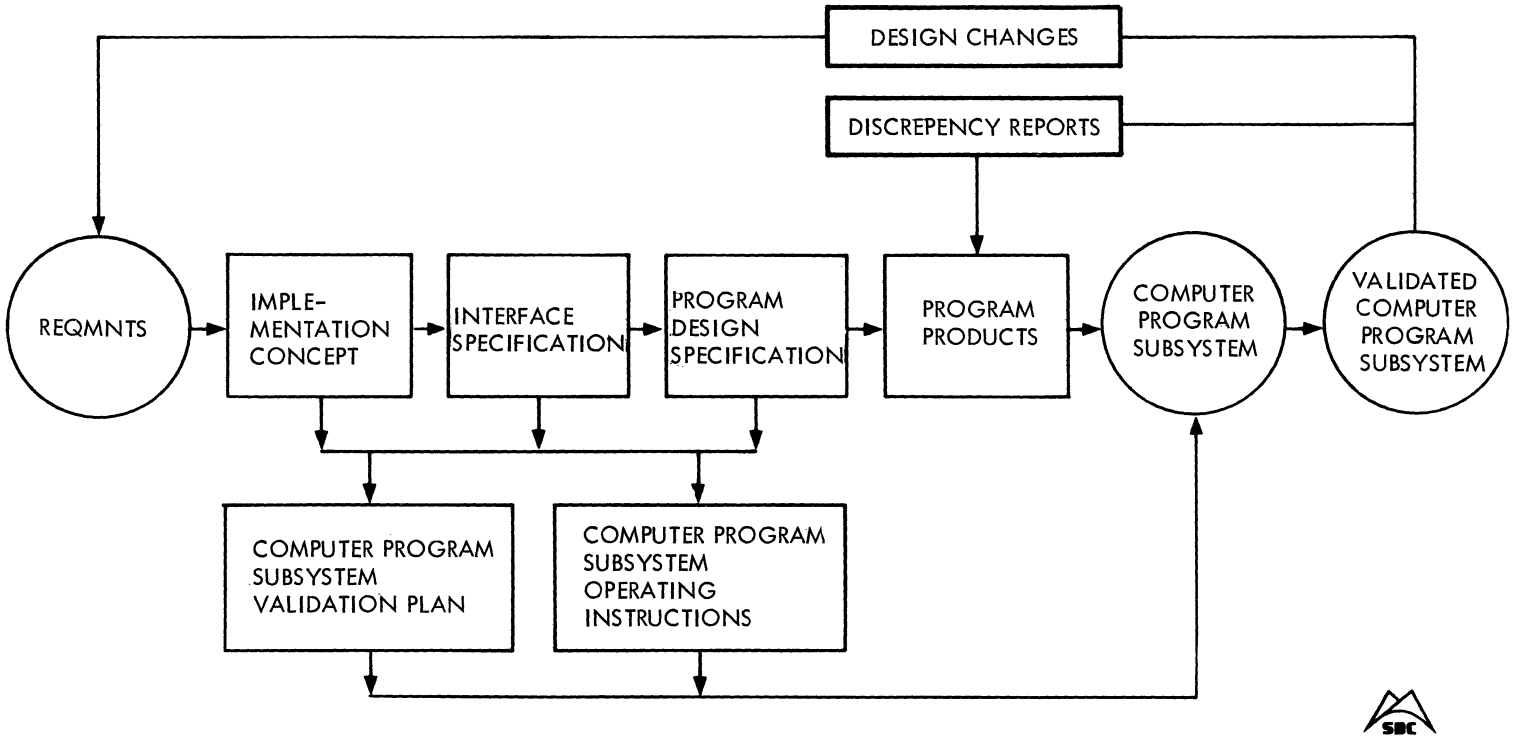
The system is then exercised by the Integrating Contractor, in accordance with this plan, until all agencies are convinced that their requirements have been met and demonstrated.

Additionally, the Integrating Contractor prepares system operating procedures which tell the operations people how to use the system to accomplish their mission and conducts training sessions for each new model.

Figure 7 shows how the development procedures provide for a feedback loop and provide for change control during the development cycle.

As indicated, two types of change are recognized. These are, basically, the correction of errors, which is when the system does not perform as specified, and the change of the specifications and associated programs to accommodate new or modified requirements. Although the results of either type of change are similar in the way they are received into the system - octals or new program mods - the control process is entirely different.

SCF COMPUTER PROGRAM DEVELOPMENT



THE ADAPTATION OF SYSTEMS MANAGEMENT CONCEPTS TO COMPUTER PROGRAM ACQUISITION

Milton V. Ratynski
Electronic Systems Division, AFSC

Lloyd V. Searle
System Development Corporation

The work we are reporting here results from an on-going project at the Electronic Systems Division to develop technical standards pertaining to the acquisition of computer programs. Because of ESD's extensive interest in the L-Systems, and because computer programs are becoming increasingly important to Air Force systems in general, the project is emphasizing requirements which exist within the context of a system program. Thus, the work is closely concerned with the precepts of "Systems Management" as set forth in the 375-series of Air Force regulations and Systems Command manuals.

Our presentation is being divided into two parts. The second part, to be presented next by Mr. Neil and Mr. Piligian, will emphasize the special topic of Configuration Management. In this first part, our purpose is to review certain general ways in which computer programming relates to the systems management framework, to provide perspective for the subsequent discussion.

***

The concept of "Systems Management", as a structured way of acquiring new military capabilities, is by no means a new, or very recent, invention. Although the present series of AFSC manuals is comparatively recent--and steadily changing in particulars--the process now has a substantial history, in the Air Force, of about 15 years.

In some part, this may have been a function of its relative independence from the engineering disciplines. Traditionally, military systems are built around a basic core of operational hardware; and the concepts of systems management reflect this fact. The "pacing items", to which the other associated and supporting items must be related, have typically been the major items of operational equipment. Hence, many of the terms, concepts, and procedures have been influenced by their established connotations in the world of hardware engineering.

Unfortunately, our "common English" language has already supplied too many terms which have acquired special--and different--meanings in the hardware and software fields. One purpose in this session is to discuss a few of these terms as they are now being used in the language of systems. Where many people from different organizations and varied technical specialties are involved in a joint enterprise--as they are, in building a large system--semantics can become a critical matter.

***

As a starting point, we might expand briefly on the meaning of "Systems Management". This refers to the general process of planning, organizing, and directing the multitude of activities which are required to bring a new system into being and put it into operation. In the Air Force, it is accomplished by a centralized SPO (System Program Office), located in one of the four Systems Divisions of the Systems Command. Typically, it encompasses the five sub-areas of management which are illustrated in Figure 1.

consideration and responsibility. They pertain, most directly, to the major system elements of equipment, facilities, and computer programs.

***

Since we are speaking of terms, it must be observed that "Software" is one which has proved to be increasingly difficult to use in this context. With so many different, but firmly established, definitions, its principal virtue with a mixed audience seems to be in keeping people from being quite sure what we are really talking about! While the troubles may not be so apparent when one really means to refer just to computer programs, or to a limited class of computer programs, they become more evident when it is intended to refer to the activities of designing and developing, or to computer programs plus something--e.g., plus the specifications, user documentation, testing, system analysis, or even associated human factors considerations. It is true that some such variety of things is (or should be) normally associated with computer programming, particularly in the context of a military or space system program. --But it is just as true that a similar variety of things is also associated with the engineering development of equipment.

In system programs, it has been found necessary to "break out" different elements into a variety of classifications and groupings, because they have different implications for both technical work and managing contracts. The rules which apply to a piece of equipment, for example, are not the same as those which apply to handbooks and manuals, or reports. And sometimes, certain elements cannot be adequately specified at all in terms of identified and deliverable products.

We find it useful to note that essentially all of the elements--or "things"--which can be bought via contract can be grouped into three very broad categories, namely; Manufactured Products, Data, and Services. For the purpose of defining the role of "Software" in systems,

research, design, development, testing, operation, or maintenance.

Now, if we confine our attention for the moment to a Computer program -- and further, disregarding development and documentation, to the resulting item itself in the form of a magnetic tape or card deck--, the question arises: "Should we treat it as a Data Item, or as a Manufactured Product?" The question is by no means academic. In the case of any given computer program which is designated as deliverable under contract, a forced choice must be made among three possible alternatives: It is either (1) a Data Item, (2) a part of another item which is classified as a Manufactured Product--namely, a computer, or (3) a Manufactured Product in its own right.

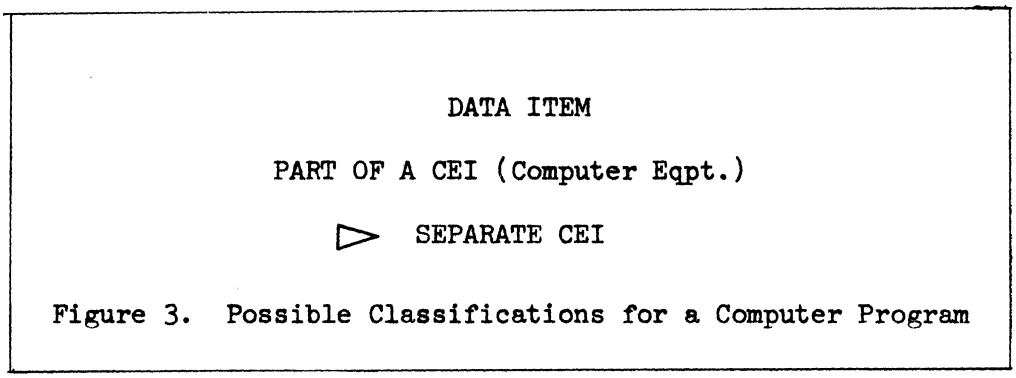


Figure 3. Possible Classifications for a Computer Program

Considering its intrinsic properties, one might easily be led to state categorically that it is an item of Data. And that position is highly defensible.

However, both the Air Force and NASA are now insisting that most computer programs for military and space systems be treated as Manufactured Products (and classified as Contract End Items) for the reason that certain requirements have much more in common with equipment than with Data.

Thus, certain management techniques which are suitable to equipment development are also indicated for computer programs. These include:

> Contractor-developed specifications, at both the performance and design levels;
>
> Design reviews & inspections, to monitor design, coding, and documentation;
>
> A formal test program, to verify compliance with approved performance requirements, including specified functional interfaces.

Now--this may sound like we are saying that: Software is Hardware!

However, although many of the same management principles might apply, in the broad sense, computer programs are definitely not like equipment in many significant ways. For example:

Unlike hardware, computer program instructions do not "wear out". And, we don't have to build and maintain an elaborate special production facility to produce each computer program in quantity--since, regardless of its configuration (if we happen to want copies), any number can be duplicated on a standard machine, at small cost.

Thus, we essentially eliminate a whole host of such concepts as:

> Reliability, Maintenance & Repair Cycles; Useful life; Provisioning, Interchangeability, & Substitution of Spare Parts; Production; Quality Control; Acceptance Testing; Logistics...

--that is, in the sense of the established connotations of these concepts for hardware.

By the same token, the management procedures which have been established for equipment have to be revised extensively, and carefully tailored for realistic application to computer programs.

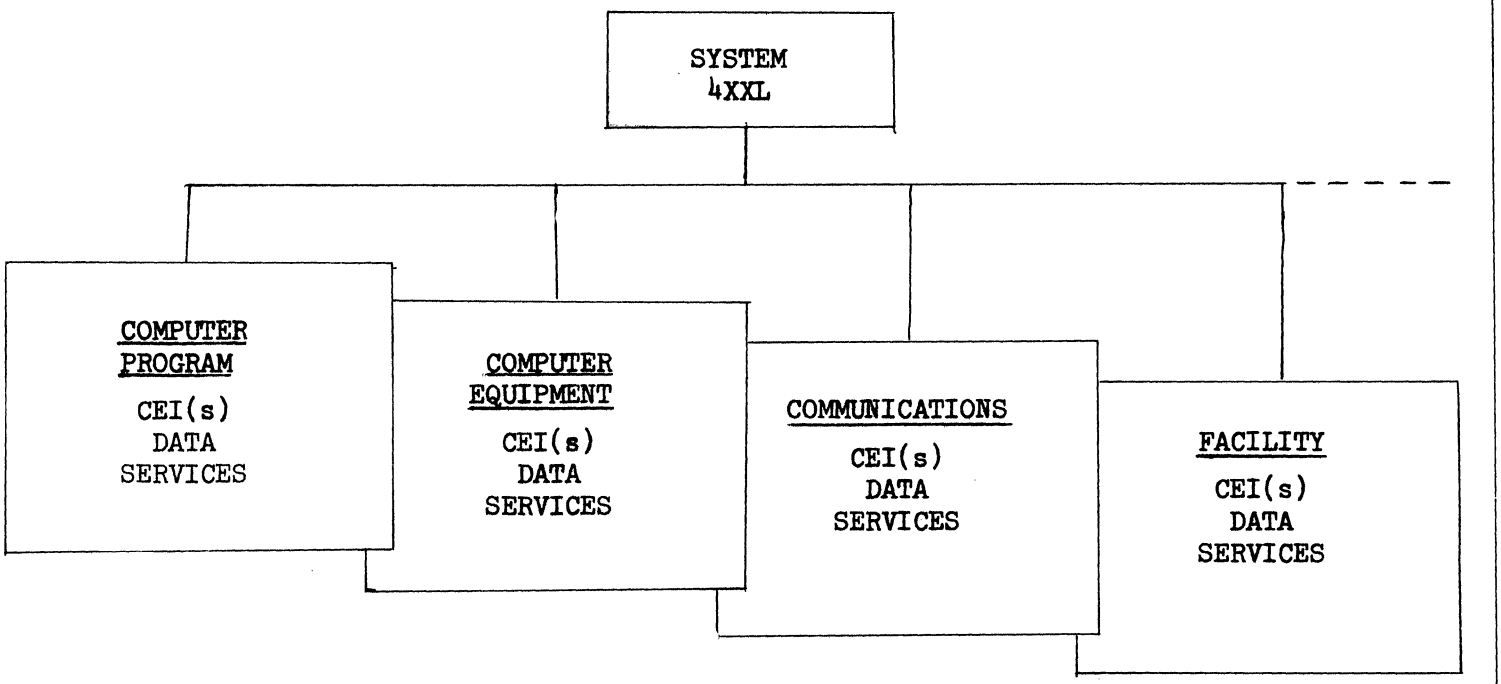


Figure 5. Major System Segments (L-Systems)

That whole package, incidentally, is also often known to carry the label of "Software". And, it is the total package with which our project is concerned:

Figure 6 is a very gross-level illustration of the major activities and events (mainly for the computer programming System Segment) which could be expected to occur during the Conceptual, Definition, Acquisition, and Operational phases of a typical system life-cycle.

Contractor activities associated with this and other System Segments occur mostly during the Definition and Acquisition phases--following certain system-level events which provide necessary starting points:

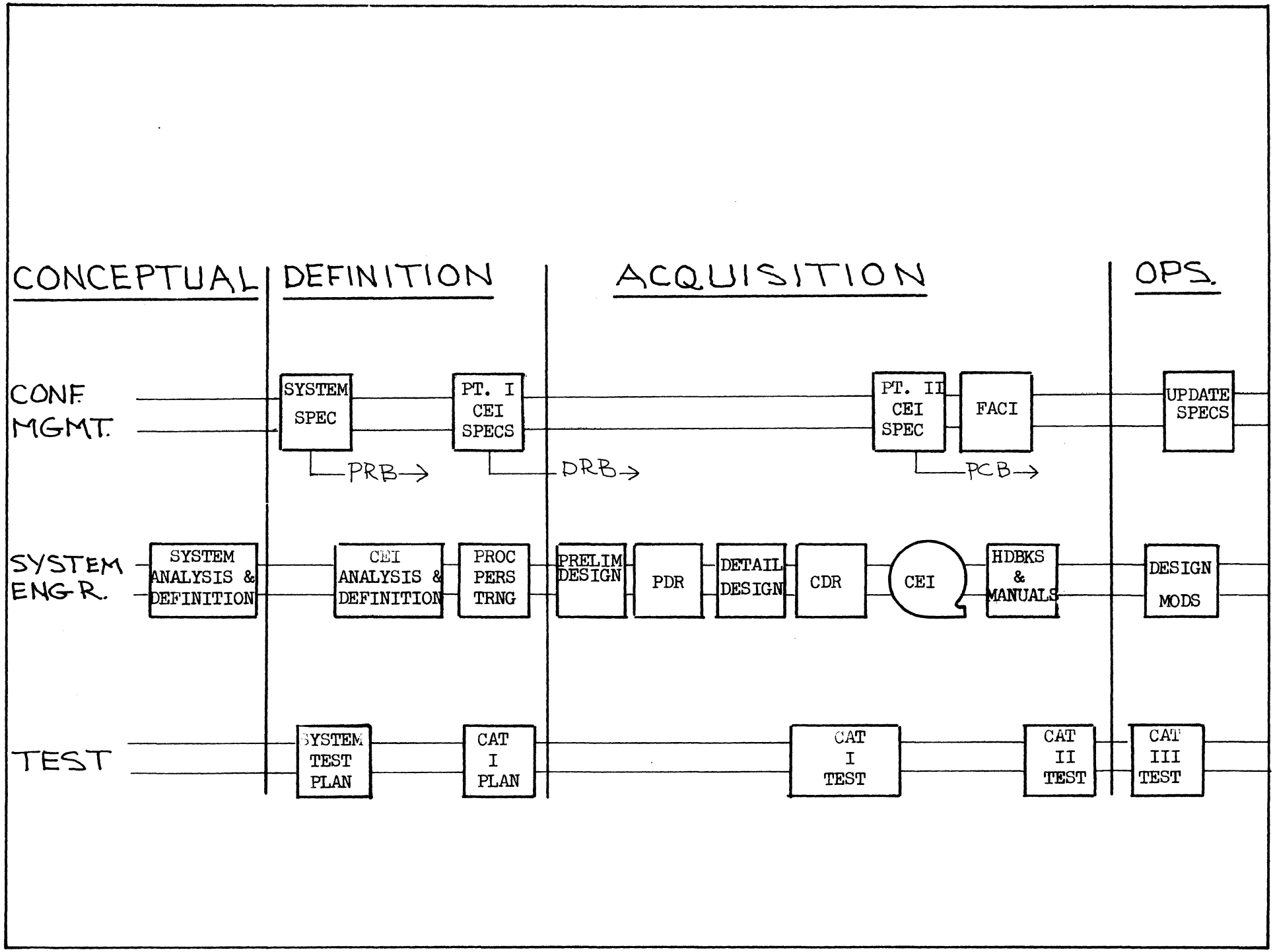


Figure 6. Information Processing Events During a System Life-Cycle

which have been completed and issued are the following:

(1) A set of procedures and requirements for computer program configuration management, which is presently being incorporated into the forthcoming revision of AFSCM 375-1, and has been issued separately for interim use as ESD Exhibit EST-1.

(2) A supplement for electronic systems to AFSCM 375-4, which has been issued as ESD Exhibit EST-2.

(3) A number of AFLC/AFSC Forms 9 for items of deliverable data associated with the computer program process, prepared for incorporation into Vol. II of the joint AFSC/AFLC Manual 310-1.

CONFIGURATION MANAGEMENT OF COMPUTER PROGRAM CONTRACT END ITEMS

by

M.S. Philigian, Air Force Electronic Systems Division
and
G. Neil, System Development Corporation

The procedures being described here have been published in Electronic Systems Division (ESD) Exhibit EST-1. The purpose of this exhibit is to augment AFSCM 375-1, "Configuration Management During Definition and Acquisition Phases," dated 1 June 1964. The exhibit is intended for use in conjunction with the parent manual prior to its incorporation in the revised version of AFSCM 375-1, which is expected to be available early in 1967.

In developing these procedures we found it necessary to examine all facets of systems management to determine the relationship of the computer program to a total system. The procedures, as documented in the exhibit, have been coordinated extensively with Systems Command and industry and to a limited extent with NASA. The procedures are currently being used on several system programs at ESD and have been found to operate quite successfully.

The computer program configuration management procedures relate to:

(a) Configuration Identification

(b) Configuration Control

(c) Configuration Accounting

CONFIGURATION IDENTIFICATION

EST-1 contains a complete and separate exhibit to provide contractors with instructions for the preparation of the detailed specification for computer program contract end items. This exhibit is equivalent to Exhibit II for prime equipment CEIs in the present manual. The computer program specification is in line with the Uniform Specification Program, as introduced in AFSCM 375-1, where we have a format which is established at the system specification level and is followed through to the Part II specifications for all CEIs within a system.

Part I - Contains the Performance and Design Requirements.
This part of the specification is needed to specify requirements peculiar to the design, development, test, and qualification of the CEI.

Part II - Contains a Detailed Technical Description of the CEI.
This part of the specification is used to describe, in detail, the exact configuration of the computer program CEI.

Now having computer program CEI specifications in line with the uniform specification program, the concept of baseline management can be applied in the same manner as for the other CEIs. The Part I of the specification technically defines the Design Requirements Baseline and the Part II of the specification technically defines the Product Configuration Baseline.

The contents of the Part I specification are as follows:

Performance Requirements

This section defines the performance requirements for each function within the CEI. It is written in mathematical, logical, and operational terms.

Interface Requirements

This section specifies the requirements imposed on the design of the computer program in order to satisfy the requirement to interface with the other elements of the system, e.g., message formats, card formats, display formats, etc.

Design Requirements

This section specifies any design requirements for the computer program. These may include specific language to be used, requirements for expansion or design modifications, programming standards, etc.

Test Requirements

This section will specify the requirements for formal verification of the performance of the CEI in accordance with the performance requirements.

PART II CEI SPECIFICATION (COMPUTER PROGRAM)

The Part II specification for computer program CEIs contains a technical description of the computer programs. Unlike the prime

duplicate those contained in Exhibit IX, this addendum is intended to be complete and self-sufficient in its coverage of procedures pertaining to changes to computer programs. While the procedures conform with the format and intent of ANA Bulletin No. 445, they are tailored to reflect the absence of many special requirements to equipment production, retrofit, and supply and they provide additional information for processing and evaluating changes to computer program CEI.

At the outset of the Acquisition Phase the contractor-prepared Part I CEI specification is approved by the procuring agency. This approval establishes the Design Requirements Baseline as a defined point of departure for configuration control. Once the Part I CEI specifications have been baselined, any changes to the Part I will be submitted, on an ECP form, as a design requirements change. The ECP will be formally approved by the Configuration Control Board (CCB) prior to the implementation of the change.

During the Acquisition Phase as the CEI is being developed, the Part II CEI specification is being prepared to describe the exact configuration of the CEI. Immediately prior to Category II testing a First Article Configuration Inspection (FACI) is conducted on the computer program CEI. At FACI the Part II CEI specification is accepted as an audited and approved document. At the successful completion of FACI the second computer program baseline may be established, i.e., Product Configuration

End Item Configuration Chart

Spec. Change Notice (SCN)

Spec. Change Log

Configuration Index

Change Status Report

Version Description Document

FIGURE 3
SPEC. MAINTENANCE & ACCOUNTING DOCS.

The End Item Configuration Chart is a summary record which identifies approved changes (ECPs) to the end item specification.

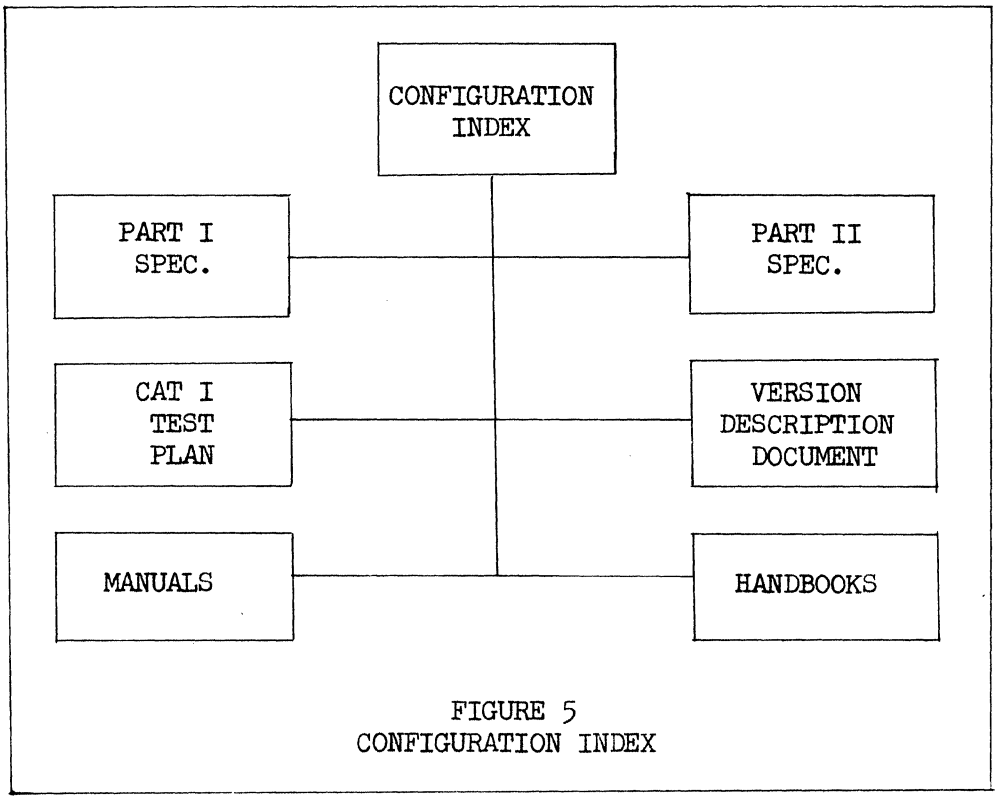


FIGURE 5
CONFIGURATION INDEX

The Configuration Index provides an official listing of the specifications, and significant support documents. It also reflects all approved changes to these documents.

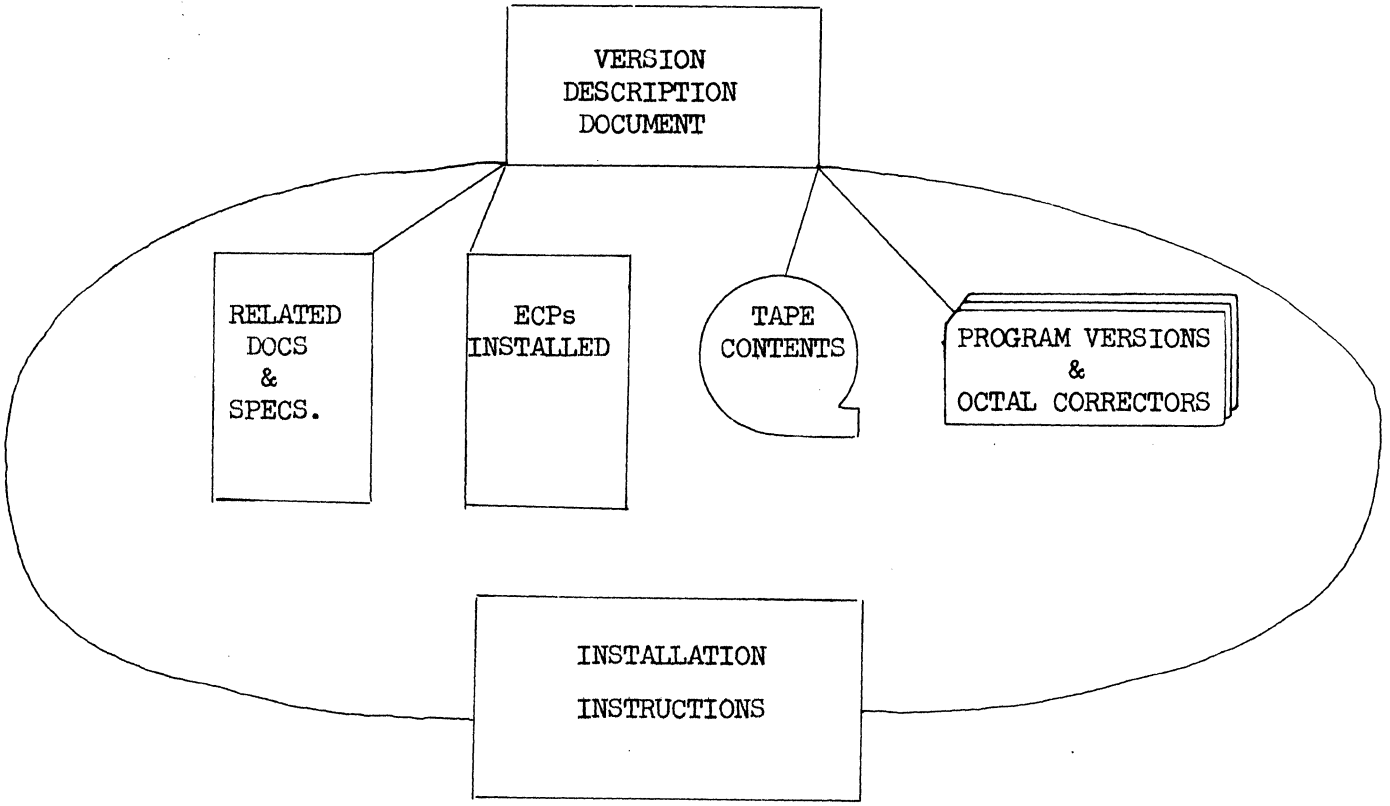


FIGURE 7
VERSION DESCRIPTION DOCUMENT CONTENTS

The Version Description Document shall be used to accompany the release of a computer program CEI, either as a whole or in part. Its purpose is to identify the elements of the computer programs delivered and to record pertinent additional data relating to status and usage.

the integrity of the computer program design prior to coding and testing. While the exhibit defines the CDR for a computer program as basically a flow chart-level review, it also provides for flexible application in the case of a complex computer program CEI which is scheduled to reach any given stage of the design in increments of individual computer programs, or blocks of programs. In these cases the CDR may also be scheduled in increments.

The First Article Configuration Inspection (FACI) is a formal technical review which establishes the adequacy of the Part II specification as an accurate and complete description of the computer program CEI. The primary product of the FACI is formal acceptance, by the procuring agency, of (1) Part II of the end item specification as an audited and approved document and (2) the first unit of the computer program CEI.

DEVELOPMENT OF AIDS FOR THE MANAGEMENT OF COMPUTER PROGRAMMING*

by

V. LaBolle

System Development Corporation

*Reprinted from the November 1966 issue of the Journal of Industrial Engineering, official publication of the American Institute of Industrial Engineers, Inc., 345 E. 47th St., New York, N. Y. 10017.

intangible nature of the operational product slows the development of such standards. A complete computer programming job may yield a variety of tangible products--documents for users, operators, and maintenance staff, as well as tapes, listings and card decks--but these merely represent the actual computer program that directs data processing operations within the computer. Another obstacle in establishing standards is the rapid technological change that is coupled to the growth in computer application and techniques for using them. The technology won't sit still long enough to allow us to appraise it.

In planning computer programming efforts, the tendency has been to underestimate both cost and development time. Some experts who help make estimates forget to discount the fact that they, the experts, won't be on the job, but rather, people with less experience and proficiency will be involved. Also, in planning there is a tendency to neglect some of the many tasks that are needed to complete a computer programming job. In the absence of quantitative guidelines based upon experience, such oversights lead to underestimates

Recently, the decision-making problems that face managers in computer programming and the buyers of the resulting products have been identified more clearly (9). A landmark in this area, Brandon's Management Standards for Data Processing, describes techniques for establishing standards for methods, and subsequently, performance standards for the men and machines used in computer program development (2). Such standards are aimed at improved mangement control, cost estimation, and cost control, particularly in the field of business data processing. Also, the Federal Government has been addressing questions on how to plan, control, and evaluate computer programming efforts. For example:

- The Bureau of the Budget, the General Services Administration, and the National Bureau of Standards, starting with the formulation of policies on computer acquisition and use, have now become interested in standards for computer programming (14).

planning by first-level supervisors, the guide is designed to stimulate more complete consideration of the entire computer programming process. Within the guide a set of prescribed planning and management tasks are applied to the computer program development process. The process is described in terms of eight phases: (1) (Information) Systems Analysis, (2) (Information) System Design, (3) (Computer) Program Development, (4) (Computer) Program Coding, (5) (Computer) Program Development, (4) (Computer) Program Coding, (5) (Computer) Program Checkout, (6) User Documentation, (7) User Training and Assistance, and (8) Turnover. Each of these phases is further divided into tasks-- a total of 36 for all eight phases. Each task is then described on a two-page format that includes inputs, outputs, subtasks, cost factors, and the characteristic task environment. Comprising more than half the 170-page document, these detailed task descriptions provide checklists for more accurate and complete planning. The sequence of planning steps together with some guidelines for estimation, help the manager plan, schedule, and cost these tasks in the development process. Various forms (e.g., see Figure 1) are supplied to record the planning results and to serve as abbreviated checklists for the required work. The forms and procedures also provide a basis for control of computer programming projects and for collection of experience data that may be used to improve future estimates.

The guide has been in use for more than a year as a reference for planning computer programming projects at NAVCOSSACT. In addition, the Air Force has requested more than 100 copies to distribute to organizations responsible for computer programming. At least one programming staff in a large corporation has planned to adopt a modified form of the guide as a standard for planning and control. In other words, the feedback to date indicates that the planning guide has approached the goal of supplying a useful aid for managers of computer programming.

ANALYSIS OF COST DATA

To help managers make better estimates of costs for computer programming, Project members began exploratory work in 1964 to derive estimating equations using actual experience data as inputs to the analysis (4). This work is being done under contract with the Air Force Electronic Systems Division, Deputy

for Engineering and Technology, Directorate of Computers. It is a pioneer task since efforts to gather and analyze numerical data on computer programming costs are still unique. The equations being derived are rules for using numerical values for cost factors that characterize the requirements, resources, and environment for a computer programming effort to calculate estimates for cost such as manpower, measured in man months, and computer time, measured in hours. These estimating relationships are intended to help the manager plan a program production effort in the early stages of computer programming, e.g., before program design begins. However, the results may also be used to evaluate completed efforts by comparing actual costs with estimates in a framework provided by the derived equations.

The analysis work has been conducted in cycles, each marked by collection and analysis of new data to improve upon earlier results. A cycle of analysis consists of the following:

- Design (or redesign) of the questionnaire used to collect the data.
- Collection of data that characterize completed programming efforts.
- Validation of these data by identifying anomalies and gaps and then coordinating with the original respondents to clarify and complete the questionnaire.
- Repeated application of statistical techniques, e.g., multivariate regression, coupled with intuition and experience. These techniques are used first to reduce the total number of cost factors to be considered as independent variables, and then to derive the equations that relate the remaining cost factors (independent variables) to the cost measures (dependent variables).

A first cycle, which used data on 27 programming efforts completed at SDC, was conducted in 1964 (5). For the second cycle more data were collected at SDC, increasing the sample to 74, and analyzed in 1965 (11, 8). A third cycle, using additional data on 104 programs completed by computer programming organizations in the Air Force and in industry, is now under way.

- Programming Personnel. Characteristics of the personnel needed to develop the computer program, e.g., number of programmers classified as coder, programmer, senior programmer, system programmer; years of experience for each category of programmer with language used, computer used, and specific application.

- Utility Computer Programs. Characteristics of the computer programs used as tools to produce the subject computer program, e.g., programming language used in coding, number of free support programs available.

ENVIRONMENT

- Management Procedures. Factors associated with the plans, policies, practices, and review techniques used in the administration of all phases of program development, e.g., existence of a documented management plan for processing of program design changes and standards for coding and flow charting.

- Development Environment. Factors describing relationships with external organizations, including customers and other contractors, e.g., number of agencies concurring on design specifications and computer facility operated on the basis of open shop, closed shop, time-sharing.

c. The analyses to derive estimating equations are restricted to costs of computer program production, i.e., the computer program design, code, and test activities including associated documentation and design and development work on the data base. Chosen because these activities are common to almost all computer programming work, this set does not include work that may be in a more general model of computer programming for large information processing systems with men, machines, and computer programs as components, such as information processing system design and analysis.

The Second Cycle--An Analysis of 74 Data Points

In a second cycle, more data were gathered to conduct similar analyses aimed at obtaining estimating equations with increased precision, and with cost factors used as predictors that are relatively easy to estimate before a programming job begins. Differences in the second cycle are discussed below:

- The Data Collection Questionnaire. The initial questionnaire was revised as a result of feedback and the experience gained in the first cycle. For example, questions were amplified to gather more detailed information and to remove ambiguous terms (e.g., the five levels of system complexity were briefly described).

- The Sample. In the second, as in the first cycle, no deliberate sample design was used; more managers throughout SDC were asked to complete questionnaires for representative programming activities. After a check of the collected questionnaires for accuracy, a total of 74 data points, including 24 points from the first cycle, remained, representing a variety of programming applications--command and control, compilers, information retrieval, management information, and utility programs.

  This larger sample included more small jobs, resulting in a larger range for the cost measures (number of man months, computer hours, new machine language instructions, and months elapsed) as well as many of the cost factors. (For example, 36 data points with less than 20 man months of effort were added in the second cycle.) So their frequency distributions showed clusters of data at the low and medium values and few values at the high end.

  The very high values from such exponential distributions dominate the equations derived by multivariate regression techniques, and estimates for low values have poor precision. In analyzing the data in the entire sample, the logarithmic transformation was applied to all the cost measures and to some cost factors, to compress the range for a variable, thus drawing in the large values toward the origin.

TABLE I

REDUCED SET OF COST FACTORS

USED AS INPUTS FOR FINAL REGRESSION ANALYSIS

Believed to Increase Costs

Innovation in system

Complexity of overall system

Log_{10} number of subprograms

Log_{10} number of words in data base

Log_{10} number of classes of items in data base

Log_{10} number of words in tables and constants not in data base

Log_{10} number of input message types

Log_{10} number of output message types

Complexity of program design

Percent math instructions

Percent logical control instructions

Percent generation to produce desired output

Insufficient memory capacity

Insufficient I/O capacity

Stringent timing requirements

First programming effort on computer

Log_{10} average turnaround time with the computer

Computer operated by agency other than developer

Program developed away from operational location

Computer at operational site different than at development site

Program developed at more than one location

Log_{10} number of reused instructions*

Percent error rate--100 x "scrap" instructions/ total instructions coded*

Percent operational discards--100 x "scrap" instructions due to changes/total instruction coded*

Believed to Decrease Costs

Percent clerical instructions

Percent self-checking-fix instructions

Percent information storage and retrieval

Estimated customer experience

Time-sharing

Management index--the ratio of "yes" answers to the total set below:

Existence of a documented management plan for:

- processing of system design changes
- processing of program design changes
- dissemination of error-detection and error-correction information
- use of computer facility.
- contingency for computer unavailability
- communication with other agencies
- design specification concurrence procedures
- cost control
- management information control
- document control
- standards for coding, flow charting

Percent programmers participating in design

Log_{10} production rate--instructions/man month

Percent senior programmers

Factors with Neither Hypothesis

Percent I/O instructions

Open/closed shop

*Measured in number of machine language instructions

TABLE II

DEFINITIONS AND CODING FOR VARIABLES USED IN THE EQUATIONS

Cost Variables

Y_1 - Total number of man months including first line supervision to program design, code, and test and document this program not including the cost of any associated executive or utility program.

Y_2 - Total number of computer hours used by all developmental computers.

Y_3 - Number of new machine language instructions written for this program (system) not including reused subroutines, logical blocks, and subprograms.

Y_4 - Months elapsed--completion data for program delivery minus start date for program design. At the time of program delivery the program is ready to be installed in the operational computer to begin system test. The program design activity uses the operating system description and operational specifications as inputs to develop program design specifications and flow charts.

Predictor Variables

X_1 - MOL versus POL, coded POL = 1; MOL = 0. POL uses procedure-oriented or compiler language for source statement--MOL uses machine-oriented assembly symbolic language source statements.

X_2 - Small- versus large-scale developmental computer systems, coded small = 0; large = 1. Machines with less than or equal to 16,000 words of core memory are small--those with more than 16,000 are large.

X_4 - Stringent timing as a constraint on programming design, coded yes = 1; no = 0.

X_5 - First programming effort on computer, coded yes = 1; no = 0.

X_6 - Program developed at more than one location, coded yes = 1; no = 0.

X_7 - Number of subprograms in this program (system)--divisions in the program design for logical reasons and/or division of programming labor.

X_8 - Total number of classes of items in the data base. Classes means categories of types of items such as names of people, salaries, cities, states or any characteristics of information for which there are many items or entries.

X_{10} - Estimate of customer knowledge or experience with the development of automatic data processing systems, coded extensive = 3; limited = 2; vague = 1.

X_{11} - Percent programmers participating in design = $\frac{\text{Number programmers participating in design}}{\text{Maximum number of programmers}}$ coded in decimal. Design may include both requirements analysis conducted to specify in detail the performance requirements of this information processing system, and the operational design activity to translate these requirements into operational design specifications that indicate how the needs will be satisfied.

X_{12} - Percent clerical instructions, coded in decimal--bookkeeping, sorting, searching, and file maintenance instructions as compared with mathematical input/output, logical control and self-checking instructions.

X_{14} - Percent generation functions to produce desired outputs, coded in decimal, as compared with other functions such as information storage and retrieval, data acquisition and display, control or regulation, decision making, and transformation.

Evaluation of the Results

The equations derived to date have large standard errors of estimate--the differences expected between actual and estimate may be 100 percent or larger of the actual. The Programming Management Project members recommend that the present equations only be used as an aid to support estimation by providing a basis for comparison with other techniques. Because of the tendency to underestimate costs, the general recommendation is to use the "most conservative" estimate when several are available.

In the search for more accurate results from the statistical analysis, potential errors in both the methods and in the data used to date have been identified, for example:

- The Model discussed earlier contains some gross assumptions that probably need refinement.
- The feedback suggests that a separate set of questions should be used to gather data for each cost measure. For example, to develop the equation for the cost measure, months elapsed, questions could be added to indicate how manpower was applied over the actual elapsed time by identifying intermediate milestones in computer program production.
- Not all of the factors solicited by the questionnaire are strictly appropriate for all types of programs.
- The accuracy of many individual answers depends upon the effort that the individual respondent devoted to completing the questionnaire and the availability of the data.
- Many of the responses showed the need to do more work to define terms more precisely.
- Even if the data were accurate, how representative the same is is unknown. The data are probably not a truly random sample over the range of values for cost factors and cost variables. Defining a good statistical sample that can serve as a basis for generalizing the analytical results to the population of computer programming jobs is

- Use of personal interview to obtain more reliable data rather than indirect mail and phone contacts.

- Improvement in the definition of a data point to differentiate as needed among runs, subprograms, programs, and program systems.

CONCLUSION

This research and development work in the Programming Management Project is one attach on one problem area in the industry. It represents a beginning; more work is needed. The long-range goals of such work should extend beyond the problems of improved cost estimation and planning for computer programming. Aids and standards are urgently needed to make decisions or trade-offs between cost versus value for planning (before), control (during), and evaluation (after) of computer programming projects.

Such standards are also needed to make decisions on directions for work to develop new tools such as utility computer programs for computer programming. Although most work to improve the tools for doing computer programming actually has an economic foundation, little has been done to define the requirements for these developments in terms of cost and value and then to collect the appropriate numerical data for analyses--analyses that would improve the ways in which management decisions are made in selecting, guiding, and evaluating new developments.

Although scientific approaches can only help alleviate parts of complex management problems in the computer programming industry, the gap between what could be done and what has been done is large. Improvements in understanding and in creating usable aids can be achieved by taking a more scientific approach--defining, comparing, measuring, and analyzing. Such an approach could lead to standards or benchmarks. In turn, these can serve as a coordinate system against which performance and costs can be measured. Only when such standards or benchmarks exist can high-confidence assessments be made in whether change really represents progress.

Computer Program Development Procedure for Gemini

by

R. R. Carley

Manned Spacecraft Center NASA

(This paper was not available at the time of publication of the proceedings.)

(page 344 blank)

SDC RECOMMENDATIONS FOR SPACEBORNE SOFTWARE MANAGEMENT

by

S. D. Manus
System Development Corporation

Everyone concerned admits to problems in spaceborne software development. Each has his own version of what these problems are. Assuming that each of these problems is truly legitimate, a need arises to put them into some sort of perspective that allows the development of means for their solution.

Management has the responsibility to obtain and dispense resources necessary to implement solutions. In order to accomplish this, management must be presented with an accurate picture of how things are. The purpose of this paper is to present a cohesive view, based on valid information, [1] of those aspects of spaceborne software development upon which management may have a positive impact.

For ease of presentation, this paper is divided into the following areas:

1. The Nature of the Spaceborne Software Development Cycle
2. Common Problems That Have Been Isolated
3. Recommended Solutions to These Problems
4. Recommended Means to Implement These Solutions

I. THE NATURE OF THE SPACEBORNE SOFTWARE DEVELOPMENT CYCLE

In a paper by A. Tucker[2] titled Summary of Current Spaceborne Software Systems it was stated that the sequence of major activities that make up spaceborne software development were basically the same regardless of projects and

---

1. All information in this paper appears either directly or indirectly in the System Development Corporation Spaceborne Software Systems Study reports: TM-(L)-3067/001/00, TM-(L)-3067/002/00, TM-(L)-3067/003/00, TM-(L)-3067/004/00.

2. Summary of Current Spaceborne Software System, by A. Tucker, presented at Spaceborne Computer Software Workshop, 20 September 1966.

Life Cycle phases. In a sense the collected data represents a cross section of various projects over various phases of their respective Life Cycles (see Figure 1).

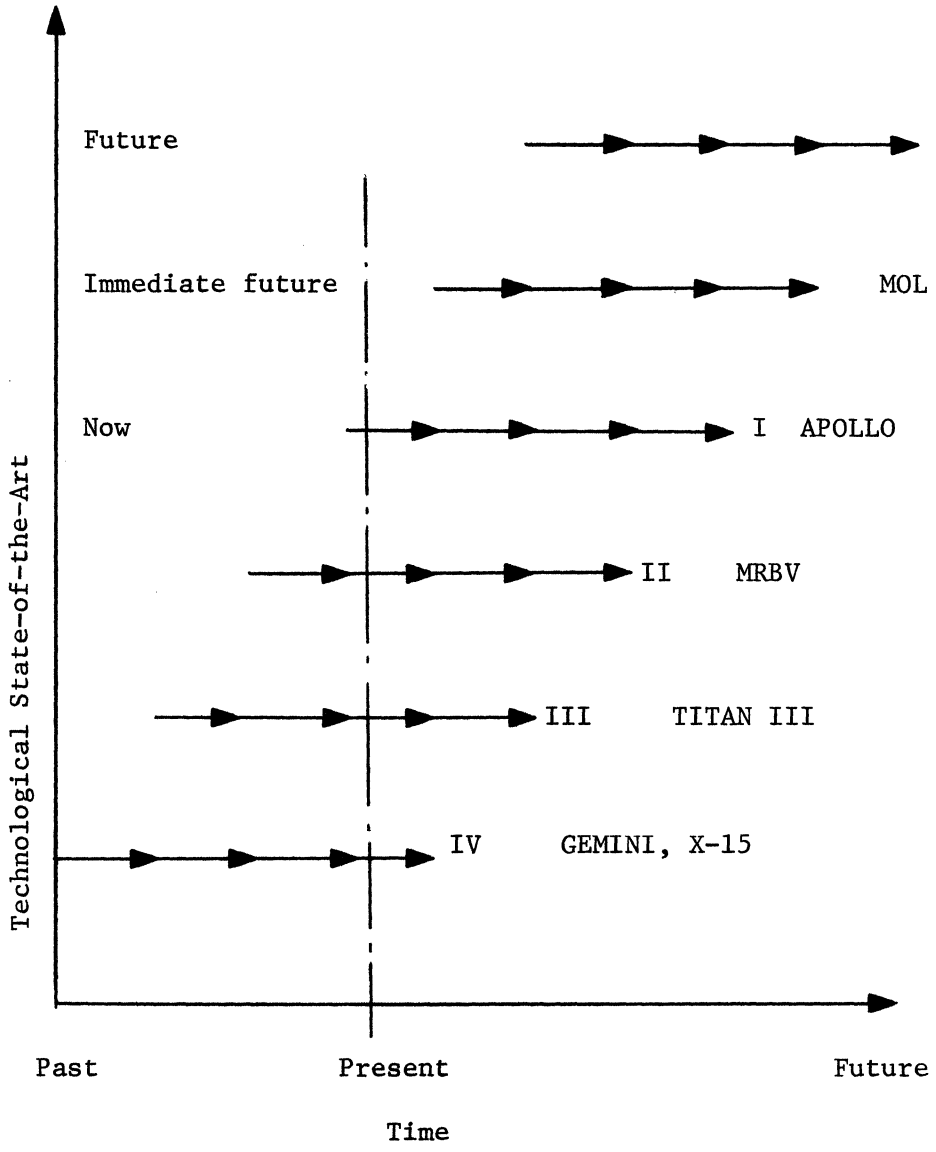


Figure 1. Life Cycles of Space Projects

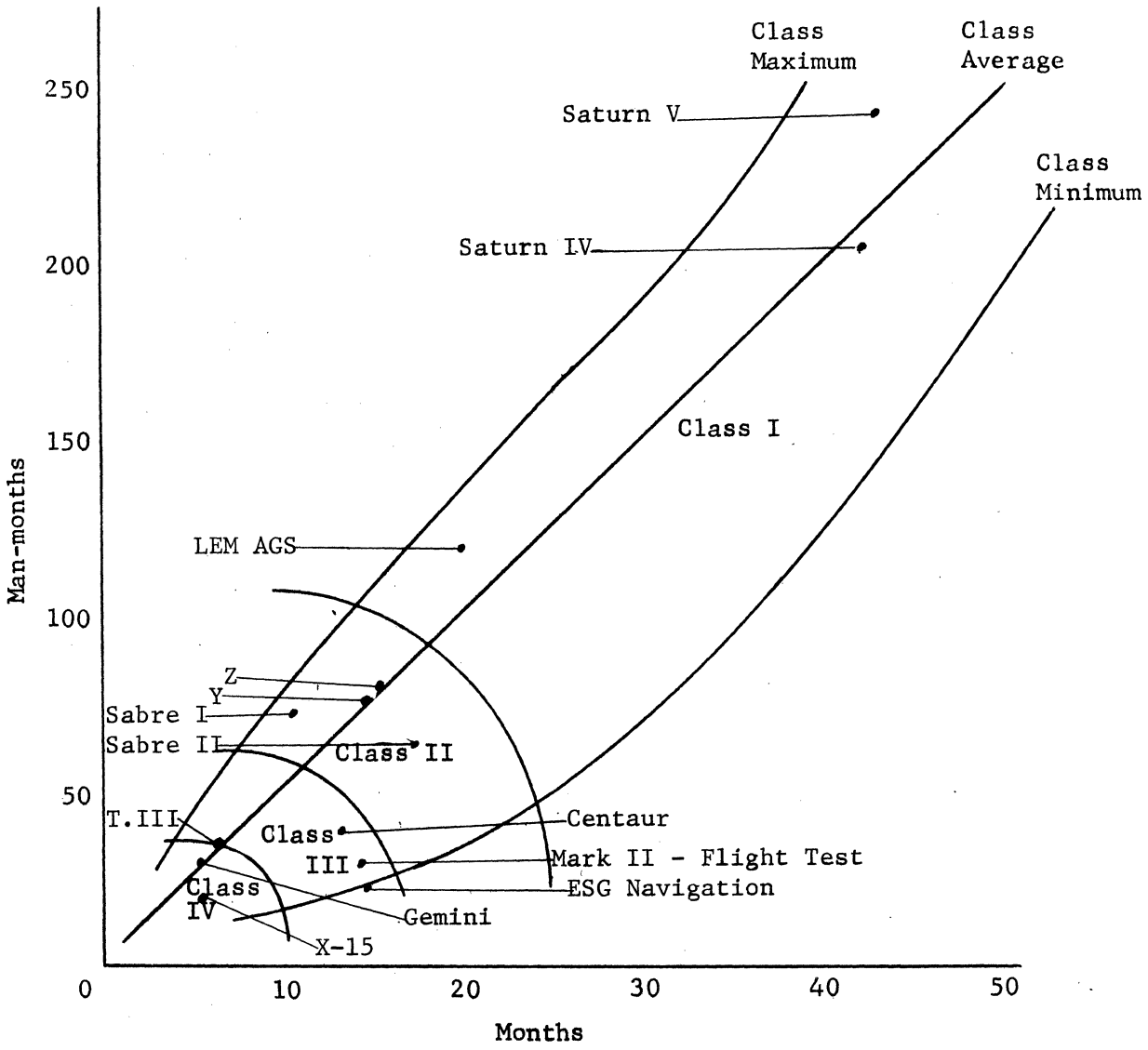


Figure 3. Total Time Vs. Total Man-Months for Programming Phase (Excluding Validation)

Examples of extremes are the LEM support computer program, in Phase 1, a 4K program utilizing 96 man months, and the Gemini computer program, in Phase IV, an 18K program utilizing 8 man months of effort, the difference being in the degree of hardness of the hardware, software, and problem statement.

It becomes apparent (see Figure 3) that resources may be considerably reduced over a Total Life Cycle, perhaps by an order of magnitude. Now, having related one spaceborne software development cycle to another, it remains to relate spaceborne software development to that of all software development. Farr, LaBolle, and Willmorth investigated four large systems, some results of which are shown in Figure 5. Similarly breaking out spaceborne software development systems and comparing, it will be noticed that the Phase IV data closely reflects the general systems. That is, by Phase IV, the problems in spaceborne have the same impact on resource utilization as any other system. This is further amplified by the results in Figure 6, showing comparative programming rates.

In short, the first three phases of the Total Life Cycle demand extensive, but decreasing resources, due to the softness of hardware, software and problem statement, but by Phase IV when all of these areas are hardened, spaceborne software development reduces to resource utilization comparable to that utilized in general ground systems.

II. COMMON PROBLEMS THAT HAVE BEEN ISOLATED

The seven most common problems in order of severity are:

1. Changing specifications
2. Inadequate specifications
3. Short lead time
4. Inadequate communications
5. Lack of manpower
6. Minimal support tools
7. Obsolescent hardware

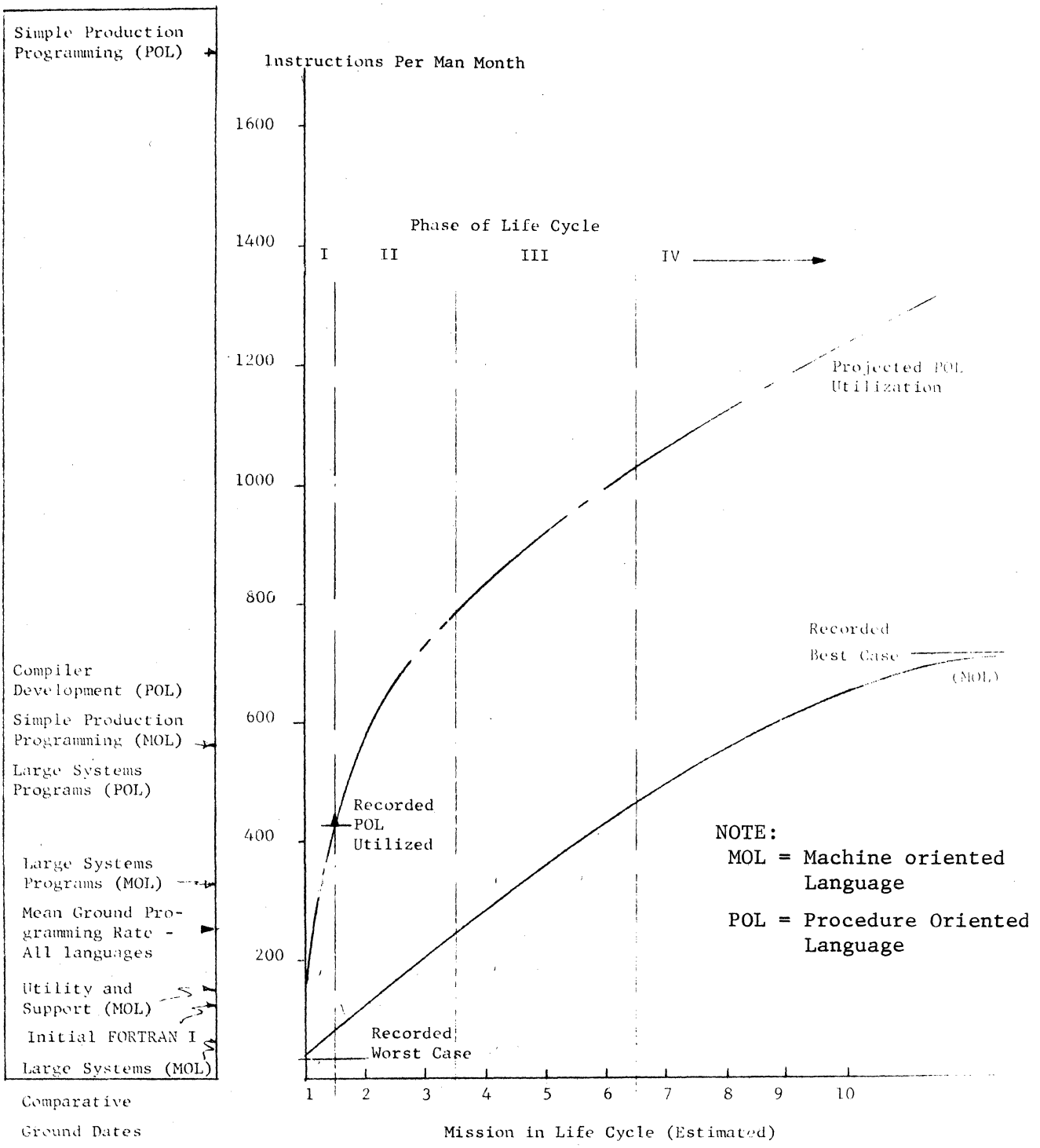


Figure 6. Comparative Programming Rates

D. INADEQUATE COMMUNICATIONS

Inadequate communications means unwanted redundancy, which leads to excessive utilization of resources. If insufficient information is passed forward to succeeding steps in a development cycle, then work must be repeated since the cycle cannot continue without sufficient information. Sometimes this is due to a contractual structure, allowing poor interfacing. Other times, this is due to poor procedures. Still other times, it is due to short lead times. In fact, redundancy of this sort seems to be the order of the day in the spaceborne software development environment. The irony of it all is that almost all necessary information for any step is available at an earlier step but is not gathered and sent forward. Even more ironic is the fact that a lot of it exists on the computer, but is not collected, and passed on.

E. LACK OF MANPOWER

Lack of manpower, on the surface, seems a minor problem, since programs are being created within allowable time spans. However, indications are that our present available spaceborne programming manpower is actually marginal. For instance, almost every organization interviewed indicated that they have sufficient manpower to create their given spaceborne software. However, when pressed with their capability to handle two or three or more such software developments concurrently, their remarks were of the sort that indicated that programmers are hard to come by. There are no effective programmer reserves for spaceborne programming! If spaceborne programs become very large, or the number of launches is increased markedly, or if lead times are to be shortened, or any combination of these, there will be insufficient manpower, at least as utilized in the present mode of spaceborne software development.

F. MINIMAL SUPPORT TOOLS

Minimal support tools impede programming and checkout. Aside from some necessary simulation programs, the only other support tools generally available to the spaceborne programmer are those tools usually supplied within his in-house general purpose operating system, tools of general usage for all the programmers in the organization, whether involved in the spaceborne area or not. Also, the spaceborne programmer must stand in line for checkout and

mission fulfillment. It is not meant to be a paper mill where most of the time is spent in filling out forms and obtaining permission through channels before each minor step is taken. In fact, most of the management information itself, could come as output of the various computer program utilized in the software development.

Second, the total time span for formulation and programming should be divided a bit differently. Formulation time should be expanded at the expense of programming time, about 25 percent more. This would allow a firmer specification before programming starts, hence less changing and redundancy. Since specifications are open ended at the beginning of the cycle, the closer the specification can be brought to the end of the cycle, the less changes or additions will occur during programming.

Third, more programming aids should be developed. If the programming time span is shortened, a new problem area is created, since this is the one area where things seem somewhat under control. But as noted earlier, this situation is marginal today, and is due for trouble tomorrow, what with more and larger missions planned. Such aids reduce the programmer requirement, and in fact, may reduce the actual amount of programming created by reutilizing programming from other or earlier efforts. Given decent problem formulation and a better computer configuration, the programming effort tends to diminish. Given a way of programming that seems computer independent, such as the utilization of a problem or procedure-oriented language, tends to harden the parameter of software mentioned in the Total Life Cycle, which would tend to eliminate Phase II. Evidence gathered already indicates the manning may be reduced threefold by utilization of an appropriate POL (see Figure 6).

Finally, inclusion of the programmer in problem formulation will lessen the chance of redundancy. This, in fact, is done by at least the Gemini project, with notable results. Since so much of the specifications are tied to computer configuration and past knowledge of programming implementation, as well as the internal logic of the programs, the programmer is in a position to supply valuable input to problem formulation.

If a common spaceborne software development system is to be implemented out of funds originally allocated for programming in the respective projects, then it would seem prudent to have each project make its needs known. This means that representatives of all interested organizations must get together in some sort of organization to determine the total needs, the mode of system implementation, the costs, and the direction of evolution of such a system, etc.

However, if such steps are not taken, it is perfectly likely that some other authoritative group will estimate the needs, system, etc., and enforce its usage upon the spaceborne community. If their estimates are incorrect, then a penalty is taken. As a point of fact, this vertical stance is historically more common than the horizontal, cooperative stance. This is due to the fact that most organizations attempt local optimization, and disregard any interface into the broader picture. Examples of such enforced systems are PERT III and 375-1. It is not clear that either of the examples are systems that do a really good job.

Consequently, SDC has recommended the horizontal stance and has formulated guidelines that would allow the development of a user-oriented management system (see footnote 1). It now remains to describe the relationship of each project relative to such a system when developed.

Go slow! Mission first, software second! This is the basic criteria for climbing aboard. If the development of a common spaceborne software development system is handled properly, there is no reason for any mission to be scrubbed because such a system interfered with mission requirements. Elements of the system should be picked up at a reasonable rate by each project and in such a way that currently used elements are superseded on an effective basis, until the total system is utilized.

For instance, the degree of implementation of the software management elements might depend on which phase of the Total Life Cycle the project is in. In Phase I, a high level system would seem sufficient, with emphasis on tracking and controlling change in hardware, software, and problem statement. By

WORKSHOP ATTENDEES

1. Attendees by name
2. Attendees by Company

(page 362 blank)

SSD/AEROSPACE SPACEBORNE COMPUTER

SOFTWARE WORKSHOP

20-22 SEPTEMBER 1966

ATTENDEES

Name	Company
Norman F. Ablett	Douglas Aircraft
Marvin E. Alberda	Aerospace, San Bernardino
L. J. Andrews	Aerospace Corporation
George Arnovick	Informatics, Inc.
Dr. Algirdas Avizienis	UCLA and JPL
E. J. Barlow	Aerospace Corporation
Capt. Jesse J. Bass, Jr.	USAF, (AFSC)
Morris L. Bernstein	Nortronics
Peter Biche	Logicon, Inc.
Stanley Blumenstein	Aerospace Corporation
Barry W. Boehm	RAND Corporation
Edward L. Braun	Aerospace Corporation
Edward R. Brooks	Aerospace Corporation, SBO
Coleman T. Brown, Jr.	Honeywell, Inc.
Norman K. Burnett	UNIVAC
T. J. Burns	Hughes Aircraft Co.
William R. Bush	Douglas Aircraft
David Caplan	Raytheon
Levi J. Carey	System Development Corporation
R. R. Carley	MSC - NASA
James W. Chapman	Control Data Corporation
William H. Cheever	General Electric Company
Leonard G. Chesler	RAND Corporation
Charles A. Clark	Raytheon
William T. Clary	Autonetics, Division of NAA
J. M. Coggeshall	IBM
Paul Colen	Aerospace Corporation, SBO
Col. James Collier	USAF (AFRST)
Ann T. Collins	Aerospace Corporation
Robert E. Conklin	USAF Wright-Patterson AFB

Name	Company
G. A. Hirschfield	System Development Corporation
Larence Hirschl	Aerospace Corporation
Gerhard L. Hollander	Hollander Associates
Helen A. Holman	Douglas Aircraft
Gary R. Howard	Douglas Aircraft
Keith H. Howes	Control Data Corporation
Henry J. Ilger	System Development Corporation
Leo F. Jarzomb	Aerospace Corporation
T. R. Jefferies	Honeywell, Inc.
C. Walter Johnson	AC Electronics Div. General Motors
Harold Johnson	IBM Corporation
V. Josephson	Aerospace Corporation
R. R. Joslyn	Douglas Aircraft
G. J. Kacek, Jr.	General Electric Company
H. A. Keit	Universal Data Systems
G. W. King	Aerospace Corporation
K. Kirkpatrick	AC Electronics Div. General Motors
Ronald D. Knight	System Development Corporation
Michael Kowalsky	Aerospace Corporation
Bal Krishan	Aerospace Corporation
Victor LaBolle	System Development Corporation
Dick Lanham	System Development Corporation
Louis I. Larson	Douglas Aircraft Company
Norbert D. LaVally	Universal Data Systems, Inc.
T. J. Lawton	MIT
Carl M. Lekven	Aerospace Corporation
Leon S. Levy	IBM
S. Herbert Lewis	Aerospace Corporation
Michael W. Lodato	Douglas Aircraft
Frank Long	System Development Corporation
Douglas T. Loughmiller	Douglas Aircraft
Glen P. Love	Douglas Aircraft

Name	Company
Carl Remstad	IBM
T. E. Rodgers	Douglas Aircraft
Capt. George L. Roeder	USAF Academy
Donald E. Root	General Precision, Kearfott Division
Capt. Robert M. Russ	USAF
Joseph E. Santa	Douglas Aircraft
Philip H. Sayre	Planning Research Corporation
W. E. Schopman	Douglas Aircraft
P. R. Schultz	Aerospace Corporation
Col. G. W. Scott	USAF (SSTD)
Lloyd V. Searle	System Development Corporation
Donald L. Segel	Douglas Aircraft
Herbert R. Seiden	System Development Corporation
Gilbert Siegel	Douglas Aircraft
Ronald R. Sikes	Hollander Associates
Gerard Sillman	System Development Corporation
Jack H. Simpson	Douglas Aircraft
E. L. Smith	Logicon, Inc.
Flint H. Smith	Douglas Aircraft
Lucile F. Solberg	Aerospace Corporation
Edward J. Solheim	UNIVAC
Paul Soulier	Computer Sciences Corporation
R. N. Southworth	Logicon, Inc.
A. L. Spence	RCA
Thomas C. Spillman	IBM
Dean G. Stark	IBM Corporation
Duane Starner	Martin Company
Thomas B. Steel, Jr.	System Development Corporation
Robert Steinert	System Development Corporation
Allan J. Stone	Hughes Aircraft Company
Everett S. Stone	System Development Corporation

SSD/AEROSPACE SPACEBORNE COMPUTER

SOFTWARE WORKSHOP

20-22 SEPTEMBER 1966

ATTENDEES

AC Electronics Div. General Motors

C. Walter Johnson
K. Kirkpatrick

Aerospace Corporation

L. J. Andrews
E. J. Barlow
Stanley Blumenstein
Edward L. Braun
Ann T. Collins
R. G. DeBiase
Lewis E. Dorough
James W. Edmundson
Robert V. Erilane
David G. Frostad
J. W. Gibson
Frank H. Harrison
Larence Hirschl
Leo F. Jarzomb
V. Josephson
G. W. King
Michael Kowalsky
Bal Krishan
Carl M. Lekven
S. Herbert Lewis
Ralph Muriello
David McColl
R. J. Mercer
A. J. Osborn
P. R. Schultz
Lucile F. Solberg
Lindley S. Wilson

Aerospace, San Bernardino

Marvin E. Alberda
Edward R. Brooks
Paul Colen
Ralph B. Conn
James Eliades
Don Farr
William J. Swartwood

Autonetics, Division of NAA

William T. Clary
W. B. Herr
Henry H. Megrund
Victor Strand

Computer Sciences Corporation

Phillip D. Nelson
Paul Soulier

Control Data Corporation

James W. Chapman
Keith H. Howes

Douglas Aircraft

Norman F. Ablett
William R. Bush
M. H. Culp
D. E. French
H. C. Hagins
Helen A. Holman
Gary R. Howard
R. R. Joslyn
Louis I. Larson
Michael W. Lodato
Douglas T. Loughmiller
Glen P. Love
David L. Mootchnik
W. D. Nason
T. E. Rodgers
Joseph E. Santa
W. E. Schoppman
Donald L. Segel
Gilbert Siegel
Jack H. Simpson
Flint H. Smith
L. J. Surfas
Brian Robert Williams

Raytheon

David Caplan
Charles A. Clark
H. H. Nishino
A. W. Yondå

RCA

E. H. Miller
A. L. Spence

System Development Corporation

Levi J. Carey
Gerard A. Hirschfield
Henry J. Ilger
Ronald D. Knight
Victor LaBolle
Dick Lanham
Frank Long
Stan D. Manus
Warren E. Meyer
John B. Munson
George Neil
Lloyd V. Searle
Herbert R. Seiden
Gerard I. Sillman
Thomas B. Steel, Jr.
Robert Steinert
Everett S. Stone
Alfred E. Tucker

TRW Systems

Howard Grossman
David L. Meginnity
W. V. Neisius

UCLA and JPL

Dr. Algirdas Avizienis

UNIVAC

Norman K. Burnett
Robert K. Draving
Walter G. Haberstroh
Philip L. Phipps
Curt W. Rangen
Edward J. Solheim

Universal Data Systems

Richard A. Hill
H. A. Keit
Norbert D. LaVally

U. S. AIR FORCE

U. S. Air Force Headquarters

Colonel James Collier (AFRST)
Captain Philip F. Gehring (AFADO)

Air Force Systems Command Headquarters

Major Dwight F. Rehberg (SCSEC)
Captain Jesse J. Bass, Jr.

Air Force Academy

Captain George L. Roeder

Electronics Systems Division

Lt. George E. Uranesh
Murad S. Piligian
M. Ratynski

Research & Technology Division

Major William J. Wilson (RTTR)
Robert E. Conklin (Wright-Patterson Air Force Base)
Dan T. Reed (Avionics Lab.)

Space Systems Division

Colonel G. W. Scott (SSTD)
Liet/Col. Charles D. Orrison (SST)
Major M. A. Ikezawa (SSTDG)
Captain Charles Osinski (SSUJ)
Captain Robert M. Russ (SSUJ)