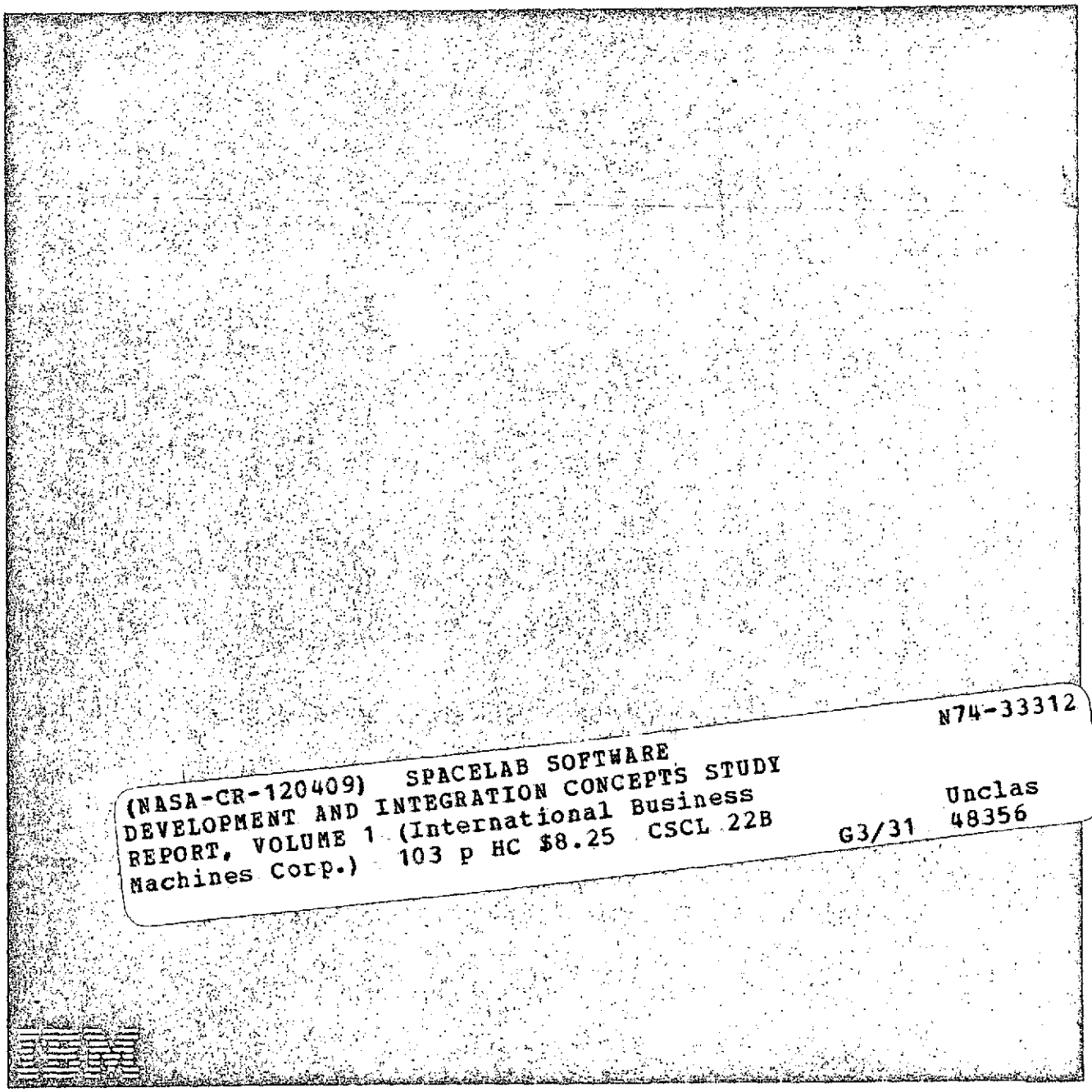
Prepared for the
GEORGE C. MARSHALL
SPACE FLIGHT CENTER
Huntsville, Alabama

31 October 1973

Contract No.: NAS8-30376
MSFC No.: MSFC-DRL-389, Line Item No. 2
IBM No.: 73W-00326

SPACE LAB
SOFTWARE DEVELOPMENT AND INTEGRATION CONCEPTS STUDY REPORT

Volume I



N74-33312

(NASA-CR-120409) SPACELAB SOFTWARE DEVELOPMENT AND INTEGRATION CONCEPTS STUDY REPORT, VOLUME 1 (International Business Machines Corp.) 103 p HC $8.25 CSCL 22B

G3/31 48356

Unclas

This report was prepared by P. L. Rose and B. G. Willis.

The Integrated Development Concept was developed by D. H. Norton.

PREFACE

This report documents the proposed software guidelines to be followed by the European Space Research Organization (ESRO) in the development of software for the Spacelab being developed for use as a payload for the Space Shuttle. This report was developed by the IBM Federal Systems Division, Huntsville, Alabama, under contract No. NAS 8-30376 from the National Aeronautics and Space Administration, Marshall Space Flight Center.

SPACELAB
DEVELOPMENT & INTEGRATION
CONCEPTS

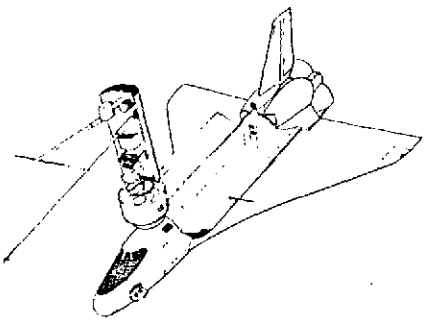


TABLE OF CONTENTS

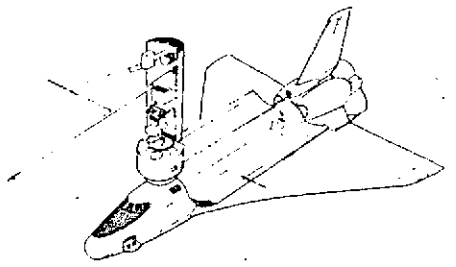
Section	Paragraph	Title	Page
1		**INTRODUCTION**	
	1.1	PURPOSE	1.2
	1.2	SCOPE	1.3
	1.3	OBJECTIVE	1.4
	1.4	STUDY BASE	1.6
	1.5	DEFINITION OF SOFTWARE TERMS	1.9
2		**CONCEPTS & PHILOSOPHY**	
	2.1	PROGRAM MANAGEMENT PHILOSOPHY	2.3
	2.2	THE SOFTWARE DEVELOPMENT PROCESS	2.4
	2.3	OPERATION	2.12
3		**TECHNIQUES**	
	3.1	TECHNIQUES	3.2
	3.2	COMPOSITE DESIGN	3.8
	3.3	STRUCTURED CODE	3.12
	3.4	PROGRAM LIBRARIES	3.19
	3.5	STRUCTURED DOCUMENTATION	3.22
	3.6	CAUSE AND EFFECT ANALYSIS	3.28
	3.7	SIMULATION	3.32
	3.8	AUTOMATED MANAGEMENT BOARDS	3.38
	3.9	SOFTWARE MANAGEMENT BOARDS	3.42
4		**SYSTEM CONSIDERATIONS**	
	4.1	HIGH ORDER LANGUAGE	4.2
	4.2	USER LANGUAGE	4.4
	4.3	ONBOARD COMPUTER	4.6
	4.4	SUPPORT SOFTWARE	4.12
	4.5	SOFTWARE	4.16
5		**SUMMARY**	
	5.0	GUIDELINES SUMMARY	5.1

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.3-1	Hardware vs Software Cost	1-5
2.0-1	Integrated Software Development Activity	2-2
3.1-1	Software Development Approach Comparison	3-5
3.1-2	Software Milestone Comparison	3-6
3.3-1	Logic Structures	3-13
3.3-2	Traditional and Structured Control Code	3-16
3.5-1	Visual Table of Contents Diagrams	3-26
3.5-2	Detail Flow/Input-Process-Output Diagrams	3-27
3.7-1	Software Debug Comparison	3-37
3.8-1	Deviation and Delivery Tracking Manpower Requirements	3-41
3.9-1	Software Management Board Evolution	3-45
3.9-2	Software Control Board Functions	3-46
4.3-1	Computer Hardware/Software Trends	4-7
4.3-2	Memory Utilization History	4-8
4.3-3	Computer Capability (CPU Memory Capacity)	4-9

INTRODUCTION

1



Paragraph	Title	Page
1.1	PURPOSE	1.2
1.2	SCOPE	1.3
1.2.1	Development	1.3
1.2.2	Operational Support	1.3
1.2.3	Management	1.3
1.3	OBJECTIVE	1.4
1.3.1	Cost	1.4
1.3.2	Schedule	1.4
1.3.3	Performance	1.4
1.4	STUDY BASE	1.6
1.4.1	Experience	1.6
1.4.1.1	Saturn	1.6
1.4.1.2	Skylab	1.7
1.4.2	Analysis	1.7
1.4.2.1	Space Shuttle	1.7
1.4.2.2	Spacelab	1.7
1.4.2.3	Experiment	1.7
1.4.3	Innovation	1.8
1.5	DEFINITION OF SOFTWARE TERMS	1.9

1.1 PURPOSE

It is the purpose of this report to provide guidelines for the development of Spacelab software--software not only for the operation of the data management subsystem, the support of experiments and space applications, but also the software used with ground support equipment, and for integration testing. These guidelines have applicability to all software associated with Spacelab and have been developed from consideration of special Spacelab characteristics, such as the wide geographic dispersion and multinational effort involved by the Spacelab development and use.

Clearly, the software for Spacelab should be characterized by:

- Systems which are functional
- Systems which are simple
- Systems which are well structured
- Systems which are well documented
- Systems which are easy to keep running
- Systems which are easy to modify
- Systems which are reliable
- Systems which are usable

Many of the concepts, tools, and techniques described within this document are refinements of existing techniques. This report integrates those techniques that have proved their worth over several years and new techniques that show high promise.

1.2 SCOPE

This report defines concepts, techniques, and tools needed to assure the success of a programming project. Experienced programmers have used these concepts, techniques, and tools on previously successful programs, but the formulation and strict adherence recommended by this report provides a programming philosophy that encourages overall reduction of cost and time. This philosophy also provides a method of creating an end product that is understandable, maintainable, and, most important, functionally correct. The report was written with the intent of bringing scientific discipline to the development of software.

All of the concepts, techniques and tools presented in this report fit logically into three main areas of software functions.

1.2.1 Development

Development includes the normal divisions of software generation. These are design, implement, verify, document, and control.

1.2.2 Operational Support

Operational support implies all necessary support required to successfully complete the software project. This includes hardware support (computers, facilities, models), and software support (compilers, simulators). Tools must be provided for the operational phase of Spacelab which will allow principle investigators and crew members to monitor and control the varied experiments envisioned.

1.2.3 Management

Management covers all systems necessary to control cost, schedule, and performance. Configuration Management and Change Control techniques will be discussed in terms of applicability for NASA requirements.

1.3 OBJECTIVE

The objective of this study is to recommend concepts, techniques, and tools that, if applied to the initial Spacelab software development, will reduce cost, support strict schedules, and ensure the performance level required in a manned space environment.

1.3.1 Cost

Experience has proven that in today's computer applications hardware unit costs are less than software unit costs (see Figure 1.3-1). The quality of work done during program development has a direct influence on the operational cost. These two facts will be of prime concern to NASA since the majority of NASA's cost will be in the operational aspect of the program. Any concepts, techniques, or tools that can be incorporated into the development phase to provide:

- Understandability
- Usability
- Maintainability

will certainly aid in a smooth transition from the development phase (ESRO) to the operational phase (NASA) of Spacelab software.

1.3.2 Schedule

Cost is an important aspect of software generation; however, just as important in the Spacelab program are the strict schedule requirements that the developed software must support.

The proposed 50 flights per year present a software scheduling challenge that has never been attempted in a space application. This report outlines concepts and techniques that will aid in the development of software that is understandable, easily changed, and easily verified in an operational state. It is imperative that the Spacelab software adhere to these attributes if the strict schedules and time lines are to be met once the software system is delivered.

1.3.3 Performance

Cost and time must be controlled during the life of a long term project like Spacelab, however, many previous software development phases have controlled these important factors yet failed to meet the functional requirements. In basic terms, cost and time objectives were met, but the program did not do what was intended. This, of course, is an inadequate product and additional resources must be expended to modify or change the developed software to perform the required functions.

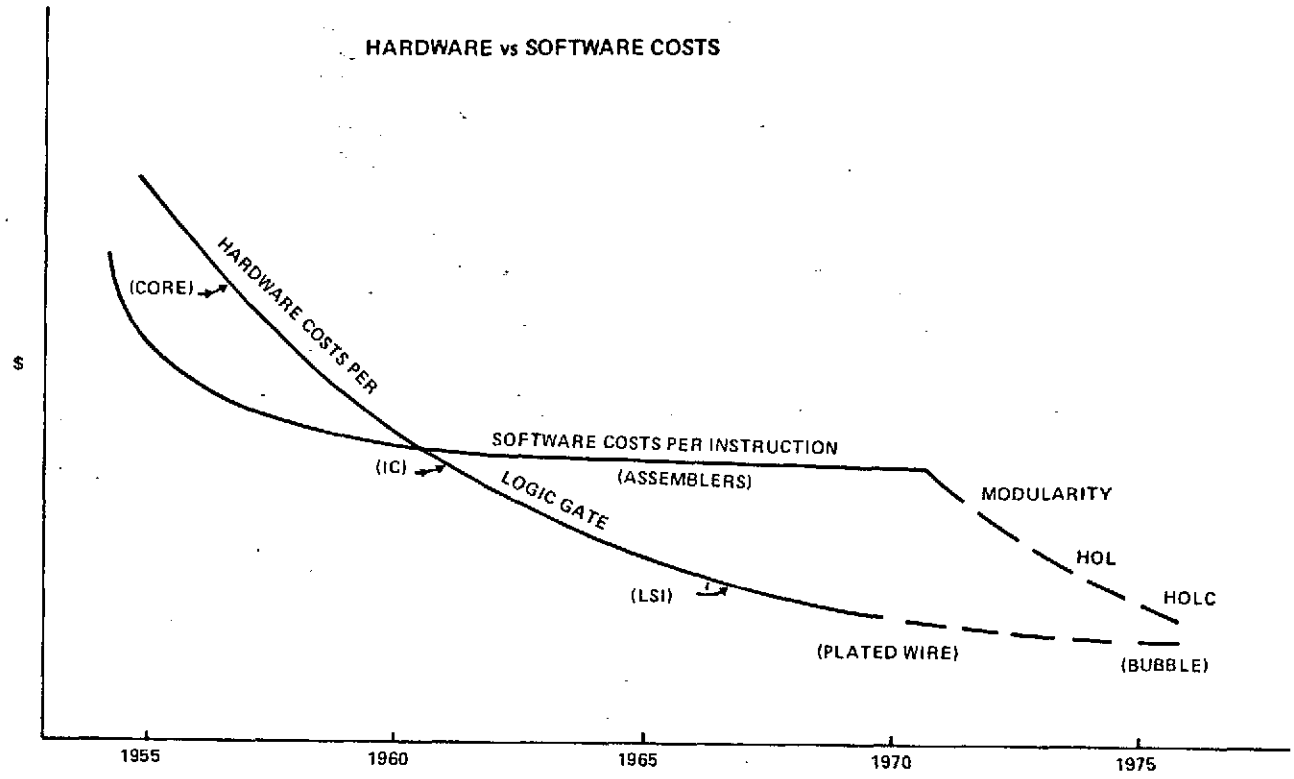


Figure 1.3-1 Hardware vs Software Cost

1.4 STUDY BASE

In the performance of this study, the data provided herein was obtained from experience gained on projects such as Saturn, Skylab, Spacelab, Manned Orbiting Laboratory, Real Time Computer Complex, and non-space related projects such as Tactical Aircraft Guidance System, FAA, Airborne Warning and Control System, and A-701E. For brevity, the related Spacelab applications are summarized below. Analysis of information gained from these programs and innovative approaches of problem solving contributed significantly to the detailed sections of this study.

1.4.1 Experience

1.4.1.1 Saturn

IBM has a broad experience base in the checkout and control of space vehicles. The complexities inherent in the diverse mission requirements supported by these systems and the use of the systems at multiple sites have demanded the employment of sophisticated technical and management techniques. Integral components of this base include:

Saturn Ground Computer System - consists of general purpose computers connected together via a high speed data link. The Operating System which executes in the two computers performs task management, display services, discrete processing, and redundancy management.

Application programs were developed by IBM which execute under the control of the Operating System. Of particular significance, an engineering-oriented user checkout language was developed for the purpose of providing a simple language for engineers to use which would not require an intensive programming knowledge. This language was the Acceptance Test or Launch Language (ATOLL). These application programs perform the automatic prelaunch checkout and verification of the Saturn vehicle.

Saturn Operational Display System - consists of a Display Control Computer and its associated equipment. This system provides a communication link between the test engineer and the launch vehicle. The operating software controls the displaying of data being input from many varied sources.

Digital Events Evaluator - consists of three high speed general purpose computers. Communication between these computers takes place over both high and low speed data links. This system monitors the status of discretes input from the Saturn vehicle and prints a record of each change in discrete status.

Saturn Launch Vehicle Flight Programs - performs navigation, guidance, attitude control, launch vehicle sequencing, telemetry, command processing, and programmed backups to specified hardware functions for the Saturn vehicle.

Operational Support Systems - Simulators played an important part in developing and validating the software. A System 360/75 was used for interpretive simulation; and a Simulation Laboratory and Flight Program Checkout Facility were used to develop and verify the Saturn Launch Vehicle Flight Program software. The ground system software was developed and validated on a hardware breadboard. The System 360/75 was also used in the configuration/control and delivery process for all developed systems.

1.4.1.2 Skylab

The Skylab is controlled by two special purpose computers, with one performing control functions and the second available as backup. IBM programmed these computers to perform the attitude control functions, subsystem monitoring, and experiment control.

The software for the Skylab control computer was developed using the top-down programming technique. The program development and validation was aided by the use of simulation. The same simulators used for the Saturn Launch Vehicle Flight Program were modified to be used for Skylab.

1.4.2 Analysis

The experience gained on the Saturn and Skylab programs provides a base to review additional areas that might apply to the Spacelab program. These areas included Space Shuttle, Spacelab, and proposed Spacelab experiments.

1.4.2.1 Space Shuttle

Space Shuttle documents were reviewed to evaluate development plans and management control aspects of the Shuttle system for their applicability to the Spacelab program. Management concepts, techniques, and tools of the Space Shuttle have been embodied within this report.

1.4.2.2 Spacelab

Existing Spacelab documents were reviewed to acquire a better understanding of the intended Spacelab application. These documents include the Computer Software Development Plan and the Phase B-2 reports. These provide an engineering overview of the Spacelab system along with the required control functions.

1.4.2.3 Experiment

Experiment sizing and data flow were studied to reassure that experiment performance would not be degraded by any concepts, techniques, or tools considered in this study. Areas reviewed included the high data rates of specific experiments, quantities of data gathered by specific experiments, general types of data analysis required, general experiment support, and payload descriptions.

1.4.3 Innovation

The review of existing trade journals and technical publications formed an active part of this study. New techniques were reviewed for applicability to Spacelab.

Research in both the commercial and military programming activities by the IBM Corporation has resulted in new techniques that have been successful in improving quality and reducing costs of programming products. These techniques have been reviewed and integrated where applicable.

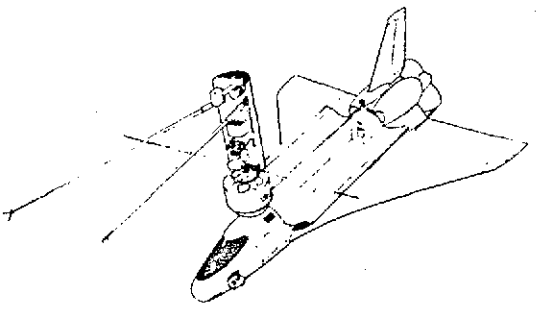
1.5 DEFINITION OF SOFTWARE TERMS

This section defines software terms that are subject to misinterpretation and defines them as they are being used in this document.

- Onboard - Onboard software includes all software executed onboard the Spacelab computer or computers.
- Ground - All support software not used onboard Spacelab.
- Operating System - The operating system is a continuously executing set of programs that manages interfaces, supports applications, and provides overall data management.
- Applications - Programs written to control, test, and support Spacelab experiments and subsystems.
- High Order Language (HOL) - A programming language that provides computer instruction in a pseudo-English environment. A HOL supports programming techniques at a level that is transparent to the programmer. Examples of High Order Languages reviewed were Houston Aerospace Language (HAL) and FORTRAN.
- User Language - An engineering oriented language designed to facilitate Spacelab checkout. Examples of user languages are the Saturn Acceptance Test or Launch Language (ATOLL), and the Ground Operational Aerospace Language (GOAL).
- Verification - Verification testing ensures that the product performs to specification.
- Validation - Validation testing ensures that the Spacelab software system meets the functional requirements.
- Integration - Integration is the process of: 1) verifying that software interfaces perform to specifications as the modules are merged into a programming system, and 2) verifying that the software interfaces with the Spacelab hardware and performs to specifications.
- Modules - A program or program element which can be defined, developed, and to some extent, tested independently of the remaining system parts.

CONCEPTS AND PHILOSOPHY

2



Paragraph	Title	Page
2.1	PROGRAM MANAGEMENT PHILOSOPHY	2.3
2.2	THE SOFTWARE DEFINITION PROCESS	2.4
2.2.1	Definition	2.4
2.2.2	Program Design	2.6
2.2.2.1	Introduction	2.6
2.2.3	Program Implementation	2.8
2.2.4	Program Verification	2.9
2.2.5	Validation	2.9
2.2.6	Acceptance and Delivery	2.10
2.3	OPERATION	2.12

SECTION 2

CONCEPTS AND PHILOSOPHY

Software development embodies management and technical concepts which work in concert to produce cost effective, useful software systems. The application of compatible concepts results in an integrated development process. Integrated development is based on the assumption that design, implementation, verification, and validation can occur simultaneously during the software development phase. It also assumes that documentation and control are active participants in this development phase. It is the purpose of this section to describe some of the advantages of integrated development that best support the Spacelab software development process.

Through interlocking techniques, as shown in Figure 2.0-1, the activities of definition, design, implementation, and validation are serially related, while the verification and documentation activities are depicted to represent the fact that these are integrated with and span the definition, design, and implementation activities. This interlocking approach creates a new emphasis on aspects of software development that has not previously been present:

- Verification is considered before system code is complete.
- Documentation spans all areas of software activity.
- Management and control become useful functions instead of necessary requirements to meet contract specifications.

Advantages realized from an Integrated Development Approach applied specifically to the Spacelab development phase are:

- Milestones can be set which notify management of early problem areas allowing appropriate action to be taken at a time when needed.
- Ambiguities and inconsistencies are identified before complete development, thus reducing the possibility of major modification during system integration.
- Documentation is an integral part of the development phase thus providing training and familiarization information long before system delivery.

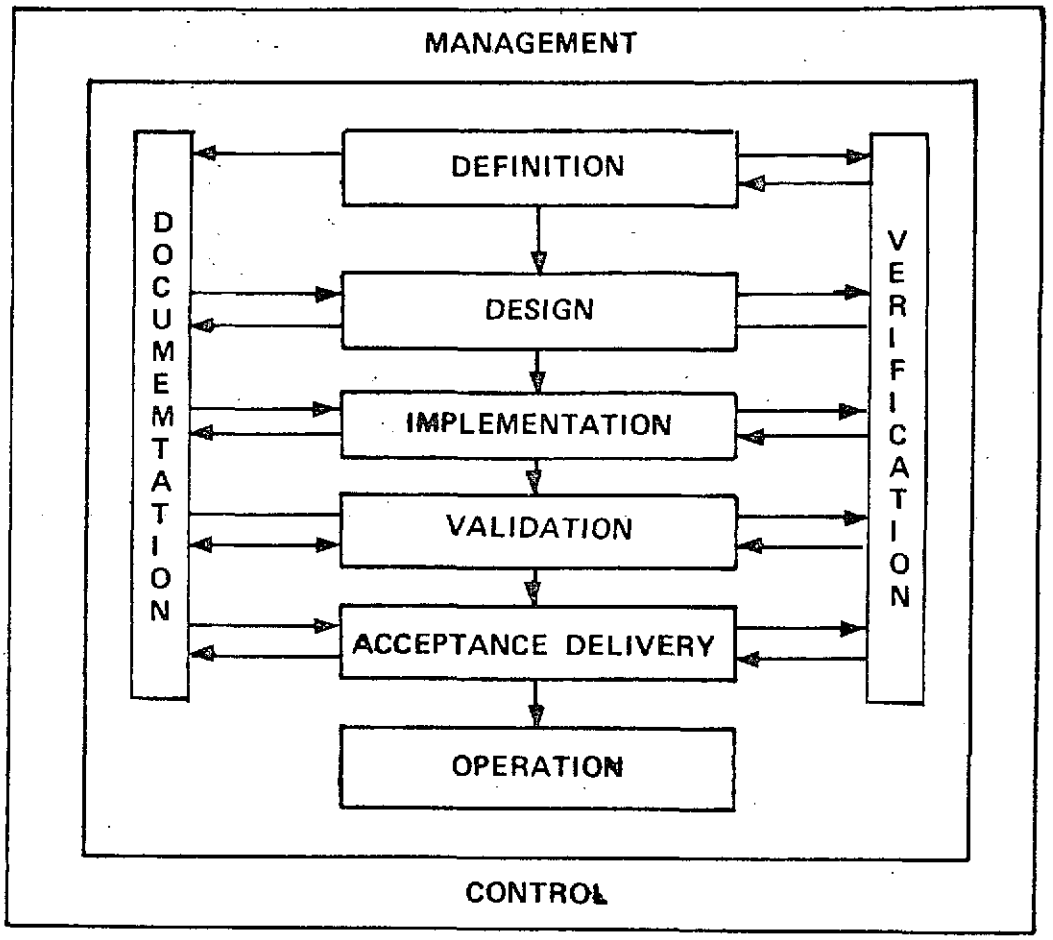


Figure 2.0-1 Integrated Software Development Activity

2.1 PROGRAM MANAGEMENT PHILOSOPHY

The collection of principles which span the software development activities is referred to as the program management philosophy. These principles run as common themes throughout the various methods which are described in this document as a set of guidelines for software development.

The primary objective of the program management philosophy is to guide the application of the various techniques for software development so that a set of principles is embodied in the effort and reflected in the end products.

One of the most important principles is concerned with the purpose of the development methodologies. It arises from recognition that it is not enough for the requirements of the problem to be addressed; but that the ability to validate the program against the requirements must be aided to the greatest extent possible.

The primary objective of the development methodologies is to provide a usable, functional end product.

Other important objectives are to ensure that the program system can be enhanced and modified without difficulty and to increase the comprehensibility of the program system at all levels of abstraction.

There should be common and equal management visibility over both the hardware and the software development activities. This is justified by the rising cost of software and the declining cost of hardware. The project leaders for software and hardware should be at an appropriate level. An attitude of mutual support and accommodation should prevail between the hardware and software groups. Problems should be conveyed across the hardware/software interface to determine if the alternate discipline can assist. Management visibility should be established and employed to ensure that this close working relationship begins at the start of the system definition activity and continues throughout the system design, implementation, validation, and acceptance activities.

To the maximum extent possible, the working papers, or byproducts, of the development process should serve as documentation. This can be accomplished by the judicious choice of methods and provides the advantage of integrating the documentation effort throughout the other activities. Lucid results are thereby made available at numerous intermediate points in the schedule for software development, from the definition phase, through the design and implementation phases. Use of byproducts as documentation avoids the nemesis of documentation which is prepared late in the schedule for the ostensible purpose of providing following efforts with insight into the characteristics of the system. It is this late documentation which inevitably suffers when schedules are revised. In addition, documentation prepared after the development phase is not available to support the program reviews which must occur throughout the schedule, requiring special purpose review documents to be developed instead.

2.2 THE SOFTWARE DEVELOPMENT PROCESS

The software development process consists of distinct, but interrelated, activities. These activities are distinct because we can assign and discuss methods of performance for each. They are interrelated by their mutual support and by their integrated nature which is evident, once the software development process begins. The definition activity is where the problem is described in its most abstract form. The design activity is where an approach is used to synthesize a system which will meet the specifications of the definition phase. The verification activity is where each level of abstraction of the evolving system solution is reviewed for conformance to the specifications for its appropriate level. The implementation activity is where coding and testing is done for the modules comprising the system whose design has been completed and verified. The validation activity is where the coded modules are analyzed, exercised, and evaluated with respect to their ability to function exactly as required. The acceptance and delivery activity assures that the system is in acceptable accord with the requirements and that the working system is delivered at the specified change level. Permeating the entire development process are the control and management functions. All of these activities culminate in an operational system.

2.2.1 Program Definition

In the definition phase it is determined what the software is to do. This activity is a refinement of the statement of work, for the purpose of thoroughly defining the problem before a solution is begun. Emphasis in this phase should be on what the problem is, as opposed to how it is to be solved. Although consideration of some design concepts is necessary, the urge to get on with the job and enter the design phase prematurely should be resisted.

A requirements specification document should be the product of the definition phase. The preliminary document is reviewed, analyzed, and revised to make the final requirements specification which is used as input to the program design phase. In the requirements specification, the problem should be described in detail, in non-technical language. The users should be identified. The problem environment should be described. Subsequently, the technical problem should be described in increasing levels of detail. The capabilities to be included in the system should be specifically stated. Where helpful in clarifying the requirements specification, capabilities which are not to be included should be detailed.

It is during the definition phase that verification should start. Each level of the technical definition of the problem should be verified to determine its ability to meet the requirements of the level immediately above. This parallel verification process is concerned with all aspects of software development, not just the implementation phase as previously practiced.

Software and hardware should be configured and changes controlled in a common manner. This means that the relationships among software modules making up the system must be defined and maintained at all times during the development effort, just as the analogous relationships are maintained for hardware. In addition, the record of changes to each module should be maintained and, when a delivery of the system is made to support a given mission or set of missions, the change level of each module should be accurately specified and audited.

Change activity for both hardware and software should be managed by change control boards whose members have the technical expertise and management responsibility to assess the impact of proposed changes to the function of the system and conduct trades between software and hardware implementation of the changes. In no case should a change to hardware or software be made without an assessment of whether the alternate discipline should be affected.

It is also during the definition phase that the tools which begin to establish test cases should be applied. These tools support the verification activity by providing test cases for use in the review of technical problem definition. The technique is to convene a review board of persons who will manually carry out the actions of the system as it is stimulated by each test case and examine the requirements specification to verify that it is technically sound.

Using the tools and methods for test case generation, the requirements specification is examined for observable effects and the causes which, when invoked, will cause these effects. Normally, the cause and effect relationships will be simple, often involving the logical functions AND, OR NOT, NAND, and NOR in addition to some direct relationships. When these relationships are displayed in graphic form, with causes as nodes on one side, effects as nodes on the other, and the logical connectives indicated, a picture begins to emerge as to the quality of the requirements specification. This is an application of Boolean algebra to the very areas where the English language and other conversational languages are inadequate - the area of vigorous definition of logical relationships which require graphics and precision.

It should be possible to spot ambiguities, causes without effects, effects without causes, and inconsistencies in the requirements specification. Therefore, a beneficial review can be accomplished which results in improvement in the requirements specification. This benefit, in addition to contributing much needed clarity for management review purposes, passes on to the design phase a better quality document for use as a baseline. The process also spots problems which might remain hidden and become troublesome much later in the software development cycle.

A beneficial effect of using the tools and methods of test case generation during the definition phase is to involve the designers in consideration of the requirements of the validation phase. Members of the design team should be used as validators.

It must be realized that misunderstandings concerning the problem to be solved will arise between contractors and customers, between different organizations within the same company. In most such cases, the root of the misunderstanding can be traced to a lack of written definition of the problem. The strict requirement of an organized, written Spacelab software specifications will reduce this misunderstanding.

2.2.2 Program Design

2.2.2.1 Introduction

The primary objective of the program design process is to specifically avoid the all-too-common practice of programming a system directly from a definition of the problem, without first translating the requirements specification into a sound program design.

The product of the design phase is the software design specification which defines the functions and interrelationships of the program modules which comprise the system. Intermediate outputs of the design phase are produced as byproducts of the design effort. The purpose of the intermediate outputs is to convey design ideas as they evolve, for use in conducting management reviews and verification.

The process of verification, which began in the definition phase, continues in the design phase. Here, it is a process of feedback, criticism, itteration, and testing of assumptions. It proceeds, as does the design effort, from the top down; each successive level being verified with regard to its ability to meet the requirements of the higher level.

2.2.2.2 Composite Design

Composite design is a method of program design which provides criteria for decomposing a system into modules. It is oriented toward reducing the difficulty and cost associated with creating and maintaining large programming systems. It provides substantial benefits to the validation phase, as well as the design and implementation phases because it tends to define modules which are highly independent of one another.

Composite design examines the function performed by each module. Function is defined to be the transformation performed by the module on its inputs, in order to produce its output. Any module's function is related to the function performed by its subordinate modules as well as its internal operation.

In two complementary ways, the program design is structured so that modules are highly indpendent. By focusing on the relationships among internal parts of a module, composite design seeks to maximize the strength of individual modules. By examining the ways in which modules reference data and the nature of that data, composite design seeks to minimize the coupling between modules. One and only one complete function is performed by each module. All input and output data is passed as parameters when this design method is followed to the fullest extent.

In addition to providing design methodology, measures of how well the design is progressing can be defined. On an ascending scale of strength, different module strengths are rated. For module coupling, a descending scale is available. These measures provide management with tools with which to evaluate an emerging design. They also provide viable alternatives when, during implementation, system constraints force the use of a less than optimum design. Instead of reverting to an arbitrary division of the system into modules, or an ad hoc methodology for managing data, the design can be modified in specific areas to meet the constraints. In these specific areas, the design method should be to move one position on the module coupling or module strength scale, rather than revert to less formal methodologies for the entire system.

Byproducts of composite design lend themselves well to documentation of the system and review material for intermediate milestones. There are program structure diagrams showing the function of a module and the relationship to its immediate subordinates. There are input/output tables showing parameter names and the type of module coupling exhibited by each module interconnection. Since composite design is a top-down process, meaningful design reviews may be held at any time during the design phase. During such reviews, the design will be more lucid, and potential problems can be detected earlier than might otherwise be the case.

Reliability must be designed into a system; it cannot be tested into a system. That is why it is important to have design methodology which has a positive effect on the ability to comprehend and modify the system. System reliability requires that each module be reliable and the most thorough way to test the system is to exercise it through all its possibilities. The problems with such a method have been in identifying what the possibilities are, and in the sheer size of the aggregate. While composite design may not reduce the total number of possibilities to a practical level, it does have a positive effect on identifying the possibilities of a single module and reducing the interactions with other modules of the system. In a correct design, modules are less complex, and thus validation is easier and more straightforward.

The reduction in complexity also increases the likelihood that the system can be easily understood, from the top, through all lower levels of abstraction. The increased independence among modules reduces the implementation costs by reducing the interactions among programmers. When a change is necessary to one function of the system, composite design increases the probability that only one module will need to be changed. At worst, the change should affect only one part of the system and the impact should be more easily assessed than in an arbitrarily modularized system, where any change is apt to ripple throughout the system in ways that are difficult to detect and validate.

2.2.3 Program Implementation

The objective of the implementation phase is the transformation of the design specification into verified code.

The output of the design phase is a detailed design specification that has been verified against the requirements. The specification includes both coding and test procedures. Low cost implementation of reliable software requires that the programmer convert these specifications into code without further design. Implementation proceeds along two parallel paths - code and test.

These paths join during the verification of the code using the test procedures. The outputs of the implementation phase are updated design specifications, verified code and test cases used in the verification. The updated design specification including the organization of the code, a source listing, and the implementation limitations becomes the documentation of the computer program.

Costs can be reduced by using tools such as higher order languages, standard subroutines, test generation programs and automated programming aids.

Coding is the process of converting the design into a machine readable form. This code normally takes the form of assemble language instructions or higher order language statements. The organization of the code should agree with the design specification. The code is divided into a tree like structure using the techniques of composite design that were employed in the design phase. This technique of separating tasks into independent modules that perform a function, is continued by using the concepts of structured code that permits definition of each function in terms of basic coding elements. Code segments are kept small by using subroutines. Subroutines may be those previously developed, or may be generated as part of implementing the module.

Testing is planned just as coding. Test cases are generated that exercise all paths of the module. Probable exception conditions are defined and test cases generated. If exceptional conditions exist that cannot be tested, then this fact is reflected in the design specification as a limitation on the implementation.

Verification tests the revised design specification against the code. Verification is complete when the revised specification is reviewed and approved.

2.2.4 Program Verification

This activity of the software development process is discussed here simply for completeness. It is not a separate activity which is in line with definition, design, and implementation. Instead, it is an integrated activity within the development process.

Verification began in the definition phase where, as each level of technical definition emerged, it was examined for compatibility with the next higher level of problem definition. It continues throughout the design phase as the top-level design is verified with respect to its ability to solve the problem defined in the requirements specification. In the implementation phase, the internal algorithm for implementing each module is verified against the function and data interface requirements developed in the design phase. Thus, verification is a continuing, iterative activity which is integrated throughout other phases of the software development cycle from definition through implementation.

Since verification begins long before the implementation phase, the methods involved do not rely on the existence of written program statements. The principle technique uses the test cases identified by a cause and effect analysis of the design area to be verified. These cases are manually walked through to determine if the requirements of the higher design level are met.

2.2.5 Validation

System validation demonstrates that the software and hardware interface as a system, providing confidence in the system's ability to perform its intended function.

The goal of validation is to develop a high quality cost effective programming product on schedule. To accomplish this objective, testing should be performed as early as possible in the development cycles. Problems found early in the development phase permit flexible response where problems found late may cause major impacts in the completed program structure.

The validation team should consist of system designers, programmers, and systems engineers with expertise in the software design, the software interfaces and a knowledge of the hardware interfaces that are utilized by the software. This validation team is independent of the implementation group and assures that the system meets design requirements and user requirements. Experience gained in the development of large complex systems has proved the value of independent software testing to ensure the high quality required for space oriented programming systems.

The validation team must work closely with the programmers thereby developing a working relationship which will permit validation to maximize the results of program development testing which is normally done by the programmer independently. During the software implementation, programmers will develop and execute test plans for each module. The validation team will audit the test plans and provide feedback to the programmer to assure that the testing will meet all requirements. This permits the validation team to get involved and exert influence during the early development phases of the project. Additionally this approach assures formalization of program test plans and reduces the duplication of testing efforts between the validators and programmers.

The top-down programming approach being followed by the programmers will permit initiation of validation tests early in the development phase to assure the quality of those interfaces and functions which have already been developed without waiting for the entire system to be completed. This approach maximizes the benefits which can be attained from early systems testing. This would include identification of system problems early enough in the development phase to permit a flexible response without impact to project schedules.

When the development is completed a final set of testing will be performed to ensure the completed integrated system performs to design requirements on the actual hardware. These tests will be developed and the segments verified as the system is being tested during development.

2.2.6 Acceptance and Delivery

The acceptance and delivery activity defines the completion of the development process and the start of the operational phase. It is imperative that all parties agree to the components of this activity to assure a smooth transition from ESRO responsibility to NASA assumption of the developed products. This activity consists of the end item acceptance testing; verification of the consistency of end item products, including actual programs and supporting documentation; and initiation of configuration controls.

Software does not lend itself to the usual means of verifying end item configuration. Inspection of the physical computer programs is inadequate to prove either function or consistency of components. Therefore, it is imperative that acceptance testing within an operational or simulated operational environment is performed and controlled under a frozen baseline configuration. This testing effort is defined by a set of acceptance test procedures written to prove functional as well as performance requirements. All errors are formally documented and placed under configuration control.

Supporting the actual testing of the system is an analysis effort to prove that the computer programs in their object form (as they operate in the computer) agree with the program source form (as they are coded by the programmer) and the supporting documentation. The result of this activity becomes the baseline configuration for the operational phase. It is from this activity that the delivery is accomplished and baseline accountability is established.

To effect adequate configuration control of the contract end items a configuration control system must be established prior to the start of the acceptance and delivery activity. This system is comprised of a configuration control board, a set of procedures that determine contract end item identifications to assure consistent end item levels between programs and documentation, and program change and deviation accountability. The configuration control system must accommodate multiple site usage of the end items.

Identification of contract end items and end item components must be established early in the development process. Contract end item types would consist of operational programs, simulators, development support systems, assemblers, compilers, debug aids, and other developed software that would be required during the operational phase. End item components would probably comprise computer programs in their object and source forms, computer program listings and flow charts, requirements specifications, design specifications, user documentation, verification/validation/acceptance plans and results, and configuration accounting documents.

2.3 OPERATION

Due to the long term nature of the Spacelab program, operational software costs will exceed development costs; however, the extent of the operational cost will be directly related to the attributes of the developed operational system and its supporting software tools. Imposition of rigid disciplines at the inception of a programming project will allow the development of functional, reliable, and easily maintained systems at lower cost than systems less amenable to support over a long life span.

Changing requirements ordinarily are reflected in the evolution of software. The economical way to approach this fact is to manage-to-change and properly prepare those who will make the changes. NASA should begin an intensive training program early in the Spacelab project to equip programmers with the knowledge of the deliverable systems that will allow a smooth transition of the maintenance responsibility. The training process can be shortened and enriched with adoption of the design and documentation techniques discussed in this report.

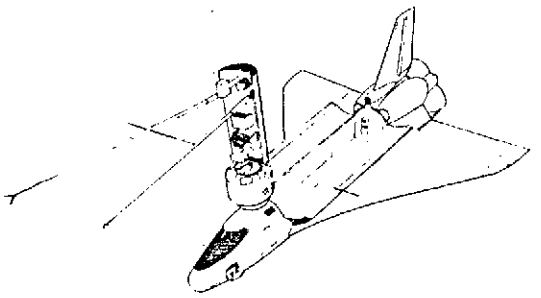
Integration of the experiments on Spacelab will require the development of software to monitor and control these experiments. The principle investigators should not be expected to be programming experts, instead they should be provided a system which is easy to use and requires minimal training. If the design of the operational system is conceived as a tool of the end user, it will contain subroutines, macros, and utilities. These can be invoked through simple procedures to aid the experimentor and crew in developing the automated applications to assist them in gaining their objectives.

Development of the experiment software applications will fall almost entirely within NASA's responsibility. It is imperative that a user language be defined early in the Spacelab development process, because the user language will not only impact, but depend on the design of the deliverable system.

Training of the men and women who will fly with Spacelab is an activity that will begin before delivery of the Engineering Model, and continue throughout the program. The crew members will not only have to learn how to survive in a space environment, but also have to gain a knowledge of the Spacelab systems and the experiments they will activate. In the current Skylab missions, a major complaint of the astronauts has been the inability to interface with the computer systems controlling the experiments. The onboard systems should provide an easily learned, interactive capability which would allow the crew to invoke predefined procedures or to develop new procedures during a mission. This interactive control and display language should be syntactically compatible with the user language. Apollo and Skylab have proven that man, given the tools, can correct or circumvent failures that occur on space missions. Spacelab should capitalize on this knowledge.

TECHNIQUES

3

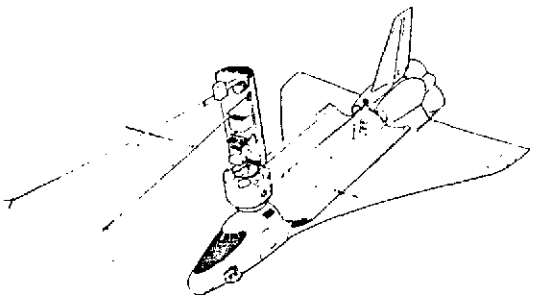


Paragraph	Title	Page
3.1	TOP-DOWN DEVELOPMENT	3-2
3.1.1	Theme	3-2
3.1.2	Conclusions	3-2
3.1.3	Description	3-3
3.1.4	Study Data	3-7
3.2	COMPOSITE DESIGN	3-8
3.2.1	Theme	3-8
3.2.2	Conclusions	3-8
3.2.3	Description	3-9
3.2.4	Study Data	3-11
3.3	STRUCTURED CODE	3-12
3.3.1	Theme	3-12
3.3.2	Conclusions	3-12
3.3.3	Description	3-13
3.3.4	Study Data	3-17
3.4	PROGRAM LIBRARIES	3-19
3.4.1	Theme	3-19
3.4.2	Conclusions	3-19
3.4.3	Description	3-20
3.4.4	Study Data	3-21
3.5	STRUCTURED DOCUMENTATION	3-22
3.5.1	Theme	3-22
3.5.2	Conclusions	3-22
3.5.3	Description	3-23
3.5.4	Study Data	3-25
3.6	CAUSE AND EFFECT ANALYSIS	3-28
3.6.1	Theme	3-28
3.6.2	Conclusions	3-28
3.6.3	Description	3-29
3.6.4	Study Data	3-31

TECHNIQUES

3

(Continued)



Paragraph	Title	Page
3.7	SIMULATION	3-32
3.7.1	Theme	3-32
3.7.2	Conclusions	3-32
3.7.3	Description	3-33
3.7.4	Study Data	3-35
3.8	AUTOMATED MANAGEMENT SYSTEM	3-38
3.8.1	Theme	3-38
3.8.2	Conclusions	3-38
3.8.3	Description	3-39
3.8.4	Study Data	3-40
3.9	SOFTWARE MANAGEMENT BOARDS	3-42
3.9.1	Theme	3-42
3.9.2	Conclusions	3-42
3.9.3	Description	3-43
3.9.4	Study Data	3-43

SECTION 3

TECHNIQUES

State-of-the-art programming techniques have reached the point that formal descriptions have been developed. Many of these techniques have recieved wide acceptance in principle, and have been applied to programming projects. As a result a firm base for programming theory is beginning to emerge. This theory covers program design, implementation, verification, validation, and management.

This study has identified some of these formal techniques and has integrated them into a consistent theory.

Techniques for design and implementation are:

- Top-down development
- Composite design
- Structured code
- Hierarchical documentation
- Programming libraries

Techniques for verification and validation are:

- Cause and effect analysis
- Simulation

Techniques for program management are:

- Automated management systems
- Management review boards

3.1 TOP-DOWN DEVELOPMENT

3.1.1 Theme

The traditional development approach for software has been to design from the top-down and then to implement from the bottom-up. The lowest level software modules are developed first, then the next level, and then these levels integrated. This continues level by level until the entire package has been integrated and tested. The top-down approach applies disciplined system design concepts to software development; and, as a result, design and implementation proceed from the highest level downward. The top-down approach makes software development more systematic and controllable.

3.1.2 Conclusions

Application of the top-down approach to software development has demonstrated the following advantages:

- Interface problems have been significantly reduced.
- A uniquely planned software integration task is eliminated.
- Overall software system compatibility is maintained.
- A functional software system becomes available in stages.

Spacelab benefits most from the last of these. Because functional software becomes available in stages, ESRO will be able to deliver operational software to NASA early in the development cycle. NASA can use these early deliveries for design validation and operational training. This will reduce the time required to transfer the software from ESRO to NASA and relieve schedule pressure.

3.1.3 Description

The top-down development technique is the application of the natural system of design approach. This technique requires the software control architecture and interfaces be established and developed first and succeeding levels of detailed logic implemented in a downward fashion. Top-down development provides an ordering of development which allows for continual software integration of system components and provides for well-defined interfaces at each level.

Although software design has always followed a top-down approach, the actual implementation has been conducted in an opposite direction.

Traditional software implementation has evolved as a bottom-up procedure where the lowest level processing programs are coded first, unit tested, and made ready for integration (see Figure 3.1-1). Driver programs are needed to perform the unit testing and lower levels of integration testing. Data definitions and interfaces tend to be simultaneously defined by more than one person and often are inconsistent. During integration, definition problems are recognized; however, integration is delayed while the data definitions and interfaces are correctly defined and the programs are reworked, and unit tested again, to accommodate the changes. It is often difficult to isolate a problem during the traditional integration cycle because of the large number of possible sources. Management control often is ineffective because there is no coherent, visible product until integration. Modules that have been coded, debugged and verified become obsolete and must be modified and reverified.

In top-down development, the system is organized into a tree structure of segments. The top segment contains the highest level of control logic and decisions within the program and either passes control to lower level segments, or identifies lower level segments for in-line inclusion. The process continues for as many levels as required until all functions within a system are defined in executable code.

Many system interfaces may occur either through data base definition or calling sequence parameters. The top-down approach requires the data base definition statements be coded and actual data records be generated before exercising any segment which references them.

The top-down approach provides the capability of evolving the product in a manner which maintains the characteristic of being always operable, modular, and always available for the successive levels of testing that accompany the corresponding levels of implementation. The quality of a system produced using this approach is increased, as reflected in fewer errors in the module integration process. Inconsistencies are eliminated, and lower level segments can be generated by referencing implemented code.

The top-down approach introduces a significantly improved capability for management control of the software development effort by providing continuous product visibility. Since the developing system is undergoing continuous integration, the system status is accurately reflected through the contents of the system library; therefore, completeness is measured objectively in terms of how much of the system is operational. Managers can review the completed code to verify status and appraise the quality of the software product.

The top-down approach alters or eliminates some of the traditional milestones usually associated with the program production process (see Figure 3.1-2). Probably the most obvious milestone elimination is the disappearance of an identified software integration period. It is no longer required since the system parts are continually being integrated as development proceeds.

Another area affected is documentation. In the past, given a set of funtional system specifications, system design proceeded down until a complete set of detailed program specifications was written prior to coding. Using the top-down approach, the various levels of documentation describe the developing system, thereby reducing inconsistencies between programs and their documentation. Design can be verified and validated by levels both as a document and as a program.

Conceptually, top-down implementation proceeds from a single starting point while conventional implementation proceeds from as many starting points as modules in the design. The single starting point does not imply that the implementation must proceed down the hierarchy in parallel. Some branches intentionally will be developed earlier than other branches. For example, user or other external interfaces might be developed to permit early training or hardware/software integration.

In systems with user interfaces, the user can interact with the system much earlier in the development phase. This early interaction provides an opportunity to prepare user and operator guides top-down as user facilities are developed to validate the guides.

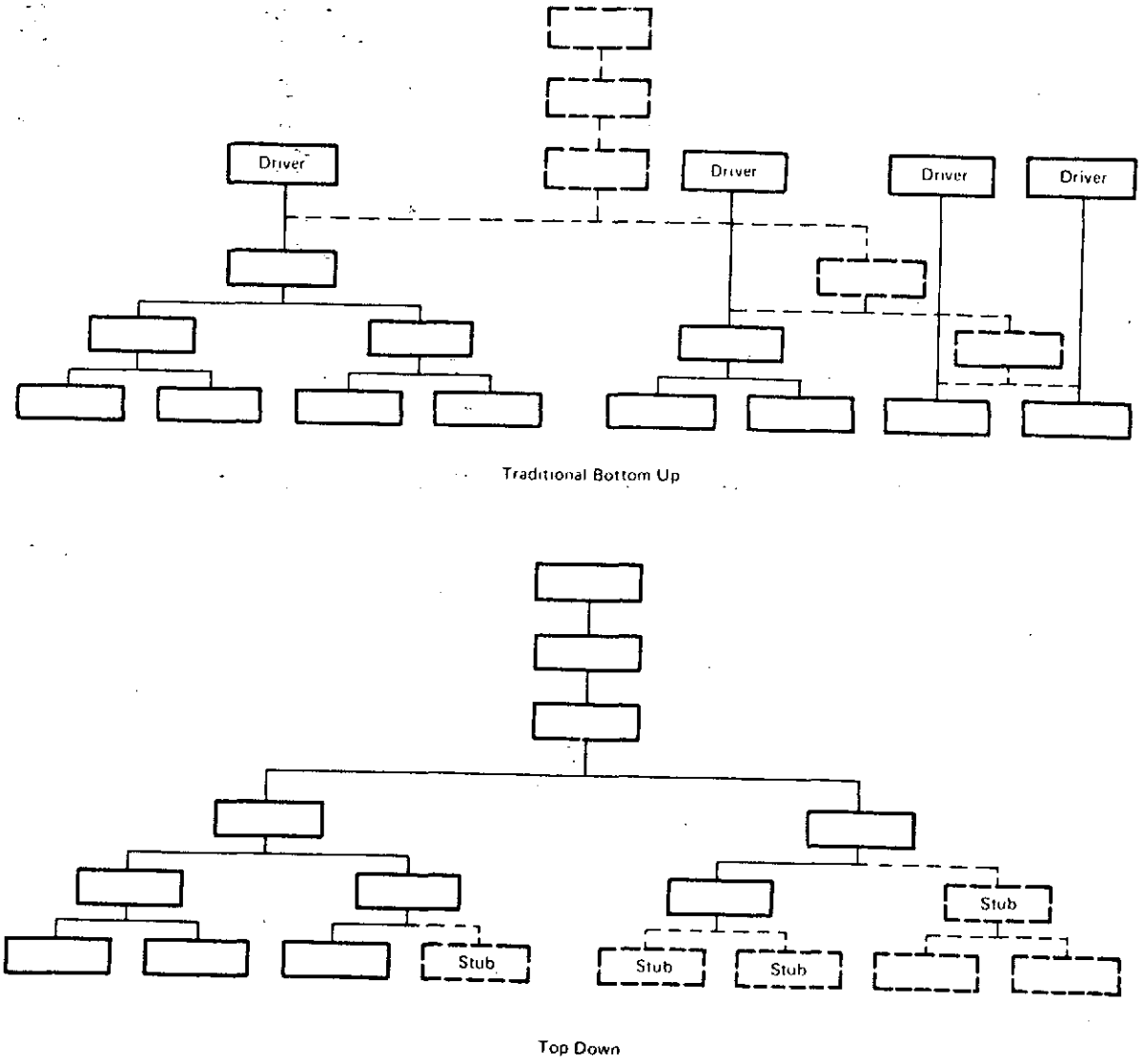


Figure 3.1-1 Software Development Approach Comparison

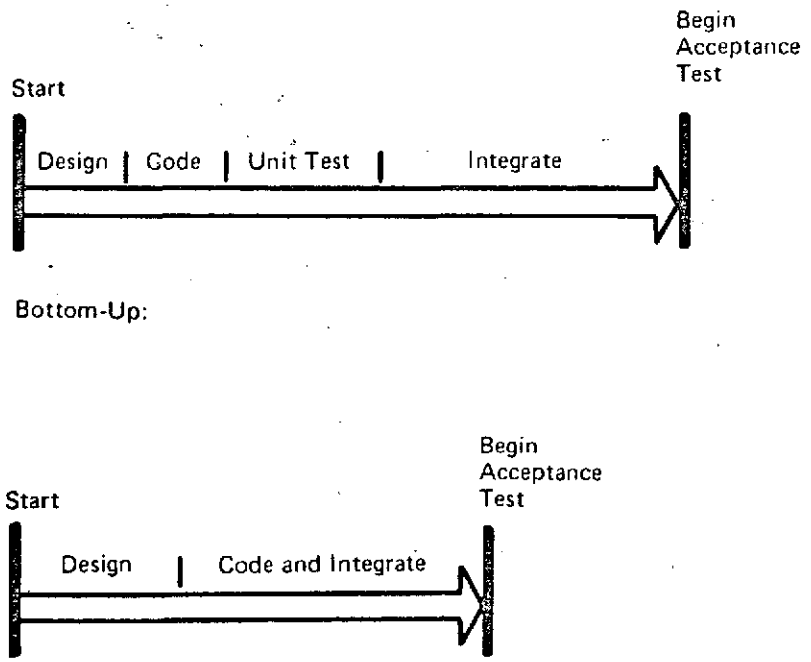


Figure 3.1-2 Software Milestone Comparison

3.1.4 Study Data

Top-down has been recognized as the way to do system design for both hardware and software.

The use of these concepts for software implementation has evolved in recent years. However, the fact that systems could be implemented and used before all components are designed is employed frequently by making intermediate deliveries of systems.

Top-down implementation has been recommended for use in Shuttle software development and Shuttle payload software.

A formal top-down integration approach was utilized during initial development of the Skylab flight program. A skeleton nucleus was created which contained the executive program and major subroutines, and the remaining application subsystem modules were added in a systematic manner until all requirements had been satisfied. Utilization of this approach provided an orderly approach to program integration and provided a thorough test of system design concepts during early program development.

In the development phase, three distinct integration steps were made between baseline of requirements and delivery of the initial program. The resulting program at completion of each step was utilized as the development baseline for the following step.

The first baseline was delivered at six months from start of design, the second at eight months, and the final flight version at eleven months. The final program used the full 16K word memory of the Skylab computer.

3.2 COMPOSITE DESIGN

3.2.1 Theme

Composite design is design methodology that consists of a strategy, rules, and measures to produce modular computer programs. Specifications are composed of independent, functionally organized modules.

3.2.2 Conclusions

Composite design has a positive effect on the cost of developing and maintaining a program because:

- Composite design creates programs composed of many relatively independent parts, thus reducing interactions and dependencies. Therefore, programmer productivity in implementing a program is higher.
- Design changes are easier because they normally affect only one part of the program.
- The design of the program is highly visible and easily understood. This factor has a positive effect on program maintenance and modification.
- Testing of the program can proceed in a straightforward sequence of steps. Designs produced using this technique are excellent candidates for top-down structured programming implementation.
- Validation can be done by function, producing high quality programs.

The functional organization into independent modules will permit the software to be maintained, modified, and expanded over the operational life of the Spacelab. This is a key item in the control of operation cost of software.

The ease of validation will reduce the verification cost that normally represents 40 percent of flight software costs.

3.2.3 Description

Composite design is a tool used for defining the modular structure of a computer program. Its use requires that a new activity be included in the software development process at some point after system or program definition but prior to the coding and testing activities. This often-ignored process might be called the internal structural design of the program. Much of today's software is produced as a result of a direct translation from a flowchart to a coding sheet. The problem involved in maintaining and modifying software produced in this way are well known. Proper application of the principles of composite design can improve this situation by producing systems and programs with an inherently simple design which is easily understood and therefore easily maintained and modified.

Classically, a good modular structure for a program is defined as one in which the interrelationships of the modules which make up the program are minimized and the internal strength (dedication of elements of the module to performing a single function) are maximized. Composite design defines measures for evaluating both the interdependence of modules and the internal strength of a module. These measures are used concurrently in evaluating the modular design of a program. Good design is refined until the measure is acceptable.

Programs designed using this concept are composed of modules. A module is a group of program statements which have the following characteristics:

- The statements are lexically together.
- The statements are bounded by identifiable boundaries.
- The statements can be collectively referenced by a name from any other part of the program.

Composite design provides the software designer two sets of measures for evaluating the modular structure of a program. The first and most important of these measures is modular strength. Modular strength is a measure of how tightly the internal elements (statements, code segments) of a module are bound together. Composite design recognizes six types of binding and associates a weight with each type on a relative scale. The optimally bound module is one in which all elements within the module are an integral part of, or essential to, the performance of a single function. This type of module is to produce program structure which is composed of modules which exhibit functional binding.

The second set of measures of program modularity is coupling. Coupling is a measure of the relationship between program modules. Modules that exhibit low coupling tend to be independent of the program structure in which they are used. Composite design recognizes five different types of coupling; the lowest, and therefore best, is data coupling. A module which is data coupled to the environment in which it resides communicates with other modules by accepting input and returning output in the form of parametric data.

Composite design produces a flexible, step-by-step strategy for developing the modular structure of a program from the basic structure of the problem to the solution. This top-down process, entitled composite analysis, channels the designer to think of a problem as being composed of several smaller problems in which a portion of the software solution can be allocated. Composite analysis is an interactive process of continually reviewing the emerging design to produce independent program modules with clearly defined interfaces. The results of the process are a graphic representation of the program's modular structure.

3.2.4 Study Data

The terms and methods that are called composite design have been adopted from a paper by G. J. Meyers.

The techniques of composite design have been recognized and used extensively in software projects for years. Composite design has identified and named the techniques that have been used. The best examples of these techniques are arithmetic subroutines available in most computer installations (all parameters are passed through calling sequences and perform a single function).

Modern operating systems have extensive library management systems to handle individually developed modules. These systems evolved because most users recognized the value of modular programming systems.

The experience of the Saturn flight program can be cited as an application similar to Spacelab. The early Saturn flight program was not organized as functional modules. This was identified as a problem and the programs were redesigned as functional modules. Verification cost savings were dramatic--390 hours of computer time per flight before, 175 hours after. Errors due to programmer interaction dropped from 26 percent to 12 percent of the errors reported during debug and verification. Redesigned into a modular system increased core utilization only one percent--from 86 to 87 percent of the total available. Program change implementation time was reduced by 30 percent--a critical factor in the Spacelab operational environment.

3.3 STRUCTURED CODE

3.3.1 Theme

In concept, structured coding is an extension of the approach utilized in hardware design and development, of constructing complex circuits from elementary AND, OR, and NOT gates. The software analogies to the hardware components are:

- Sequence of two operations.
- Conditional branch to one of two operations and return.
- Repeating an operation while some condition is true.

Structured coding is the technique of coding a program such that all logical functions to be performed are comprised of these three basic software structures.

3.3.2 Conclusions

Application of the structured coding technique to software development has shown the following advantages:

- Readability/maintainability of logic flow.
- Detail design of logic flow prior to initiation of coding.
- Reduction in complexity of software with a corresponding reduction in testing requirements and an increase in overall reliability.
- Increased visibility for communication and audit purposes.
- Eliminate possibility of incorrect branches.

Spacelab, as all space systems, requires reliable software. Structured coding takes one more step in the evolutionary process of eliminating programmer errors, thereby improving reliability.

In addition, the readability of structured code will reduce transfer time and expense when ESRO delivers the software to NASA.

3.3.3 Description

Structured coding is based on the structure theorem which states that any proper program (a program with one entry and exit) is equivalent to a program that contains as logic structures only:

- o Sequences of two or more operations.
- o Conditional branch to one of two operations and return (IF Then Else Statement).
- o Repetition of an operation while a condition is true (Do While).

The physical representations of these logic structures are shown in Figure 3.3-1.

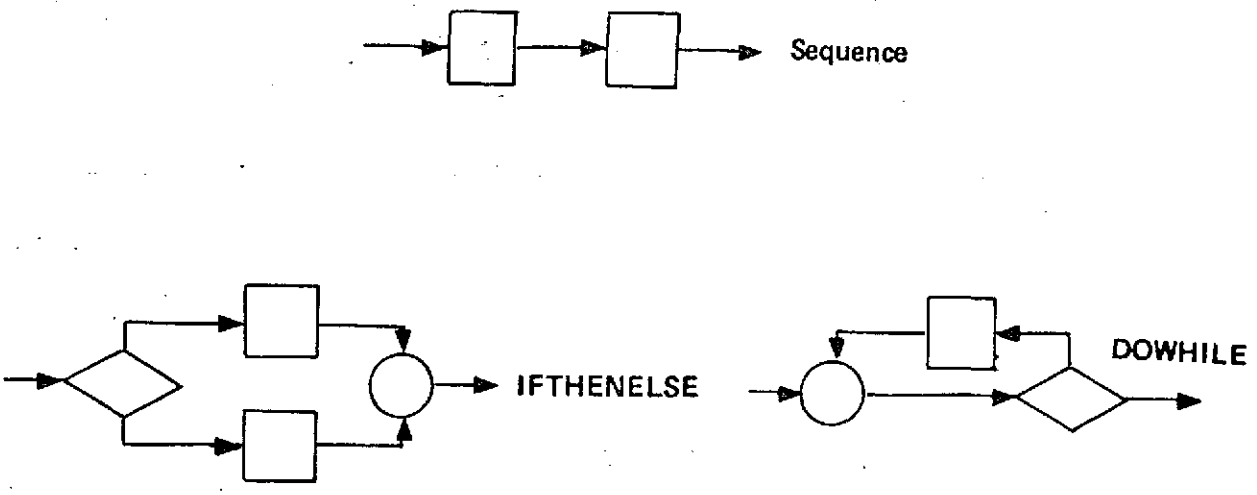


Figure 3.3-1 Logic Structures

Each of the above figures constitutes a proper program and, through combinations of these basic structures, any size program can be developed.

Traditionally, programmers have been unrestricted as to the format of code and organization of logic. The primary objective has been to develop software which will solve the immediate problem at hand with little thought given to readability, understandability, maintenance, transferability, and documentation. The standards established through use of structured coding will address these significant problem areas through the following techniques:

- o Absence of arbitrary "branches" in logic.
- o Formatted coding.
- o Picture-on-page organization.

A major characteristic of programs written in structured code is that they can be literally read from top to bottom typographically; there is never any jumping around as is typical in trying to read code which contains GO TO statements. The property of readability is a major advantage in testing, maintaining, or otherwise referencing code.

Another advantage of structured code, of possibly even greater benefit, is the program design work that is required to produce structured code. The programmer must think through the processing problem, not only writing down everything that needs to be done, but writing it down in such a way that there are no afterthoughts with subsequent jump-outs and jump-backs. He must ensure no indiscriminate use of a section of code from several locations because it just happens to do something at the time of the coding. Instead, the programmer must think through the control logic of the module completely at one time, in order to provide the proper structural framework for the control. This means that programs will be written in a much more uniform way because there is less freedom for arbitrary use of code.

As illustrated in Figure 3.3-2, structured coding is performed according to indentation standards which group logical sequences for readability and which organize the program logic such that the logic flow is significantly easier to comprehend. It may be seen from analysis of the figure that programmer thought is required to arrive at the structured approach; however, it is more obvious that traditional code is significantly more complex and more difficult to read, maintain, or transfer.

Inherent in achieving the readability objective of structured code is the necessity for restricting the physical size (number of statements) which comprise a logical entity. Although a program may be highly structured and readable, it is extremely difficult to comprehend 15 to 20 pages of code without significant analysis. For this reason, the picture-on-a-page technique should be utilized. This technique requires that the programmer utilize a top-down design on his assigned area and subdivide the logic into small stand-alone elements of approximately 50 to 100 lines of code. The first page would then contain the necessary controlling logic for subsequent pages containing lower levels of coding. Through this technique, a programmer or program user can access any level of detail from the highly summarized top-level to the complete detail at lower levels in a systematic manner. This can be recognized as the application of top-down concepts to coding.

An important task in assuring that a program meets its design requirements is the detailed audit procedure. Since this audit is best performed by someone not intimately involved in the program development, readability of the code is of prime importance. One of the advantages of the structured code technique is that it provides a basis for systematic reading of the program. The reading sequence within each segment is strictly from top to bottom. As a result, audit procedures are made more systematic and result in higher reliability of software.

TRADITIONAL

```
        IF p GOTO label q
        IF w GOTO label m
        L function
        GOTO label k
label m M function
        GOTO label k
label q IF q GOTO label t
        A function
        B function
        C function
label r IF NOT r GOTO label s
        D function
        GOTO label r
label s IF s GOTO label f
        E function
label v IF NOT v GOTO label k
        J function
label k K function
        END function
label f F function
        GOTO label v
label t IF t GOTO label a
        A function
        B function
        GOTO label w
label a A function
        B function
        G function
label u IF NOT u GOTO label w
        H function
label w IF NOT t GOTO label y
        I function
label y IF NOT v GOTO label k
        J function
        GOTO label k
```

STRUCTURED

```
IF p THEN
   A function
   B function
   IF q THEN
      IF t THEN
         G function
         DOWHILE u
           H function
         ENDDO
         I function
      (ELSE)
      ENDIF
   ELSE
      C function
      DOWHILE r
         D function
      ENDDO
      IF s THEN
         F function
      ELSE
         E function
      ENDIF
   ENDIF
   IF v THEN
      J function
   (ELSE)
   ENDIF
ELSE
   IF w THEN
      M function
   ELSE
      L function
   ENDIF
ENDIF
K function
```

Figure 3.3-2 Traditional and Structured Control Code

3.3.4 Study Data

Work by H. D. Mills forms the basis for the structured coding concepts of this study. His work was based upon earlier works by Bohm, Jacopini, and Dijkstra.

The principle drawing factor in utilization of structured coding is to increase programmer productivity through significant reduction in coding errors. The measurement parameter utilized for assessing programmer productivity is the number of source statements generated per programmer day or programmer month when spread over the lifetime of software development to cover all associated development activity. As would be expected, this measurement parameter varies widely depending on such factors as complexity of software system, user requirements, programmer experience, change activity, software system application, hardware limitations, etc.

F. T. Baker's work on the New York Times Information Bank System has been the most notable application of structured coding. This system required over 83,000 PL/I source statements. The programmer productivity figures for that application were 65 source statements per day with a minimum of error encountered as shown in Tables 3.3.1 and 3.3.2.

Comparison of programmer productivity among widely varying applications is extremely subjective; however, the results shown indicate that an improvement by a factor of two should be possible over existing coding techniques. The use of structured coding as a software development tool adds a new dimension to programmer productivity.

TABLE 3.3.1 ERRORS IDENTIFIED DURING ACCEPTANCE TESTING

Subsystem	Source Lines	ERROR TYPE: Incorrect Function	Omitted Function	Misinterpreted Function	Total
File Maintenance	12,029	0	0	0	0
Conversational	38,990	9	8	3	20
Data Entry Edit	13,421	0	0	1	1
Other	18,884	0	0	0	0
Total	83,324	9	8	4	21

TABLE 3.3.2 ERRORS IDENTIFIED DURING OPERATION

Subsystem	Source Lines	ERROR TYPE: Incorrect Function	Omitted Function	Misinterpreted Function	Total
File Maintenance	12,029	1	0	1	2
Conversational	38,990	4	3	0	7
Data Entry Edit	13,421	8	5	3	16
Other	18,884	0	0	0	0
Total	83,324	13	8	4	25

New York Times Information Bank System Error History

3.4 PROGRAM LIBRARIES

3.4.1 Theme

The program library provides an automated method of generating, updating, and releasing programming products.

3.4.2 Conclusions

Automated control of the software by programming libraries provides a cost effective method to control software development.

Spacelab will benefit during both the development and operational phases in several ways:

- Product delivery quality is improved.
- Measurable milestones can be established and monitored.
- Product visibility is improved.
- A data base is established for configuration control.

3.4.3 Description

Programming libraries are those classes of data files on mass storage devices (disk or tape) that represent the source and object code that is the programming end product.

Libraries used to assist the programmer have four distinct sections:

- Development Library - The working file of source statements and object decks that are the programmer's evolving product. The programmer can modify this file as required to produce a finished product.

- Verification Library - The file of modules that have been declared complete by the programmer. Modules will be subject to formal verification procedures by the testing organization.

- Qualified Library - The file of completed, tested modules that have met all specifications. These modules can be selected and used as finished products, but have not been submitted to the ultimate user as CPCEI.

- Production Library - The file of programming products that have been qualified for and by the customer. This file includes source statements and object code for individual modules as well as the complete, integrated load modules that represent deliverable CPCEIs.

Automated libraries become a valuable management tool by providing control points for milestone measurement and statistics for performance measurement.

3.4.4 Study Data

Numerous library tools exist. Most operating systems developed by computer manufacturers provide the basic building block. Using these blocks, library management and control systems have been developed.

The IBM Federal Systems Division Programming Production Library (PPL) discussed by F. T. Baker represents another, as does the IBM Federal Systems Division Saturn Software Release System.

3.5 STRUCTURED DOCUMENTATION

3.5.1 Theme

Structured documentation is a strategy for obtaining low cost, usable documentation of programming products. This is accomplished by integrating the concepts of top-down composite design with the documentation process. As a result, documentation by program review and final description become byproducts of the programming process.

3.5.2 Conclusions

Spacelab costs can be reduced by adopting structured documentation techniques that will:

- Produce usable documents on time.
- Reduce the number of documents.
- Reduce the volume of documentation.
- Produce consistent documents.

In addition, software quality will be improved because:

- Programmers will produce programs that agree with the document, not documents that agree with the program.
- Programs will reflect approved requirements.

3.5.3 Description

Documentation should be products of the definition and design activity. Implementation, verification, validation, and delivery should only modify design documentation.

Employing top-down design will require that documentation be developed in a hierarchical, tree-like fashion. High level documents will reflect high level program modules. These high level documents will in turn be explained and/or expanded by lower level descriptions, and the process continued to the lowest module level. This produces documents that can be read and understood at any level. In addition, physically separate documentation required by volume can be done without impacting readability.

When the design of a program module has been documented (at any level), that document is verified against the specification and then is reviewed and approved for implementation. The programmer must then program the document, not document the resulting program. Any modifications from the approved design must be reflected by revised documentation that must be returned for the verify, review, and approval cycle. This includes limitation and/or constraints. The only addition permitted to the design document is the program listing and verification test results.

Usability of documentation is enhanced by using the principle of composite design in determining content. Composite design calls for modules that describe functions. So, documentation should reflect functions to be performed. This is done by describing what are the inputs, what outputs are required, and what transformations occur to translate inputs into outputs. Assisting in this design is IBM HIPO: Design Aid and Documentation Tool. This tool uses visual techniques. They are shown in Figures 3.5-1 and 3.5-2.

Readability can be enhanced by using these visual display techniques. Program operation is described in the form of:

- Visual Table of Contents
- Overview diagrams.
- Detail diagrams.
- Supporting text and annotated data.

Functional documentation proceeds from a top level Visual Table of Contents diagram down to a basic module diagram of Input, Processing, Output, functional chart. For each task to be performed (see Figure 3.5-2), a module diagram is developed. For simplicity sake, only one module is described in the figure.

The top level Visual Table of Contents (Figure 3.5-1) identifies major functions to be performed and also points the reader to additional lower level charts via text data within the diagram.

The major functional block is documented by a lower level table of contents. This table of contents identifies the next level tasks to be performed by the selected function. Each block illustrated on the second level Visual Table of Contents is further described by function (see Figure 3.5-2). Functions are described in relation to INPUT, PROCESSING, and OUTPUT. Pertinent additional descriptive data relating to processing functions are described in tabular format.

3.5.4 Study Data

Structured documentation has two major features--hierarchical organization and input/output/process. The hierarchical organization is a natural byproduct of the top-down design and an accepted concept in engineering for description of hardware. The technique has gained acceptance in programming in the form of block diagrams. Dr. H. Trauboth (NASA/MSFC) and others have proposed hierarchical organization of software documentation.

The Input/Output/Process format for describing computer programs is the basic organization of the Part I CPCEI format in the Apollo Configuration Management Manual (NHB 8040.2).

These two concepts were merged in the IBM-developed concept of HIPO: Design Aid and Documentation Tool used to document the System/370 Operating System for virtual storage operating system.

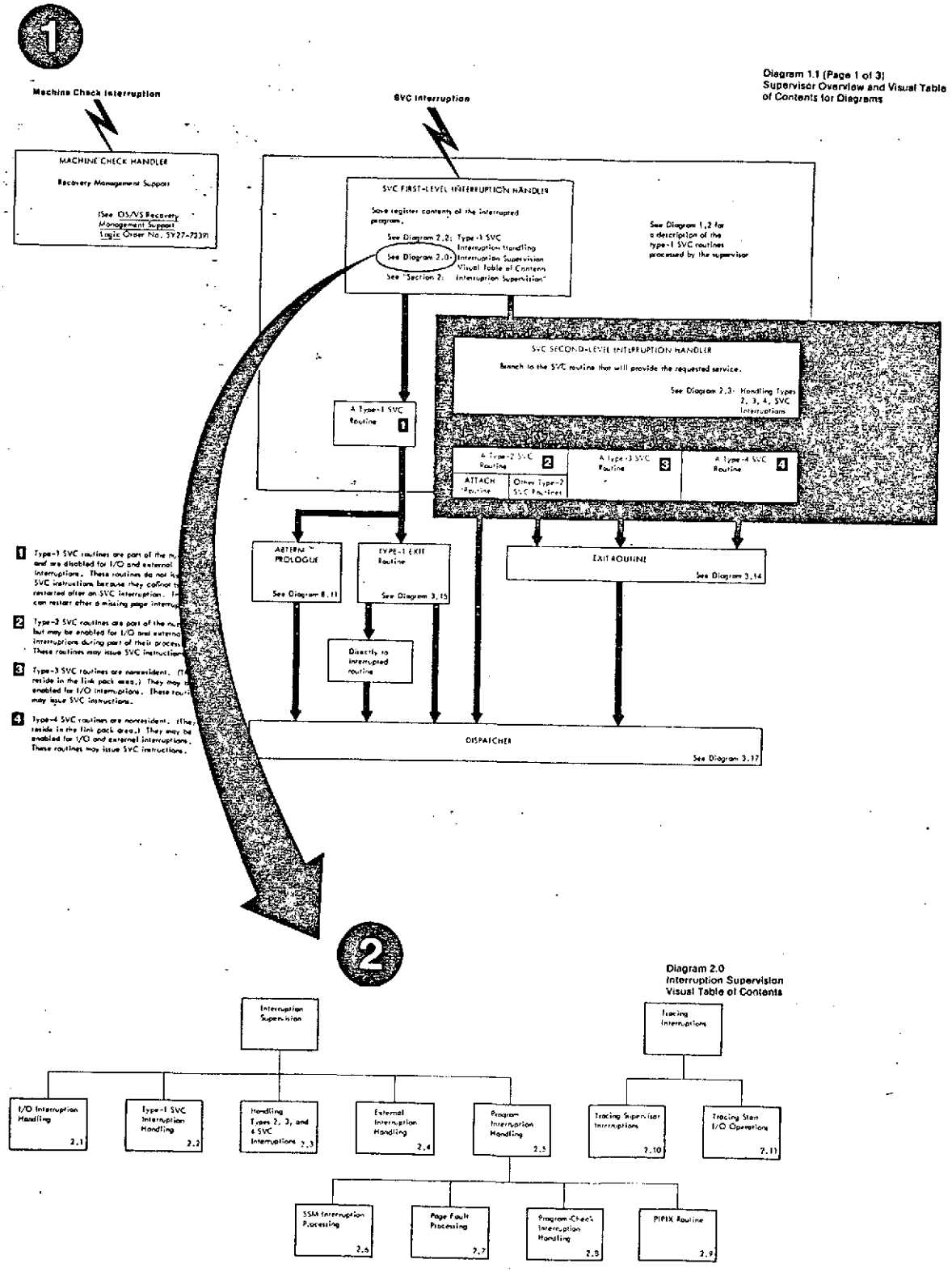
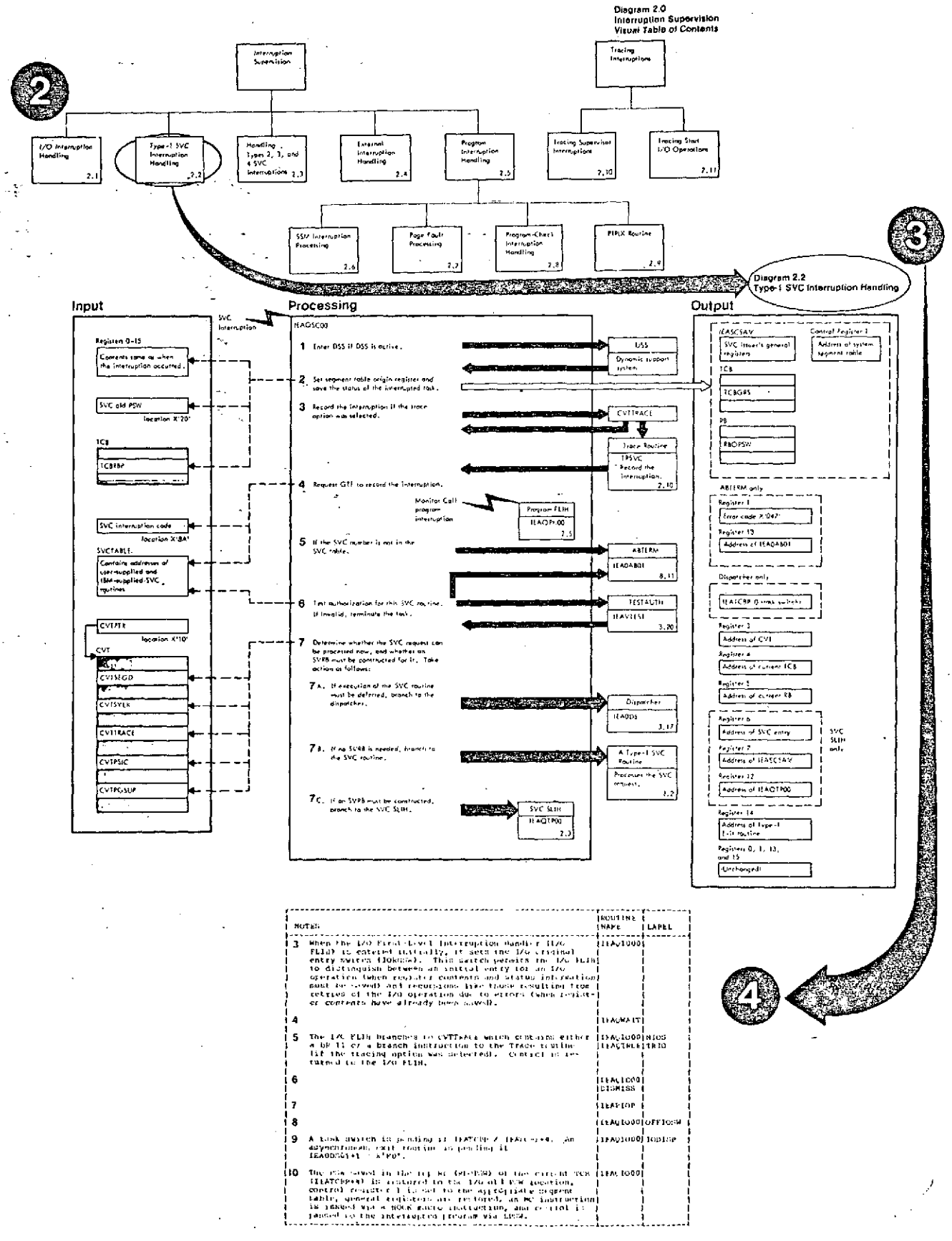


Figure 3.5-1 Visual Table of Contents Diagrams



NOTES:	ROUTINE NAME	LABEL
3 When the I/O First-Level Interruption Handler (IEAQFLIH) is entered initially, it sets the I/O original entry switch (IOORIGNL). This switch permits the I/O FLIH to distinguish between an initial entry for an I/O operation (when register contents and status information must be saved) and recursions like those resulting from retries of the I/O operation due to errors (when registers or contents have already been saved).	IEAQI000	
4	IEAQWAIT	
5 The I/O FLIH branches to CVTTRACE which contains either a BR 11 or a branch instruction to the trace routine (if the tracing option was selected). Control is returned to the I/O FLIH.	IEAQI000 IEAQTRLE	NIOS TRIO
6	IEAQI000 DISMISS	
7	IEAPIOP	
8	IEAQI000	OFFIOSM
9 A link switch is pending if IEAPIOP ≠ IEAZero. An asynchronous exit routine is pending if IEAQDCS1+1 ≠ X'FO'.	IEAQI000	IOPISP
10 The PSW saved in the I/O FLIH (X'0-X'08) or the current TCB (IEATCBPSW) is restored to the I/O old PSW location, control register 1 is set to the appropriate segment table, general registers are restored, an RC instruction is issued via a HOOK macro instruction, and control is passed to the interrupted program via LPSW.	IEAQI000	

Figure 3.5-2 Detail Flow/Input-Process-Output Diagrams

3.6 CAUSE AND EFFECT ANALYSIS

3.6.1 Theme

Cause and effect analysis, based on techniques used in hardware logic design, is a means of rigorously defining a software element in terms of its inputs (causes) and outputs (effects). The techniques involved include annotation of the written program definition to identify which causes produce which effects, and graphic representation of the results to form Boolean graphs.

3.6.2 Conclusions

Cause and effect analysis is an excellent specification review tool providing:

- A concise means of documenting complex interrelationships which are difficult to express in writing.
- A complete set of validation tools used to verify the quality of a software product.
- A vivid description of system interrelationships when Spacelab is transferred to NASA.

3.6.3 Description

Cause and effect analysis produces a Boolean graph that rigorously describes the structure of a program's functions. It shows relationships in precise, logical, Boolean terms that are difficult to express in English. It shows them graphically so that they can be analyzed.

The graphs and associated techniques were originally developed as a software verification tool. Prior to the development of these techniques the process of software validation was an error-prone activity. The steps involved analysis of the program specification and the code if available in an attempt to identify all possible functional variations that could occur during program execution. The findings produced by this somewhat intuitive process were then documented in the form of test cases. The problem was that no one using this method could be sure that every possible situation had been covered by a test case. The result was that many times software was released into a production environment without being thoroughly tested. Cause and effect analysis makes the process of software validation a more disciplined activity in which all possible functional variations of a program are identified, and therefore can be tested.

To produce a Boolean graph, it is necessary to thoroughly analyze the written program specification. The process of analysis involves identification of each cause and effect in the document and numbering them as unique entities. The result of this process is called a node list. The graph itself is then developed from the node list. The process is to list the causes on one side of the graph and the effects produced on the opposite side. Some causes will produce immediate effects; others must be associated using the Boolean operators AND, OR, NAND or NOR to produce the required effect. The node for each cause that is involved in process of producing an effect must be connected by a line to the appropriate effect node. Each Boolean association of causes is identified by an appropriate symbol on the graph. During the development of the graphs, errors in the specification document are identified. The nature of the graphing process is such that there can be no ambiguities or inconsistencies or the graph cannot be drawn. This characteristic is one of the major benefits of cause and effect analysis. The system designer and the programmer jointly can develop a complete graph which rigorously defines the problem the program is required to solve. The completed graph then becomes a valuable communication tool for the remainder of the development cycle. The graph is also an excellent addendum to the normal program documentation providing a clear and concise definition of how the completed program will perform.

Using this rigorously defined technique permits the development of automated tools to document these interrelationships.

The process of developing the necessary test cases to prove a program's validity normally is conducted concurrently with the actual implementation effort. The validation group assigned to test the finished program can develop a computer

program that provides the computational and data handling power required to produce an effective set of test cases. Input to this program would be the encoded information contained on the cause and effect graph. The program would produce four kinds of output information which complete the process of developing an exhaustive test plan for the program being validated. These outputs are:

- A list of the distinctive functional variations eligible for testing.
- A pattern of causes (inputs) to be invoked or suppressed for each test case.
- A pattern of effects (outputs) to be observed in a correctly executed test.
- A pattern of faulty variations potentially responsible for a failing test.

3.6.4 Study Data

The base for this study was the advanced techniques used by the IBM System Development Division in validation of software systems (such as OS/370 VS).

3.7 SIMULATION

3.7.1 Theme

Software in space must execute correctly the first time. No test runs are permitted. Therefore, simulation must be employed to debug, verify and validate software prior to actual operation.

3.7.2 Conclusions

Both hardware and digital simulation of the Spacelab will be required to ensure reliable software that works.

Digital simulation provides these advantages:

- o Repeatability - with digital simulation, exact repeatability is possible including timing and error conditions.
- o Error Response - with digital simulation, hardware errors can be introduced that are physically impossible, either because of timing constants or possibility of hardware damage, in the operational environment.
- o Traceability - all simulated elements of the system can be observed at any point of operation. This permits detailed analysis of all software errors.

Hardware simulation provides these advantages:

- o Authenticity - real time operation is more closely obtained. Any software simulation is a model and as such never describes all physical operation.
- o Speed - digital simulation at the detail level normally is much slower than hardware simulation in real time.

Spacelab will require both types of simulations for the development of onboard software. The hardware and software used for these simulations should be duplicated (or moved) when Spacelab delivery is made from ESRO to NASA.

3.7.3 Description

Development of onboard software requires five modes of simulation.

- o Modeling -

  The operation of the onboard processor and system dynamics is grossly modeled for a ground-based computer to permit rapid execution of software programs. Most programs' logical errors can be detected and corrected at low cost.

  NOTE: System dynamics includes all input and output functions to the computer as well as dynamic models' external functions.

- o Interpretive Computer Simulation (ICS) -

  A detailed logical model of the onboard processor and the related system dynamics is developed for a ground based system. This simulates the onboard processor at the bit level. The simulation can perform diagnostic checks during execution to produce trace data, memory dumps and accumulator values to enable a more detailed analysis of a problem. Information can be obtained to produce a measure of the load placed on the computer at any given time. Random but controlled error conditions can be introduced into the software to evaluate software response. The actual code under test need not be altered to obtain expected results or induce specific software paths. The simulator can perform separate checks on the output of the software under test to insure that the calculations are being performed correctly. Every variable used in the program computation is available to facilitate problem analysis. The ICS can provide a record of critical timing paths, frequency of instruction reference and even a record of unused (or untested) instructions. In addition, error conditions can be traced back to the source.

- o Target Computer Independent Simulation -

  Interpretive Computer Simulation suffers in performance as the target computer speed approaches the host computer speed. The ratio of simulation execution time to mission real time has grown from approximately a 1.8 to 1 ratio to over a 100 to 1 ratio over the past few years. To maintain acceptable software development schedules, alternative simulation techniques are required. The target computer independent simulator becomes a useful tool if the operational software is written in a higher order language. The programmer source statements, being computer independent, can be compiled into host computer instructions instead of target computer instructions. The simulator takes advantage of this attribute. The programmer coding in the high order language requests that the compiler generate host computer instructions rather than target computer instructions. The compiler provides

timing information with each source statement code to assure that the simulation will proceed based on realistic timing relationships. Diagnostic checks similar to those afforded by an ICS are available. The advantage of this type of simulation is that it proceeds at a rate close to the host machine's speed, affording quick testing of program logic. This type of simulator is being developed for the Space Shuttle software development effort.

- o Hybrid Simulation -

  The ground based computer is interfaced with an onboard processor. This provides actual hardware execution of instructions but system dynamics is simulated digitally. This allows the software to execute in a manner which will obtain exact timing and interface signals without the use of the actual hardware which may or may not be developed at that time. The hardware is simulated by means of mathematical models and allows the design engineers to evaluate hardware changes before implementation. This technique also provides a simulation of a real time environment so that software can be tested as if in actual conditions. Hardware problems can be simulated that may be impossible to do with a hardware breadboard. The use of hybrid simulation allows situations to be simulated which would cause physical damage to actual hardware in proving software response.

- o Hardware Simulation -

  This is a complete hardware buildup of the onboard system. It provides a realistic response to the onboard software, while providing easy access to induce requested hardware response. This type simulation provides the most realistic environment for software validation.

No software validation is complete until it executes on the real hardware. Onboard software approaches real hardware execution in stages--first on the onboard processor driven by a software model of system dynamics, then to a breadboard and finally to the actual flight.

3.7.4 Study Data

The IBM Federal Systems Division has developed and used digital simulation for most of their flight processors. This included the Saturn LVDC, the Skylab Apollo Telescope Mount Digital Computer (ATMDC), all of the 4/Pi series processors, and the Shuttle AP-101.

The Saturn flight program was developed with these simulators.

Skylab used these same facilities with additional equipment representing Skylab functions.

Shuttle has identified these simulation facilities for operational software development.

Simulation used in the Saturn/Apollo and Skylab Programs provides an excellent study base for evaluation of the simulation techniques.

The Six Degree of Freedom/LVDC Simulator, Simulation Laboratory, and Flight Program Checkout Facility (360/44) have been utilized as tools in verification of both Saturn and Skylab flight programs. The following figures demonstrate the value of simulation in delivering an error free software package by raising questions and exposing possible problems.

SATURN FLIGHT PROGRAM (LVDC)

Saturn Flight Program For IU	Questions Exposed
SA-206 Launch Vehicle	232
SA-207 Launch Vehicle	199
SA-207A Launch Vehicle	203
SA-208 Launch Vehicle	105
SA-209 Launch Vehicle	30

SKYLAB FLIGHT PROGRAM (ATMDC)

System	Questions Exposed
16K Programming System	403
8K Programming System	317

While simulation uncovered a large number of questions the fact remains that:

- On a system as complicated as Saturn, there has never been a software problem to prevent achieving complete mission objectives.
- The ATMDC on Skylab has reported two minor software problems in over six months of continuous operation. Neither affected mission performance.

This record provides proof that the simulation techniques used on those projects produce high quality software.

Simulation saves time during program verification on the actual hardware. Simulation uncovers problems such as coding errors, logic problems, and data specification errors, thus allowing all the time used for verification on hardware to determine interface problems and erroneous hardware specifications. Proof of this time saving technique is illustrated in Figure 3.7-1. This represents two programs that were written for the Tactical Aircraft Guidance System application. These are two delivered programs that were a part of this application. These programs performed different functions, yet did the same basic operations of read input, reformat, and output. One used simulation during program verification, the other did not. Note that the program using simulation completed verification approximately four weeks early.

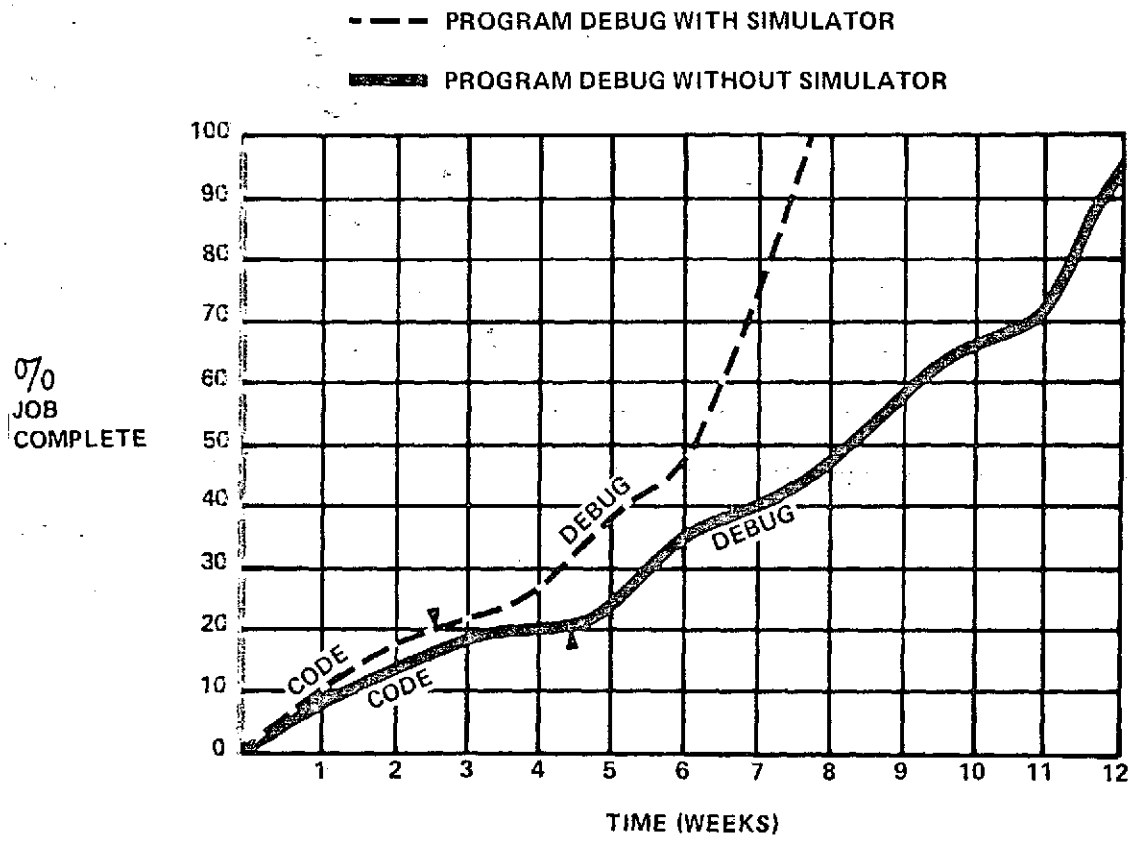


Figure 3.7-1 Software Debug Comparison

3.8 AUTOMATED MANAGEMENT SYSTEM

3.8.1 Theme

The Automated Management System provides an automated method for assuring controlled planning, development, and implementation of the software. This is accomplished through written guidelines, procedures, and reports which provide visibility into the status and progress being made at any point in time of the software development. Additionally, facilities are provided which assist in controlling the generation and maintenance of the software and documentation.

3.8.2 Conclusions

Experience gained in the development of techniques for the Saturn/Skylab programs has proven that an automated management system is a mandatory requirement for accurate, cost effective system control of software programs.

The magnitude of the Spacelab software system presents a large paper-control problem to control the change activity within the system. The ability to track change activity, the programs in development, documentation, and the coordination of hardware and software implementation, is vital to the operation of those charged with the responsibility of a successful mission. This important fact is accented because of the short lead times caused by the refurbishment cycle of the Spacelab at multiple sites.

3.8.3 Description

The Spacelab software consists of many interactive software modules and exists in many forms and change levels. These include source modules, object modules, and load modules in the different phases of development. This assortment creates a complex control problem during the software development and delivery process. The problem is not isolated to computer programs alone. Documentation, such as specifications, user's manuals, and interface control documents exhibit the same problem characteristics.

To accomplish this timely and accurate configuration management, the use of automated methods is required and must encompass the complete spectrum of software design, development, and maintenance. These automated methods must be flexible to meet the varying Spacelab requirements and yet provide a straightforward approach to control of the entire system from cradle to grave.

The automated management system should support:

- Remote terminal access for software program generation, configuration accounting, and management information tracking activities.
- A security system to prevent inadvertent destruction of the data base.
- Processes for management and control of the software program libraries.
- Automated generation of the delivery package (source modules, object modules, and documentation).

3.8.4 Study Data

The recommended use of the automated management system is based on IBM's success with these methods in the Saturn/Skylab programs. These methods have significantly increased visibility into the system and provided better control of change by management. The delivery process has been significantly improved and streamlined both in time, manpower, and accuracy in making a Saturn software delivery. Figure 3.8-1 shows the manpower saving realized by an automated system.

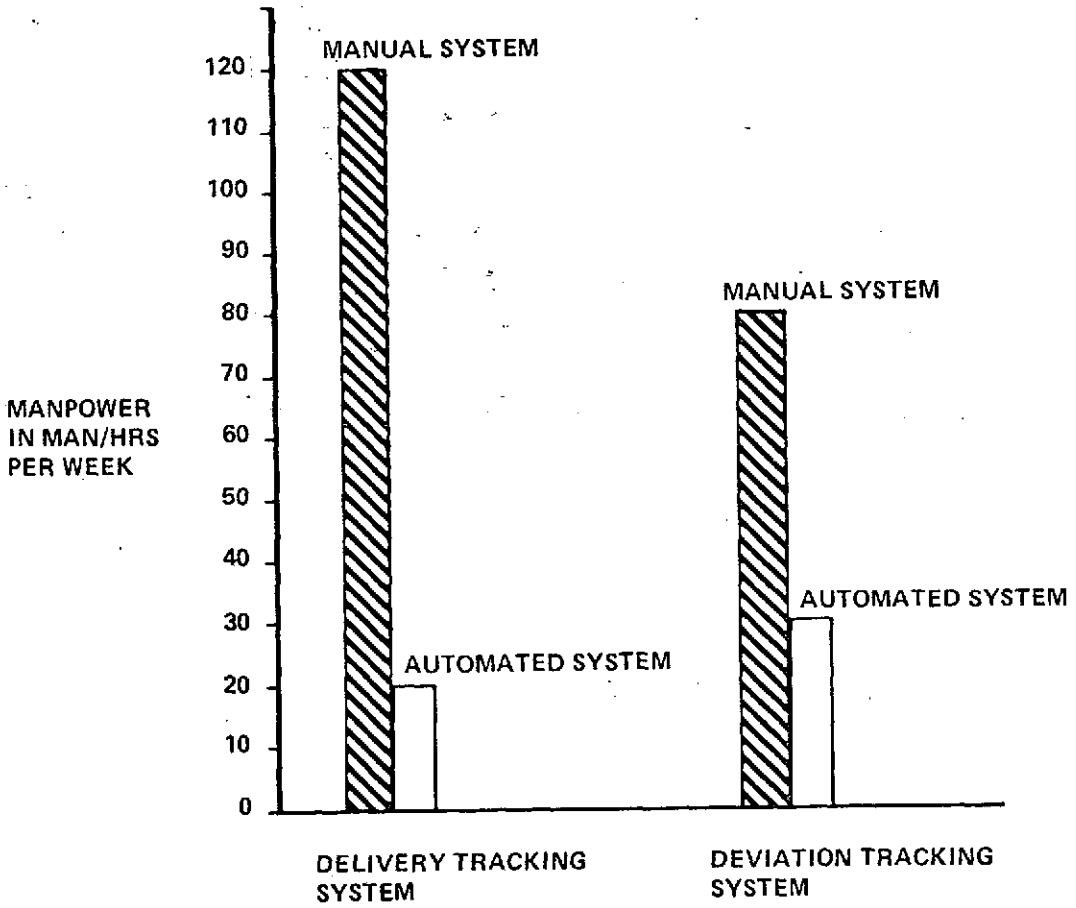


Figure 3.8-1 Deviation and Delivery Tracking Manpower Requirements

3.9 SOFTWARE MANAGEMENT BOARDS

3.9.1 Theme

Proper management of the software development and operational phases is required to produce computer programs that satisfy operational requirements as well as cost and schedule commitments. Properly organized Software Management Boards have proved to be effective vehicles to provide software development precepts in the pre-development phase, control of the design during the development phase, and configuration control of the end products during the operational phase. Software Management Boards serve as focal points for software requirements emanating from many sources, assuring quick and decisive action.

3.9.2 Conclusions

Software Management Boards have proved to be effective software management techniques on several large and small software development projects. With the number of government organizations and contractors involved in the Spacelab project, positive management control of the software is mandatory and can be effected through the use of Software Management Boards.

3.9.3 Description

The requirements for the control of software change as the software transcends from concepts, requirements, development to operation. To satisfy these changing responsibilities, the management boards' makeup and spheres of control must be adjusted accordingly as depicted in Figure 3.9-1. Three types of boards that have served successfully for diverse types of software requirements and products are described below.

SOFTWARE REVIEW BOARD

This board, in its preliminary form, consists of ESRO and NASA personnel and is expanded to include contractor personnel when appropriate. This group's primary purpose is to manage the design and development of the Spacelab's systems.

During the software pre-development period, the Board will define software standards, programming languages, programming techniques, documentation requirements, and verification techniques. The Board will ensure that these sets of software standards and procedures are established and used by the software contractor.

During the development phase the board acts as the clearing house for software requirements imposed by the users. The Board will forward approved requirements to the software contractor for inclusion into the system specifications. The responsibilities of the Board during this phase are:

- Establish schedules and monitor these schedules to ensure timely and orderly controlled development of the software system.
- Receive and approve software requirements baselines and baseline changes.
- Review and approve the software requirement specifications, products of the definition activity.
- Review and approve the design specifications, products of the design activity. This point establishes the design baseline from which the computer program is manufactured and also establishes the point at which changes to the requirements are rigorously controlled.
- Review and approve all requirement changes and provide positive change accounting procedures. This responsibility of the Board is important and is usually the determinate in providing on-time, on-cost software products.
- Assure that the hardware and software development processes are compatible through establishment of interface control documents, hardware/software trade studies, and interlocking board memberships.

- Review and approve software acceptance test plans to assure the test will adequately prove that the software is reliable and meets the operational requirements.

SOFTWARE CONTROL BOARD

The Software Control Board consists of contractor personnel, and it is their responsibility to provide direct management over the software during the software development phase. The Control Board will ensure that the guidelines and the designs approved by the Review Board will be implemented. The types of functional groups comprising this Board are shown in Figure 3.9-2.

The Software Control Board will meet as required to review current status, additional requirements, and/or implementation or integration difficulties. This Board will have the authority to resolve all technical problems within the scope of the guidelines and design requirements approved by the Review Board. It will also refer any design requirements or technical problems that cannot be resolved within the framework of the contract to the Review Board for final resolution.

The Software Control Board will be active during the software development phase to enable fast response to problems that appear during development.

SOFTWARE CONFIGURATION CONTROL BOARD

The Configuration Control Board consists of the combination of the Software Review Board and the Software Control Board and is placed into operation at the software acceptance test milestone. The most experienced technical people from ESRO, NASA and the contractor are combined into a comprehensive, integrated management control unit. This is necessary to provide fast action on changing requirements during the operational phase. With up to 50 launches a year, fast software response to requirement changes is imperative.

The Configuration Control Board will assess new requirements or change recommendations and analyze the impact on the system performance. If the change is approved, the Board will conduct a design review and, upon approval, implement the requirement and determine the change break-in point into the Spacelab system.

3.9.4 Study Data

Saturn experience has proved the effectiveness of the Software Management Board concept. During the early stages of Saturn development no board existed as a formal, recognized entity. As the project progressed and the number of software users expanded, it became imperative to establish a single point of control for the software development/operation processes. The spheres of authority and responsibility were modified as experience with the Board concept grew. The concept proved so valuable that it was imposed upon other Saturn software projects and was implemented at the inception of the Skylab Apollo Telescope Mount Digital Computer (ATMDC) software system.

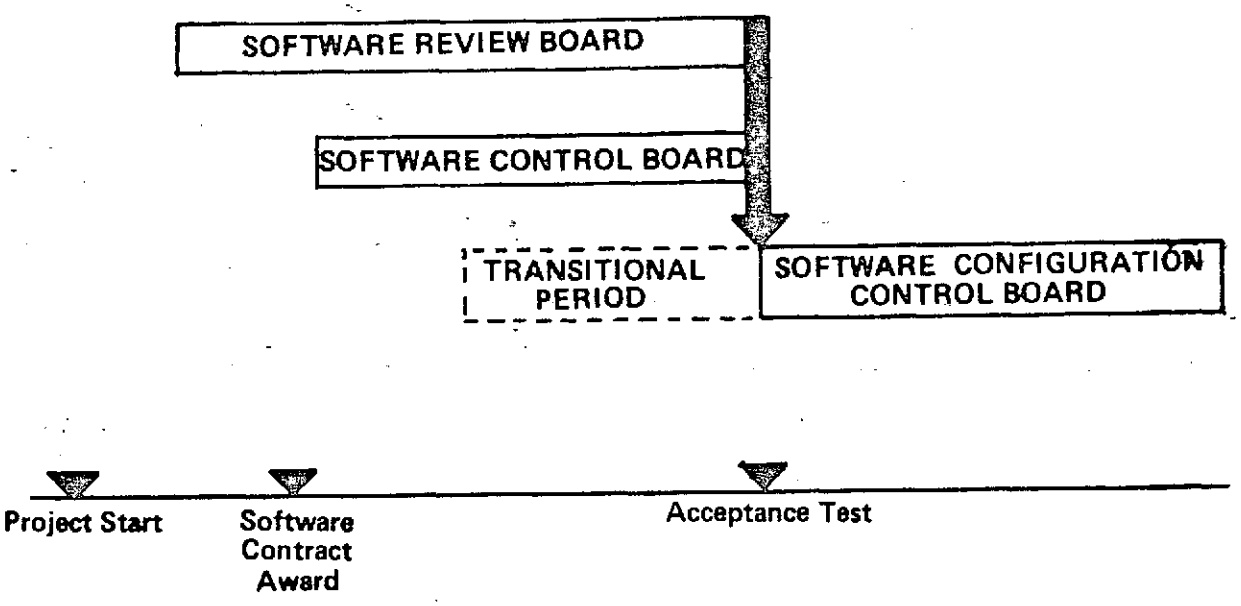


Figure 3.9-1 Software Management Board Evolution

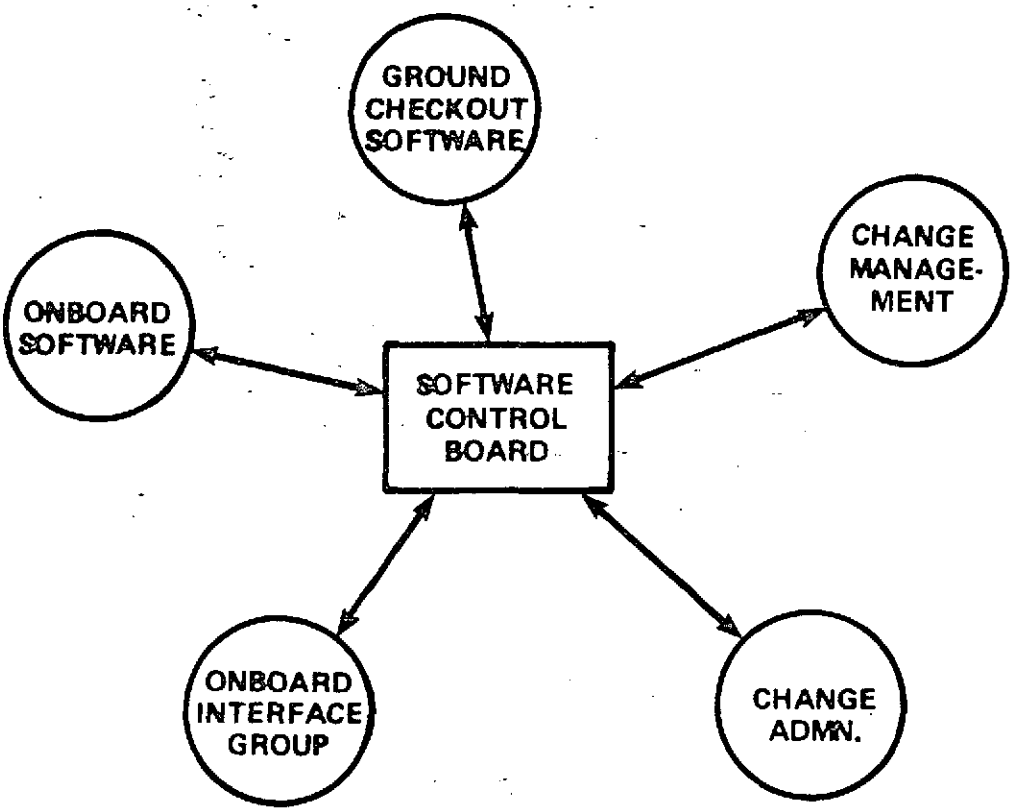
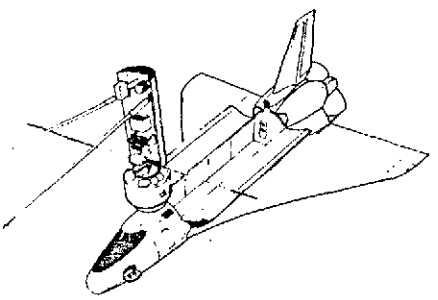


Figure 3.9-2 Software Control Board Functions

SYSTEM CONSIDERATIONS

4



Paragraph	Title	Page
4.1	HIGHER ORDER LANGUAGE	4-2
4.2	USER LANGUAGES	4-4
4.3	ONBOARD COMPUTER	4-6
4.4	SUPPORT SOFTWARE	4-12
4.5	SOFTWARE DEVELOPMENT FACILITIES	4-16

SECTION 4

SYSTEM CONSIDERATIONS

Spacelab programmatic decisions must be made that affect the cost of software and yet cannot be considered techniques of programming.

For Spacelab, these items present opportunity for significant cost savings in software development and operation:

- Higher Order Language
- User Language
- Onboard Computer
- Support Software
- Software Development Facilities

4.1 HIGHER ORDER LANGUAGE

The term Higher Order Language (HOL) is used to identify a class of programmer tools that permit coding in English language statements. The statements are then translated into object code by a compiler for that HOL.

The HOL is problem-oriented and is relatively machine independent. The compiler is the tool that implements the HOL on a computer, and is normally designed to produce code for only one class computer. One HOL may have several compilers, each compiler producing code for different computers. One compiler may execute on one computer (the Host computer) and produce object code that executes on another computer (the target computer). The compiler may be implemented in a HOL other than the one it translates.

Higher order languages are designed to be an effective tool in expressing problem solutions. This is an advantage since the HOL programmer can express his software solution in statements which are natural for him to use. The key issue here is that the programmer avoid having to translate his solution into the terms understandable to the machine. In time, an HOL programmer comes to think in terms of the language he is using much like mathematicians think in mathematical notation. This results in improved programmer productivity. The resulting programs are easier to understand, easier to learn and easier to maintain.

The machine-independent characteristic of a high-order language has two aspects. The first of these is that programmers using the language do not need a detailed knowledge of the hardware on which their program will run. This is not to say that hardware knowledge is not a valuable asset for a programmer, it is, because a basic function of a programmer is to use the computer hardware effectively. Nevertheless, it is true that it takes less education time to become a productive programmer using a high-order language than using an assembly language, primarily because a detailed knowledge of complex hardware is unnecessary.

The second aspect of the machine-independent characteristic is program transferability. If the language is sufficiently removed from the machine, then it may be run on any computer if a translator exists for the language on the target machine. This aspect is particularly important since it provides the means for recovering software investment by maximizing the use of existing programs.

These advantages come at a price. Historically, HOL's do not take advantage of the hardware features of the target computer. As a result the code produced has been less than optimal both in size (too much memory) and performance (slow).

Memory and performance have been critical factors in computers designed for the space environment and HOL's could not meet the restrictions. Recent improvements in technology and manufacturing techniques have significantly reduced the restrictions on these computers. State-of-the-art space computers can now provide greater memory and performance in a smaller, lighter package that requires less power and can be purchased for fewer dollars.

Higher order languages now present a viable alternative to assembly level coded systems. The reduced software development costs can be evaluated against the cost of providing an onboard computer with more memory and performance capability required to compensate for the less than optimum output from the HOL compiler. The reduced software cost realized by using a HOL are a result of:

- Improvement in programmer productivity
- Improvement in product quality
- Improvement in program readability
- Improvement in program documentation
- Improvement in program maintainability
- Improvement in program development schedules

The availability of a HOL compiler should be a consideration in the selection of the onboard computer.

Several candidate HOL's exist and have been evaluated for space applications. Each has advantages and disadvantages. As a result of the evaluations, JOVIAL and some of its dialects have been selected for use on Air Force programs, and HAL (the Houston Aerospace Language) has been selected for the Space Shuttle software. Current analysis indicate that 80 to 95 percent of the Space Shuttle software can be done in HAL.

The use of a HOL for Spacelab software should be carefully evaluated as a cost-effective alternative to assembly language coding tehcnique. The operational advantages offered by the HOL's readability, reliability and maintainability weigh heavily in such a decision.

4.2 USER LANGUAGES

A user language is a subset of the class of higher order languages that is oriented to the non-programmer. The language is defined to solve problems stated in terms of a particular discipline. Languages exist for electrical engineers, mechanical engineers, traffic engineers and many others. For Spacelab a language is required for the test engineers, principle investigators and the Spacelab crew members.

Spacelab language requirements are:

- The language must be self-documenting.

  This minimizes the requirement for additional documentation and provides written definition of checkout procedures for test personnel.

- The language must support engineering nomenclature.

  Engineering terms and symbols must be integrated into the user language to provide usability and readability.

- The language must provide automatic execution of defined test procedures.

  To eliminate operator errors and excessive response time, test procedures can be defined and created in an off line environment and be cycled in an automatic condition in the Spacelab.

- The language must provide capability of building special checkout procedures in real time.

  Problems will arise during checkout where automatic test procedures will not completely fulfill the necessary test. Because of this it is very important that the capability exist to interrupt automatic operations and allow the crew or principle investigator to build a special procedure to address the problem area.

- The language must support machine language and special purpose subroutines.

  It is probable that some aspects of checkout and experiment control cannot be successfully accomplished by using aspects of the user language. When these cases arise, it will be necessary for the user language to relinquish control to a machine language or special purpose subroutine. This must be done automatically by the user language without interruption of the procedure.

- A language must be functional and easy to use.

  It is of prime importance that the user language provide the functions required for the Spacelab program in a clear and understandable manner.

Several examples of languages exist that meet most of these requirements. These languages are normally classified as test and checkout languages. Examples are:

- Abbreviated Test Language for Avionic Systems (ATLAS) developed for commercial airline use by Aeronautical Radio, Inc. in 1968.
- Acceptance Test or Launch Language (ATOLL) developed for checkout of the Saturn vehicle by NASA/Marshall Space Flight Center in 1965.
- Basic English for Testing Applications (BETA) developed for aircraft testing by the General Dynamic Corporation in 1970.
- Ground Oriented Aerospace Language (GOAL) developed by NASA/Kennedy Space Center for Shuttle checkout in 1971.
- Vast Interface Test Application Language (VITAL) developed for the Navy's Versatile Avionics Shop Test System by PRD Electronics in 1970.

A user oriented language should be developed for the Spacelab to reduce development and operational costs by giving engineering and scientific personnel access to both onboard and ground computer systems without the necessity of going through the software development cycle.

4.3 ONBOARD COMPUTER

With recent and projected computer hardware improvements in memory capacity, computational ability, and power utilization combined with a decreasing computer cost, the impact of software development cost on overall system cost has become an increasing concern. In addition, increasing reliance is being placed on software to perform functions previously performed by hardware. As a result, the software has become a critical gating item both in system development and the ability to meet overall system objectives. Figure 4.3-1 indicates the trend in software costs versus computer hardware costs.

A common problem in all previous space programs has been a lack of onboard computer capability to satisfy all the demands made upon it. The most significant hardware limitations have been in the memory capacity of the computer and the CPU computational speeds.

Previous experience in memory utilization on three projects is shown in Figure 4.3-2. In all cases, the memory capacity was chosen based on preliminary software requirements analysis and was thought to be more than adequate for any future needs. In all cases a significant increase in software requirements occurred during software development. This failure to plan for adequate growth in software resulted in requirements exceeding the memory capacity of the computer.

As with memory utlization, CPU computational capability has largely been based on preliminary data, and growth in software requirements has likewise placed a significantly increased computational burden on the computer.

The effects of marginal computer capability on software development costs is shown in Figure 4.3-3. The reason for the rapid increase in development costs as the computer capability is approached is that a slight gain in software capability can only be bought at the cost of increased logic complexity. This increased complexity makes the program harder to write, to checkout, modify, verify, and coordinate with other operations.

Because of the tendency to build unique computers to satisfy particular requirements, spaceborne computers have had unique architecture, instruction sets, input/output capabilities, memory capacity, and CPU speeds. These unique characteristics have resulted in significantly increased software development cost. The costs were largely the result of inability to transfer software having the same capabilities and the necessity for development of special-purpose support software for every application/computer.

Studies conducted by IBM on airborne software development projects which utilized the same computer hardware indicate that transferability of common software from application to application can reduce cost per program statement by as much as 75% and increase productivity by a factor of five. In addition, transfer of operational software improves system reliability.

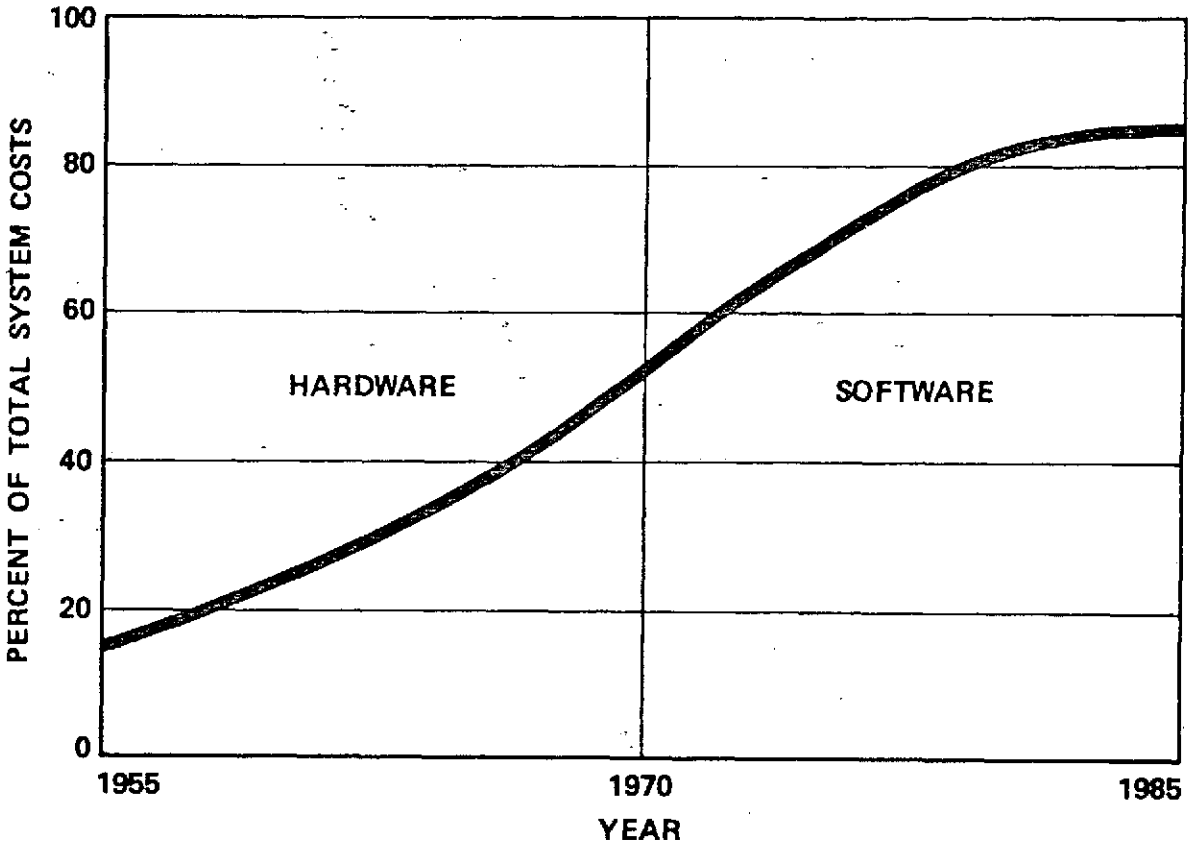


Figure 4.3-1 Computer Hardware/Software Trends

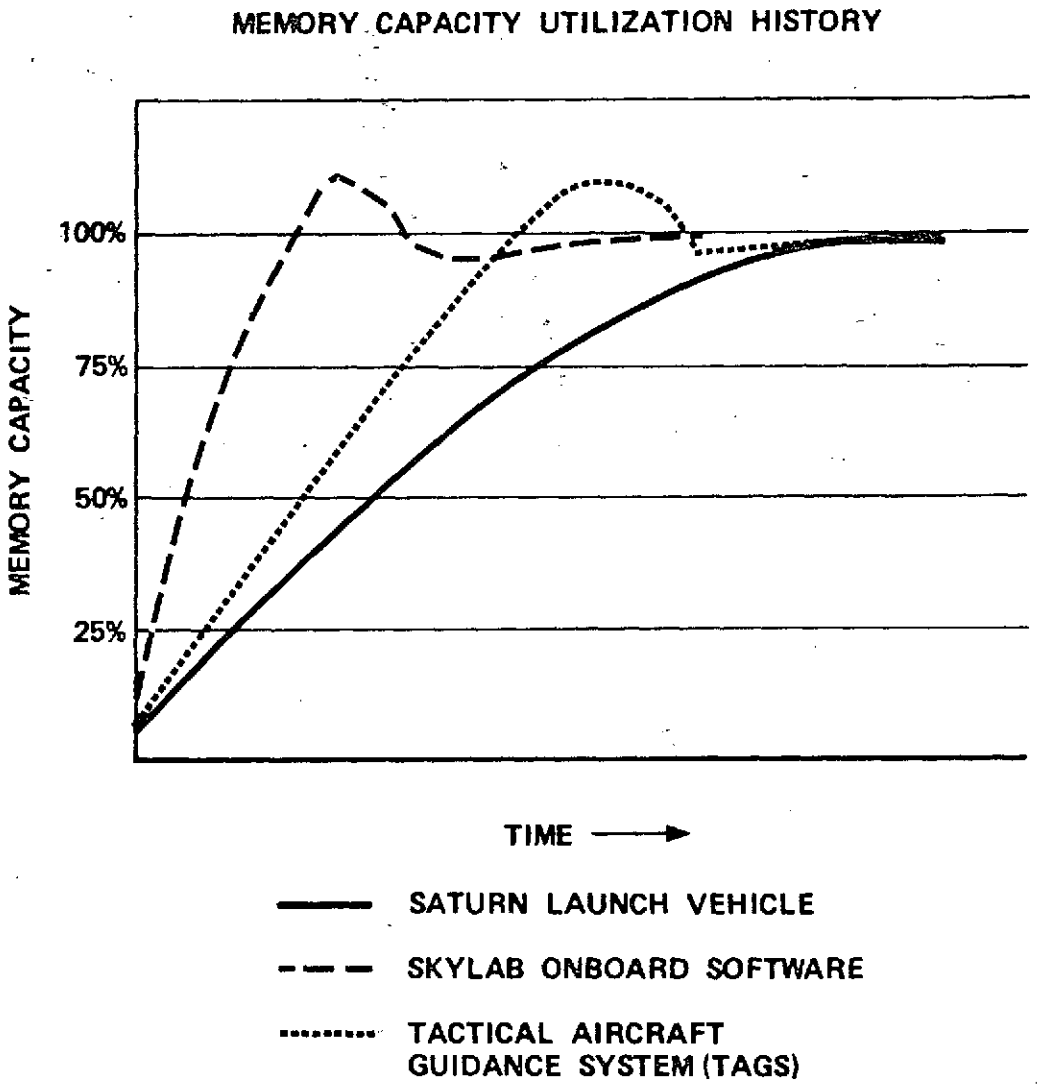


Figure 4.3-2 Memory Utilization History

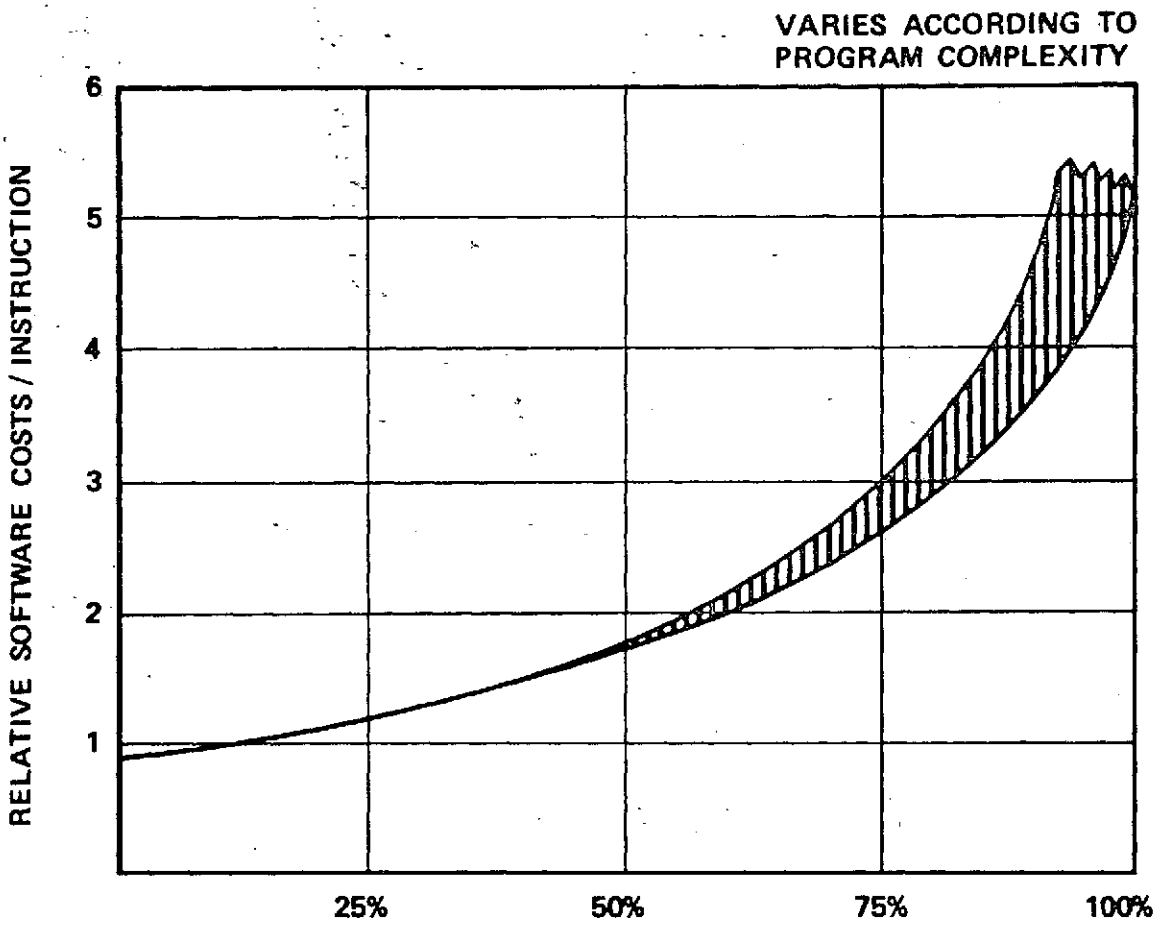


Figure 4.3-3 Computer Capability (CPU Memory Capacity)

A result of commonality between onboard/ground based computers will be the capability to utilize support software (high-order languages/ assemblers/linkage editors/debug aids, etc.) which already exist. This commonality also results in a reduction in the simulation area in that an interpretive simulation of the onboard computer will not be required.

A significant problem in the development of spaceborne software is programmer training. In the majority of previous applications, the programmer had to become knolwedgeable in the details of computer operations, support software, and test equipment for each new project. Commonality can significantly reduce this programmer training.

Studies have indicated that fixed-point arithmetic adds between six percent and seventeen percent of the overall software costs. The use of fixed-point arithmetic requires that the following tasks be performed:

- Scaling of all data to fit within the architecture constraints.
- The range of values of all parameters must be determined.
- Programmer must insert shift operations in equations to assure proper alignment for arithmetic operations.
- Programmer must resolve conflicts between scaling requirements and accuracy requirements.
- Extensive testing of adequacy of scaling is required.

The range of cost increase is a function of the program type--computational functions require more scaling operations than logical data handling functions. With the increasing trends toward more capability in data analysis/computation burden in the onboard computer, floating point hardware is becoming a requirement.

As a result of the trend toward less hardware cost for computers and increased cost of onboard software, it is recommended that the impact of computer architecture, speed, memory, and configuration on software cost be closely evaluated. Rather than making computer cost the main guideline, more emphasis must be placed on overall system cost impact resulting from selection of the particular computer.

Because of their direct relationship on software cost, the following recommendations are made:

- Onboard computer should have a memory capacity of at least 50 percent over that thought necessary in preliminary design.
- Onboard computer should have CPU computational ability exceeding the original requirements by 50 percent.
- Onboard computer should have floating point arithmetic built into hardware.

In addition to hardware capabilities, commonality between onboard computers and ground-based computers should be a prime objective in space software development. Commonality will reduce costs of support software and will allow use of existing capabilities such as high order languages to increase programmer productivity and allow transferability of software Spacelab experiments. Retraining of programmers will also be reduced significantly.

4.4 SUPPORT SOFTWARE

In the development of Spacelab software, support software packages/capabilities must be utilized to increase programmer productivity and to provide the means to ensure that the software system satisfies requirements.

Support software consists of those software packages/capabilities which are utilized by the programmer/analyst in software development. Included in this category are the following:

- o Assembler/compiler/linkage editor.
- o Source program library maintenance and remote job entry.
- o Source code flowcharter.
- o Source code analyzer.
- o Data analyzer.
- o Program tracing and snap/dump.
- o Automatic simulator event sequencer.

Assembler/Compiler/Linkage Editor

Provide the capability to prepare modularized avionic specifications into relocatable object modules which in turn can be combined into an executable and loadable program.

The assembler and a high order language compiler must be provided to facilitate the programmer's task of transforming the avionic specifications into relocatable object modules. The linkage editor is required to combine these relocatable object modules into an executable and loadable program.

The assembler is employed to produce relocatable object programs from the programmer's source code. It allows macro processing and conditional assembly. Outputs consist of an assembler listing, a relocatable object module, assembler error diagnostics, and symbol table statistics.

The compiler is employed to process high level language source code into acceptable assembler input. It provides good memory and register utilization and an easy to use language to express data and processing requirements. Outputs consist of assembler acceptable input, a compiler listing, compiler diagnostics, and symbol table statistics.

The linkage editor combines the relocatable object module outputs from the assembler and resolves program linkages. It outputs a linkage editor module map of the output program, diagnostics, and a program which can be used as simulator input or be loaded into the flight computer.

Source Program Library Maintenance and Remote Job Entry

To provide an efficient and flexible tool to maintain source and data code and the capability to prepare, maintain, and submit jobs remotely.

Using video terminals, the programmer will be able to maintain, add, delete, and update the source and data code in real time on a mass storage device. In addition, jobs to be batch processed can be prepared, maintained, and submitted remotely via a terminal.

To increase programmer efficiency, job turn-around, and dynamic library accessibility, a terminal management system should be employed as an on-line computer facility, to provide programmers with remote job preparation and submission capability. This system must provide multi-terminal, multi-user capability.

User source and data code card images can be maintained on mass storage devices. They can be retrieved, reviewed, updated and then returned to the mass storage device in one terminal session. Batch accessing is also available if desired.

Source Code Flowcharter

Provide a visible representation of the avionic specifications as derived from the source code itself.

Using the module source code as input, logic flowcharts can be generated which depict the actual avionic software. Flowchart outputs will provide programmers with a logic checkout tool and double as a documentation and specification review aid.

Flowcharts provide visibility into the detailed logic of the avionic software. The flowcharts can be generated from encoded flowcharting instructions or based on each source code instruction itself. The flowcharts can be used in specification comparisons, alterations, and reviews. While specifically augmenting the programmer's coding task, they also complete the programming cycle in the form of documentation.

Source Code Analyzer

The source code analyzer provides a means of automatically checking for adherance to established programming standards through analysis of the source listing.

Each program module developed for the Spacelab should be examined through use of the source code analyzer. Items to be checked are of the following types:

1) Correct use of macros
2) Reentryable routine utilization
3) Structured coding format
4) Routine linkages
5) Priority level interrelationships

The use of the source code analyzer support program will result in the programmer's being assured that certain established standards have been satisfied. Such a support software package can be used to reduce problems in integration of experiment software developed by principle investigators responsible for specific experiments.

Data Analysis

Provide the capability to assemble avionic software telemetry outputs into meaningful analyzable reports.

As an addendum of the avionic software simulator, telemetry outputs from the avionic flight program will be scanned and assembled into formats for performance analysis, program debug, and logic sequencing checkout.

Telemetry outputs in raw form are meaningless unless they are assembled into decipherable reports. Such reports will be in the form of strip charts, graphic plots, and flight variable tabulations.

From these reports, avionic specification performance analysis can be performed for nominal and off-nominal conditions. Flight logic sequencing and program output formats can be verified during program debug.

Program Tracing and Memory Snap Dumps

Provide the programmer with detailed instruction tracing of selected avionic software segments and memory image dumps at any selected point during program execution.

As an addendum of the avionic software simulator, program instruction tracing will be provided to check out logic path execution. Memory snap dumps will also be provided at program logic points as desired to verify memory contents.

The debug phase of program development necessitates a pseudo hands-on environment to verify program logic sequencing. Through the manual or automatic simulator sequencer modes, the avionic software simulator will permit the programmer to initiate program execution, trace logic paths including register contents, and to dump memory contents and selected points to verify proper or improper software logic.

Automatic Simulator Event Sequencer

Provide an interactive automatic simulator sequencing system which will maximize simulator utilization in a hands-on environment.

The avionic software simulator will provide an interactive graphics display in which to sequence the simulation runs. The standard programmer mode will be manual operation for maximum debug capability. An automatic simulator sequencer will augment this mode to provide remote and more efficient simulator utilization.

The automatic simulator sequencer will duplicate the same user stimuli through punch card input as available through the tutorial graphics display. Simulation runs can be run completely, remotely or with user intervention and resumption permitted.

Each automatic simulator sequencing run will be coded in a special language to duplicate light pen and compose field operation. This input is processed as source code and link edited as an executable driving program to the avionic software simulator.

Since development of support software can be a significant cost item in the overall software development cost, the ability to utilize existing support software must be a prime cost consideration for Spacelab. Since support software capability is a function of computer architecture, instruction sets, etc., an attempt should be made to select an onboard computer which is compatible with existing ground-based computer systems. If such a selection is made, the support software available with the ground-based computer system will be directly applicable to the Spacelab onboard software development activity, and a significant cost savings can be realized.

The various support capabilities discussed have proven invaluable in previous onboard computer software development projects and are recommended for use on Spacelab. It should be noted that the capabilities discussed are not all-inclusive, but merely a synopsis to increase the understanding of the term 'support software.'

4.5 SOFTWARE DEVELOPMENT FACILITIES

Historically, the the areospace industry, Simulation and Functional Prototype Facilities have been used to evaluate engineering and software designs and to train personnel who check out and man space vehicles. These facilities have proven to be extremely effective especially since the use of operational equipment is impractical due to cost and availability.

The three types of facilities proposed to be used in the Spacelab program for software development are:

- o Program Development Facility
- o Integration Facility
- o Prototype Facility

The Program Development Facility will be the first facility utilized. The software design personnel will utilize this facility which will consist of a large computer and its associated peripheral equipment. This facility will simulate the Spacelab computer and its environment. The Interpretive Simulator will be executed in this facility and will allow software development to be accomplished while the hardware development is being accomplished. This facility will provide the support needed for the software generation process and will provide automated tools required for the management and control of the development process and an automatic system for delivery package creation.

The Integration Facility is the second facility utilized. This facility will consist of the Spacelab onboard computer interfaced through special interface equipment to a large scale ground-based computer. This facility will simulate the Spacelab vehicle hardware and dynamic-control characteristics will be simulated. The primary utilization of the facility will be in overall onboard software system testing and verification. Although the Program Development Facility and the Integration Facility perform separate functions they may reside in a single physical environment sharing the large computer complex.

The Prototype Facility consists of an engineering model of the Spacelab and associated equipment and a large computer to simulate the Shuttle craft. This facility will provide the final check of the interfaces between the Spacelab hardware and the operational software. Advance training of checkout personnel and crew members will be performed. Engineering changes will be finalized on this facility before being implemented on a flight model of Spacelab. The man-machine interface will be tested in this facility.

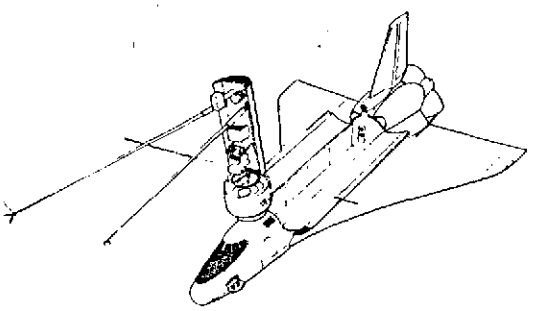
The proper use of the three facilities described above will ensure that the design of the flight model will be performed as error free as possible with the result being a greater reliability factor in the flight model. This will keep cost of the program at a minimum and prevent mission failures.

To determine if the need of a checkout facility exists in the Spacelab program, past experience in the Apollo/Saturn program was used. The Apollo/Saturn program utilizes all three facilities described in the preceding sections. To date, only one mission had to be aborted; however, no loss of life was realized. The crew returned safely. Numerous examples can be cited where a malfunction occurred during a mission and, due to simulation of the failure on a prototype facility, a work-around was established which resulted in the main objectives of the mission being performed. Based upon the performance record of the Apollo/Saturn program, the checkout facilities will be an invaluable tool in the Spacelab program.

Simulation and prototype facilities are recommended for use in the development and integration of Spacelab concepts. These facilities will become invaluable tools used in evaluating engineering and software designs, crew training and selection, and check out personnel training. The success or failure of the mission very possibly depends on the successful use of these facilities.

SUMMARY

5



Paragraph	Title	Page
5	GUIDELINES SUMMARY	5.1

SECTION 5

GUIDELINES SUMMARY

This section provides a summary of recommended guidelines formulated in the body of the report. Although the recommendations are listed as separate items for clarity, maximum benefit can be attained through adoption of all those techniques basic to the integrated software development concepts.

- o Software Standards

  Software standards must be imposed at the beginning of the software acquisition phase (contract award). These standards should include programming standards such as structured coding, listing comments and annotations, picture-on-a-page listings; documentation standards including structured documentation, flowchart conventions; and configuration control standards such as program naming conventions and program baseline/ revision level identification.

- o Program Definition

  Program definition is the process of defining the problem before a solution is begun. The product of the program definition is a requirements specification. A thorough validation of the requirements specification should be performed. This is the prime technique for development of a low cost, high quality programming product.

- o Top-Down Development

  The top-down approach applies disciplined system design concepts to software development. Design and implementation proceed from the highest level downward. The approach should be applied to Spacelab. It will provide the capability of evolving the product in a manner which maintains the characteristics of being always operable, modular, and always available for the successive levels of testing that accompany the corresponding levels of implementation.

- Composite Design

  Composite design is a design methodology that consists of a strategy, rules, and measures to produce modular computer programs. The resulting programs are composed of relatively independent parts that are highly visible and easily understood. Applying these concepts to Spacelab will discipline the design technique.

- Structured Code

  Structured coding is the technique of coding a program such that all logical functions to be performed are comprised of three basic software structures. Programs written in structured code can be literally read from top to bottom typographically. Using structured code for Spacelab will produce code that is readable.

- Structured Documentation

  Structured documentation is a strategy for obtaining low cost, usable documentation of programming products. This is accomplished by integrating the concepts of top-down composite design with the documentation process. Documentation for program review and final product description become byproducts of the programming process. This will reduce Spacelab documentation cost while producing a quality product.

- Planned Verification

  Verification is the process of ensuring that the product performs to specification. Verification is a planned activity at each step in the program development cycle. It applies to documentation as well as code. Planned verification will produce programs that work.

- Planned Validation

  System validation performed by an independent group demonstrates that the hardware and software interface as a system, providing confidence in the system's ability to perform its intended function.

- Acceptance Testing

  The acceptance testing activity consists of verification testing within an operational or simulated operational environment performed against a frozen baseline configuration. The activity defines the completion of the development phase and the start of the operational phase. Applying this concept to Spacelab will sharply define milestone completion.

- Deliverable Items

  Deliverable contract end items must be identified early in the development cycle and consist of acceptance tested operational programs, simulators, development support systems, assemblers, compilers, debug aids, and other software required during the operational phase. End item components are computer programs in their object and source form, listings and flowcharts, requirements and design specifications, user documentation, test procedures and results, and configuration accounting documents.

- Software Review Board

  The Software Review Board should be established early in the project life. The primary purposes of the Board are to define software standards, languages and techniques; and to manage the design and development of the software systems. The Board has the responsibility of reviewing and approving software requirements and design documentation, employing Preliminary and Critical Design Reviews as necessary. The levels of the members should be such that quick problem resolution is assured.

- Software Control Board

  The Software Control Board consists of lead contractor personnel having the responsibility to provide direct management over the software during the software development phase. The Board will review current status, requirements changes, and implementation problems. It will insure that standards and guidelines directed by the Software Review Board are implemented.

- Configuration Control Board

  The Configuration Control Board is placed into being at the software acceptance test milestone (FACI). It consists of members of the Software Review and Control Boards which cease to operate at this time. This Board will assess, analyze, and approve new requirements. It will be the controlling body for the control of the configuration of all operational and support software at each site.

- Software Reviews

  Adequate technical and management reviews must be conducted during the software development cycle to assure that development is proceeding in a satisfactory manner. These reviews will be conducted through the auspices of the Software Review Board. Types of reviews include Preliminary Design Reviews (PDRs), Critical Design Reviews (CDRs), First Article Configuration Inspection (FACI), software delivery reviews, and software standards reviews. Adherance to the integrated development approach will assure project visibility at any time during the development process.

- Higher Order Language

  A higher order language (HOL) is a programmer tool that permits coding in English language statements. A HOL improves program productivity by making it easier to express problems. The resulting product is easier to use, easier to maintain, and more reliable than assembly language code. A higher order language should be selected and used for Spacelab.

- User Language

  A user language is a higher order language defined in terms of a particular discipline. It is oriented to the non-programmer. Spacelab requires a language for the test engineers, principle investigators, and crew members that give them access to the onboard and ground computer systems through that command and display terminals without going through the software development cycle.

- Simulation

  Simulation is the process of reproducing the functional requirements of an operational environment with software and hardware. Simulation is employed to debug, verify and validate software prior to actual operations. The results are programs that execute correctly the first time. Multiple level simulators should be employed in the development of Spacelab software.

- Automated Management System

  Experience has proved the worth of an automated management system during the development as well as operational phases. Spacelab is a multinational, multicontractor endeavor lending itself to all of the advantages of a real time, interactive

management system. An automated management system should be implemented to provide program status, program revision levels, requirements change status, program delivery milestones, program defect status and other information required to properly manage this set of many complex systems encompassing onboard software, support software, software support facilities software, etc.

- Computer Selection

  Selection of an onboard computer must be made which will adequately meet the requirements of the Spacelab program as it is defined and as it is sure to evolve. The computer is of prime importance in developing cost effective, flexible software systems. The computer should have design reserves of approximately 50 percent in both computation speed and memory capacity. It should contain floating point arithmetic. Instruction set compatibility with the ground host computer is highly desirable and allows utilization of support software which already exists. Commonality results in significant simulation savings because an interpretive simulator is not required. Commonality reduces the programmer training time. Selection criteria for the onboard computer should consider the magnitude of available support software such as compiler, simulators, program testing aids, etc., to minimize the support software that must be developed.

- Support Software

  Adequate support software is mandatory for cost effective development of software systems and must be defined and developed in the initial development phase. Proper computer selection will serve to minimize the amount of support software that must be developed. Support software consists of those software packages/capabilities which are utilized by the programmer. Included in this category are assemblers, compilers, linkage editors, program library maintenance, flowcharts, source code analyzers, data analyzers, program traces and snap/dump, and automatic simulator event sequencer. The higher order language compiler is of extreme importance due to the large cost associated with developing one.

- Software Development Facilities

  It is imperative that proper software development facilities are available throughout the software development cycle. Only through utilization of these types of facilities can highly reliable software be developed within critical cost and schedule requirements. The types of facilities recommended for Spacelab are the Program Development Facility, the Integration Facility, and the Prototype Facility.

REFERENCES

Apollo Configuration Management Manual NHB 8040.2. National Aeronautics and Space Administration, Washington, D. C., January 1970.

Baker, F. T., "Chief Programmer Team Management of Production Programming." IBM System Journal, Vol. 11, No. 1, 1972.

Baker, F. T., Mills, H. D., Chief Programmer Teams. IBM, Gaithersburg, Maryland, 1973.

Barnett, T. O. and Constantine, E. L., Editors, "Proceedings of the National Modular Programming Symposium." (Pre-print) Information and Systems Press, Cambridge, Massachusetts, July 23-24, 1968.

Brief Survey of Languages used for System Implementation. IBM Gaithersburg, 1971.

Computer Software Development Plan, MBB, Doc. No. PL-A40000-0210, 1973.

Dillon, T. J.; Jacobs, J. H., "Interactive Saturn Flight Program Simulator." IBM System Journal, Vol. IX, No. 2, 1970.

Flight Program Checkout Facility Specification Document, Contract NAS8-14000, MSFC-DRL-008A, Marshall Space Flight Center, Huntsville, Alabama.

Glans, Thomas, B.; Grad, Burton; Holstein, David; Meyers, William E.; Schmidt, Richard N.; Management Systems; New York: Holt, Rinehart and Winston Inc., 1968.

Ground Operations Aerospace Language, NASA, 1972.

Hamilton, M., and Zeldin, S., Top-Down, Bottom-Up Structured Programming and Program Structuring, Revision 1, The Charles Stark Draper Laboratory, MIT, Cambridge, Massachusetts, December 1972.

Hertel, H. F. and Stanley, W. I., The Performance Measurement and Analysis of System/360 Multiprogrammed Systems. IBM, Houston, 1966.

Hetzel, W. C., Program Test Methods. Englewood Cliffs; Prentice-Hall Inc., 1973.

Higher-Order Language Study for Avionics Programming, Technical Report AFAL-TR-71-154, IBM Owego, 1971.

"HIPO: Design Aid and Documentation Tool," SR20-9413, IBM Corporation 1973.

Hoskins, J. F., Space Shuttle Software Management Plan," IBM No. 73-55-732, IBM Houston, September 1973.

Hughes, J. S., "Space Shuttle Software Development Plan," IBM No. 73-C04-001, IBM Houston, January 1973 (Rough Draft).

Hughes J. S. and Witzel, T. H., Flight Software Development Laboratory, IBM No. 69-226-0036, IBM Huntsville, Alabama, 1969.

IMS Software Study for Manned Shuttle Payloads, Volumes I and II Technical Appendices. Information Management System Steering Group, Ad Hoc Committee on Software, NASA MSFC.

Lord, D. R., "Spacelab Guidelines and Constraints for Program Definition, Level 1", NASA No. MF-73-1, March 1973.

McHenry, R. C., Management Concepts for Top-Down Structured Programming, IBM No. FSC 73-0001, Gaithersburg, Maryland, February 1973.

Metzer, P. W., Managing a Programming Project, Prentice-Hall Inc., 1973.

Mills, H. D., "Mathematical Foundations for Structured Programming," IBM FSD, Gaithersburg, Maryland, FSC 72-601, February 1972.

Mills, H. D., "Top-Down Programming in Large Systems," Courant Computer Science Symposium, July 1, 1970.

Myers, G. J., "Composite Design: The Design of Modular Programs," IBM SDD Poughkeepsie, New York, TR00.2406, February 1972.

Newbold, P. M.; Helmers, C. T., Jr.; Meuse, L. A., "HAL/S Language Specification," Intermetrics Incorporated, Cambridge, Massachusetts, September 1973.

Program Verification Plan for the Skylab I Flight Program, Contract NAS8-14000, MSFC-DRL-008A, Line Item 244, 1972.

Schoderbeck, Peter P., Management Systems: New York, John Wiley & Sons, Inc., 1967.

Spacelab Phase B2 Report, Volume II Programme Definition Part 2 Baseline Programme. ERNO VFW-Fokker, July 31, 1973.

Spacelab Phase B2 Report, Volume II Programme Definition Part 2 Baseline Programme. Messerschmitt-Bolkow-Blohm-GMHB, July 31, 1973.

Spacelab Programme System Requirements for Project Definition, ESRO, Nordwijk, The Netherlands.

Space Station Program Development Definition, Volume III Software Requirements Document. MSFC-DRL-160, Line Item 18. McDonnell-Douglas Astronautics Company-West, July 1970. Huntington Beach, California.

Trauboth, H., "Proposal for Hierarchical Description of Software Systems," NASA Technical Note: NASA TN D-7200. George C. Marshall Space Flight Center, National Aeronautics and Space Administration, Washington, D. C., March 1973.

GLOSSARY

ATMDC	Apollo Telescope Mount Digital Computer
CPT	Chief Programmer Team
DDAS	Digital Data Acquisition System
FCDD	Flight Computer Data Device
FPDD	Four Pi Data Device
GOAL	Ground Operations Aerospace Language
HAL	Houston Aerospace Language
HIPO	Hierarchy Input Output Processing
HOL	High Order Language
IC	Instruction Counter
IF	Integration Facility
IO	Input Output
LCC	Launch Computer Complex
LSI	Large System Integration
LVDA	Launch Vehicle Data Adapter
LVDC	Launch Vehicle Digital Computer
MDI	Mobile Launcher Discrete Input
MDO	Mobile Launcher Discrete Output
MIS	Management Information System
MLC	Mobile Launcher Computer
PCR	Program Change Request
PDF	Program Development Facility
PF	Prototype Facility
PRN	Program Release Notice
PTR	Program Trouble Report
SP	Structured Programming
SS	Switch Selector
STCR	System Tape Configuration Report

SVC	Supervisory Call
TM	Telemetry