AD-A090 159 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
TECHNIQUES AVAILABLE FOR IMPROVING THE MAINTAINABILITY OF DOD W--ETC(U)
JUN 80 R D PILCHER

UNCLASSIFIED NL

1 OF 2
ADA
090159

LEVEL

(2)

AD A090159

# NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC
OCT 14 1980
C

# THESIS

TECHNIQUES AVAILABLE FOR IMPROVING THE MAINTAINABILITY OF DOD WEAPON SYSTEM SOFTWARE,

by

(10) Russell [redacted]/Pilcher

(11) Jun[redacted]980

Thesis Advisor: N. F. Schneidewind

Approved for public release; distribution unlimited

DDC FILE COPY

80 2 007

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| | | |
|---|---|---|
| **REPORT DOCUMENTATION PAGE** | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO.<br>AD-A090 159 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>Techniques Available for Improving the Maintainability of DoD Weapon System Software | | 5. TYPE OF REPORT & PERIOD COVERED<br>Master's Thesis;<br>June 1980 |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Russell Dean Pilcher | | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS ✓<br>Naval Postgraduate School<br>Monterey, California 93940 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Naval Postgraduate School<br>Monterey, California 93940 | | 12. REPORT DATE<br>June 1980 |
| | | 13. NUMBER OF PAGES<br>184 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)<br>Naval Postgraduate School<br>Monterey, California 93940 | | 15. SECURITY CLASS. (of this report)<br>Unclassified |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Maintenance; DoD Software; Tactical Software; Software Management; Software Quality

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Problems associated with the production and operational support of DoD weapon system software are examined. Emphasis is placed on identifying techniques that are currently available for improving the maintainability of this software. A discussion of the software life cycle, structured programming methodologies, use of high order languages, and documentation requirements for software is included with a review of applicable DoD policies. Among the conclusions is that there exists a critical need to recognize maintainability as a primary design objective for DoD weapon system software.

Accession For
NTIS GRA&I ☑
DTIC TAB □
Dist
A

Approved for public release; distribution unlimited

# Techniques Available for Improving the Maintainability of DoD Weapon System Software

by

Russell D. Pilcher
Major, United States Marine Corps
B.S., Utah State University, 1969

Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1980

Author: Russell D. Pilcher

Approved by:

N. F. Schneidewind
Thesis Advisor

Melvin B. Kline
Second Reader

Chairman, Department of Computer Science

D A Schrady
Dean of Information and Policy Sciences

# ABSTRACT

Problems associated with the production and operational support of DoD weapon system software are examined. Emphasis is placed on identifying techniques that are currently available for improving the maintainability of this software. A discussion of the software life cycle, structured programming methodologies, use of high order languages, and documentation requirements for software is included with a review of applicable DoD policies. Among the conclusions is that there exists a critical need to recognize maintainability as a primary design objective for DoD weapon system software.

# TABLE OF CONTENTS

# I. INTRODUCTION

## A. NEED FOR IMPROVED SOFTWARE MAINTENANCE

On 9 November 1979 the North American Air Defense Command Headquarters in Colorado received an alert of a Soviet missile attack [1]. Fortunately, within 6 minutes it was determined to be an apparent computer malfunction but not before 10 U.S. and Canadian interceptors took off from their bases. While not triggering the nuclear holocaust that looms over the modern world, such an event at least shatters the confidence of many individuals in Department of Defense (DoD) computer systems.

Articles, such as the one appearing in the San Francisco Sunday Examiner [2], which highlight a wide variety of large scale, expensive DoD computer system failures and refer to Federal Computer Systems as a "multi-billion-dollar quagmire" do little to convince the public that DoD personnel are capable of designing, developing or maintaining complex computer systems.

Ample examples illustrate that software in DoD computer systems is the main culprit behind these highly visible failures. Since it appears unlikely that complex, weapon system software will be produced error-free in the foreseeable future, the maintenance of this software takes on a critically important role.

Besides the ramifications that non-maintainable software brings, the cost associated with the software life cycle is cause for increasingly serious concern. In fact, a Defense Science Board Task Force on Technology Base Strategy [3], composed of members from industry, medicine, government and universities, concluded that the cost of software has become a national problem and is of particular concern to DoD.

When costs associated with weapon system software are more closely analyzed, it is found that maintenance activities account for a large percentage. The Rome Air Development Center gives the figure of up to seventy percent [4]. Actual projects can be used for illustration. For instance, SAGE, a military defense system, had an average software maintenance cost of approximately 20 million dollars per year after 10 years of operation, compared to an initial development cost of 250 million dollars [5]. De Roze [6] explains that Air Force Avionics software costs around $75 per instruction to develop, but the maintenance for this software costs around $4,000 per instruction.

These large percentages for software maintenance costs can be confirmed by examples from industry. Mills [7] points out "in only 25 years 75 percent of data processing personnel are already taken up with maintenance, not development." On the IBM operating system, IBM 360 OS, approximately four times as much time was spent on maintenance as on development [8]. Boehm [9] reports that "a recent analysis of software activities at General Motors indicated that about 75 percent of GM's Software effort goes into maintenance, and that GM is fairly typical in this respect of industry at large."

There are indications that maintenance problems are compounded for real-time system software. Daly [10], for example, found that programmers were able to maintain only one-fourth to one-third as many instructions of on-line, real-time programs as other type software.

The study of software maintenance becomes so important because of the need to keep DoD real-time, weapon system software operating as error-free as possible and the need to check the escalating cost associated with modifying this software that the study of software maintenance becomes so important.

The software associated with the U. S. Navy's new TRIDENT class submarine, known as the TRIDENT Command and Control System (TRIDENT CCS), is a current, real-time weapon system software project that provides an interesting and beneficial example for illustrating the need for weapon system software maintenance activities.

The original source code was written in the Navy's high order language (HOL), CMS-2. Even though this code was generated by highly experienced software engineers and, according to Oxman [11], "was of a very high caliber and quality", the maintainability of the CCS software has become a matter of concern. In part, this is a result of the way software errors found during the integration test and evaluation stages were corrected. Logic fixes were applied directly via the object code rather than by using the source code. Now the TRIDENT CCS has over thirty-five thousand words of object level only code. An effort is currently underway to improve the maintainability of the TRIDENT CCS.

## B. PURPOSE AND APPROACH

The purpose of this thesis is to evaluate available maintenance techniques that are applicable for use with DoD weapon system software such as the TRIDENT CCS. This evaluation is based upon the current state of the art as discussed in the technical literature and existing DoD policies. Where possible, actual TRIDENT CCS software has been used to provide a realistic example for comparing various maintenance techniques.

The approach used will be to present in the next chapter, Chapter II, a discussion of the overall software life cycle illustrating the relationship maintenance has to the various life cycle phases. Software life cycle management methodologies useful for obtaining improved software maintainability will be incorporated, such as the use of design reviews and configuration management. Some significant differences between software and hardware acquisition will also be included.

Chapter III covers the techniques that must be applied during the development phase of the software life cycle, for obtaining more maintainable software, specifically, the use of structured programming methodologies, use of high order languages, and automated aids.

Chapter IV addresses the important issue of software documentation. A full set of applicable DoD documents used to support the maintenance of weapon system software is identified. Emphasis, however, is placed on comparing those techniques that are currently available for representing the program logic to the maintenance programmer: flowcharts, hierarchy plus input-process-output (HIPO) diagrams, decision tables. Nassi-Shneiderman charts, and program listings.

Chapter V concerns specific software maintenance policies within DoD. This includes an identification of the current directives, instructions, and standards that impact on weapon system software maintenance; the results from a limited survey of some DoD organizations that are involved in software maintenance activities; and trends that exist for research in the area of software maintenance technology.

Finally, Chapter VI contains conclusions and recommendations.

## C. DEFINITIONS

Before any further discussion, exactly what is meant by the term "software maintainability" should be made clear. Unfortunately, there is no universally accepted definition; therefore, some perceptions from various authors will be presented.

Myers [5] lists maintainability as one of ten major categories of software objectives: generality, human factors, adaptability, maintainability, security, documentation, product cost, schedule, efficiency and reliability. It is important to understand the relationships among these categories so that appropriate tradeoffs can be made during the process of software development. He explains that maintainability and adaptability are closely related and that both are compatible with obtaining software reliability. The definition presented for "maintainability" is that it "is a measure of the cost and time required to fix software errors in an operational system." The associated term, "adaptability", is defined as "a measure for the ease of extending the product, such as adding new user functions to the product."

More formalized definitions are offered by Tausworthe [12]:

> Maintenance: alterations to software during the post delivery period in the form of sustaining engineering or modification not requiring a reinitiation of the software development cycle.

Sustaining Engineering: Software related activities in the post-delivery period, principally supportive in form, which keep that software operational within its functional specifications. . . The holding or keeping of software in a state of efficiency or validity despite interface fluctuations in system, subsystem or applications capabilities.

Adaptation: Modification of existing software in order that it may be used as a module in a program development, as opposed to developing another module for that same purpose.

Modification: The process of altering a program and its specification so as to perform either a new task or a different but similar task. In all cases, the functional scope of a program under modification changes.

Figure 1-1 [13] is a chart that brings many of these similar terms together as they are related to the more general concept of software quality. It illustrates what attributes are associated with each of three factors of software quality (operation, revision, and transition). Notice that maintainability is listed as an attribute associated with product revision.

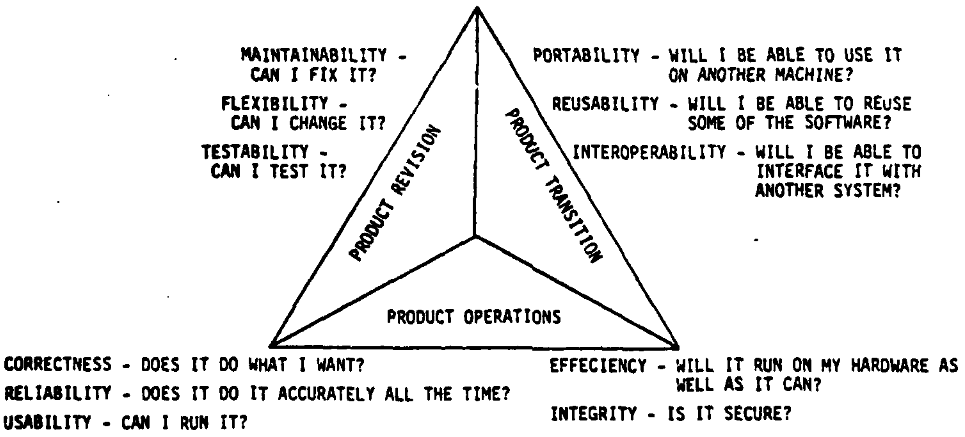

MAINTAINABILITY - CAN I FIX IT?
FLEXIBILITY - CAN I CHANGE IT?
TESTABILITY - CAN I TEST IT?
PRODUCT REVISION
PORTABILITY - WILL I BE ABLE TO USE IT ON ANOTHER MACHINE?
REUSABILITY - WILL I BE ABLE TO REUSE SOME OF THE SOFTWARE?
INTEROPERABILITY - WILL I BE ABLE TO INTERFACE IT WITH ANOTHER SYSTEM?
PRODUCT TRANSITION
PRODUCT OPERATIONS
CORRECTNESS - DOES IT DO WHAT I WANT?
RELIABILITY - DOES IT DO IT ACCURATELY ALL THE TIME?
USABILITY - CAN I RUN IT?
EFFECIENCY - WILL IT RUN ON MY HARDWARE AS WELL AS IT CAN?
INTEGRITY - IS IT SECURE?

Figure 1-1. Software Quality [13]

Yet another attempt to provide a relationship among the various factors in quality software is given in Figure 1-2 [14]. The factors are categorized into two classes: (1) measurement of what is quality and (2) control over software production to ensure that quality is obtained. Note that maintenance falls under flexibility which in turn falls under the measurement of what is quality.

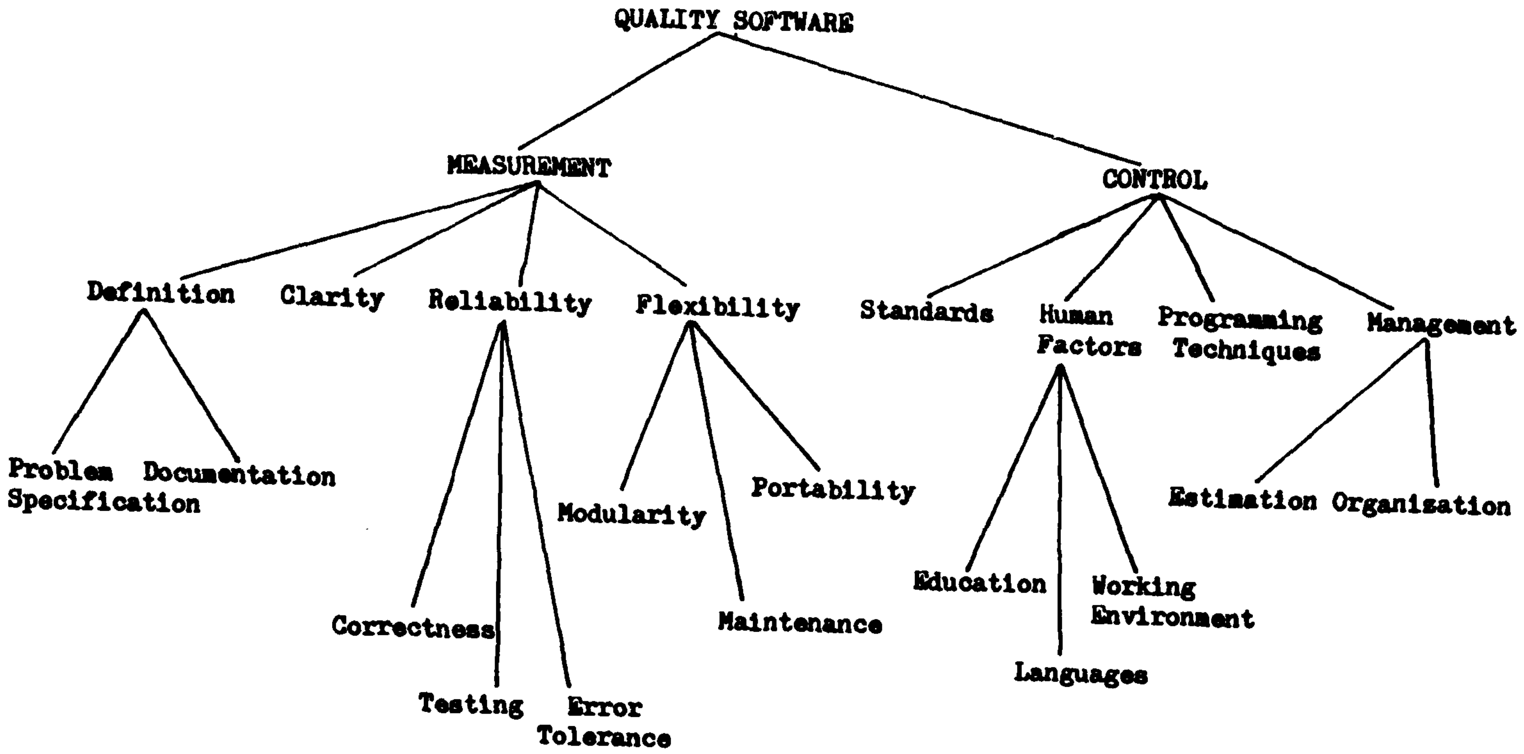


Figure 1-2. The Quality Software Tree [14]

Swanson [15] has attempted to provide a basis for an understanding of the "dimensionality" of the maintenance problem. He feels it is important to distinguish between types of software maintenance activities. He categorizes maintenance into three major types: corrective maintenance, adaptive maintenance, and perfective maintenance. Corrective maintenance is performed in response to failures such as the abnormal termination of a program or the failure in meeting performance criteria. Adaptive maintenance is performed in response to changes in environments such as the installation of a new generation of system hardware. Perfective maintenance is performed to make the program a more perfect design implementation such as to improve processing efficiency or to add new features.

It is interesting to note that there are proponents for dropping the terminology "software maintenance" altogether. The EDP Analyzer [16] suggests a better name for "maintenance" type activities would be "production programming." The contention being this would help alleviate the stigma that maintenance is technician level rather than professional level work. Kline [17] argues that misconceptions about software reliability and maintainability have been, to some extent, due to inappropriate terminology. In order to minimize confusion with hardware maintainability, he suggests replacing the term "software maintainability" with the more descriptive term "software configuration management."

It is evident that no standard terminology exists for this area. Rather than pursue the search for even more definitions it will simply be stated that software maintainability, as used in this thesis, will refer to the degree a software product facilitates updating to satisfy new requirements or modification to correct mistakes (adapted from [4]).

The tools and techniques that currently exist for producing more maintainable software are addressed next. Throughout the remaining chapters it should be kept in mind that, while specifically addressing software maintenance, the principles presented are generally applicable to the many other nuances of successfully accommodating changes to software (e.g., portability, flexibility, adaptability).

Also, it is extremely important to be aware that there are a variety of parameters which can be used to measure the quality of a software product, as the previous discussion has illustrated. An attempt to optimize one parameter is often at the expense of other parameters. For example, optimizing the maintainability of software may be at the expense of development schedule or, conversely, and what appears to have been a common pitfall of past projects, to optimize development schedule may be at the expense of subsequent maintainability. These opposing objectives must be understood and appreciated by all levels of management before tradeoff decisions are made.

# II. THE SOFTWARE LIFE CYCLE

## A. SOFTWARE LIFE CYCLE MODELS

The first step in studying techniques associated with maintainability of weapon system software is to examine all the phases through which software transitions prior to and including the operational point where maintenance is performed. This is commonly called the software life cycle. It is important that this is understood, because the decisions made throughout the earlier phases will ultimately affect the software's maintainability. Unfortunately, as opposed to hardware, there is no universal agreement on the phases of the software life cycle, with well-defined boundaries, so several models will be discussed in order to provide a broader understanding.

The first software life cycle model discussed will be one proposed by Manley [18]. This model is only a slight modification of the already well-understood DoD system life cycle, as presented in DOD INST 5000.1, and as shown in Figure 2-1.

One advantage of using this model is that the terminology appearing in existing DoD documents need not be replaced but simply modified. A disadvantage is that it does little to illustrate the interrelationships that exist among the various phases.

An interesting conclusion reached in Manley's report is that one software life cycle model applies equally to all types of software. This includes both weapon system software as well as automated data processing software. The report recommends that further research be conducted in order to add conceptual detail to the individual life cycle subphases and further recommends that research efforts should be concentrated on the support phase where maintenance is performed.

| DEFENSE SYSTEM LIFE CYCLE MAJOR PHASE | SOFTWARE LIFE CYCLE SUBPHASE |
|---|---|
| Conceptual | Requirements Definition |
| | Requirements Validation |
| Validation | Validation |
| Full-Scale Development | Full-Scale Development |
| Production | Production |
| Deployment | Debugging |
| | Fine tuning |
| Support | Maintenance |
| | Modification |

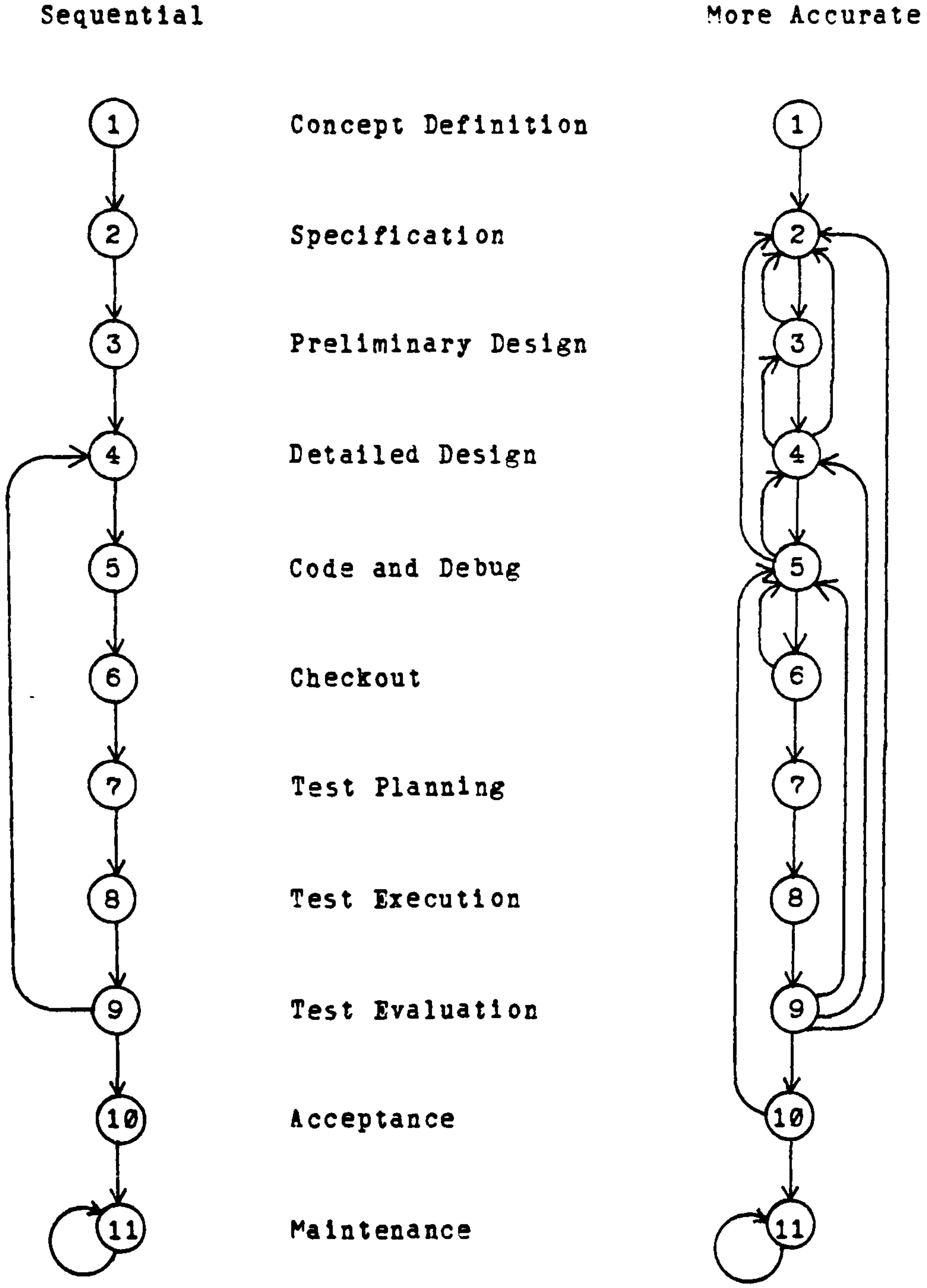
Figure 2-1. Software Life Cycle Model [18]

Brown [19] provides a good contrast of two views of the software life cycle. One view as a fixed sequence of the following events and the other, more accurate view, as a complex and highly dynamic interaction of the following events (see Figure 2-2):

1. Concept ( Requirements ) Definition
2. Detailed Requirements Specification
3. Preliminary Design
4. Detailed Design
5. Code and Debug
6. Checkout
7. Test planning
8. Test execution
9. Test evaluation
10. Acceptance and Use
11. Maintenance (Modification) and Re-test

While Figure 2-2 represents the interrelationships among the phases of the software life cycle, it overly simplifies the importance of the maintenance phase (node 11). This bottom loop really illustrates what should be considered as a mini-life cycle which would include many of the same phases and interrelationships shown by the previous nodes.

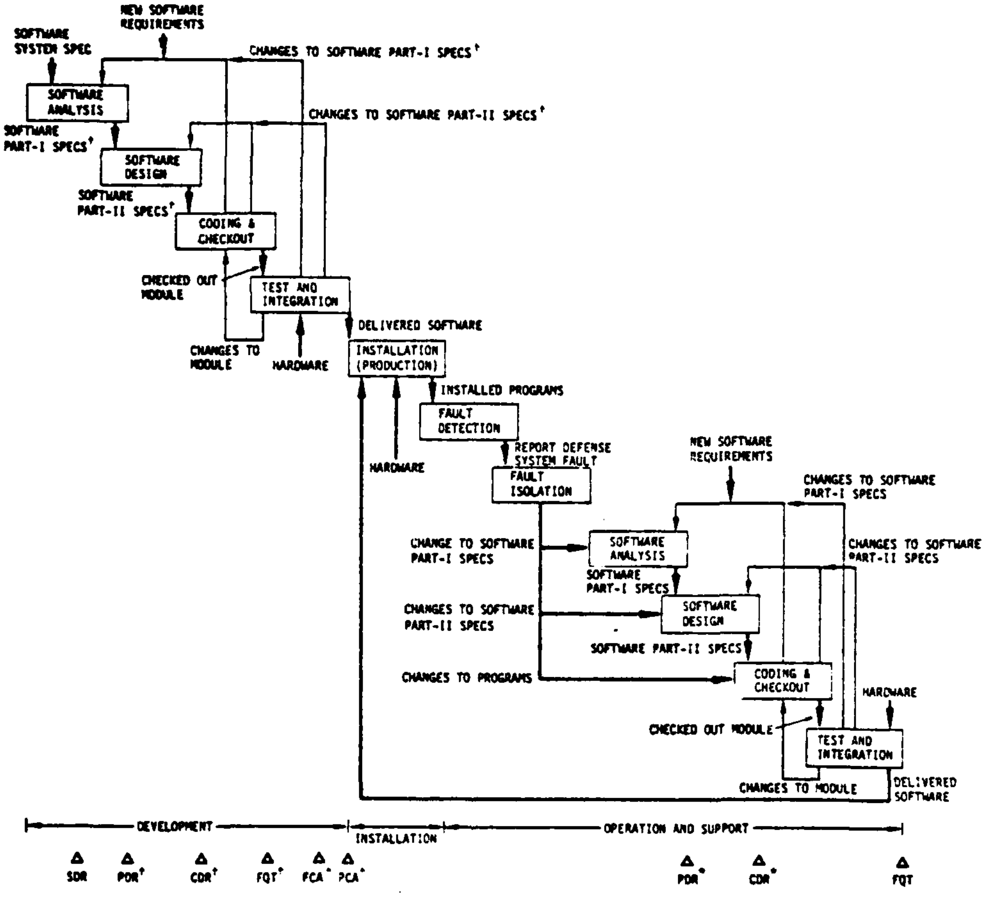
Figure 2-2. 'Sequential' View and a 'More Accurate' View of Software Production [19]



McHenry [20] describes weapon system software life cycle management from a contractor's perspective. He states that today's procurement processes still use the traditional life cycle model consisting of the sequential steps of "define, design, develop, integrate, test, and operate." After evaluating four different procurement strategies being used for the procurement of weapon system software today, he concludes that this is not a satisfactory way to envision or to manage the software development process. The deployment and operation phases of the software life cycle, where maintenance becomes a key issue, are said to be often overlooked or neglected because of the pressures and crises which occur during the development phases. To compound this problem, there is a tendency to apply low skill persons to "maintenance" tasks.

He recommends more emphasis be placed on software design so that the product is less costly to maintain and advocates the use of, what he terms, readiness management (planning for change) by doing such things as conducting exercises where simulated modifications occur.

The software life cycle model described by the Rome Air Development Center [4] seems to accurately model the software life cycle (Figure 2-3).



* MAY NOT BE FORMALLY HELD

† EACH CPCI HAS A SEPARATE SET OF SPECIFICATIONS AND SEPARATE REVIEWS

Figure 2-3. Software Life Cycle [4]

Figure 2-3 shows that the process of software development is highly interactive, as indicated by the feedback arrows to accommodate new software requirements and changes to software specifications. More significantly, it highlights the importance of the operation and support phase where maintenance is performed through a series of subphases. Note that these subphases incorporate the same interactive steps shown for software development: software analysis, software design, coding and checkout, and test and integration.

A variety of models have been presented in an effort to better understand how maintenance relates to the overall software life cycle. It must be emphasized that even though maintenance appears chronologically last it must be properly considered and thoroughly planned for early in the life cycle.

## B. MANAGING THE SOFTWARE LIFE CYCLE

### 1. General

Now that a conceptual framework has been presented for envisioning the life cycle of software and highlighting the importance of the phase where maintenance is performed, attention is turned to software management considerations. This is important because the decisions made by managers of weapon system software projects will often mean the difference between whether the final product is maintainable or non-maintainable.

There has been some argument that regardless of what management techniques are employed, successful development of large, complex software projects is not always possible. For example, an Air Force assessment [21] of why its large, complex computer system, the Advanced Logistics System (ALS), failed concluded that "...the ALS is beyond the software state-of-the-art."

This view is contrasted to one offered by Cave [22]. In an article which describes project management methods used for controlling the life cycle of large-scale software systems, he states "...project failures are generally the result of improper or inexperienced management and not the lack of technical ability." The article goes on to conclude that successful development of large software systems can be achieved in a consistent manner.

This thesis is based on the premise that Cave's view is correct. It further assumes that software maintenance problems can be largely avoided if knowledgeable project management is applied.

Cooper [23] explains that, in the past, one of the common pitfalls in project management has been that it was development-oriented and, therefore, management attempted to optimize the development process in trying to meet budget and schedule constraints. This tends to create an initial design with little documentation, resulting in increased difficulty in maintaining the software and a corresponding increase in overall life cycle costs.

Another problem with management´s ability to produce maintainable software identified by Cooper was that high level decision makers lack computer-related experience. This, undoubtedly, results from the fact that, as a discipline, software management is still in its infancy.

While there is no simple series of steps for managers to follow which will ensure successful development of maintainable software, experience has revealed some general policies that appear to help. For example, Daly [10] has reported on his experience in managing developments. Table II-1 provides a comparison of two approaches. Method 1 is the preferred approach to producing a more cost-effective, more maintainable software product. Note that he recommends the application of strict management objectives to guide development.

Table II-1. Software Design Methods [10]

| Method 1 | Method 2 |
|---|---|
| High level language | Assembly language |
| Structured Code | Tight Complex Code |
| Composite design (hierarchy of small segments) | Large blobs of code |
| Parallel, top-down, bottom up design all optionally used | Bottom-up design |
| Simple data structures and work areas (not) tightly packed | Tight, efficient, data structures and work areas (every bit used, no data duplicated) |
| Team approach to design (egoless programming) | One program - One man concept |
| IMB's structured walk through for reviewing detail design and code | No detailed technical review of design or code |
| Three separate teams one team design, one tests one evaluates | Original coder tests, integrates and helps evaluate his program |
| Complete set of hierarchy charts, sequence charts data maps and narratives, well commented listings | Detailed flow charts and general narratives, no consistency listing comments |
| Detailed test plans for all test phases | No formal test plans |
| Program maintained by 30% senior programmers | Program maintained by inexperienced programmers or technicians |
| Only commercial documentation generated during development | Extensive, noncommercial technical memorandum generated and placed in library |
| Strict management objectives established to guide development | No management objectives |

## 2. DoD Management Policies

Within DoD the need for improving weapon system software management has been recognized and action has been initiated. On 3 December 1974 a DoD Software Steering Committee was established with a charter to identify critical weapon system software problems and to recommend policies for their solution.

In support of the first phase, the MITRE Corporation in conjuction with The Applied Physics Laboratory of Johns Hopkins University [24, 25], conducted a study of weapon system software management. The study concluded "The major contributing factor to weapon system problems is the lack of discipline and engineering rigor applied to the weapons system acquisition activities."

Incorporating recommendations from this study, the Software Management Steering Committee formulated a comprehensive plan comprising policy, practice, procedure and technology initiatives. This plan was released in March 1976 and is available through the Defense Technical Information Center [26]. Part III of this plan recommends management policy with the purpose of supplementing principles put forth in DoD Directives 5000.1 and 5000.2. The first management policy listed states, "Ease of maintenance and modification will be a major consideration in the initial design."

The policies provided in this plan have the effect of establishing visibility and management control to weapon system software. Two important techniques used to provide visibility and management control are design reviews and configuration management.

### a. Design Reviews

MIL-STD-1521 (USAF) prescribes the requirements for the conduct of the following technical reviews and audits on computer programs:

Systems Requirements Review (SRR)
System Design Review (SDR)
Preliminary Design Review (PDR)
Critical Design Review (CDR)
Functional Configuration Audit (FCA)
Physical Configuration Audit (PCA)
Formal Qualification Review (FQR)

For detailed definitions and specific requirements for these reviews the reader is referred to the standard. It should be noted that the standard fails to list requirements to be specifically considered for optimizing the maintainability of the software. An available software maintenance guidebook [27] does, however, provide as a supplement to MIL-STD-1521, checklists of maintenance considerations for use with the various reviews and audits.

### b. Configuration Management

The elements of software configuration management are configuration identification, configuration control, configuration status accounting and configuration auditing. Configuration identification involves specifically identifying and labeling the configuration items at selected baselines during the software life cycle. Configuration control provides the means to manage changes to the (software) configuration items and involves three basic ingredients:

> -Documentation (such as administrative forms and supporting technical and administrative material) for formally precipitating and defining a proposed change to a software system.
>
> -An organizational body for formally evaluating and approving or disapproving a proposed change to a software system.
>
> -Procedures for controlling the actual changes to a software system

Software configuration status accounting provides the mechanism for maintaining a record of how the software evolved and where the software is at any current stage of implementation. Software configuration auditing provides a means to determine how well the software product matches its associated documentation.

DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems, states:

> Defense system computer resources, including both computer hardware and computer software will be specified and treated as configuration items.

As part of the proposed requirements assigned to contractors for the development of weapon system software, MIL-STD-1679, Weapon System Software Development, states:

> The contractor shall establish and implement the disciplines of configuration management; namely configuration identification, configuration control, and

> configuration status accounting. The contractor shall be cognizant of the requirement for long-term life-cycle support of the weapon system software. The appropriate degree of configuration management shall be applied to ensure completely accurate correlation between descriptive documentation and the program in order to facilitate post-delivery maintenance by software support personnel.

MIL-STD-52779(AD), Software Quality Assurance Program Requirements, further requires that the contractor provide audits by independent personnel to ensure that the objectives of the configuration control program are being attained.

This need for software configuration management, as reflected in current standards and directives, has been only recently recognized in DoD. Fortunately, it is now accepted as an essential task if software maintenance is to be successfully performed. In fact, as previously mentioned, Kline [17] proposes replacing the term "software maintenance" with the term "software configuration management." This highlights the central role it plays in the maintenance of software.

As Bersoff [28] points out, the problem with configuration management of software in the past has been that it fell under the umbrella of configuration management of the entire system (Figure 2-4). Hardware, being more visible, has been treated in great detail, but software, being less mature as well as less visible from a total system viewpoint, has been largely neglected.

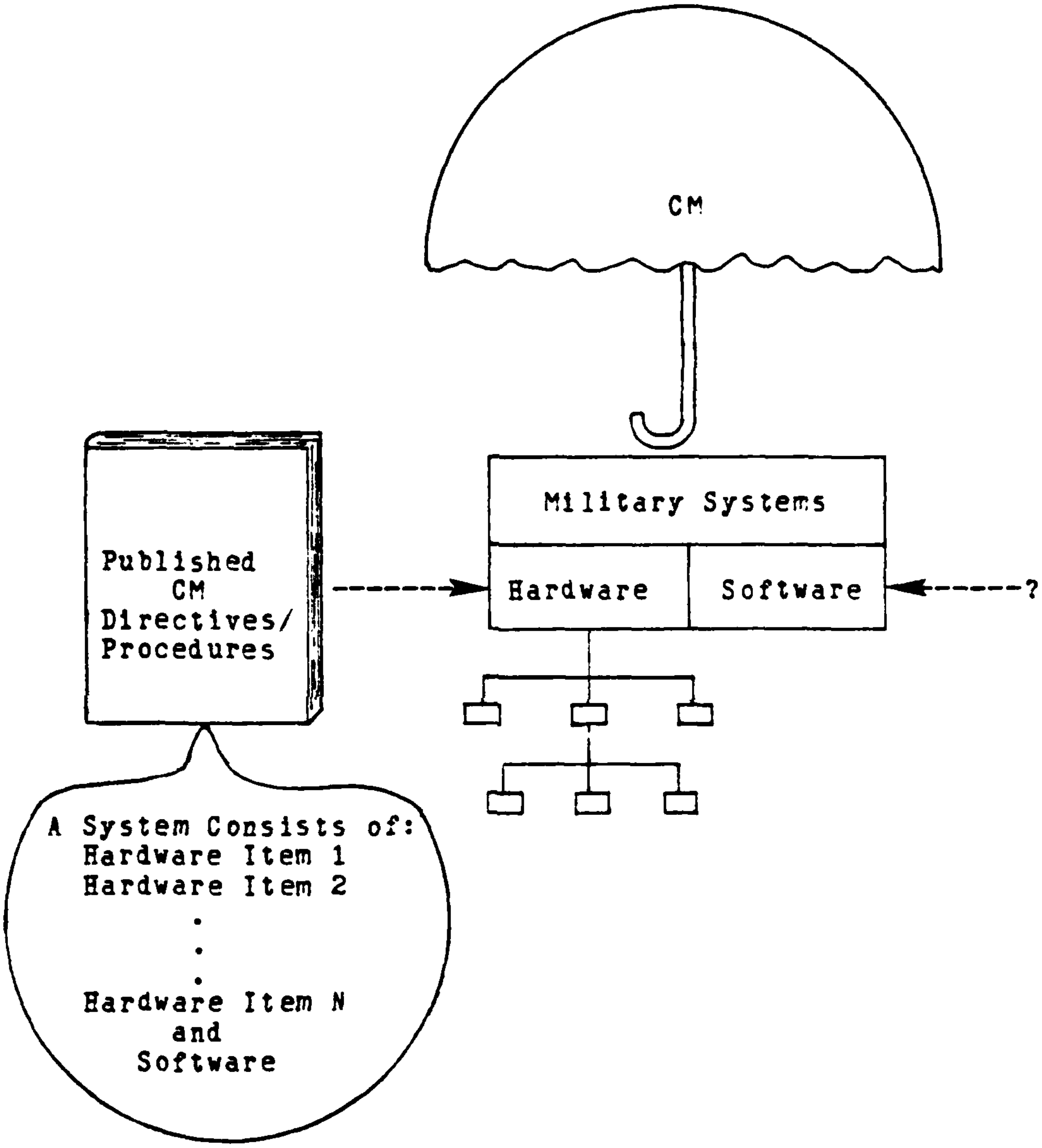


Figure 2-4. Configuration Management Umbrella [28]

There is probably no aspect more important to software maintenance than managing change since software maintenance is really a matter of correctly applying changes. Clearly, software configuration management must be applied to discipline this process. A word of caution, however, is that the same change control procedures do not apply equally to all software projects; therefore, configuration management must be properly tailored to the organization performing maintenance and to the software product itself.

### 3. Software vs Hardware

The theme pervading the evolving initiatives for managing software is to elevate it from an artistic enterprise to a true engineering discipline, or--to put it another way--to treat software more like hardware throughout its complete life cycle [10, 22, 29]. There are, however, differences between software and hardware that merit consideration.

A major difference is in the maintenance requirements. Hardware is maintained primarily by replacement of worn or failed components with new ones meeting the original specification. Software, unlike hardware, requires that the product specification and design be changed when maintenance is performed [20].

Among the differences Schneidewind [30] has pointed out are: (1) the passage of time is an important parameter in predicting hardware failure, but has little significance in predicting software failures and (2) hardware is usually assumed to have a constant failure rate during its operational phase as compared to software's variable failure rate.

Kline [17] has also identified many significant differences between software and hardware in the area of reliability and maintainability. Among his conclusions are that there exist well-established statistical relationships for hardware reliability and maintainability which is not yet the case for software.

Because there are many differences between hardware and software, caution should be applied in using the same techniques which have been successful for developing maintainable hardware to development of maintainable software.

Selectively, however, some hardware management techniques can be successfully employed for improving software. Significant examples are the use of design reviews and configuration management as described in the previous sections.

# III. DEVELOPMENT ISSUES FOR IMPROVED MAINTENANCE

## A. GENERAL

As mentioned in chapter II, decisions made during the development phases of the software life cycle will have a significant impact on how maintainable the software is during its operational phase. There is little disagreement on the observation made by Mills [7] that better development procedures can reduce the need for maintenance. This chapter is concerned with briefly discussing those "better development procedures."

## B. STRUCTURED PROGRAMMING

Structured programming is becoming one of the more promising approaches to reducing the ever increasing cost of producing and maintaining software. Meyers [5] states that structured programming will probably be recorded in history as one of the great steps forward in programming technology.

The Naval Surface Weapons Center [31] and The Naval Air Development Center [32] are two Navy R & D centers that have obtained successful results in producing improved quality weapon system software by using structured programming techniques.

Professor E. W. Dijkstra, of the University of Eindhoven, Netherlands, is credited with being one of the first to advocate structured programming principles with his 1965 paper [33]. Since 1965, many books have been published covering the topic of structured programming [5, 34, 35, 36, 37, 38, 39]. A complete review of these works will not be attempted here, but the following selected items provide a general overview.

As with the term "software maintenance", no specific, widely accepted definition exists for "structured programming." Jensen [40] surveys many definitions and concludes that one proposed by Wirth [41] is the most accurate: "Structured programming is the formulation of programs as hierarchical, nested structures of statements and objects of computation." Meyers [5] gives his favorite definition of structured programming as "the attitude of writing code with the intent of communicating with people instead of machines."

A goal of structured programming is to organize and discipline the program design and coding process in order to reduce logic type errors [8]. Three important characterisitcs of structured programming will serve as the framework for further explanation: top-down design, modular design, and structured coding.

1. Top-down Design

One characteristic of structured programming is the use of top-down design. In a very general sense, this involves first specifying a program in the broadest terms and in a step-wise fashion gradually refining the structure to fill in details. At each step, major functions to be accomplished are identified, a given task is broken into a number of subtasks until the subtasks are simple enough to be coded into modules. If a module requires more than a line or short paragraph to describe, then the module should be redefined.

The rationale behind this approach is that the mind is capable of comprehending only so much at a time and most problems are too large to be attacked all at once.

Top-down design is illustrated in Figure 3-1 [27] where successive levels of design provide additional details of the eventual solution. This approach will provide visibility to the design which is an important need of the maintenance programmer.

Top-down development has been described as perhaps the least appreciated area of modern software technology [42] and includes much more than the simplified description just presented. It is a rich and powerful technique for project implementation and for system integration.

It is interesting to note that an adaptation of the top-down approach, conceived by O'Neill in 1972, was used for the TRIDENT CCS [42, 43, 44]. This was the first time a top-down design was specified for use on a Navy weapon system software development project [25].

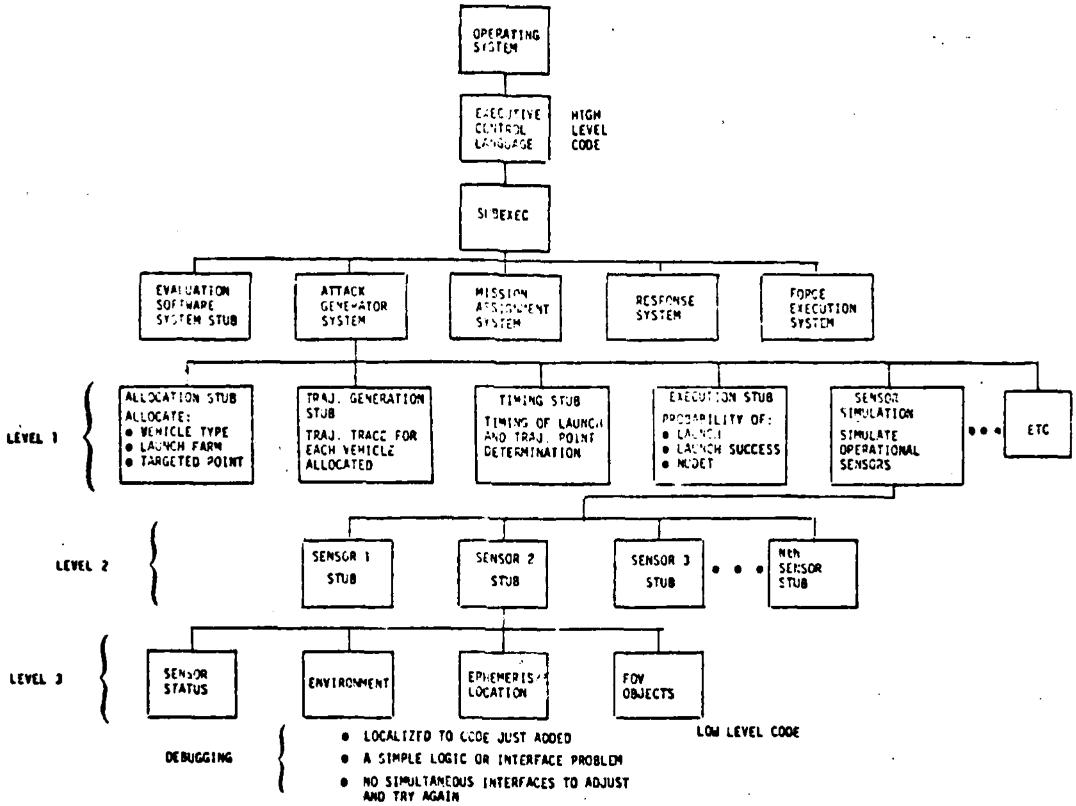


Figure 3-1. Top-down Design [27]

### 2. Modular Design

Another characteristic of structured programming is modular design. A good description of principles and practices for module design is provided by Meyers [5]. The first step, Meyers explains, in designing a module is defining its external characteristics. This is information needed by interfacing modules, nothing more, and includes: module name, function, parameter list, inputs, outputs, and external effects. It is recommended that this information be located in comment statements at the beginning of the source code. Only after defining the module's external characteristics, is design and coding of the internal logic accomplished.

No hard and fast rules exist for what constitutes the optimum size for a module. Van Tassel [8] states as a general rule that modules should contain between 10 and 100 high level language instructions. Meyers [5] gives as a commonly used limit 60 lines of code. The main point is that a module should be easy to keep in mind and comprehend. It should be noted, though, that programs can increase in complexity as the number of modules increases.

A goal in using modules is to reduce complexity, which improves maintainability. Complexity can arise from three sources: functional complexity, distributed complexity and connection complexity. Functional complexity occurs when a module is made to do too many things. Distributed complexity occurs when a common function has not been properly identified and separated, resulting in its being accomplished by many different modules. Connection complexity occurs when modules interact on common data in unexpected ways.

Tausworthe [12] describes two important measures for modularity (originally defined by Meyers [45]): module coupling and module strength. An optimal design for improved maintainability minimizes the relationships between modules (minimal connections) and maximizes relationships among components within each module (maximum strength).

Table III-1 [46] shows the various categories of both module coupling and module strength and ranks these categories from the best situation to the worst.

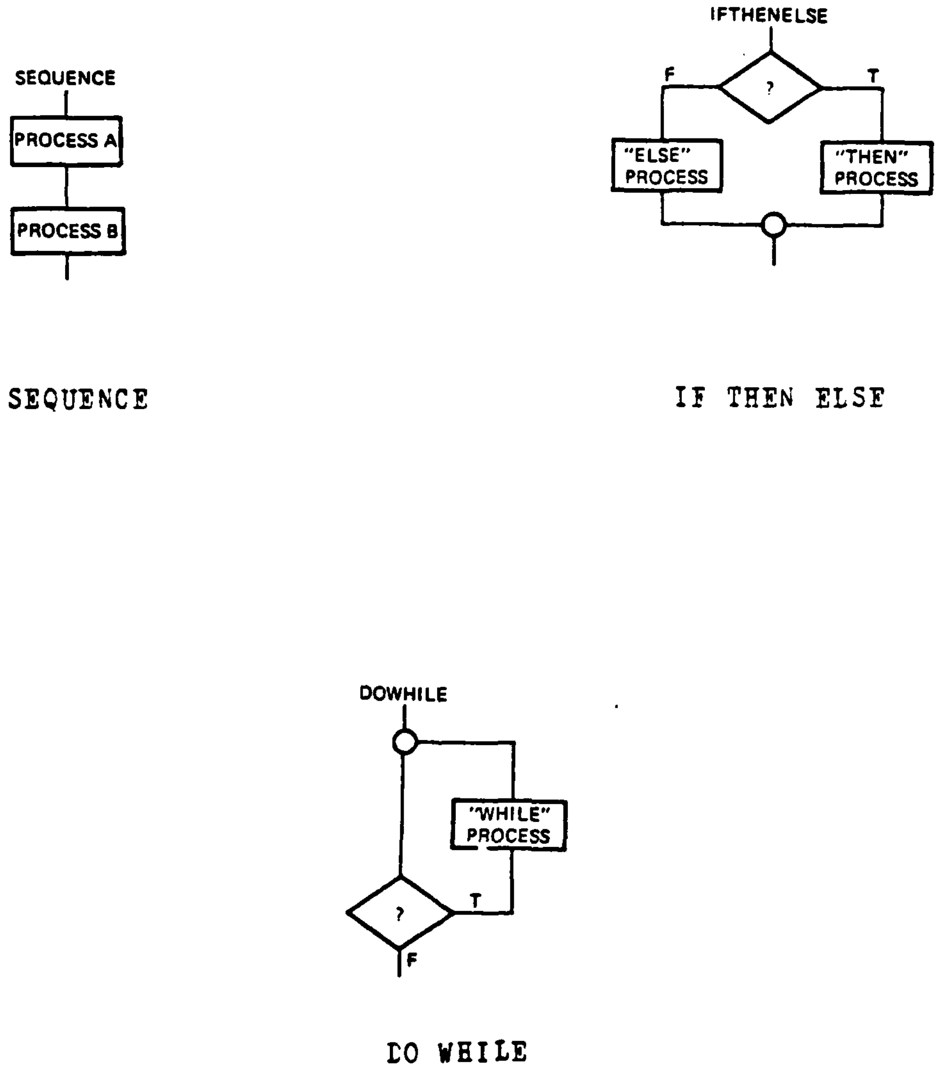
|  |  |
|---|---|
|  | MODULE COUPLING |
| Data: | all communications between them is via arguments that are data elements |
| Stamp: | their communication includes an argument that references a data structure (some of whose fields are not needed) |
| Control: | an argument from one knowingly influences the flow-of-control of the other, e.g., flag |
| External: | they reference an externally declared data element |
| Common: | they reference an externally declared (i.e., common) data structure (some of whose fields are not needed) |
| Content: | one references the contents of the other |
|  | MODULE STRENGTH |
| Functional: | modules perform a single specific function -- "write a record to output file" |
| Clustered: | module is a group of functions sharing a data structure usually to hide its representation from the rest of the system..only one function is performed per invocation--"symbol table with insert and look-up function" |
| Sequential: | module action comprises several functions that pass the data along--"update and write a record" |
| Communicational: | module action consists of several logical functions operating on some data--"print and punch a file" |
| Procedural: | module elements are grouped for algorithmic reasons--"loop body" |
| Temporal: | module functions are all related in time--"initialization" |

Table III-1. Module Characteristics [46]

### 3. Structured Coding

A third characteristic of structured programming is the use of structured coding. Structured coding is a method of writing programs which are more easily understood and maintained. It is based on the fact that arbitrarily large and complex programs can be written using a small set of basic programming structures.

Bohm and Jacopini [47] demonstrated that three basic control structures were sufficient for expressing any flowchartable program logic (Figure 3-2): "sequence", selection ("if then else"), and iteration ("do while"). These three control structures are often expanded to include "do until" and "case" type constructs (Figure 3-3). MIL-STD-1679, for example, limits control structures used in programming to these five basic types.

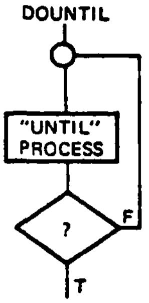


SEQUENCE

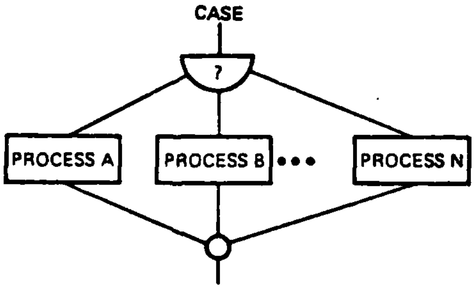
IF THEN ELSE

DO WHILE

Figure 3-2. Basic Control Structures



DO UNTIL



CASE

Figure 3-3. Additional Control Structures

Meyers [5] provides a list of seven basic elements of a structured program which should be applied to help reduce program complexity, promote clarity of thought by the programmer, and enhance readability of the program:

- The code is constructed from sequences of three basic elements.
- Use of the GOTO statement is avoided wherever possible.
- The code is written in an acceptable style (e.g. use meaningful variable names, avoid statement labels, avoid language tricks)
- The code is properly indented on the listing so that breaks in execution sequence can be easily followed (e.g. a DO statement can be easily matched with the statement ending the loop)
- There is only one point of entry and one point of exit in the code for each module.
- The code is physically segmented on the listing to enhance readability. The executable statements for a module should fit on a single page of the listing.
- The code represents a simple and straightforward solution to the problem.

Often, a program is written with a clear structure but is eventually modified by unstructured constructs. Even if a bit exaggerated, Van Tassel [8] offers a graphic illustration showing how a program's original logic can become completely obscured as the need for changes or corrections develops (Figure 3-4). Clearly, the maintenance of such a program would be extremely difficult.

This illustrates the point that not only the initial source code should be structured but subsequent changes to the code must also follow structured constructs. TRIDENT CCS software provides an example of a project that followed a structured development approach but eventually lost some of the benefits of structured programming by application of non-structured techniques (e.g., use of patches) [11].

Unstructured

```
        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
        GOTO label u
label w IF NOT t GOTO label y
        I function
label y IF NOT v GOTO label k
        J function
        GOTO label k
```

Structured

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

Figure 3-4. Examples of Unstructured and Structured Coding [8]

## C. LANGUAGE CONSIDERATIONS

No single development decision affects the maintainability of a program more than choosing what language it will be written in. Some aspects that should influence that choice are discussed in this section.

### 1. High Level vs Assembly Level Language

Hopkins [48], in discussing software quality, made it clear where he stood concerning the use of high level languages when he stated "The higher level the language used in programming the better."

Lang [48] provides a brief list pointing out "the very grave disadvantages of assembly languages:

-Apart from the few who delight in such intricacies, most people find assembly language programs harder to write, read, understand, debug and maintain than high level language programs.

-It provides the poorest conceptual framework for the programmer to express the computing operations he wants performed.

-It is completely machine dependent, thus requiring any machine language program to be completely rewritten when it is transferred to a different machine."

Glass [49] talks about the enormous benefit of programming in high order languages both in terms of productivity and reliability. He points out that high level language code requires many fewer statements than assembly language; thus, there are many fewer chances for errors. Also, the high level language programmer is screened from a whole class of potential error situations related to hardware intricacy since the compiler accomplishes the task of making hardware dependent choices.

To illustrate some advantages in using a high level language vs an assembly level language, a simple algorithm has been coded in both the high level language Pascal (Figure 3-5) and the Intel 8080 assembly language (Figure 3-6). The program is designed to read an integer from a console and maintain a running total; when a "Ø" is presented then the program is to print out the total. Although, most programs are more "complex" than these simple examples, they are helpful in making comparisons between the use of high level language and assembly language. No claim is made concerning the elegance of the solutions or for that matter the utility of their function.

```
Program ADD;
Var Number, Total:Integer;
Begin
  Total:=0;

  Repeat
    Read (Number);
    Total:=Total + Number;
  Until Number = 0;

  Write ('Total= ',Total)
End.
```

Figure 3-5. Integer Addition Program Written In Pascal

```
TOTAL:   DB      Ø
NUMBER:  DB      Ø
         ORG 1ØØH
INIT:    MVI C,6ØH
         MVI B,6ØH
         MVI A,ØØH
         STA TOTAL
START:   INR C
         CALL POSCUR
         CALL READ
         ANI ØFH
         STA NUMBER
         LDA TOTAL
         LXI H,NUMBER
         ADD M
         DAA
         STA TOTAL
         LDA NUMBER
         CPI ØØH
         JZ DISPLY
         JMP START
DISPLY:  CALL POSCUR
         MVI A,´S´
         CALL PRINT
         INR C
         CALL POSCUR
         MVI A,´U´
         CALL PRINT
         INR C
         CALL POSCUR
         MVI A,´M´
         CALL PRINT
         INR C
         CALL POSCUR
         MVI A,´=´
         CALL PRINT
         INR C
         CALL POSCUR
         LDA TOTAL
         RRC
         RRC
         RRC
         RRC
         ANI ØFH
         ORI 3ØH
         CALL PRINT
         INR C
         CALL POSCUR
         LDA TOTAL
         ANI ØFH
         ORI 3ØH
         CALL PRINT
         RST Ø7
POSCUR:  MVI A,ØCH
         CALL PRINT
         MOV A,C
         CALL PRINT
         MOV A,B
         CALL PRINT
         RET
READ:    PUSH B
         PUSH D
         PUSH H
         MVI C,Ø1H
         CALL Ø5H
         POP H
         POP D
         POP B
         RET
PRINT:   PUSH B
         PUSH D
         PUSH H
         PUSH PSW
         MVI C,Ø2H
         MOV E,A
         CALL Ø5H
         POP PSW
         POP H
         POP D
         POP B
         RET
END
```

Figure 3-6. Integer Addition Program Written In Intel 8Ø8Ø Assembly Code

Perhaps the most striking difference is in the program length. For the high level language program only 12 statements were used. This compares with 82 statements for the assembly language program. Another significant difference is in readability. The high level language statements are more English-like (e.g., Begin, End, Repeat, Until, Read, Write) and, hence, more comprehensible, while the assembly language instructions (e.g., LXI, MVI, INR) are generally more abbreviated, requiring increased effort for understanding.

Another notable difference is that the details associated with the hardware interfaces are hidden from the high level language programmer. Items such as memory location of the program, register usage allocation, conversion of ASCII code to binary coded decimal and back again, and cursor control for the terminal display are all items that have to be considered and accounted for in the assembly language program. This increased level of complexity provides significant opportunities for programming errors, thus increasing the difficulty of maintaining the program.

Finally, consider the degree of difficulty that would exist for correcting an error in this simple program or the amount of effort that would be required to add enhancements (e.g., to obtain the average value). Clearly, the high level language program is more suited to this "maintenance" type work.

2. DoD's Use of Eigh Level Language

a. Standard High Level Languages

DoD is taking action to reduce the proliferation of programming languages in an effort to improve the maintainability of future weapon system software and to increase the transfer of available software among new systems [29].

Under DoD Instruction 5000.31, weapon system development programmers are restricted to the use of one of the following high level languages: TACPOL, CMS-2, SPL-1, JOVIAL, FORTRAN, and COBOL.

A continuing effort is underway to standardize even further, to adopt one common high level language. A set of technical requirements for the common language was developed, and during 1976 twenty-three existing languages were evaluated against these requirements. The findings were that no language completely satisfied the requirements, that several languages could be sufficiently modified to produce an acceptable language, and that it would be possible to produce a language that would satisfy essentially all the requirements [50].

b. ADA

DoD has subsequently adopted a common programming language based on the language PASCAL to use as its future high level language for embedded computer software [51]. It has been named ADA, after Ada Augusta who became the first programmer as an assistant to Charles Babbage.

On the surface, it appears that one common programming language for DoD embedded tactical software would greatly improve maintainability through standardization and increased familiarity by a larger number of programmers. Also, a new language could be designed to incorporate the latest language methodologies for improved program clarity.

ADA is not, despite these apparent advantages, universally accepted in its present form. Dijkstra [52], for example, has the opinion "that it is neither complete, nor concise" and expresses concern over its size by pointing out that ADA's reserve word list amounts to "more than ten percent of basic English." Also, he states maintainability is hampered by the multiple ways that exist for doing the same thing.

Regardless of this lack of universal support, the ADA project is going forward and the Army plans to have a compiler ready during 1981. The Navy seems somewhat less aggressive in pursuing this common high level language effort [51, 53].

c. Navy's Use of CMS-2

The Navy is reluctant to accept ADA partially because it has already standardized to CMS-2 which was designed primarily for real-time, command and control applications. It combines features of FORTRAN, COBOL and JOVIAL and has had continuous modifications, corrections and enhancements over several years of actual use. This is contrasted with ADA which is completely new and has had no previous use.

3. Patching

Before leaving the subject of programming languages, the use of patches must be addressed because of their detrimental effect upon software maintainability.

A patch is a change made to the object program after it is assembled or compiled. Patching is generally acknowledged to be a bad programming practice yet it continues to occur. Its use is encouraged by rigid testing schedules since it provides expedient solutions [54].

Both TADSTAND 9 and MIL-STD-1679 limit the total number of patch words to less than 0.005 of the total machine instruction words in the program, but despite such attempts at limiting its use, patching can quickly get out of control. A small sample of the TRIDENT CCS software was taken and found to have five times the current limits allowed by the new MIL-STD-1679. This is one reason why the maintainability of this software has become a matter of concern.

D. AUTOMATED AIDS

There is little disagreement that, in order to produce maintainable software, the development must proceed in an orderly, flexible and measurable manner, with all phases clearly traceable from system requirements to machine readable code.

This entire process is extremely labor intensive and subject to errors of commission and omission. It is not a novel idea to suppose such an effort could benefit from automation. Many automated tools have, in fact, been designed and employed with varying degrees of success.

It is beyond the scope of this thesis to include a comprehensive study of the strengths and weaknesses of such tools, but a few methodologies are presented to serve as examples of this trend because of the significant influence it might have on the way software is maintained in the future.

A problem statement language (PSL) and a problem statement analyzer (PSA) are two tools developed at the University of Michigan to aid systems design. PSL and PSA are used by a number of large commercial organizations. Chase Manhattan Bank is one example and it feels that by using these methodologies, its software is now easier to maintain [55].

TRW, working for the U.S. Army Ballistic Missile Defense Advanced Technology Center has developed a software requirements engineering methodology (SREM) which applies specifically to large, real-time weapon systems [46]. SREM is designed to generate clear and complete requirements and to facilitate their modification. Since incorrect or missing requirements account for a large portion of errors in large software projects, the use of SREM should improve maintainability.

A highly ambitious software development and maintenance support system (SDMSS) is being designed to automate the various activities for large scale software. It is comprised of several subsystems, including requirements engineering, design, documentation, software error management, and maintenance. Reference [56] contains a more complete description of this system.

The source code control system (SCCS) is designed for controlling changes to files of text such as source code and software documentation and aids maintenance efforts considerably. The current version has been operational at Bell Telephone Laboratories since 1977 [57].

A library control program (SYSM) has been developed by Magnavox and is currently being used to control a total of 200,000 lines of code. It aids maintenance by controlling changes in a secure and traceable manner [58].

PSL/PSA, SREM, SDMSS, SCCS, and SYSM are only a limited set of automated tools being developed which will support maintenance activites. DoD must continuously study and evaluate these and similar methodologies for possible applications to its weapon system software.

# IV. DOCUMENTATION FOR MEETING MAINTENANCE REQUIREMENTS

## A. GENERAL

The "Documentation Standards," Volume VII, of IBMs Structured Programming Series [59] states that "documentation in some form should be acquired for all software developed in order to support the future needs of software maintenance." It is obvious that a computer program stored in machine readable form on a media such as tape is not adequate to meet the requirements of the maintenance programmer. The question becomes what type and how much documentation is sufficient. This question must be correctly answered if maintenance activities are going to be successful.

In determining what specific documentation should be produced and maintained concurrently with weapon system programs, some general guidelines should be kept in mind.

First, documentation must provide for complete traceability from the user's operational requirements to the actual lines of code so that if a requirement changes then the appropriate code can be correctly modified, or, conversely, if an error is found in a section of code the full impact on the user's requirements can be determined.

Second, the documentation must be easily modified. As requirements or programs are changed then corresponding changes must be made to the documentation. If this is not done, then the documentation soon becomes outdated. This need for concurrent maintenance of documentation with the software makes those documentation forms that can be computer generated preferred.

Finally, because of the high cost of documentation, the amount produced should be kept to the absolute minimum required. Tausworthe [12] provides a graphic example showing the relationship between program costs and the level of documentation (Figure 4-1). Note that there is an optimum level that must be strived for.

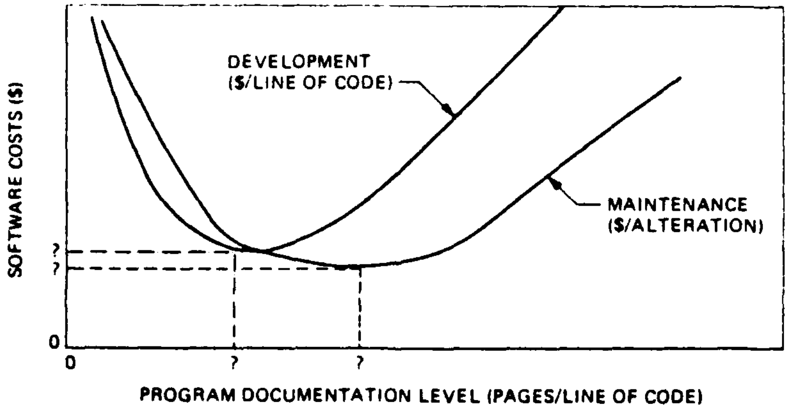


Figure 4-1. Program Costs vs Documentation Level [12]

In this chapter some examples of formal standards are identified which have been developed within DoD concerning the production of documentation for use in the maintenance of software. Also, available forms of documentation are discussed which are specifically used for representing program design, an important need of the maintenance programmer.

## B. MAINTENANCE DOCUMENTATION STANDARDS

A limited set of standards have been developed at various levels within DoD which specify the content and format of documentation to be used to support software maintenance activities. Examples of these are provided in order to demonstrate the nature and extent of these standards.

### 1. Program Maintenance Manual

DOD STANDARD 7935.1-S, "Automated Data Systems Documentation Standards," 13 September 1977, provides guidelines for the development of a Program Maintenance Manual. The purpose of this manual is to provide the maintenance programmer with the information necessary to effectively maintain a system. A copy of the format of the Program Maintenance Manual is given in Appendix A. Note that it is oriented towards documenting data base systems rather than weapon systems.

### 2. Combat System Program Description Documents

SECNAVINST 3562.1 is one of the most complete sets of documentation standards specifically for weapon system software. Within this Navy standard three documents are identified which support the maintenance of tactical software. Categorized under the general heading Combat System Program Description Group, they are called: the Program Description Document (PDD), Data Base Design (DBD), and Program Package (PP). A description of their purpose and a copy of their format is provided in Appendix B.

## C. ALTERNATIVES FOR REPRESENTING PROGRAM STRUCTURE

As the previous section illustrates, there has been some standardization for maintenance documentation to follow. The remainder of this chapter is devoted to a discussion of those tools available for representing a program's internal structure. This is an area that has not been standardized. In fact, there is considerable disagreement as to what tools are the best to use.

### 1. Flowcharts

The flowchart is a graphic representation of a program logic. Its purpose is to make it easy to see the relationships and flow of control among the various design elements. It is a technique that has been so widely used since it was developed by von Neuman in 1947 that a set of national standards exists for flowcharting symbols [62].

Many individuals, however, are opposed to the use of flowcharts. Brooks [61] calls the technique an "obsolete nuisance," and "a most thoroughly oversold piece of program documentation." Aron [62] feels that flowcharts are useless to a programmer when diagnosing errors. Weinberg [63] states "we find no evidence that the original coding plus flow diagrams is any easier to understand than the original coding itself--except to the original programmer." These comments bring into question the value flowcharts have for the maintenance programmer.

Schneiderman, et.al. [64] decribe a series of controlled experiments which test the utility of flowcharts as an aid to the full range of programming activities: composition, debugging and modification. Although their original intent was to determine when flowcharts were most helpful, the experimental results led them to conclude that flowcharts are a redundant presentation of the information contained in the programming language statements. Their conjecture is that flowcharts may even be a hindrance because they are not as complete (omitting declarations, statement lables and input/output formats).

To provide an example for illustrating some points to consider when using flowcharts as a maintenance tool, a series of four pages of flowcharts which represent the logic in a TRIDENT CCS module will be used (Figures 4-2 through 4-5). For simplification, the labels used in the flowcharts have been changed using the convention: (Ti) for terminals, (Di) for decision points, (Ci) for connectors, and (Pi) for processes.

These flowcharts were chosen as examples because they represent a small, logically clear section of code. According to the flowcharts, this section of code can be entered only through T1, Figure 4-2, and exited only through T2, Figure 4-4. A stopping condition exists at T3, Figure 4-5.

The first point to be illustrated concerns the use of connectors. The connectors used in the original TRIDENT flowcharts are statement labels and could be used as entry points from other portions of the program. The use of single connectors embedded in a sequence of code such as C1, Figure 4-2, is unnecessary since no additional entry points are designated. By checking the actual code, through the use of the cross-reference listings, it was determined that this label was, however, used by a subsequent branch point. A modified version of the flowchart in Figure 4-2, which more accurately represents the programs logic, is provided in Figure 4-6. The point is that all possible entries to a program should be clearly designated. If no entry point exists then labels are not needed and should be eliminated. Not to do so creates the possiblity for potential errors.

A second point to consider is the ability to trace through a section of logic. Going from beginning to end is relatively easy, but consider tracing through the reverse direction. Often, the maintenance programmer is left with a specific program state, and his job is to determine what conditions created it. For example, using Figures 4-2 through 4-5, if the maintenance programmer needed to determine what sequence of control could have led to the stopping condition (T3), Figure 4-5, it would be necessary to trace backwards through all four pages of flowcharts. This problem is compounded when dealing with numerous pages of flowcharts and multiple branch points.

A third point to consider is the difficulty of making changes to the documentation. Note that substituting a decision block (D2A) for a procedure block (P2) in Figure 4-2, in order to more accurately represent the programs logic, required that a completely new flowchart be constructed, Figure 4-6.

It should be noted that the Software Acquisition Management Guidebook, Software Maintenance Volume [27], recommends that DoD not procure flowcharts with delivered software, and MIL-STD-1679 states that "there is no requirement that flowcharts be a deliverable item."

In contradiction to this guidance, SECNAVINST 3560.1, when describing the Program Description Document, states "a flowchart shall be included for each major procedure or subroutine that depicts detailed operations performed by the subprogram."

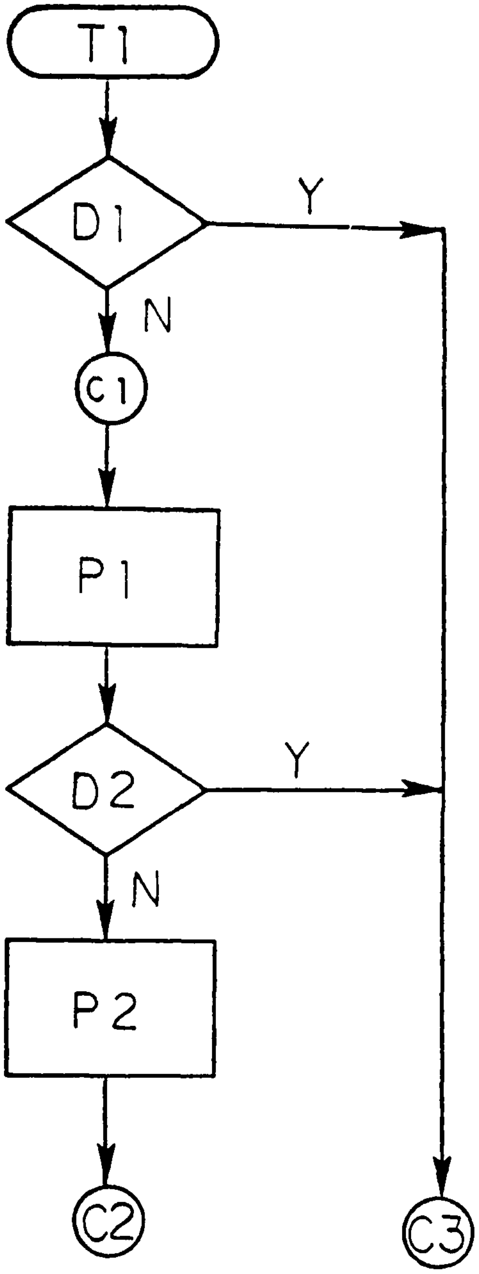


Figure 4-2. Exarple Flowchart, part 1 of 4

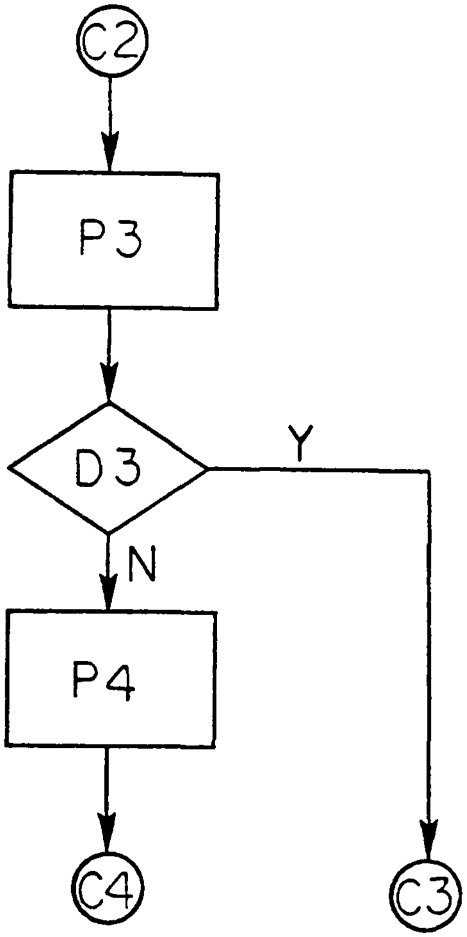


Figure 4-3. Example Flowchart, part 2 of 4

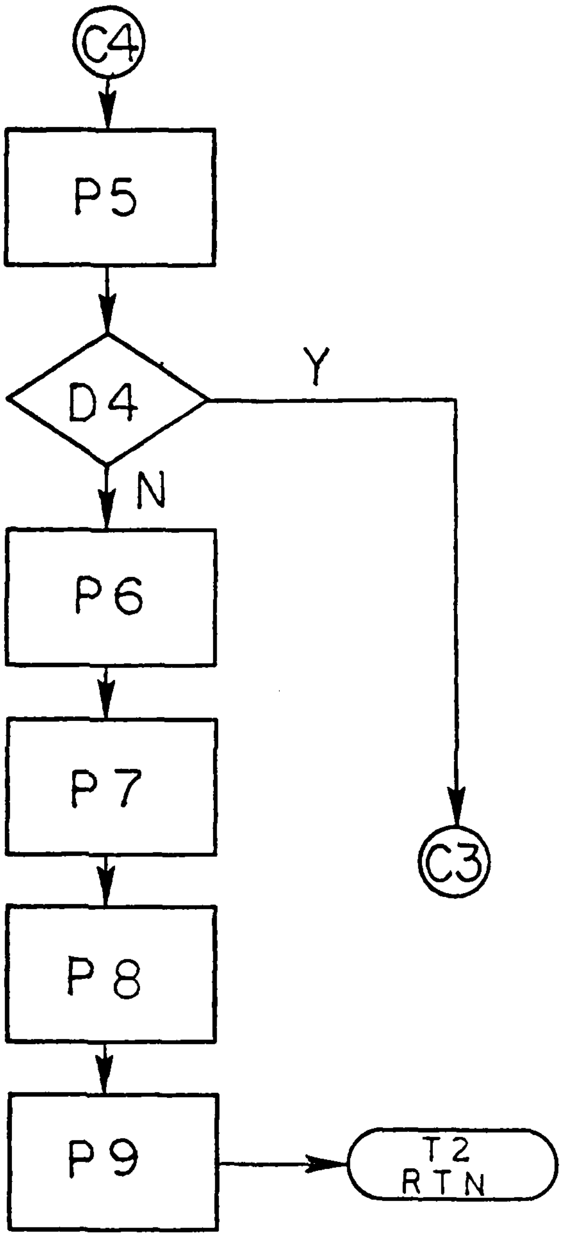


Figure 4-4.  Example Flowchart, part 3 of 4

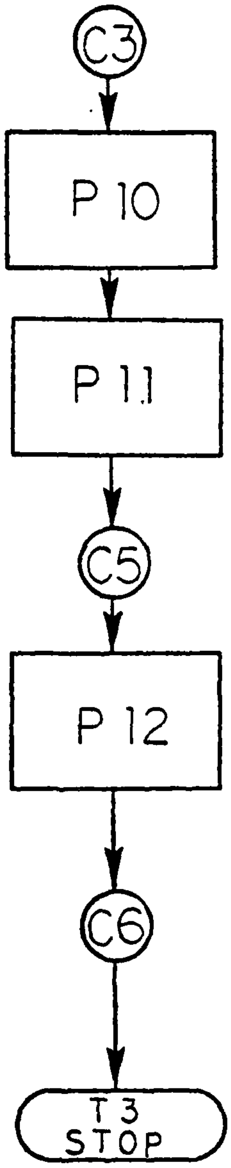


Figure 4-5. Example Flowchart, part 4 of 4

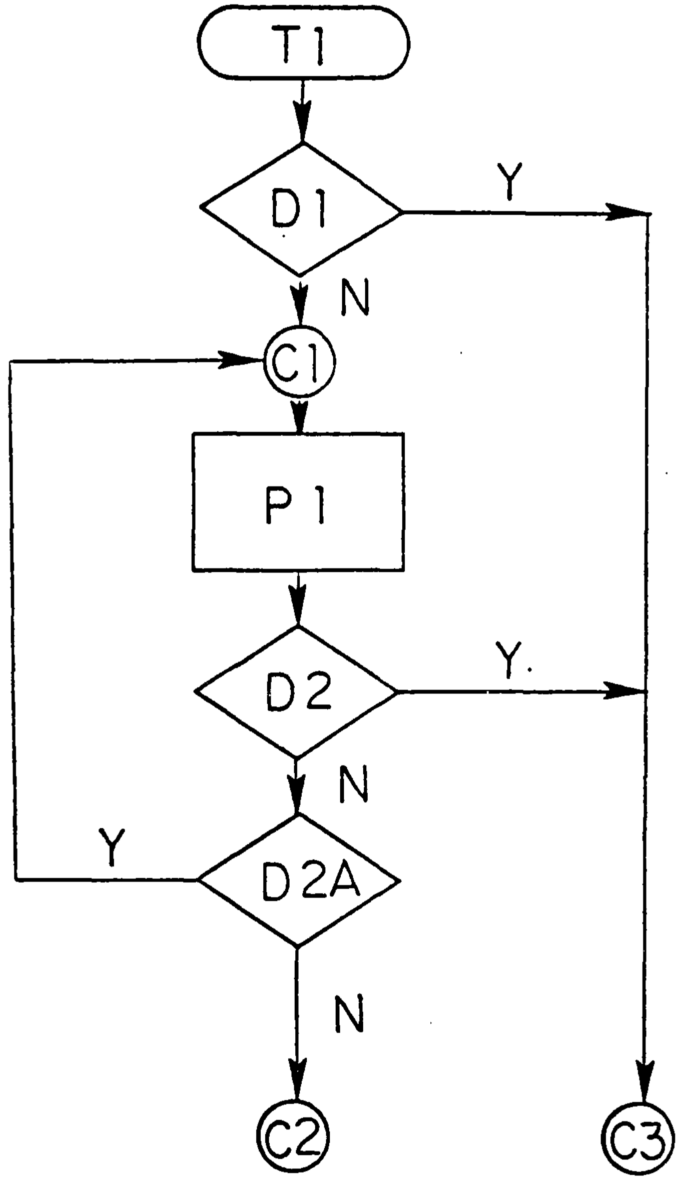


Figure 4-6. Modified Flowchart

### 2. Hierarchy Plus Input-Process-Output (HIPO)

HIPO was developed as a design aid and documentation technique by IBM and is described in [65]. It attempts to provide more than just representing the program logic as flowcharts do. It emphasizes the functional aspect of the program and its data flow. Maintenance efforts are said to be facilitated by making it easier to trace a function that needs to be modified from the documentation to the actual code.

A HIPO package consists of three kinds of diagrams: a visual table of contents, overview diagrams and detail diagrams. These diagrams provide a graphical description of the program's function from the general to a detailed level.

Figure 4-7 shows the structure of a typical HIPO package. Note that the visual table of contents shows the structure of the diagram package and relationships of the functions in a hierarchical fashion. The overview and detail HIPO diagrams contain the inputs, processes, outputs and extended descriptions at each stage of the successive decomposition of a program.

HIPO does not enjoy universal support as a maintenance tool. In a survey by Anderson and Shumate [66], conducted to find out what documentation tools were found useful by maintenance programmers, HIPO was ranked as the least preferred form when compared to the program listings, English language narratives, flowcharts, hierarchy diagrams and the data base design documents. The authors felt that HIPO documentation is an important design tool but seems to have a lesser value for maintenance activities.

Meyers [5] contends that while HIPO diagrams are superior to the flowchart because they show data flow as well as control flow, HIPO diagrams are not needed for the same reasons that flowcharts are not needed for maintenance type work. Basically, he feels both merely duplicate information that is already contained in the program listings.

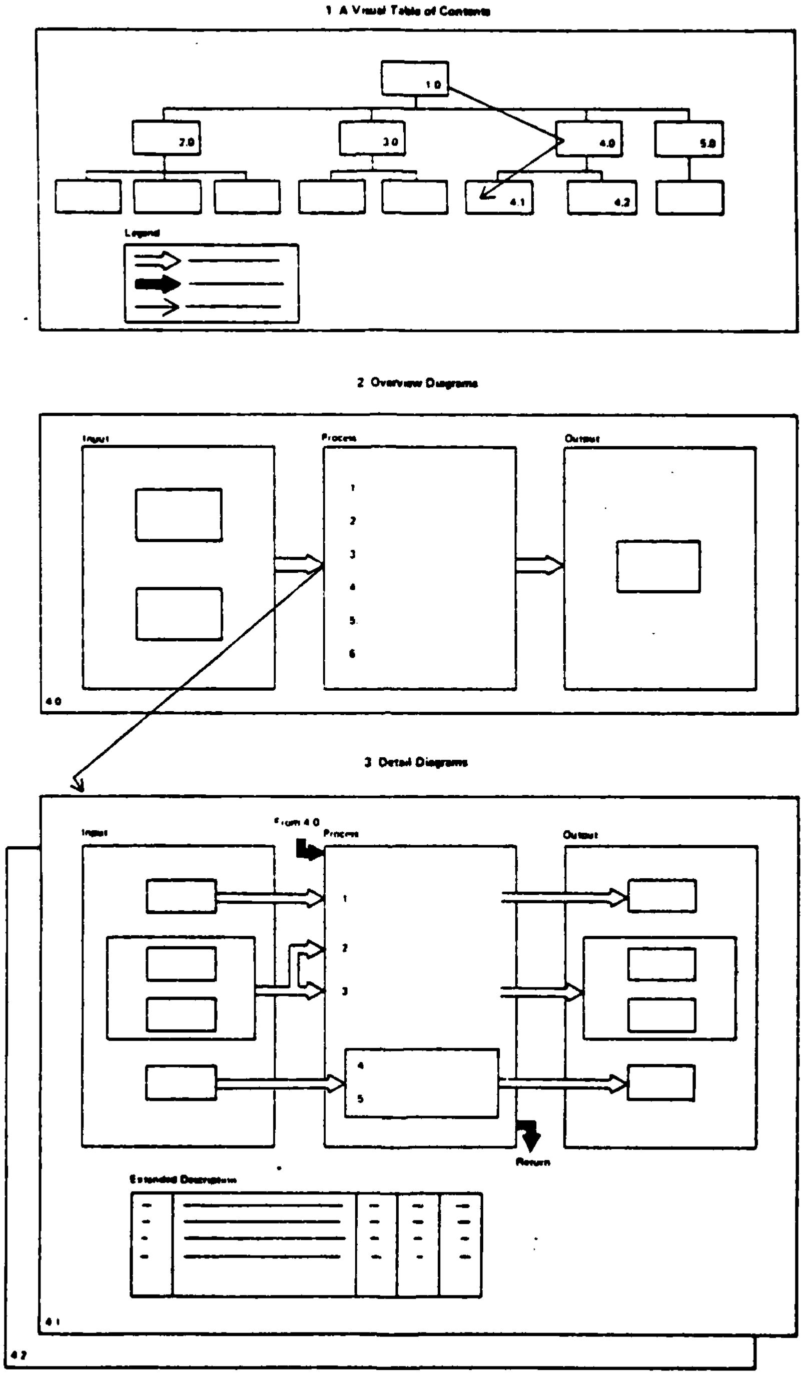


Figure 4-7. HIPO Documentation [65]

### 3. Decision Tables

Decision tables provide a tabular form of representing program design and have been used as a maintenance tool. Generally, decision tables are made up of a set of conditions, each of which may be evaluated as true or false at any given time. The truth or falsity of these conditions may be combined in various ways, along with a series of actions, to form what is called a decision rule (i.e., a set of conditions that must be satisfied in order that a series of actions be taken).

| | |
|---|---|
| CONDITION STUB | CONDITION ENTRY |
| ACTION STUB | ACTION ENTRY |

Table IV-1. Decision Table Structure

As illustrated in Table IV-1, it is divided into four quadrants. The upper left quadrant, called the condition stub, contains all the conditions being considered for a particular decision rule. The condition entry, in the upper right quadrant, combines with the condition stub to form the condtion that is to be tested. The action stub, in the lower left quadrant, contains actions resulting from the conditions tested above. Action entries, in the lower right quadrant, serve to indicate responses to the indicated combination of conditions.

If a condition in the condition stub is true, a "Y" is entered for that particular rule in the condition entry; if the condition is false, an "N" would be entered. In a situation where a particular condition is irrelevant a don't-care would be indicated by use of a dash, "-". An "X" specifies actions to be executed. An example of a decision table for representing a simple process of approving/disapproving loan requests is presented in Table IV-2.

| LOAN TABLE | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| Satisfactory credit limit | Y | N | N | N |
| Favorable Payment History | - | Y | N | N |
| Special Clearance Obtained | - | - | Y | N |
| Approve Loan | X | X | X | |
| Reject Loan | | | | X |

Table IV-2. Example Decision Table [67]

One advantage of using decision tables is that it is possible to convert them into compilable source code via a preprocessor [67, 68]. The additional computer time required for compilation can be offset by reduced effort for programming both during the initial programming phase and the maintenance phase. Another big advantage of decision tables is that their concept and structure causes the number of overlooked situations and program inconsistencies to be reduced.

The B. F. Goodrich Chemical Company is one proponent on the use of decision tables. Reference [16] reports that Goodrich has used them extensively and finds that complex logic becomes clearer and there is less chance of overlooking a logical path. Goodrich estimates that overall productivity for analysts and programmers in maintaining its COBOL-based systems has been at least double what it would have been without decision tables.

Another successful example concerning the use of decision tables is reported by Fisher [69]. An extremely complex file maintenance problem arose at the USAF Automatic Resupply Logistic System at Norton AFB. Almost seven man-years had been spent trying to define the problem using narrative descriptions and flowcharts, but to little avail. A crash program using decision tables was then implemented. Four analysts spent one week establishing the decision table format. Three weeks later the problem was solved.

To help determine whether the use of decision tables is appropriate for documenting programs such as the TRIDENT CCS, a section of logic was translated into a decision table format (Table IV-3). The logic represented is the same as that shown in Figures 4-2 through 4-5. Note that identical logic contained in four pages of flowcharts has been reduced to a clear, concise table taking less that one page. This points out, also, that revision of decision tables requires less work than modifying flowcharts. This is an important consideration for maintenance activities where revisions are expected.

| | R1 | R2 | R3 | R4 | R5 |
|---|---|---|---|---|---|
| D1 | Y | N | N | N | N |
| D2 | - | Y | N | N | N |
| D3 | - | - | Y | N | N |
| D4 | - | - | - | Y | N |
| P1 | | X | X | X | X |
| P2, P3 | | | X | X | X |
| P4, P5 | | | | X | X |
| P6-P9 | | | | | X |
| RETURN | | | | | X |
| P1Ø-P12 | X | X | X | X | |
| STOP | X | X | X | X | |

Table IV-3. Example Program Logic

Two disadvantages of decision tables are: (1) possible ambiguities may arise when "don't care" conditions are presented and (2) decision tables are of little help when the program logic involved is not decision-making oriented.

While decision tables may not always be applicable, the previous discussion illustrates that they serve as an alternative form of documentation that should be considered. Federal Information Processing Standards Publication 38, "Guidelines for Documentation of Computer Programs and Automated Data Systems," 15 February 1976, states that either flowcharts or decision tables, whichever is more appropriate, can be included or appended to documentation for software. However, SECNAVINST 3560.1 makes no mention of their use.

4. Nassi-Shneiderman Charts

With the advent of structured programming technology a form of structured flowcharts has emerged. Developed by I. Nassi and B. Shneiderman in 1972, they can serve as a graphic representation of a modules logic design and provide a maintenance programmer with a quick reference for finding the code performing any logical function. The advantages claimed for these charts include:

-The scope of IF THEN ELSE clauses is well-defined and visible; moreover, the conditions or process boxes embedded within compound conditions can be seen easily from the diagram.

-The scope of local and global variables is immediately obvious.

-Arbitrary transfers of control are impossible.

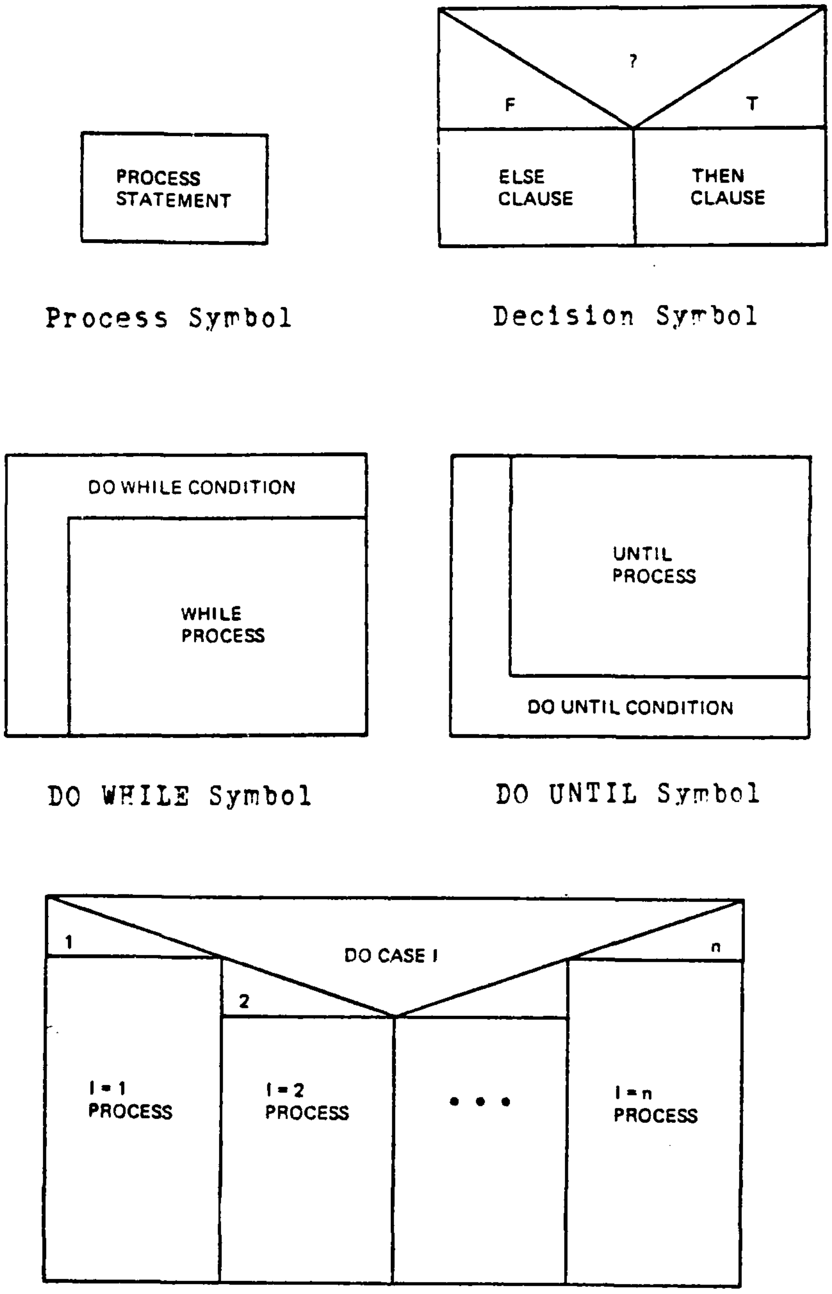
-Complete thought structures can and should fit on one page (i.e., no off-page connectors).

Yoder [70] provides a thorough description of the use of N-S charts. Briefly, the charts are constructed by combining and nesting the basic structures shown in Figure 4-8. An example showing an extension of the use of the basic symbols, which illustrates a N-S chart to calculate and print an FICA report, is shown by Figure 4-9.

N-S charts are strongly linked to structured programming constructs, thus, it may be difficult to apply this form of documentation to non-structured portions of program logic.

The method of N-S charts has not been fully exploited in actual practice and little information exists in the technical literature advocating their use. They are, nevertheless, an alternative form of documentation that may be considered for use as a maintenance tool.

The section of logic previously represented by Figures 4-2 through 4-5 and by Table IV-3 has been represented using N-S charts (Figure 4-10). This illustrates the potential of using N-S charts as a maintenance tool for software such as the TRIDENT CCS.



Process Symbol

Decision Symbol

DO WHILE Symbol

DO UNTIL Symbol

CASE Symbol

Figure 4-8. Five Basic Structures of N-S Charts [70]

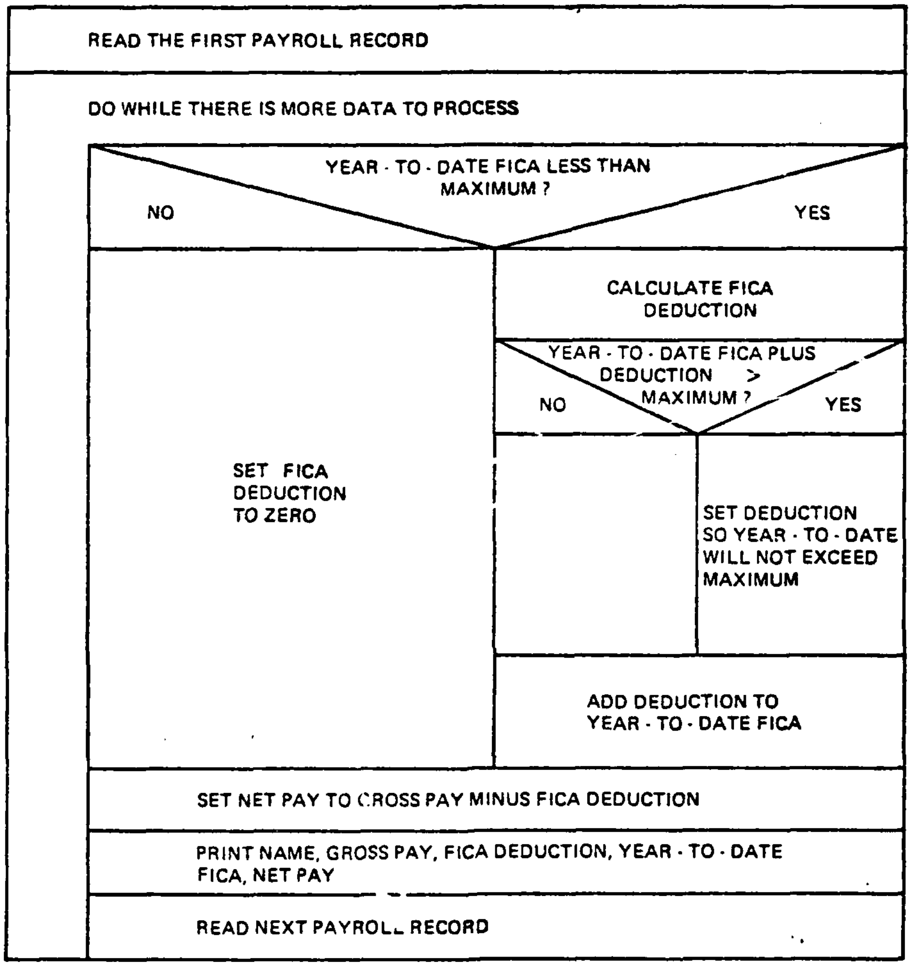

READ THE FIRST PAYROLL RECORD
DO WHILE THERE IS MORE DATA TO PROCESS
YEAR - TO - DATE FICA LESS THAN MAXIMUM ?
NO
YES
CALCULATE FICA DEDUCTION
YEAR - TO - DATE FICA PLUS DEDUCTION > MAXIMUM ?
NO
YES
SET FICA DEDUCTION TO ZERO
SET DEDUCTION SO YEAR - TO - DATE WILL NOT EXCEED MAXIMUM
ADD DEDUCTION TO YEAR - TO - DATE FICA
SET NET PAY TO CROSS PAY MINUS FICA DEDUCTION
PRINT NAME, GROSS PAY, FICA DEDUCTION, YEAR - TO - DATE FICA, NET PAY
READ NEXT PAYROLL RECORD

Figure 4-9. Exarple N-S Chart [70]

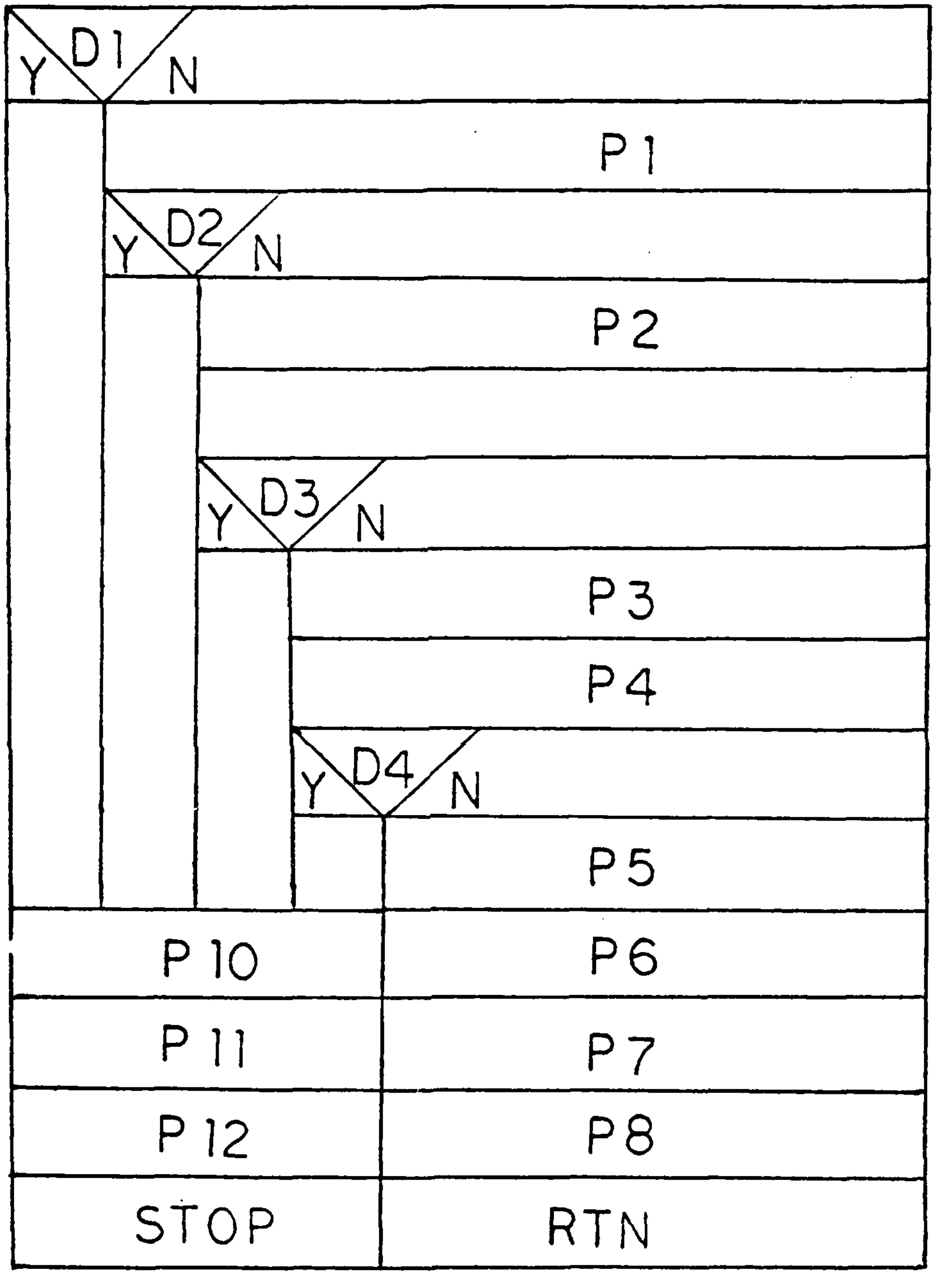


Figure 4-10. Nassi-Shneiderman Chart For TRIDENT

5. Program Listings

It would be highly desirable if programs could be made self-documenting, thereby, eliminating the necessity of maintaining multiple forms of documentation representing the same logic. Many authors advocate such an approach through structuring program listings. Meyers [5], for example, states:

> Since we already have the code, why not let it serve as the logic documentation? . . . additional documentation such as a flowchart would be undesirable because it would be redundant with the code. Redundancy in any type of documentation should be avoided because it increases the chances of conflicts. Furthermore, unless care is taken to update the documentation (which is more difficult if the logic documentation is physically separated from the code), redundant documentation often becomes totally useless after the code is modified a few times.

In his 1974 ACM Turing Award Lecture, Knuth [71] addressed the importance of program listings when he stated:

> There are many senses in which a program can be "good" of course. In the first place, it's especially good to have a program that works correctly. Secondly it is often good to have a program that won't be hard to change, when the time for adaptation arises. Both of these goals are achieved when the program is easily readable and understandable to a person who knows the appropriate language.

Anderson's study [66], discussed previously, has illustrated the importance of program listings as compared to other forms of documentation for maintenance work. Again, this study found listings were the maintenance programmer's "most useful tool."

What constitutes a self-documenting program? SECNAVINST 3560.1 states that the listing will be an exact duplicate of the delivered card decks or magnetic tape. It further states that each compiler source statement will be annotated with comments, or, if the source is assembly level, then a comment shall be listed for each assembly level line or function group of lines with not less than an average of one comment per five statements. No mention is made of the type or form of comments.

MIL-STD 1679 provides much more explicit direction. It states, in part, that:

> A narrative description shall describe the history and identify the functions of each hierarchical component of the weapon system software.
>
> Each component shall include at the beginning of the executable coding a textual description of its inputs, outputs, function or task, and algorithms; a list of other components called; and a list of all calling components. In addition to general explanations, to assist understanding, precise references to the appropriate statement labels and data-names shall be included in each module, procedure and routine descriptive abstract. The descriptive abstract shall define the allowed and tolerable range of values for all inputs and shall define the allowed and expected range of values for all outputs. A history of the original and updating programmer names, the activity or commercial company name and the activity or company division code or billet identifier with dates completed shall be included.
>
> In order to facilitate program comprehension, comment statements shall be used throughout the program code. Comment statements are non-executable (i.e., they have no effect on program executions) and are used to provide documentation and clarification of the logic, data, variables, and algorithms. Each source statement shall be self-defined or defined by a comment phrase to a level understandable by a person not associated with

> the original development effort. Logical groups of comment phrases may be included in a single comment statement. General comments on groups of source statements performing logical functions shall be included on separate comment statements.

The Tactical System Programming Support Branch of the Marine Corps Tactical System Support Activity, responsible for maintaining the Marine Corps' tactical software, considers the computer program listing to be "the single most important tool for software maintenance." It has developed a set of standards to ensure listings are properly designed and coded. This standard serves as a possible example for other maintenance organizations to follow. See Appendix C.

Both MIL-STD-1569 and SECNAVINST 3560.1 address the use of cross-reference listings which are included here as a portion of self-documentation since they can be automatically generated from the program listings. They are considered a necessary maintenance tool since they identify every place an item (e.g., variables or subroutines) appears in the program, so when the item is changed or modified the impact on the remaining portions of the program can be quickly determined.

### 6. Summary

This section has illustrated a variety of techniques used for representing program design to the maintenance programmer. Clearly, no one form completely represents all aspects of program design. As programming methodologies become more structured, the trend towards increased emphasis on the use of program listings should continue, reducing the need for supplemental forms of program documentation. Although, it seems unlikely the need for some type of graphic representation will be totally eliminated. There is an important psychological aspect of conveying meaning through pictures that cannot be duplicated with narratives. No doubt, a variety of documentation tools will always be necessary.

# V. SOFTWARE MAINTENANCE POLICIES WITHIN DOD

## A. BACKGROUND

This chapter provides an overview of policies and methodologies existing in DoD which affect weapon system software maintenance. First, the publications that contain applicable policy guidance are reviewed. Next, the results from a limited survey of agencies involved with weapon system software maintenance are presented. Finally, there is a discussion of pertinent research and development work.

It is important to realize that the policies and methodologies for procuring weapon system software have been different than that used for procuring automatic data processing equipment (ADPE). The distinction made between these two categories of automated systems is a result of the 1965 "Brooks Act" (Public Law 89-306, 40, U.S.C. 759).

The Office of Management and Budget (OMB) and the General Services Administration (GSA) administer the Brooks Act guidelines. ADPE is controlled by this act and falls under the purview of the Assistant Secretary of Defense (Controller). Weapon system software, however, is excluded from the provisions of this Act and fall under the jurisdiction of the Office of the Undersecretary of Defense for Research and Engineering.

## B. CURRENT POLICIES

There has been no centralized source of guidance with respect to weapon system software maintenance for DoD organizations to follow. Many directives, regulations, specifications, and standards have, however, influenced weapon system software maintenance to varying degrees. The most significant of these are listed in this section. Even though most of these have been introduced in previous chapters, they are consolidated here for ease of reference.

### 1. MIL-STD-483 (USAF)

MIL-STD-483 (USAF) "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs," 1 June 1971, defines the entire spectrum of activities associated with controlling changes (a critical need for maintenance work) to computer programs.

### 2. MIL-S-52779 (AD)

MIL-S-52779(AD), "Software Quality Assurance Program Requirements," 5 April 1974, requires that a Quality Assurance Program (QAP) be implemented specifically for the development of computer programs and related documentation. Even though this standard is concerned with the development phase, it is important to software maintenance because it directly affects the quality of the software.

3. SECNAVINST 3560.1

SECNAVINST 3560.1, "Department of the Navy Tactical Digital Systems Documentation Standards," 8 August 1974, identifies, names, and describes that set of documents necessary to support both the development and maintenance of tactical software.

4. DODDIR 5000.29

DODDIR 5000.29, "Management of Computer Resources in Major Defense Systems," 26 April 1976, establishes DoD policy for the management and control of computer resources during system acquisition. Maintainability of software is called out as a major consideration during initial design. It also directs that support items required for cost effective maintenance be specfied as deliverable items.

5. MIL-STD-1521 (USAF)

MIL-STD-1521 (USAF), "Technical Reviews and Audits for System, Equipment, and Computer Programs," 1 June 1976, prescribes the requirements for the conduct of technical reviews and audits in conjunction with the documents defined in MIL-STD-483. Direction is provided concerning the review and audit of computer program configuration items and their associated documentation. Each type of review or audit is described in an appendix to the standard and can serve as a basis for checking compliance with maintainability requirements.

### 6. DODINST 5000.31

DODINST 5000.31, "Interim List of DoD Approved High Order Programming Languages (HOL)," 24 November 1976, specifies the HOLs which are approved for use in conjunction with DODDIR 5000.29. Although this instruction allows for certain exceptions, it attempts to reduce proliferation and ensure control of HOLs in defense systems by limiting new development to six approved languages: CMS-2, SPL-1, TACPOL, JOVIAL, COBOL, and FORTRAN.

### 7. MIL-STD-1679 (NAVY)

MIL-STD-1679 (NAVY), "Weapon System Software Development," 1 December 1978, establishes uniform requirements for the development of weapon system software within DoD. Strict adherence to the provisions of this standard will help ensure that the tactical software so developed will be improved over current versions of tactical software.

## C. SURVEY OF DOD MAINTENANCE ORGANIZATIONS

An informal survey was taken of personnel from five different DoD organizations involved with the maintenance of weapon system software. While not providing official policy, the results can be used to derive a general understanding of the environment in which they have operated, such as what problems have been experienced and what methodologies were used in performing maintenance activities.

1. Pacific Missile Test Center

The Weapons Control and Software Systems Division of the Pacific Missile Test Center is involved with Fleet support of tactical software for selected weapon systems such as the F-14.

The software, developed largely under contract, was being maintained by in-house resources. Maintenance functions performed included configuration accounting, problem validation, training, analysis, design, change implementation, documentation, verification and tape generation. The greatest amount of work has been necessitated by software enhancements which required varying degrees of redesign. New tape versions were released approximately every 18 months.

Competing with private industry for recruiting professional personnel has been a significant problem. Another problem has been inadequate software documentation from contractors. Concern was expressed that documentation has historically been one of the first items to be cut from software development budgets, a decision that has seriously degraded the subsequent maintainability of software.

A large effort has been made to correct the problem of inadequate documentation. Guidance was being formulated which goes beyond the requirements defined in SECNAVINST 3560.1 and MIL-STD-1679 by improving the traceability from one level of system description to another.

The importance of using actual operational equipment for program debugging and verification after maintenance changes were made was stressed.

An effort to keep methodologies current is evident, but this effort is being strained by increased work loads and personnel shortages.

2. Naval Ocean Systems Center

The Software Quality Control Organization at Naval Ocean Systems Center is not directly responsible for maintaining tactical software. It did, however, perform a critical function that greatly improves software maintainability. Activities include document inspection, configuration management and test and evaluation during all phases of the acquisition cycle in order to assist procuring organizations in acquiring higher quality and more maintainable software.

One of the biggest problems encountered has been convincing managers that software requires the same degree of engineering controls as hardware.

3. Naval Surface Weapons Center

The Fleet Ballistic Missile Geoballistics Division of Naval Surface Weapons Center is responsible for both development and maintenance of Fleet ballistic missile type software such as the TRIDENT-I Fire Control System. Most of its work is accomplished in-house with very little contracting. There is no separate organizational group dedicated solely to the maintenance of software. Maintenance activities are integrated with development activities.

As expected, when software products were initially released to the fleet the vast majority of maintenance was accomplished in order to correct errors, but the ratio of improvements to error corrections increased as the time from initial release increased. One software product which had been released for two years was experiencing maintenance of approximately 50 percent for improvements and 50 percent for error corrections.

Changes to software are made according to a formalized configuration control plan. Releases of new versions have been made on the average of once per year. Patches were discouraged but used under restricted and tightly controlled circumstances such as to correct critical errors between major program releases.

Actual field equipment is used to test program changes with the capability of using some real inputs. Most inputs, however, are simulated.

A hardware monitor is used and found very useful for analyzing the performance of software. Another useful tool used is the ability to take core dumps which are analyzed via computer whenever program crashes occurred.

A specially designed HOL called Trident High Level Language (THLL), said to be even more structured than CMS-2,

AD-A090 159 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
TECHNIQUES AVAILABLE FOR IMPROVING THE MAINTAINABILITY OF DOD W--ETC(U)
JUN 80 R D PILCHER

UNCLASSIFIED

NL

2 OF 2
ADA
090159

END
DATE
FILMED
11-80
DTIC

was being used. Program listings are maintained in a structured form, and a program design language (PDL), a pseudo high level language, is used to help document programs.

The actual process of making changes to software has posed no significant problems, but understanding and verifying reported software errors from the Fleet did, at times, present difficulties.

### 4. Naval Air Development Center

The Software and Computer Directorate of the Naval Air Development Center functions as the software support agency for selected avionics software such as that in the P-3C Orion.

The maintenance of the P-3C software is complicated by the fact that it is being converted from a tape configuration system to a drum configuration system. While the functional requirements remained the same, the details of implementation differed. Both configurations must be simultaneously maintained.

The importance of defining to a fine detail maintenance requirements early in the development of software was stressed. The concepts of structured programming was advocated, but trying to implement the constructs of MIL-STD-1679 on existing software that was originally unstructured presented many difficulties and was not recommended.

New program versions were being released on the average of every 18 to 24 months and patches were being used. It was stated that patches will always be required to some extent because of constraints such as delivery schedules.

While the program listing was the cheapest form of program documentation, detailed flowcharts were considered useful as a maintenance tool. It was suggested that the automated process of producing and updating flowcharts would be helpful.

One of the biggest problems being experienced was the large personnel turnover rate that exists in the services. Maintenance of software would be an easier task if there were greater stability of personnel.

### 5. TACFIRE Software Support Group

The TACFIRE Software Support Group is responsible for maintaining the software for the Army's automated Artillery Tactical Fire Direction System (TACFIRE), a system whose software was developed under contract. Maintenance of the software is still using contractor support.

The group uses configuration control procedures much like the other organizations contacted with a configuration control board setting priorities for approved software changes. Approximately 75 percent of the changes experienced were the result of program enhancements and 25 percent necessitated by program errors. New program versions were being released about every 12 months. Patches were discouraged but practically every release had contained a limited number.

Both a programming support system (PDP 11/35) and actual TACFIRE hardware were used for program debugging and testing procedures.

The code for the software is written in the HOL TACPOL. Some code in the programs is assembly level. The ratio of HOL lines of code to assembly level lines of code averaged roughly nine to one.

The support group is beginning to do software development work for a multiple rocket system. The software for this system is being designed to fit an existing set of hardware. The language used for this new software is assembly level, called Symbolic Interpreter Routine (SIR). The use of an assembly level language is necessitated by both hardware contraints and a desire to share previously written software modules.

The only general problem mentioned in maintaining weapon system software concerned the difficulty of interpreting software trouble reports submitted by using units in the field.

6. Marine Corps Tactical System Support Activity

The Marine Corps tactical software is developed largely by contractors. Software maintenance of fielded systems, however, is centralized and accomplished in-house by the Tactical Systems Programming Support Branch of the Marine Corps Tactical System Support Activity.

The software is written in CMS-2 and kept highly structured using the conventions outlined in Appendix C. Listings provided documentation for the program's logic eliminating the need for detailed flowcharts. The software is refined to the point that no major operational errors are observed. The majority of maintenance was being necessitated by program enhancements not error corrections.

Software configuration management is strictly applied to all changes. New tape versions have been released about every 9 months. Patches had not been used in over two years and are considered contrary to good maintenance practices.

Two tools found useful to support maintenance activities are the CMS-2 librarian to control coding changes and a hardware monitor to measure system performance.

Actual field systems are available for program testing and debugging with the capability of using both actual and simulated real-time inputs.

Personnel were in favor of adopting the programming language ADA and have been involved with the Department of Defense High Order Language Commonality Program since 1977.

Problems mentioned included attracting and retaining qualified personnel and educating top level managers about the nature of software. The technical aspects of maintaining software presented no significant problem.

## D. RESEARCH TO IMPROVE SOFTWARE MAINTENANCE

Wegner [72] states:

> Software maintenance has only recently been recognized as a key area for software research. Research needs include the development of tools to allow understanding (readability) of software, modifiability of software and revalidation of modified software.

Not listed in the previous statement is the need for validating claims that new software engineering methodologies significantly improve the maintainability of large, complex, real-time weapon system software. Since claims have not been demonstrated, there has been reluctance from some system developers to incorporate their use on actual system projects.

An ambitious, exploratory research project has been initiated by the Naval Research Laboratory and the Naval Weapons Center in order to correct this situation. The project involves completely redesigning and implementing the operational flight program (OFP) for the A-7 aircraft using many of the new software engineering principles. The redesigned program will be functionally identical to the existing A-7 OFP so a direct comparision between the two can be made in areas such as software maintainability.

If successful, the final product could serve as a useful engineering model for subsequent weapon system software developments. For further information the reader is referred to the program summary, Appendix D.

# VI. CONCLUSIONS AND RECOMMENDATIONS

DoD organizations are becoming more aware of the significance that maintenance plays in the overall life cycle of weapon system software. Even as this software becomes more error-free, the relative importance of maintenance activities will continue because of frequent enhancements made to existing systems and increasing complexity of applications.

To ensure that future weapon system software can be easily and accurately modified to correct errors or accommodate changes in user requirements, maintainability must be considered as a primary design objective.

The organization which will eventually be responsible for maintaining the software of a weapon system must be allowed to participate in the development process, including the formulation of specifications and subsequent technical design reviews.

The importance of programming standardization must be stressed because of the long life of weapon system software and the relatively high rate of personnel turn-over within DoD software maintenance organizations. Although software standards have not yet reached the refinement or level of detail that exist for hardware, MIL-STD-1679 represents a good starting point. If complied with, this standard should significantly improve the maintainability of weapon system software.

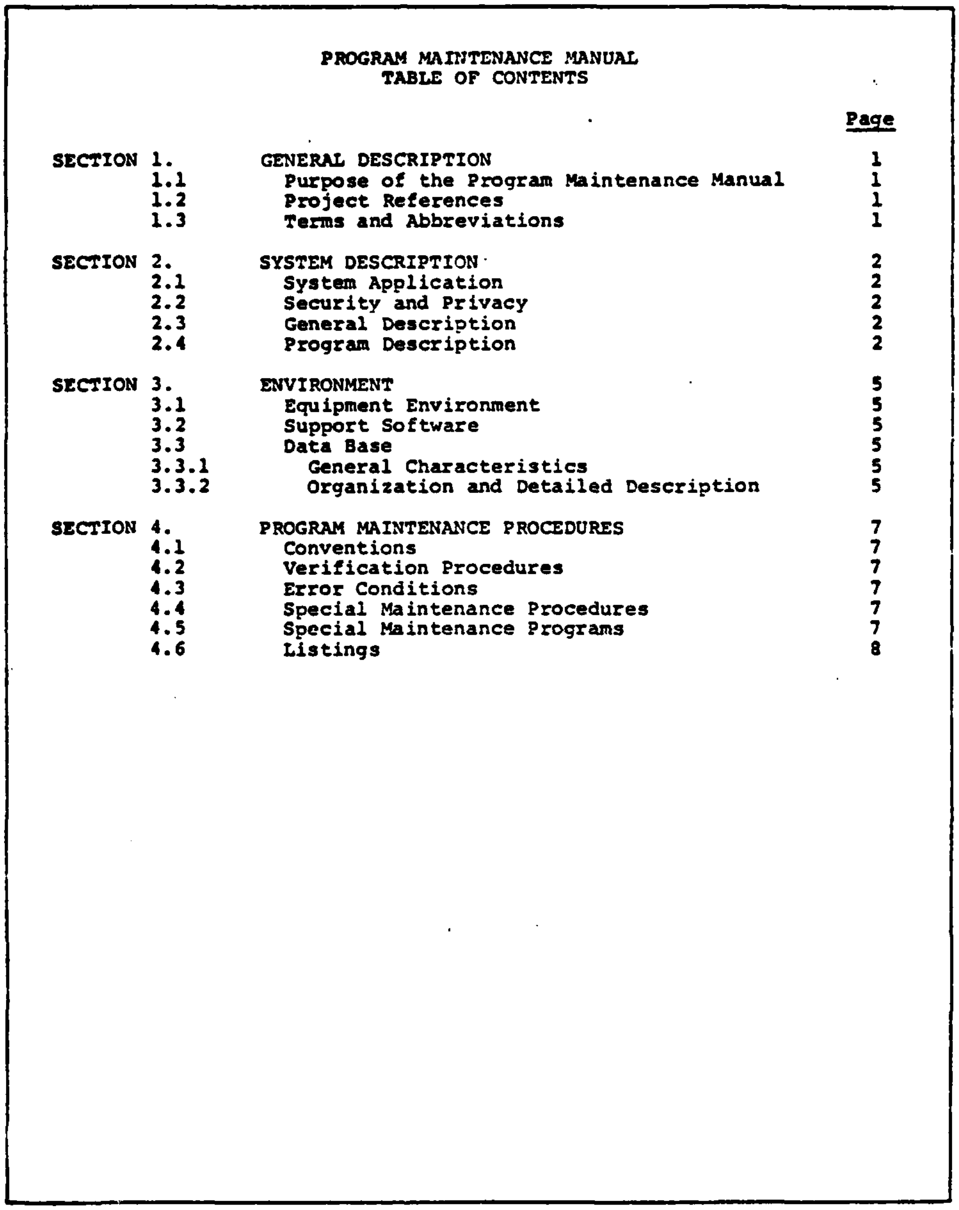
How much and what kind of documentation will be delivered with weapon system software are among the most important management decisions affecting the software's maintainability. Decisions must be based on the size and complexity of software produced and what techniques are used by the organization performing the maintenance. This thesis has illustrated a small portion of available types.

Institutions such as the Naval Postgraduate School are in a position to improve the education of future computer scientists on the nature of software maintenance. This could be done by establishing computer science program libraries consisting of student developed computer programs. Programs in these libraries would then be available for projects emphasizing program maintenance in addition to the traditional approach of emphasizing only program development. Grades based on how easily a student's program is understood and correctly modified by other students would provide an incentive for improving software maintenance skills.

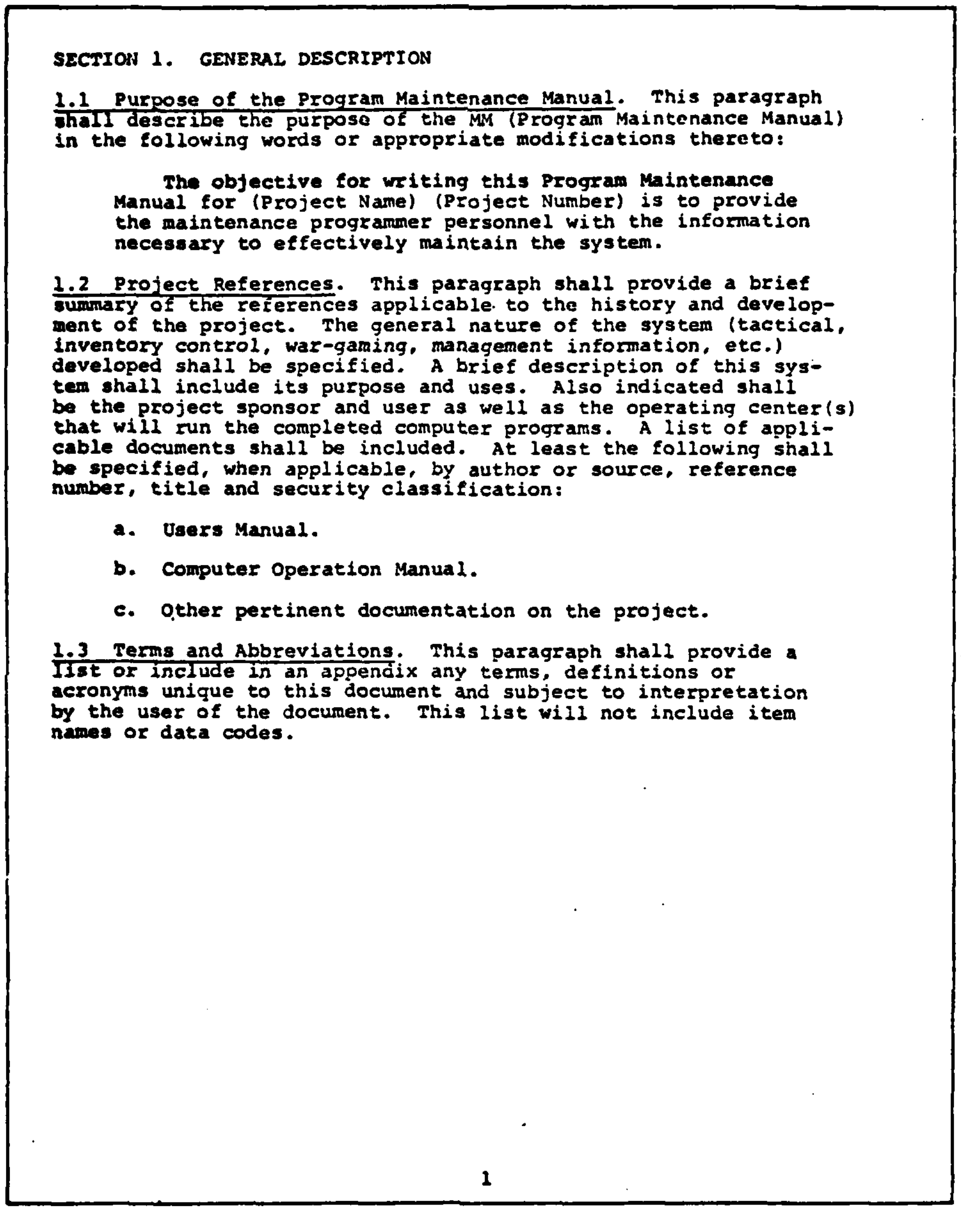
As a final thought, consider the findings of a study on software maintenance by Lientz and Swanson [73]. Their study "supports the proposition that an increase in the age of a system tends to lead to an increase in the level of effort in maintenance." This indicates that DoD must continually face a difficult question: when is it more economical to dispose of and redesign an existing system than to go on maintaining it?

# APPENDIX A - Program Maintenance Manual

from: DOD STANDARD 7935.1S, "Automated Data Systems Documentation Standards," 13 September 1977



PROGRAM MAINTENANCE MANUAL
TABLE OF CONTENTS



SECTION 1. GENERAL DESCRIPTION

1.1 Purpose of the Program Maintenance Manual. This paragraph shall describe the purpose of the MM (Program Maintenance Manual) in the following words or appropriate modifications thereto:

> The objective for writing this Program Maintenance Manual for (Project Name) (Project Number) is to provide the maintenance programmer personnel with the information necessary to effectively maintain the system.

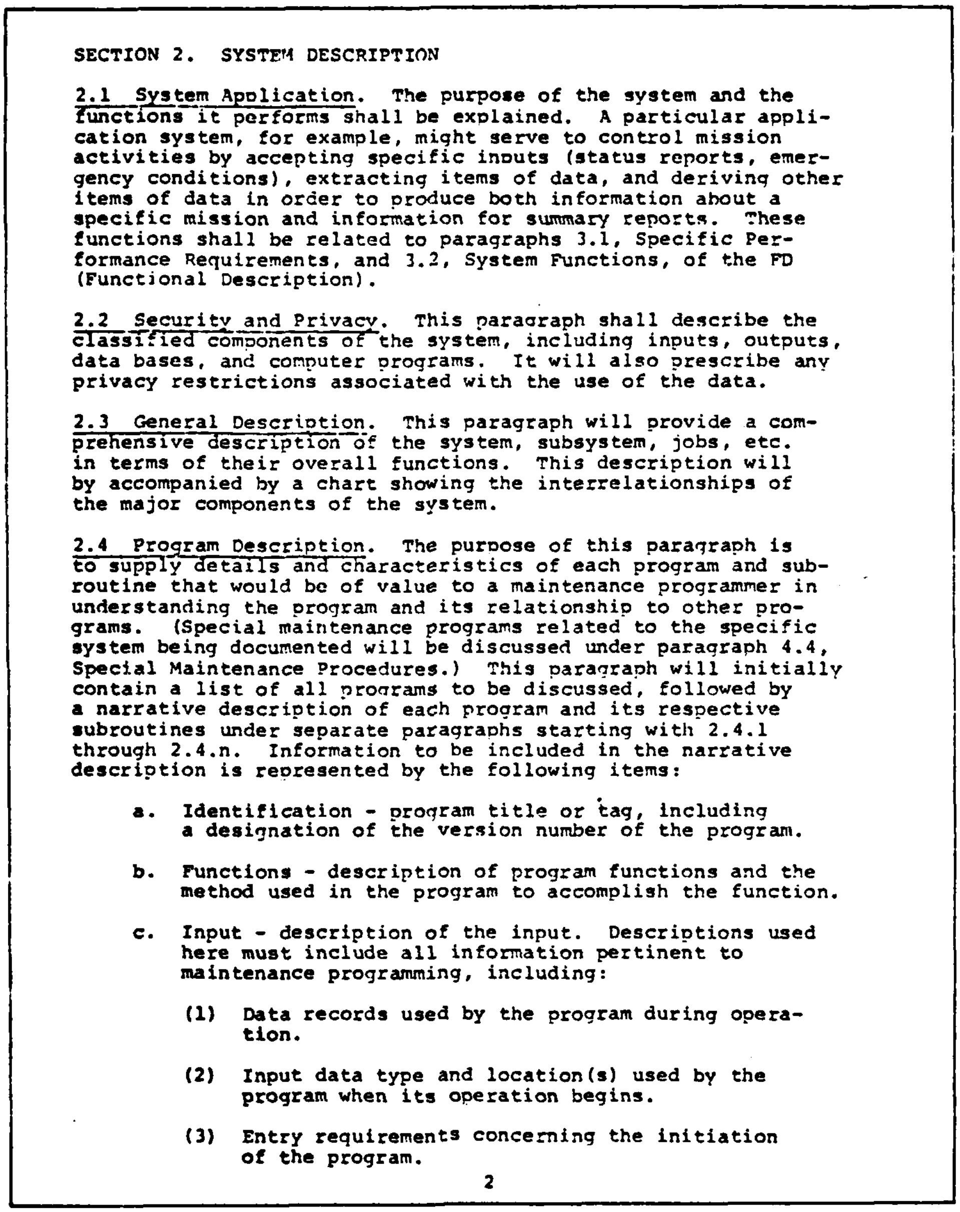
1.2 Project References. This paragraph shall provide a brief summary of the references applicable to the history and development of the project. The general nature of the system (tactical, inventory control, war-gaming, management information, etc.) developed shall be specified. A brief description of this system shall include its purpose and uses. Also indicated shall be the project sponsor and user as well as the operating center(s) that will run the completed computer programs. A list of applicable documents shall be included. At least the following shall be specified, when applicable, by author or source, reference number, title and security classification:

a. Users Manual.

b. Computer Operation Manual.

c. Other pertinent documentation on the project.

1.3 Terms and Abbreviations. This paragraph shall provide a list or include in an appendix any terms, definitions or acronyms unique to this document and subject to interpretation by the user of the document. This list will not include item names or data codes.



## SECTION 2. SYSTEM DESCRIPTION

2.1 System Application. The purpose of the system and the functions it performs shall be explained. A particular application system, for example, might serve to control mission activities by accepting specific inputs (status reports, emergency conditions), extracting items of data, and deriving other items of data in order to produce both information about a specific mission and information for summary reports. These functions shall be related to paragraphs 3.1, Specific Performance Requirements, and 3.2, System Functions, of the FD (Functional Description).

2.2 Security and Privacy. This paragraph shall describe the classified components of the system, including inputs, outputs, data bases, and computer programs. It will also prescribe any privacy restrictions associated with the use of the data.

2.3 General Description. This paragraph will provide a comprehensive description of the system, subsystem, jobs, etc. in terms of their overall functions. This description will by accompanied by a chart showing the interrelationships of the major components of the system.

2.4 Program Description. The purpose of this paragraph is to supply details and characteristics of each program and subroutine that would be of value to a maintenance programmer in understanding the program and its relationship to other programs. (Special maintenance programs related to the specific system being documented will be discussed under paragraph 4.4, Special Maintenance Procedures.) This paragraph will initially contain a list of all programs to be discussed, followed by a narrative description of each program and its respective subroutines under separate paragraphs starting with 2.4.1 through 2.4.n. Information to be included in the narrative description is represented by the following items:

a. Identification - program title or tag, including a designation of the version number of the program.

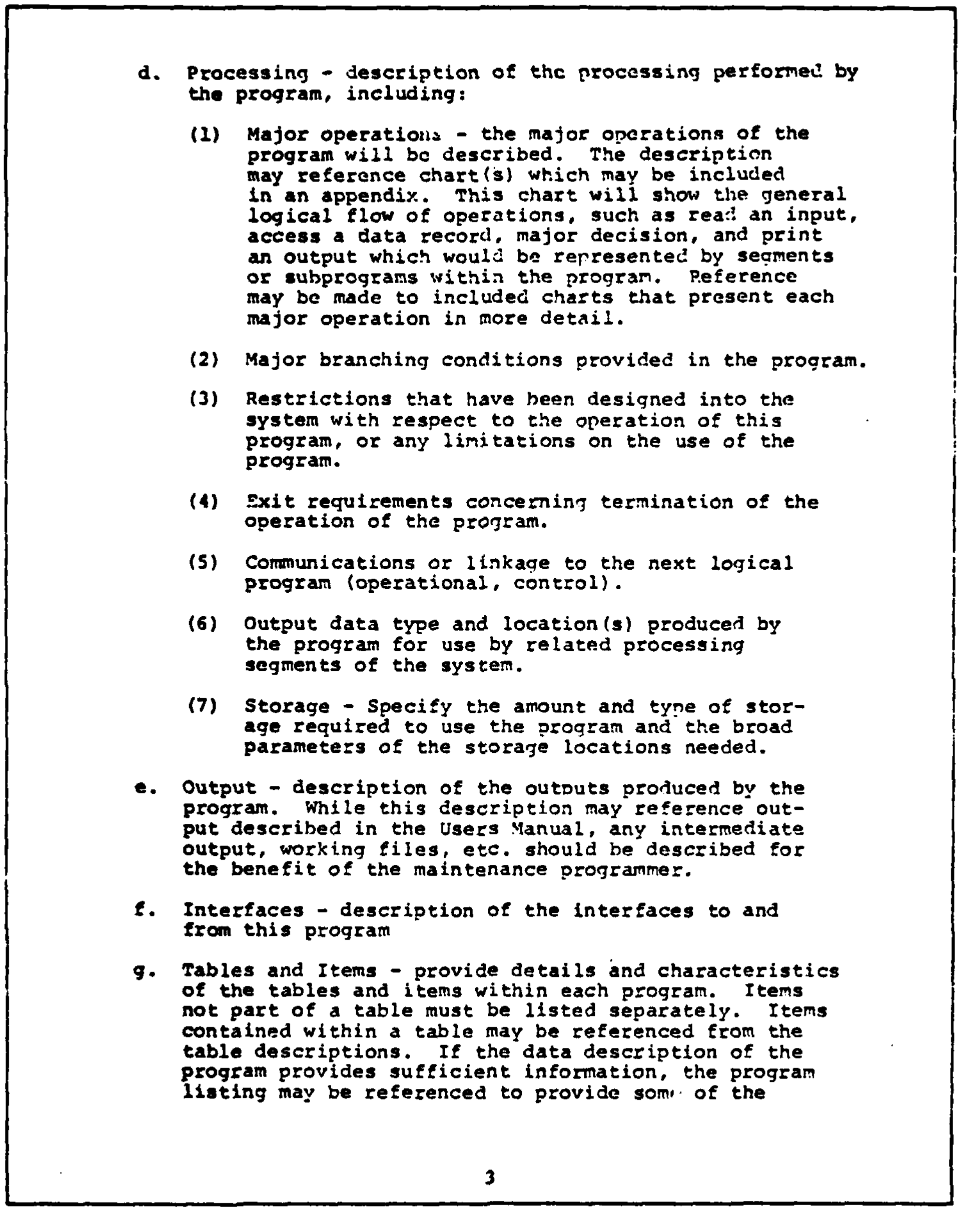
b. Functions - description of program functions and the method used in the program to accomplish the function.

c. Input - description of the input. Descriptions used here must include all information pertinent to maintenance programming, including:

(1) Data records used by the program during operation.

(2) Input data type and location(s) used by the program when its operation begins.

(3) Entry requirements concerning the initiation of the program.



d. Processing - description of the processing performed by the program, including:

(1) Major operations - the major operations of the program will be described. The description may reference chart(s) which may be included in an appendix. This chart will show the general logical flow of operations, such as read an input, access a data record, major decision, and print an output which would be represented by segments or subprograms within the program. Reference may be made to included charts that present each major operation in more detail.

(2) Major branching conditions provided in the program.

(3) Restrictions that have been designed into the system with respect to the operation of this program, or any limitations on the use of the program.

(4) Exit requirements concerning termination of the operation of the program.

(5) Communications or linkage to the next logical program (operational, control).

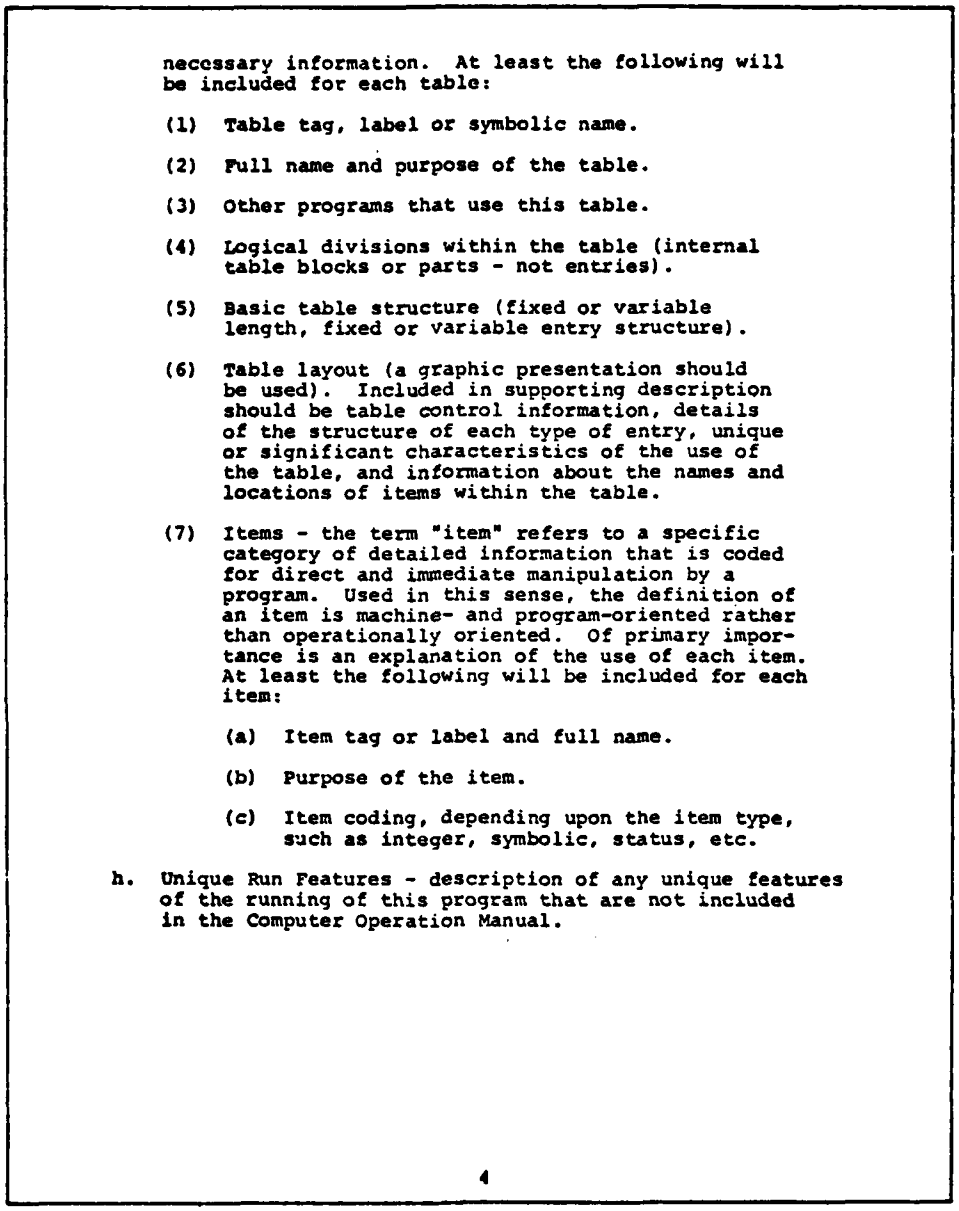
(6) Output data type and location(s) produced by the program for use by related processing segments of the system.

(7) Storage - Specify the amount and type of storage required to use the program and the broad parameters of the storage locations needed.

e. Output - description of the outputs produced by the program. While this description may reference output described in the Users Manual, any intermediate output, working files, etc. should be described for the benefit of the maintenance programmer.

f. Interfaces - description of the interfaces to and from this program

g. Tables and Items - provide details and characteristics of the tables and items within each program. Items not part of a table must be listed separately. Items contained within a table may be referenced from the table descriptions. If the data description of the program provides sufficient information, the program listing may be referenced to provide some of the



necessary information. At least the following will be included for each table:

(1) Table tag, label or symbolic name.

(2) Full name and purpose of the table.

(3) Other programs that use this table.

(4) Logical divisions within the table (internal table blocks or parts - not entries).

(5) Basic table structure (fixed or variable length, fixed or variable entry structure).

(6) Table layout (a graphic presentation should be used). Included in supporting description should be table control information, details of the structure of each type of entry, unique or significant characteristics of the use of the table, and information about the names and locations of items within the table.

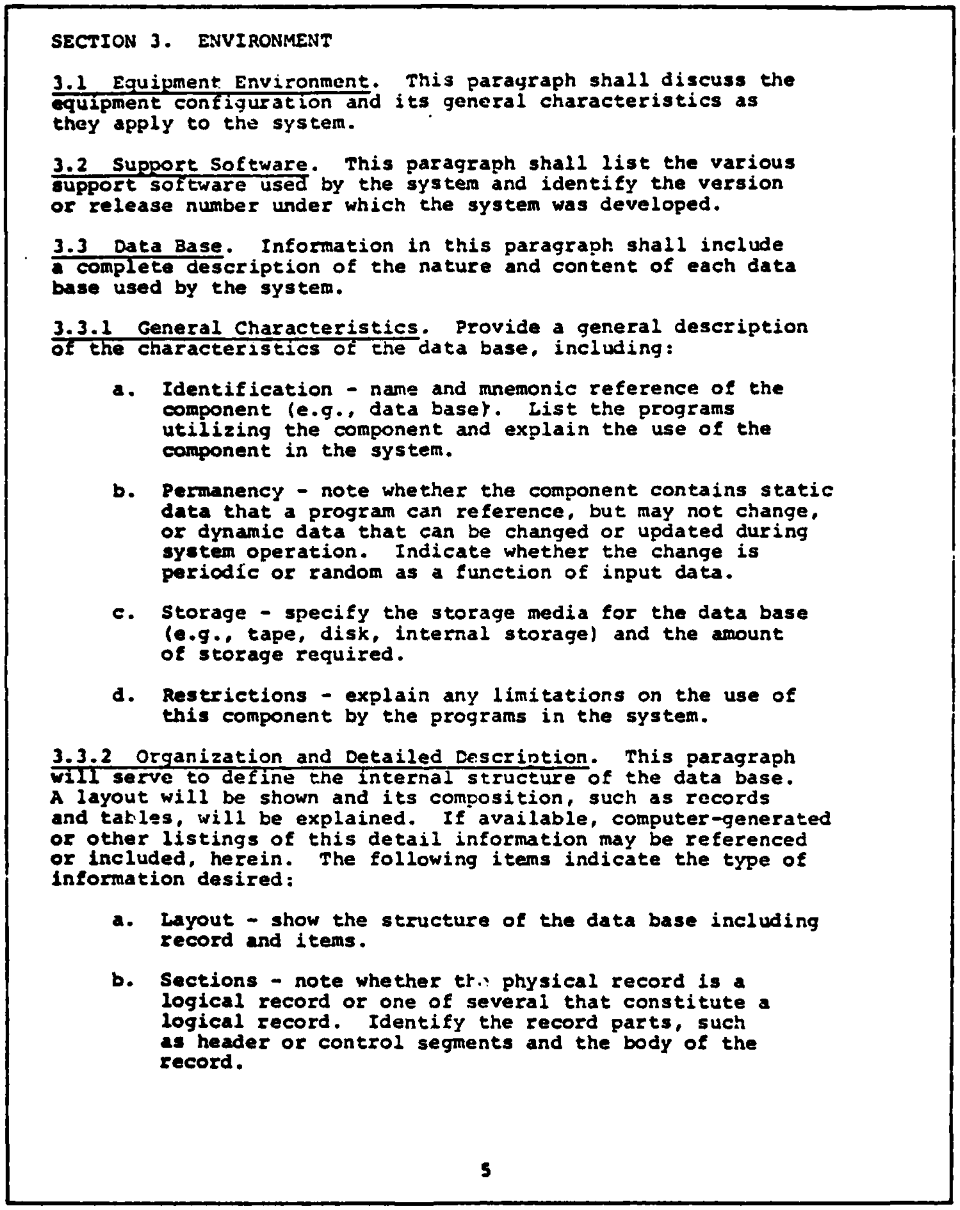
(7) Items - the term "item" refers to a specific category of detailed information that is coded for direct and immediate manipulation by a program. Used in this sense, the definition of an item is machine- and program-oriented rather than operationally oriented. Of primary importance is an explanation of the use of each item. At least the following will be included for each item:

(a) Item tag or label and full name.

(b) Purpose of the item.

(c) Item coding, depending upon the item type, such as integer, symbolic, status, etc.

h. Unique Run Features - description of any unique features of the running of this program that are not included in the Computer Operation Manual.



SECTION 3. ENVIRONMENT

3.1 Equipment Environment. This paragraph shall discuss the equipment configuration and its general characteristics as they apply to the system.

3.2 Support Software. This paragraph shall list the various support software used by the system and identify the version or release number under which the system was developed.

3.3 Data Base. Information in this paragraph shall include a complete description of the nature and content of each data base used by the system.

3.3.1 General Characteristics. Provide a general description of the characteristics of the data base, including:

a. Identification - name and mnemonic reference of the component (e.g., data base). List the programs utilizing the component and explain the use of the component in the system.

b. Permanency - note whether the component contains static data that a program can reference, but may not change, or dynamic data that can be changed or updated during system operation. Indicate whether the change is periodic or random as a function of input data.

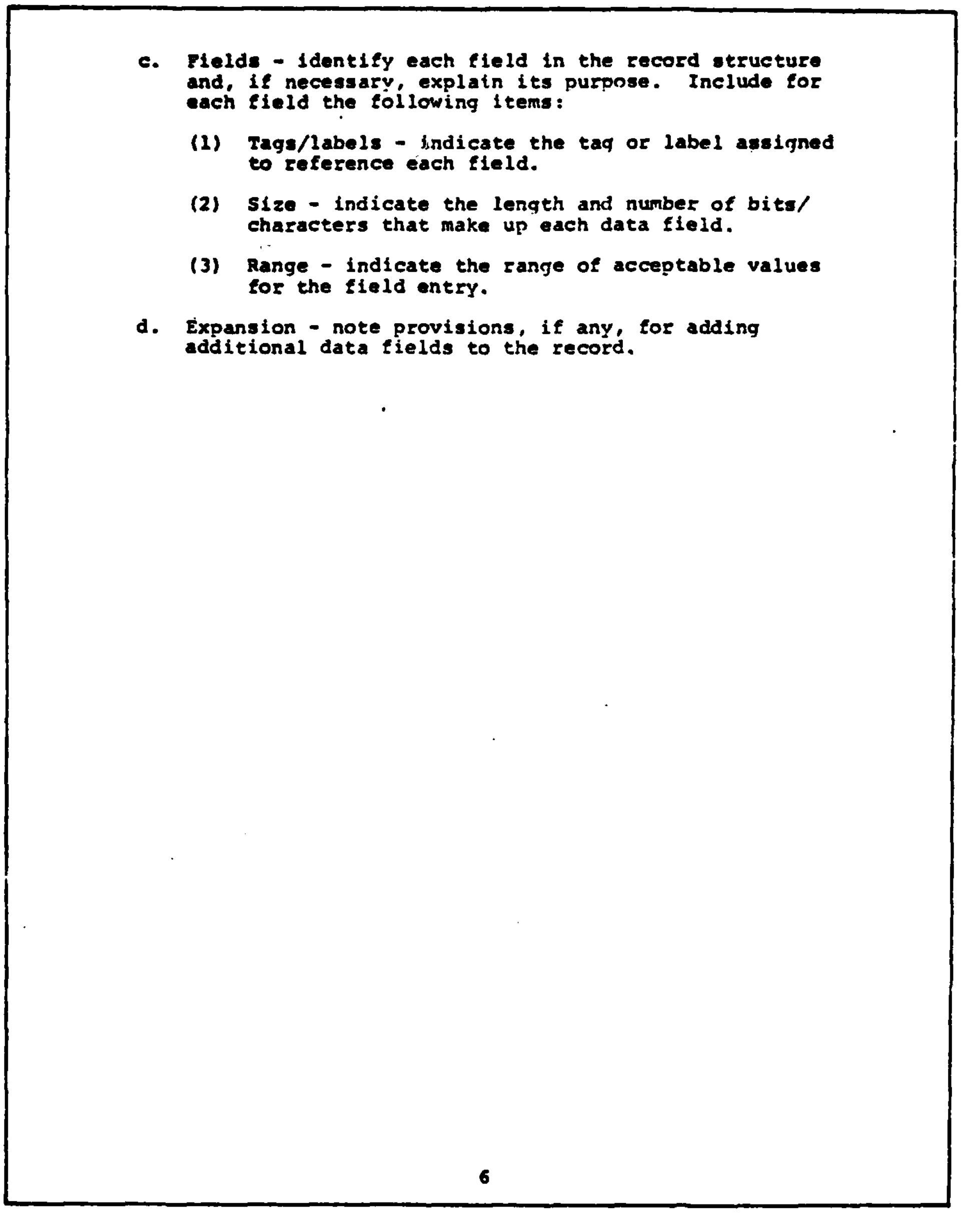
c. Storage - specify the storage media for the data base (e.g., tape, disk, internal storage) and the amount of storage required.

d. Restrictions - explain any limitations on the use of this component by the programs in the system.

3.3.2 Organization and Detailed Description. This paragraph will serve to define the internal structure of the data base. A layout will be shown and its composition, such as records and tables, will be explained. If available, computer-generated or other listings of this detail information may be referenced or included, herein. The following items indicate the type of information desired:

a. Layout - show the structure of the data base including record and items.

b. Sections - note whether the physical record is a logical record or one of several that constitute a logical record. Identify the record parts, such as header or control segments and the body of the record.



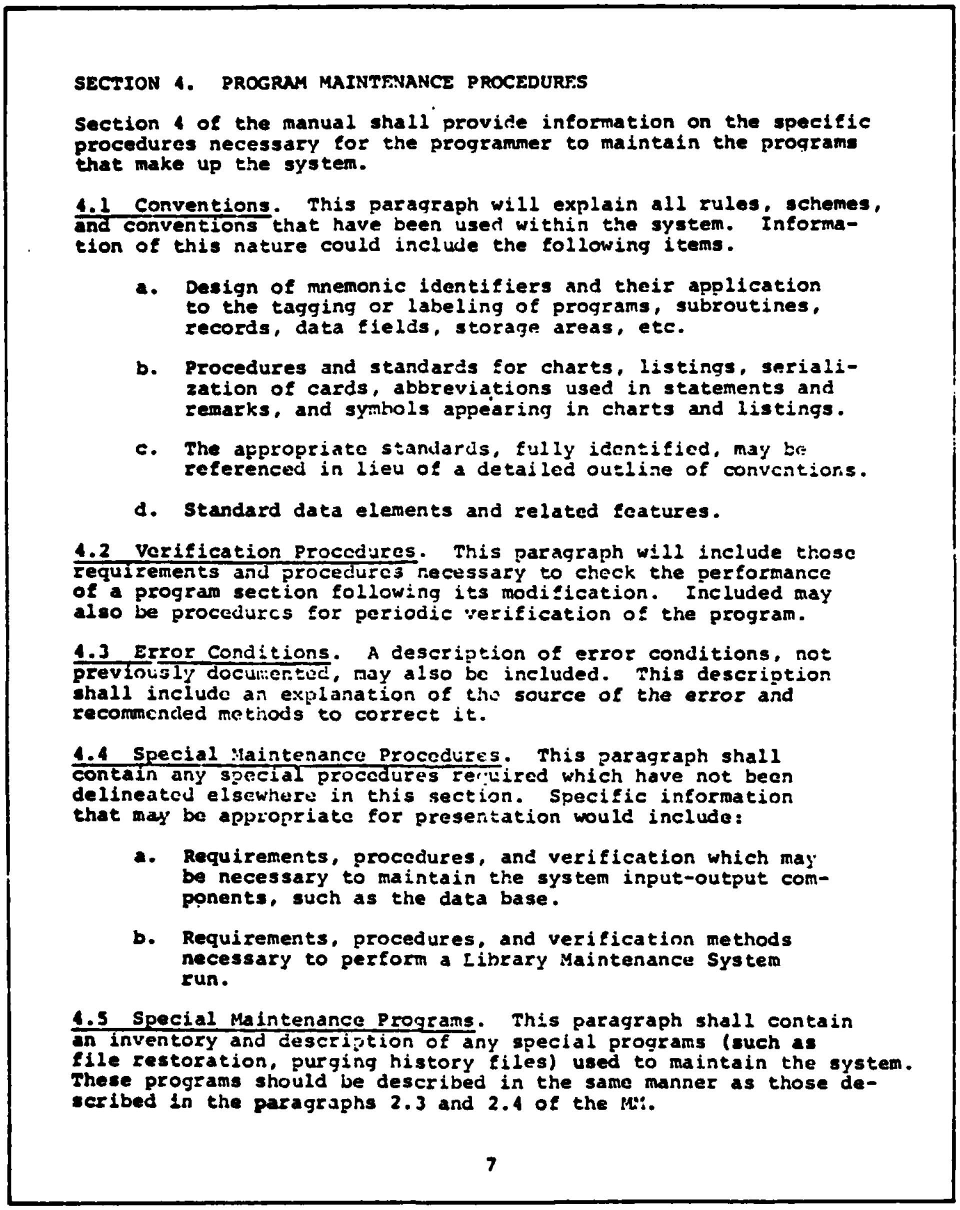
c. Fields - identify each field in the record structure and, if necessary, explain its purpose. Include for each field the following items:

(1) Tags/labels - indicate the tag or label assigned to reference each field.

(2) Size - indicate the length and number of bits/characters that make up each data field.

(3) Range - indicate the range of acceptable values for the field entry.

d. Expansion - note provisions, if any, for adding additional data fields to the record.



## SECTION 4. PROGRAM MAINTENANCE PROCEDURES

Section 4 of the manual shall provide information on the specific procedures necessary for the programmer to maintain the programs that make up the system.

4.1 Conventions. This paragraph will explain all rules, schemes, and conventions that have been used within the system. Information of this nature could include the following items.

a. Design of mnemonic identifiers and their application to the tagging or labeling of programs, subroutines, records, data fields, storage areas, etc.

b. Procedures and standards for charts, listings, serialization of cards, abbreviations used in statements and remarks, and symbols appearing in charts and listings.

c. The appropriate standards, fully identified, may be referenced in lieu of a detailed outline of conventions.

d. Standard data elements and related features.

4.2 Verification Procedures. This paragraph will include those requirements and procedures necessary to check the performance of a program section following its modification. Included may also be procedures for periodic verification of the program.

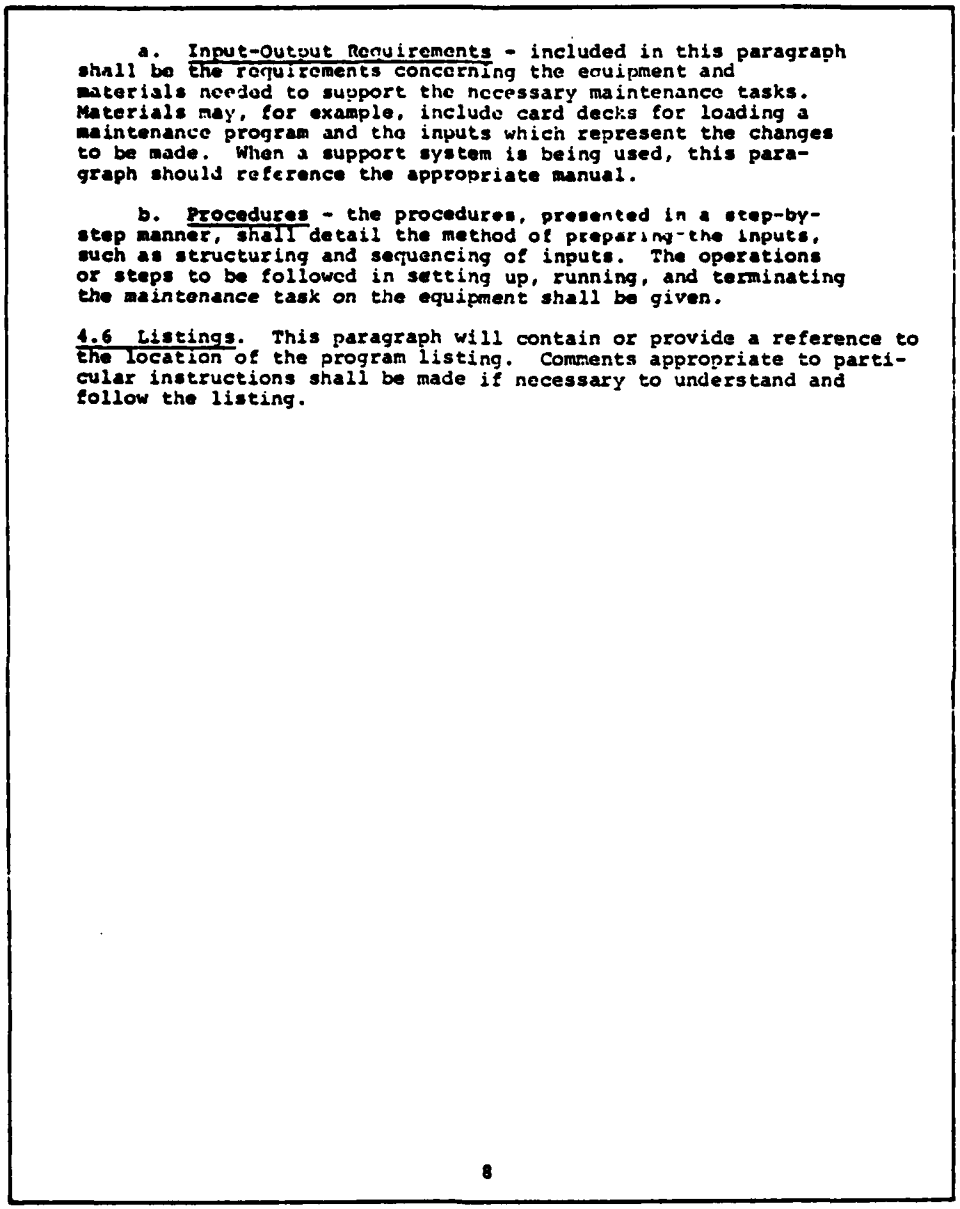
4.3 Error Conditions. A description of error conditions, not previously documented, may also be included. This description shall include an explanation of the source of the error and recommended methods to correct it.

4.4 Special Maintenance Procedures. This paragraph shall contain any special procedures required which have not been delineated elsewhere in this section. Specific information that may be appropriate for presentation would include:

a. Requirements, procedures, and verification which may be necessary to maintain the system input-output components, such as the data base.

b. Requirements, procedures, and verification methods necessary to perform a Library Maintenance System run.

4.5 Special Maintenance Programs. This paragraph shall contain an inventory and description of any special programs (such as file restoration, purging history files) used to maintain the system. These programs should be described in the same manner as those described in the paragraphs 2.3 and 2.4 of the MM.



a. Input-Output Requirements - included in this paragraph shall be the requirements concerning the equipment and materials needed to support the necessary maintenance tasks. Materials may, for example, include card decks for loading a maintenance program and the inputs which represent the changes to be made. When a support system is being used, this paragraph should reference the appropriate manual.

b. Procedures - the procedures, presented in a step-by-step manner, shall detail the method of preparing the inputs, such as structuring and sequencing of inputs. The operations or steps to be followed in setting up, running, and terminating the maintenance task on the equipment shall be given.

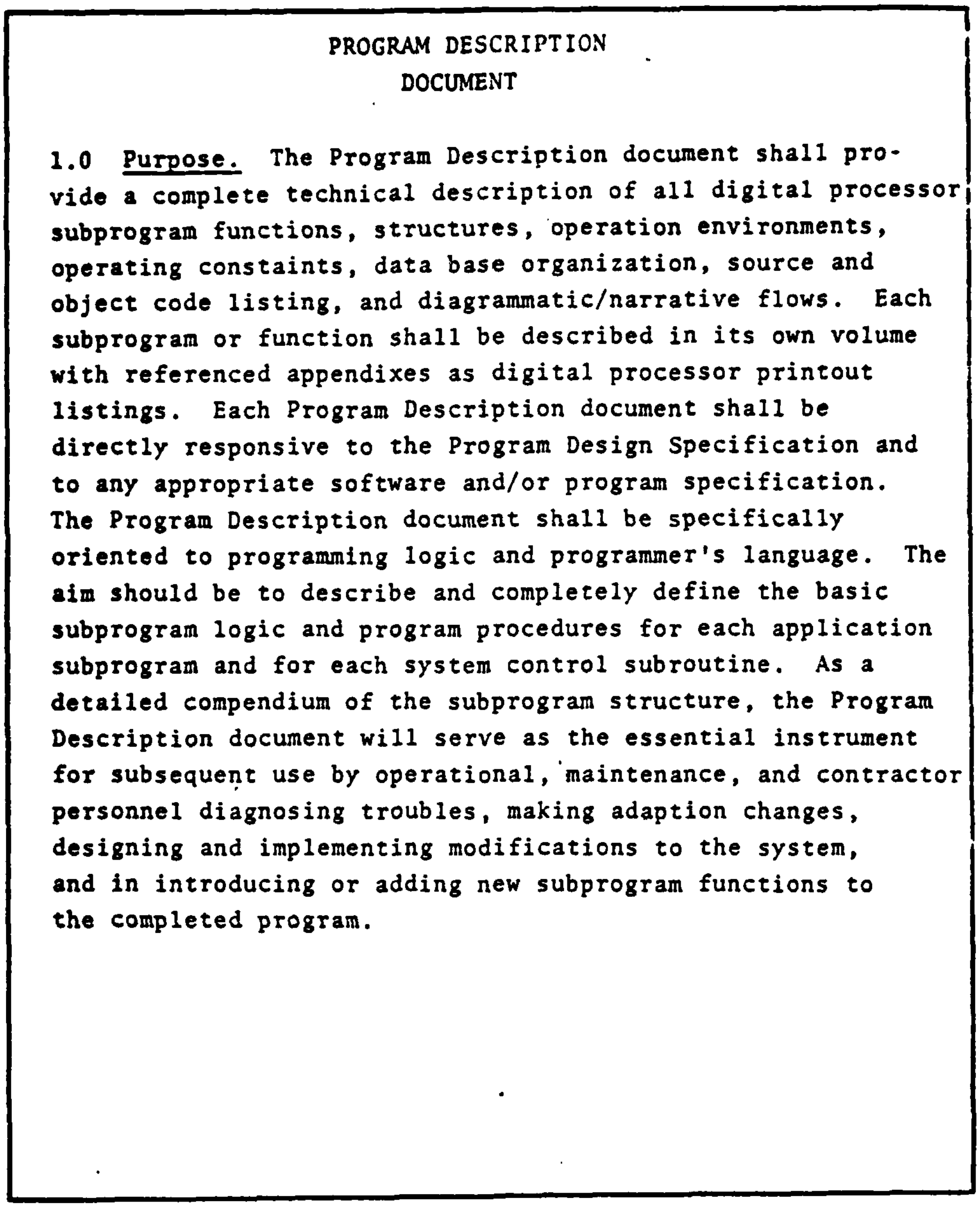
4.6 Listings. This paragraph will contain or provide a reference to the location of the program listing. Comments appropriate to particular instructions shall be made if necessary to understand and follow the listing.

APPENDIX B - <u>Combat System Program Description Group</u>

from: SECNAVINST 3560.1, "Tactical Digital Systems
Documentation Standards," 8 August 1974

## C. COMBAT SYSTEM PROGRAM DESCRIPTION GROUP

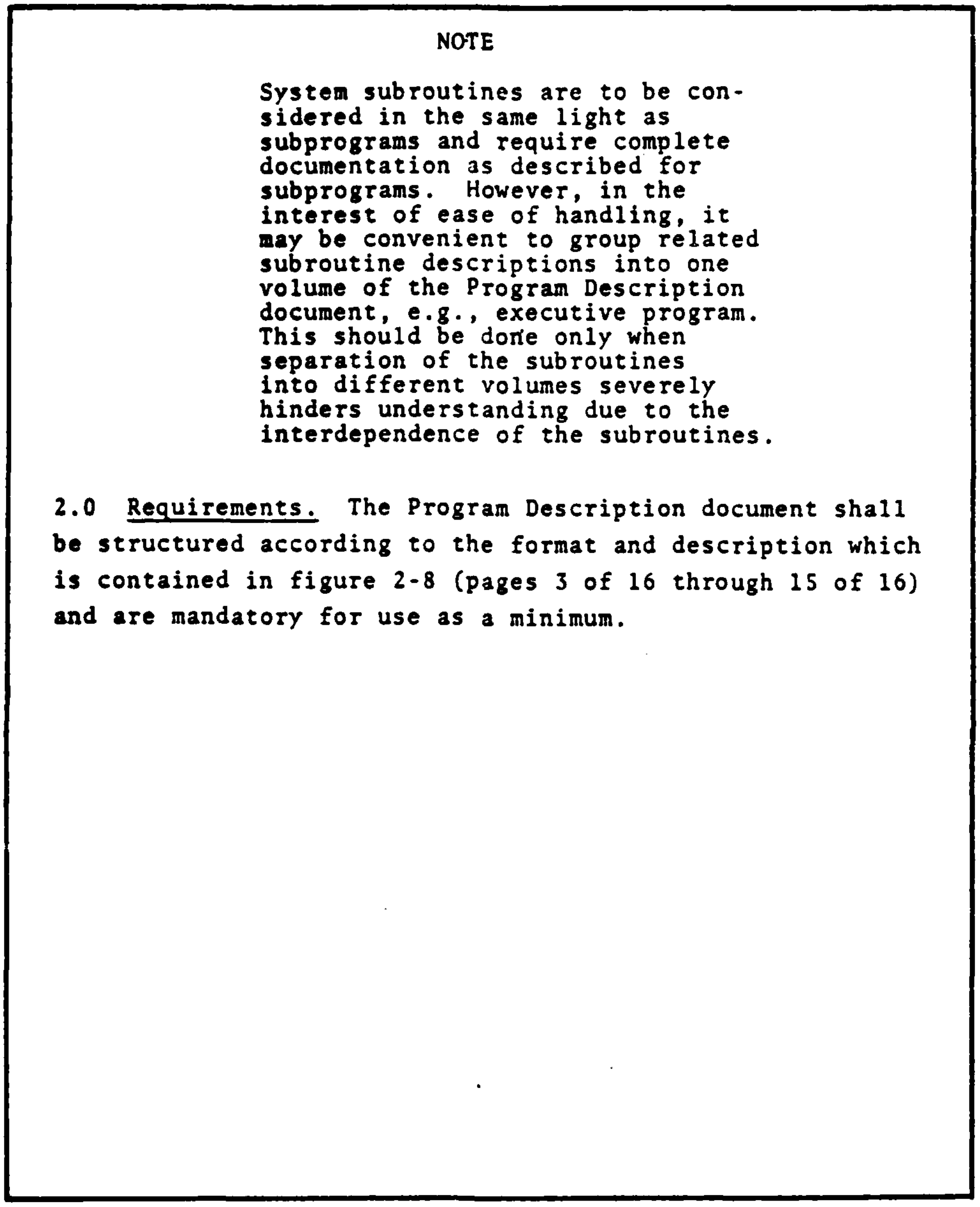
1. PDD – PROGRAM DESCRIPTION DOCUMENT
2. DBD – DATA BASE DESIGN
3. PP – PROGRAM PACKAGE



PROGRAM DESCRIPTION
DOCUMENT

1.0 Purpose. The Program Description document shall provide a complete technical description of all digital processor subprogram functions, structures, operation environments, operating constaints, data base organization, source and object code listing, and diagrammatic/narrative flows. Each subprogram or function shall be described in its own volume with referenced appendixes as digital processor printout listings. Each Program Description document shall be directly responsive to the Program Design Specification and to any appropriate software and/or program specification. The Program Description document shall be specifically oriented to programming logic and programmer's language. The aim should be to describe and completely define the basic subprogram logic and program procedures for each application subprogram and for each system control subroutine. As a detailed compendium of the subprogram structure, the Program Description document will serve as the essential instrument for subsequent use by operational, maintenance, and contractor personnel diagnosing troubles, making adaption changes, designing and implementing modifications to the system, and in introducing or adding new subprogram functions to the completed program.

Figure 2-8. Program Description Document (Page 1 of 16)



NOTE

System subroutines are to be considered in the same light as subprograms and require complete documentation as described for subprograms. However, in the interest of ease of handling, it may be convenient to group related subroutine descriptions into one volume of the Program Description document, e.g., executive program. This should be done only when separation of the subroutines into different volumes severely hinders understanding due to the interdependence of the subroutines.

2.0 <u>Requirements.</u> The Program Description document shall be structured according to the format and description which is contained in figure 2-8 (pages 3 of 16 through 15 of 16) and are mandatory for use as a minimum.

Figure 2-8. Program Description Document (Page 2 of 16)

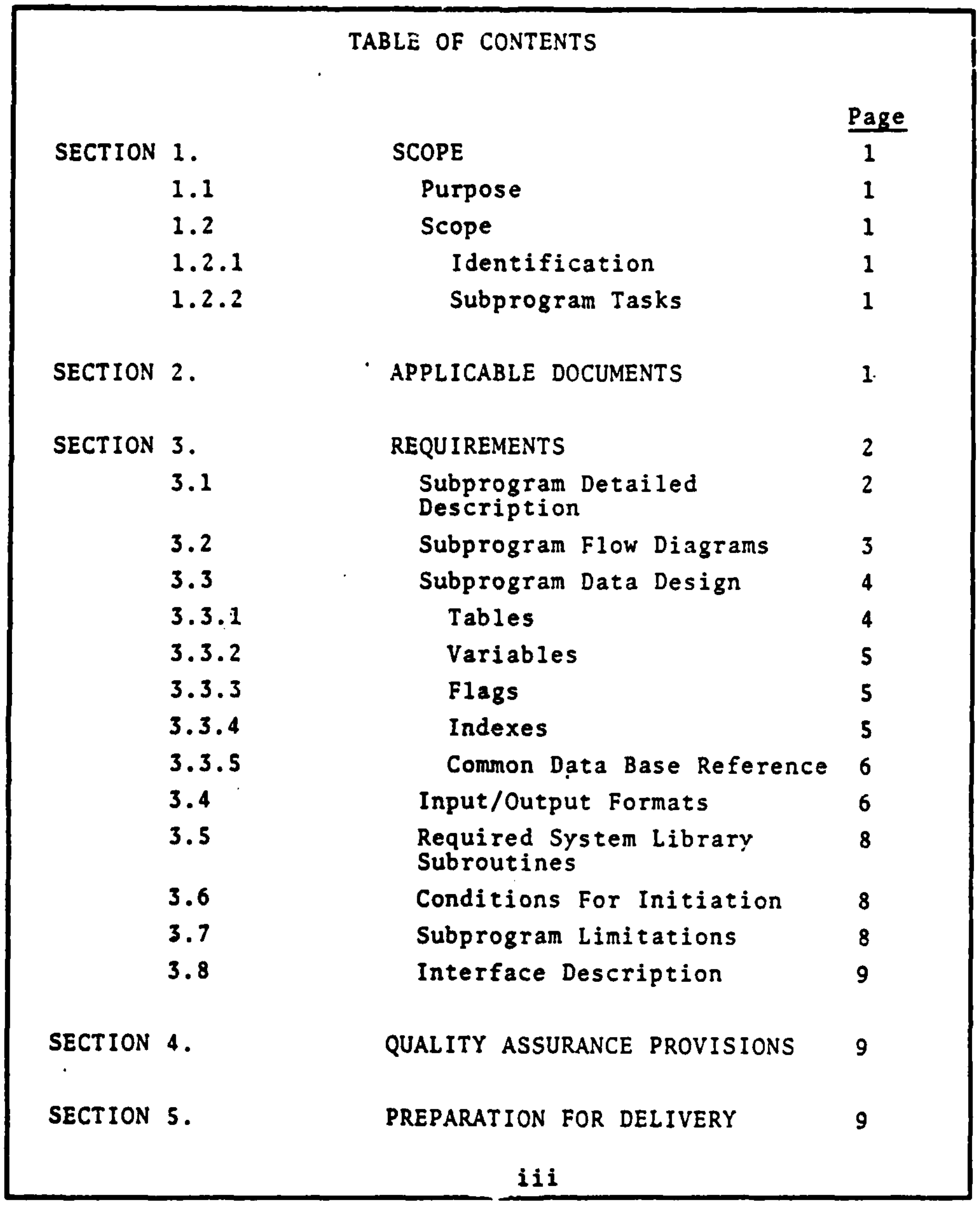


TABLE OF CONTENTS

| | | Page |
|---|---|---|
| SECTION 1. | SCOPE | 1 |
| 1.1 | Purpose | 1 |
| 1.2 | Scope | 1 |
| 1.2.1 | Identification | 1 |
| 1.2.2 | Subprogram Tasks | 1 |
| SECTION 2. | APPLICABLE DOCUMENTS | 1 |
| SECTION 3. | REQUIREMENTS | 2 |
| 3.1 | Subprogram Detailed Description | 2 |
| 3.2 | Subprogram Flow Diagrams | 3 |
| 3.3 | Subprogram Data Design | 4 |
| 3.3.1 | Tables | 4 |
| 3.3.2 | Variables | 5 |
| 3.3.3 | Flags | 5 |
| 3.3.4 | Indexes | 5 |
| 3.3.5 | Common Data Base Reference | 6 |
| 3.4 | Input/Output Formats | 6 |
| 3.5 | Required System Library Subroutines | 8 |
| 3.6 | Conditions For Initiation | 8 |
| 3.7 | Subprogram Limitations | 8 |
| 3.8 | Interface Description | 9 |
| SECTION 4. | QUALITY ASSURANCE PROVISIONS | 9 |
| SECTION 5. | PREPARATION FOR DELIVERY | 9 |

iii

Figure 2-8. Program Description Document (Page 3 of 16)

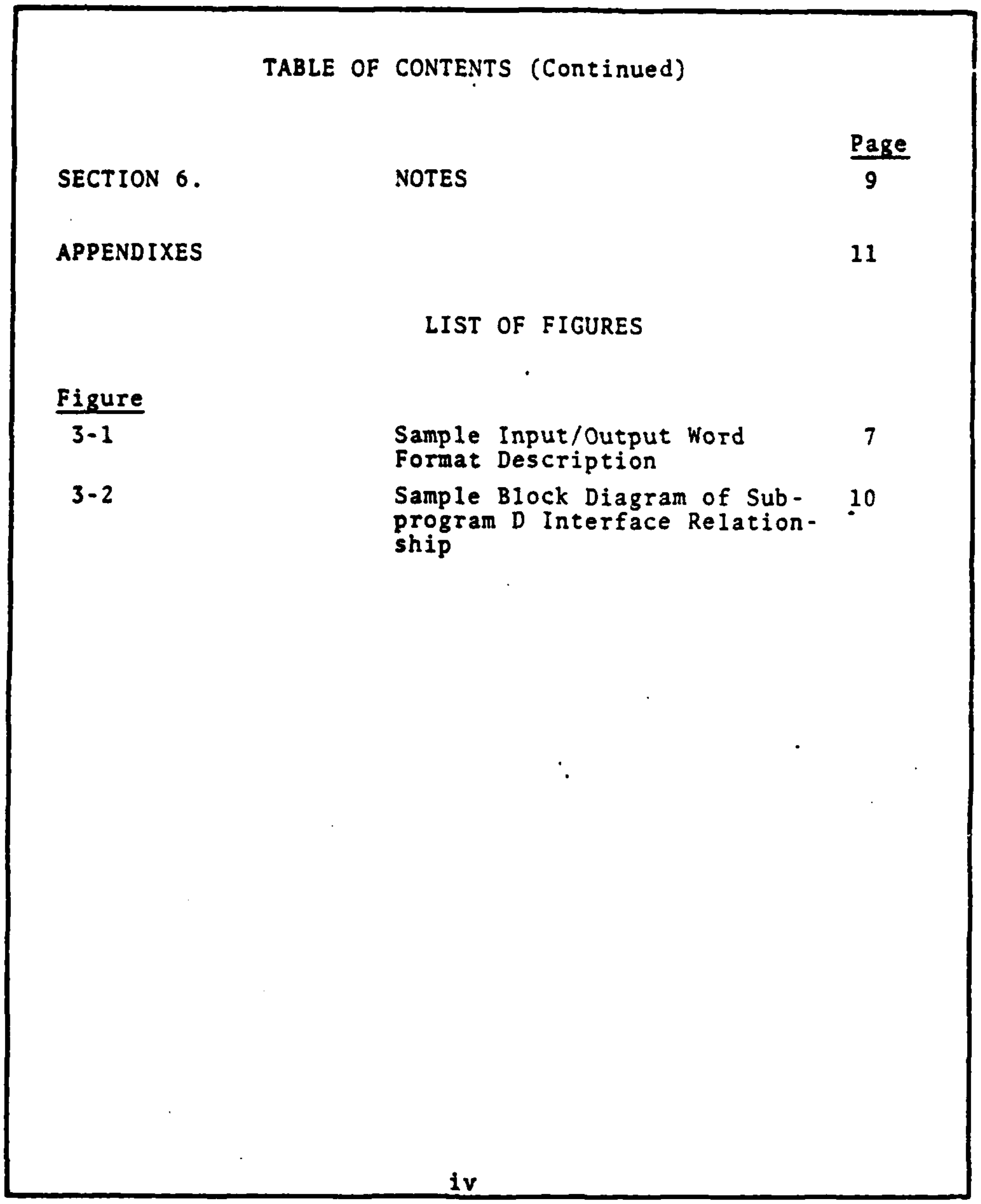
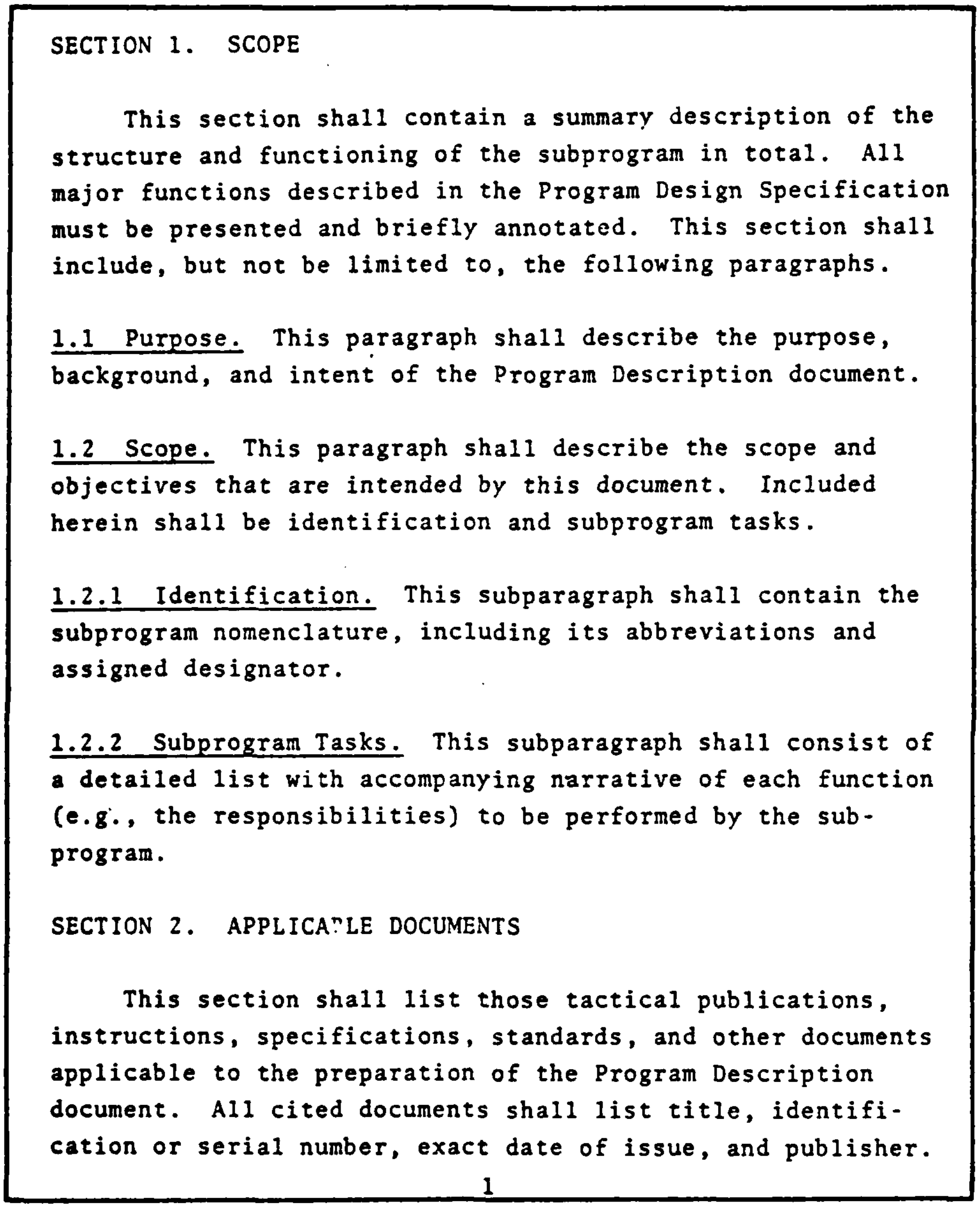


TABLE OF CONTENTS (Continued)

LIST OF FIGURES

iv

Figure 2-8. Program Description Document (Page 4 of 16)



### SECTION 1. SCOPE

This section shall contain a summary description of the structure and functioning of the subprogram in total. All major functions described in the Program Design Specification must be presented and briefly annotated. This section shall include, but not be limited to, the following paragraphs.

1.1 Purpose. This paragraph shall describe the purpose, background, and intent of the Program Description document.

1.2 Scope. This paragraph shall describe the scope and objectives that are intended by this document. Included herein shall be identification and subprogram tasks.

1.2.1 Identification. This subparagraph shall contain the subprogram nomenclature, including its abbreviations and assigned designator.

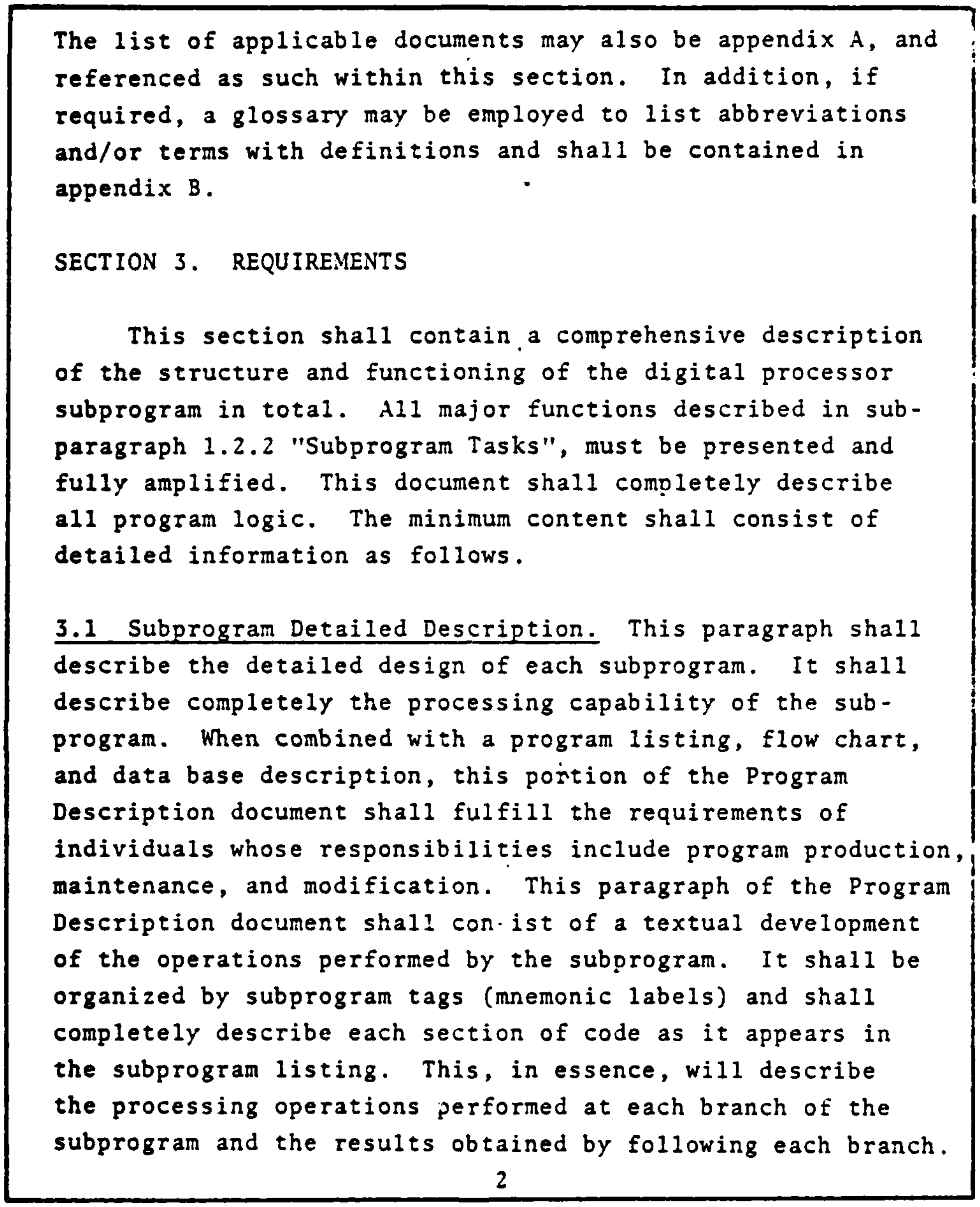
1.2.2 Subprogram Tasks. This subparagraph shall consist of a detailed list with accompanying narrative of each function (e.g., the responsibilities) to be performed by the subprogram.

### SECTION 2. APPLICA"LE DOCUMENTS

This section shall list those tactical publications, instructions, specifications, standards, and other documents applicable to the preparation of the Program Description document. All cited documents shall list title, identification or serial number, exact date of issue, and publisher.

1

Figure 2-8. Program Description Document (Page 5 of 16)



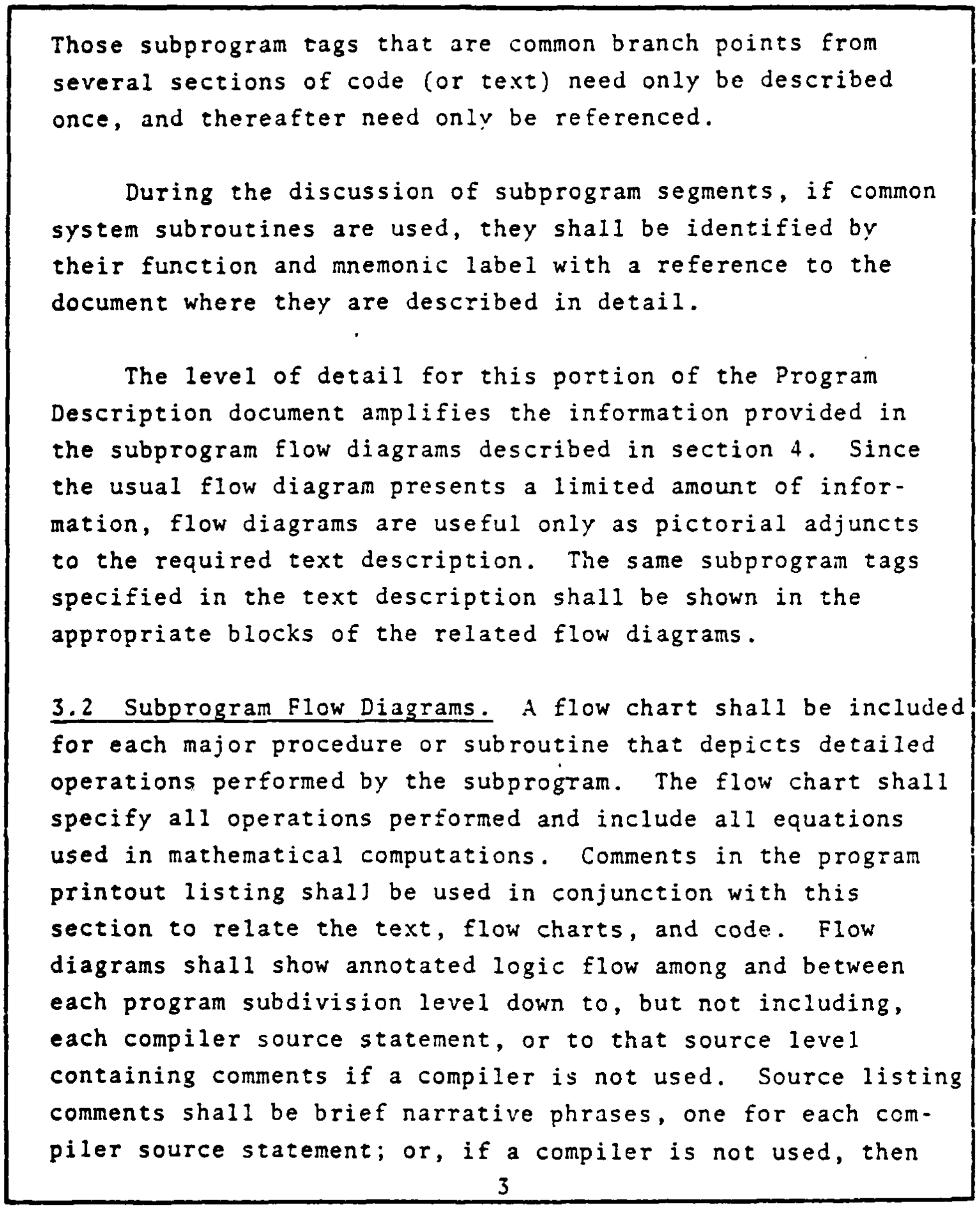
The list of applicable documents may also be appendix A, and referenced as such within this section. In addition, if required, a glossary may be employed to list abbreviations and/or terms with definitions and shall be contained in appendix B.

SECTION 3. REQUIREMENTS

This section shall contain a comprehensive description of the structure and functioning of the digital processor subprogram in total. All major functions described in subparagraph 1.2.2 "Subprogram Tasks", must be presented and fully amplified. This document shall completely describe all program logic. The minimum content shall consist of detailed information as follows.

3.1 Subprogram Detailed Description. This paragraph shall describe the detailed design of each subprogram. It shall describe completely the processing capability of the subprogram. When combined with a program listing, flow chart, and data base description, this portion of the Program Description document shall fulfill the requirements of individuals whose responsibilities include program production, maintenance, and modification. This paragraph of the Program Description document shall con·ist of a textual development of the operations performed by the subprogram. It shall be organized by subprogram tags (mnemonic labels) and shall completely describe each section of code as it appears in the subprogram listing. This, in essence, will describe the processing operations performed at each branch of the subprogram and the results obtained by following each branch.

Figure 2-8. Program Description Document (Page 6 of 16)



Those subprogram tags that are common branch points from several sections of code (or text) need only be described once, and thereafter need only be referenced.

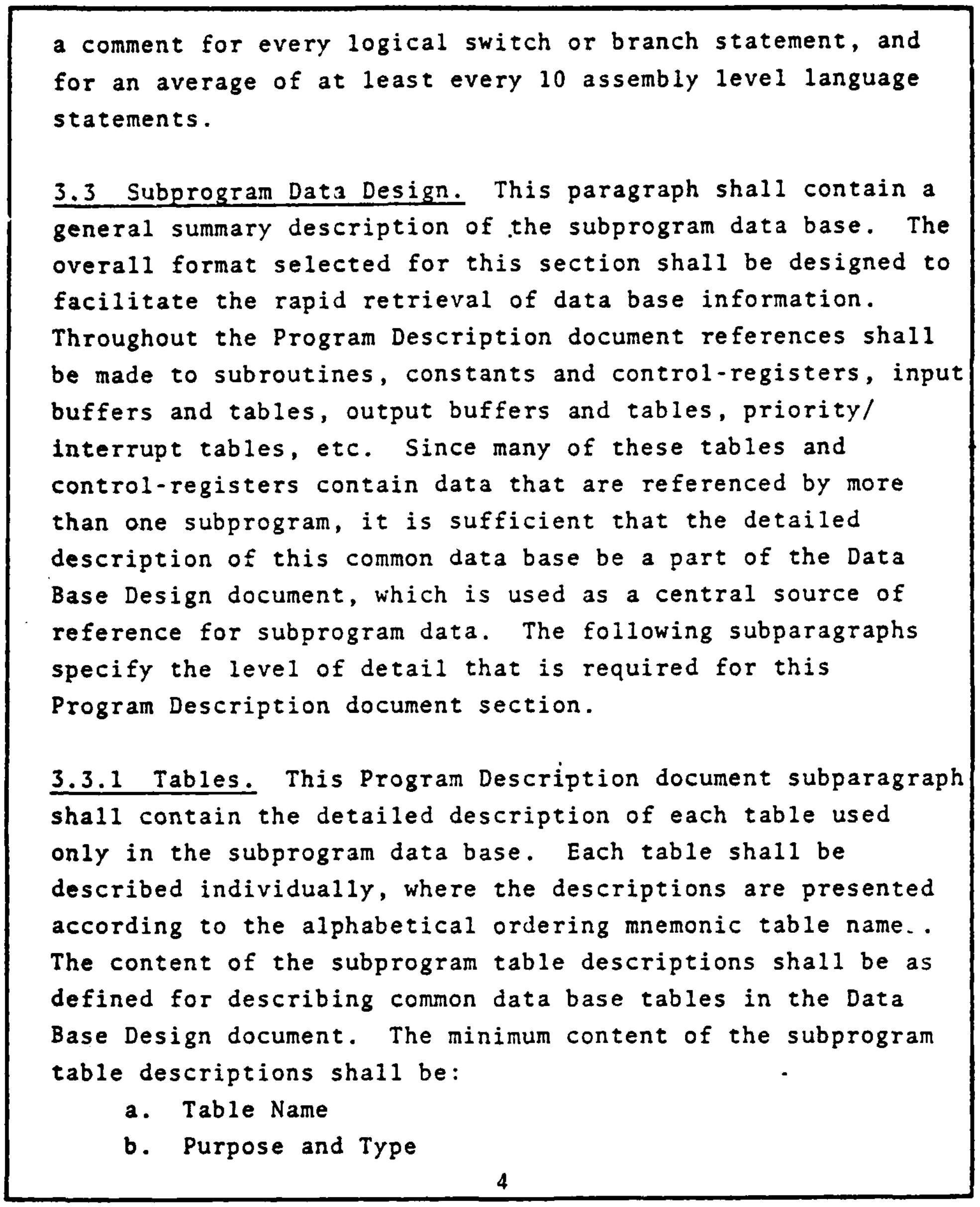
During the discussion of subprogram segments, if common system subroutines are used, they shall be identified by their function and mnemonic label with a reference to the document where they are described in detail.

The level of detail for this portion of the Program Description document amplifies the information provided in the subprogram flow diagrams described in section 4. Since the usual flow diagram presents a limited amount of information, flow diagrams are useful only as pictorial adjuncts to the required text description. The same subprogram tags specified in the text description shall be shown in the appropriate blocks of the related flow diagrams.

3.2 Subprogram Flow Diagrams. A flow chart shall be included for each major procedure or subroutine that depicts detailed operations performed by the subprogram. The flow chart shall specify all operations performed and include all equations used in mathematical computations. Comments in the program printout listing shall be used in conjunction with this section to relate the text, flow charts, and code. Flow diagrams shall show annotated logic flow among and between each program subdivision level down to, but not including, each compiler source statement, or to that source level containing comments if a compiler is not used. Source listing comments shall be brief narrative phrases, one for each compiler source statement; or, if a compiler is not used, then

3

Figure 2-8. Program Description Document (Page 7 of 16)



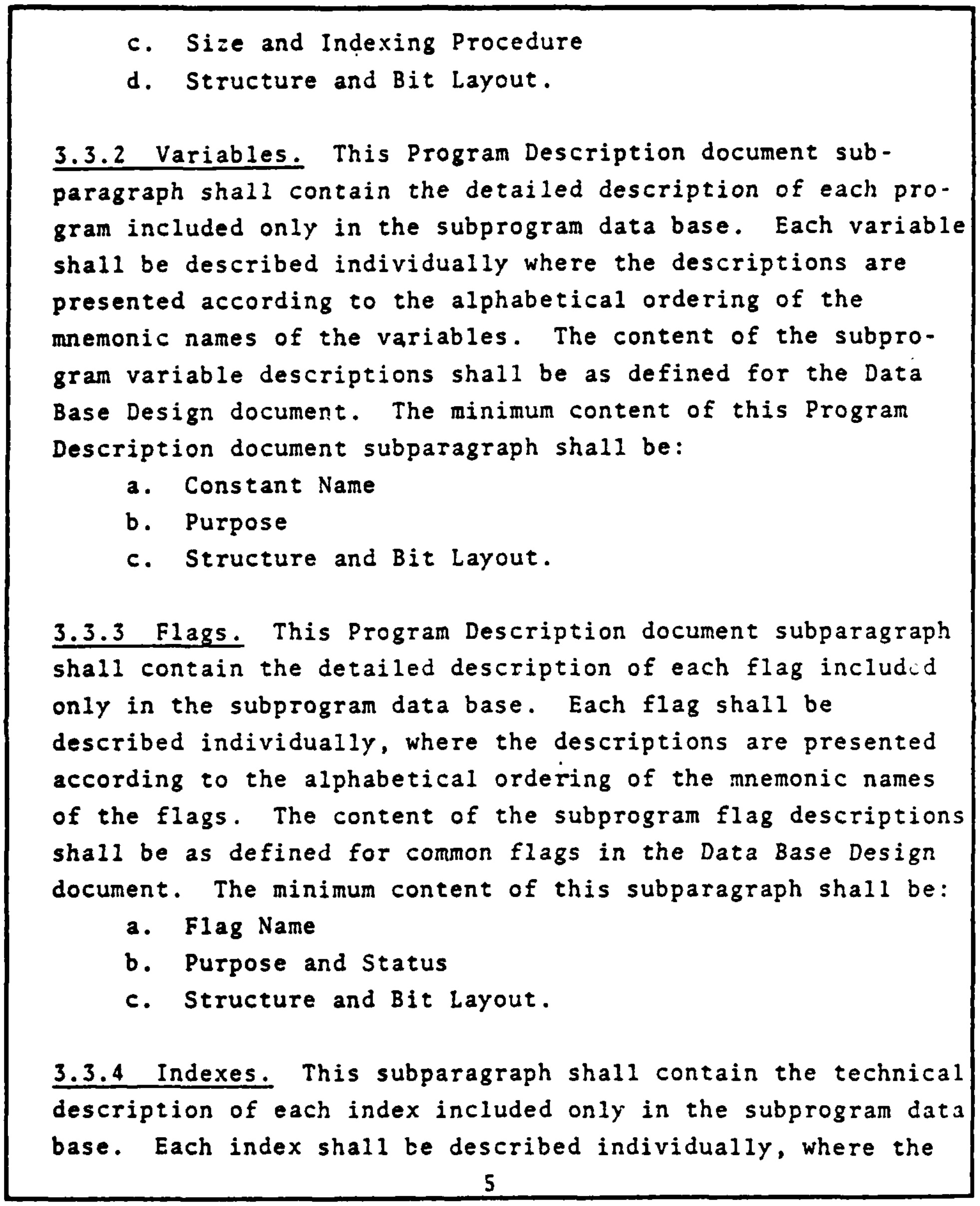
a comment for every logical switch or branch statement, and for an average of at least every 10 assembly level language statements.

3.3 Subprogram Data Design. This paragraph shall contain a general summary description of the subprogram data base. The overall format selected for this section shall be designed to facilitate the rapid retrieval of data base information. Throughout the Program Description document references shall be made to subroutines, constants and control-registers, input buffers and tables, output buffers and tables, priority/ interrupt tables, etc. Since many of these tables and control-registers contain data that are referenced by more than one subprogram, it is sufficient that the detailed description of this common data base be a part of the Data Base Design document, which is used as a central source of reference for subprogram data. The following subparagraphs specify the level of detail that is required for this Program Description document section.

3.3.1 Tables. This Program Description document subparagraph shall contain the detailed description of each table used only in the subprogram data base. Each table shall be described individually, where the descriptions are presented according to the alphabetical ordering mnemonic table name.. The content of the subprogram table descriptions shall be as defined for describing common data base tables in the Data Base Design document. The minimum content of the subprogram table descriptions shall be:

a. Table Name
b. Purpose and Type

Figure 2-8. Program Description Document (Page 8 of 16)



c. Size and Indexing Procedure

d. Structure and Bit Layout.

3.3.2 Variables. This Program Description document subparagraph shall contain the detailed description of each program included only in the subprogram data base. Each variable shall be described individually where the descriptions are presented according to the alphabetical ordering of the mnemonic names of the variables. The content of the subprogram variable descriptions shall be as defined for the Data Base Design document. The minimum content of this Program Description document subparagraph shall be:

a. Constant Name

b. Purpose

c. Structure and Bit Layout.

3.3.3 Flags. This Program Description document subparagraph shall contain the detailed description of each flag included only in the subprogram data base. Each flag shall be described individually, where the descriptions are presented according to the alphabetical ordering of the mnemonic names of the flags. The content of the subprogram flag descriptions shall be as defined for common flags in the Data Base Design document. The minimum content of this subparagraph shall be:

a. Flag Name

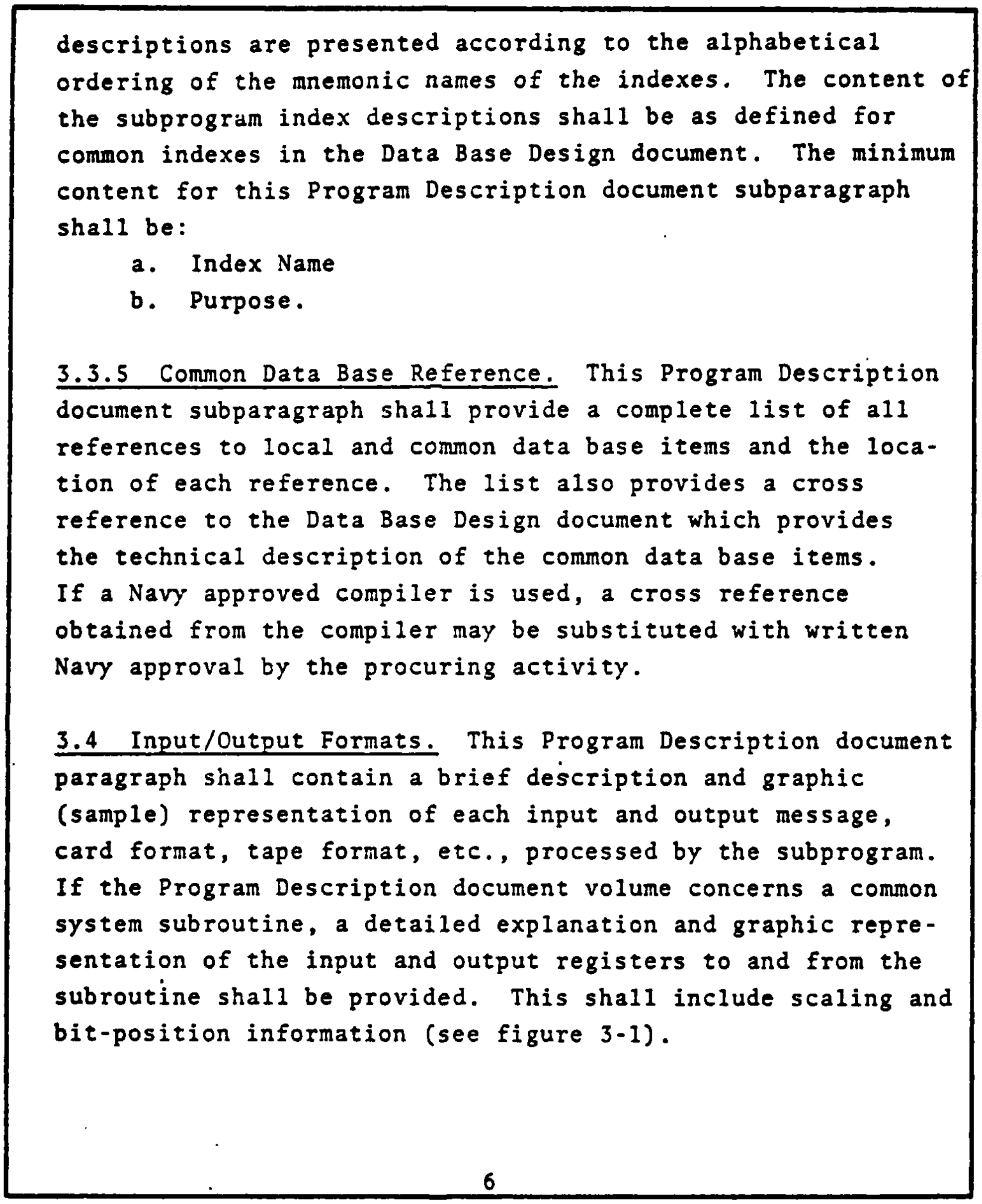
b. Purpose and Status

c. Structure and Bit Layout.

3.3.4 Indexes. This subparagraph shall contain the technical description of each index included only in the subprogram data base. Each index shall be described individually, where the

5

Figure 2-8. Program Description Document (Page 9 of 16)



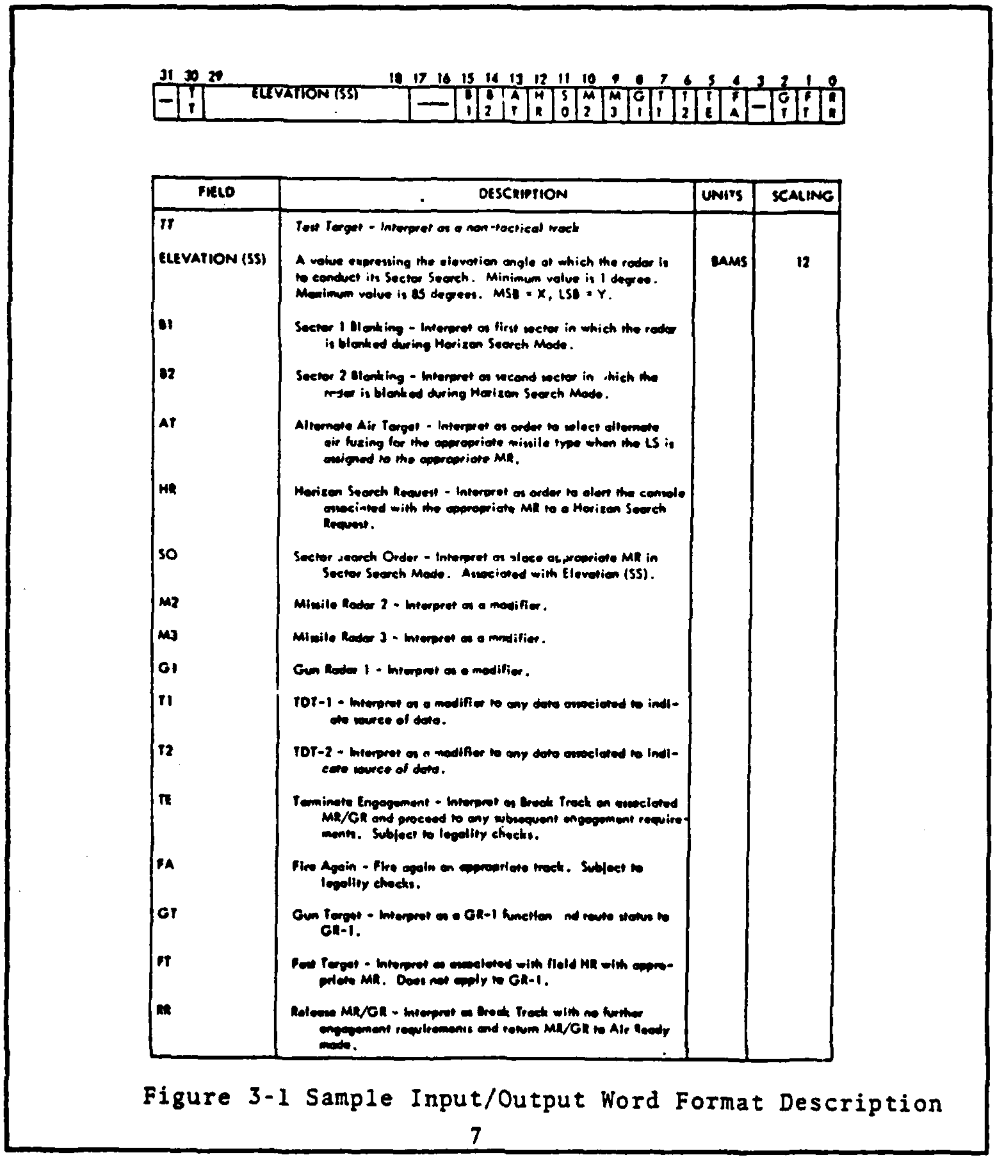
descriptions are presented according to the alphabetical ordering of the mnemonic names of the indexes. The content of the subprogram index descriptions shall be as defined for common indexes in the Data Base Design document. The minimum content for this Program Description document subparagraph shall be:

a. Index Name
b. Purpose.

3.3.5 Common Data Base Reference. This Program Description document subparagraph shall provide a complete list of all references to local and common data base items and the location of each reference. The list also provides a cross reference to the Data Base Design document which provides the technical description of the common data base items. If a Navy approved compiler is used, a cross reference obtained from the compiler may be substituted with written Navy approval by the procuring activity.

3.4 Input/Output Formats. This Program Description document paragraph shall contain a brief description and graphic (sample) representation of each input and output message, card format, tape format, etc., processed by the subprogram. If the Program Description document volume concerns a common system subroutine, a detailed explanation and graphic representation of the input and output registers to and from the subroutine shall be provided. This shall include scaling and bit-position information (see figure 3-1).

Figure 2-8. Program Description Document (Page 10 of 16)



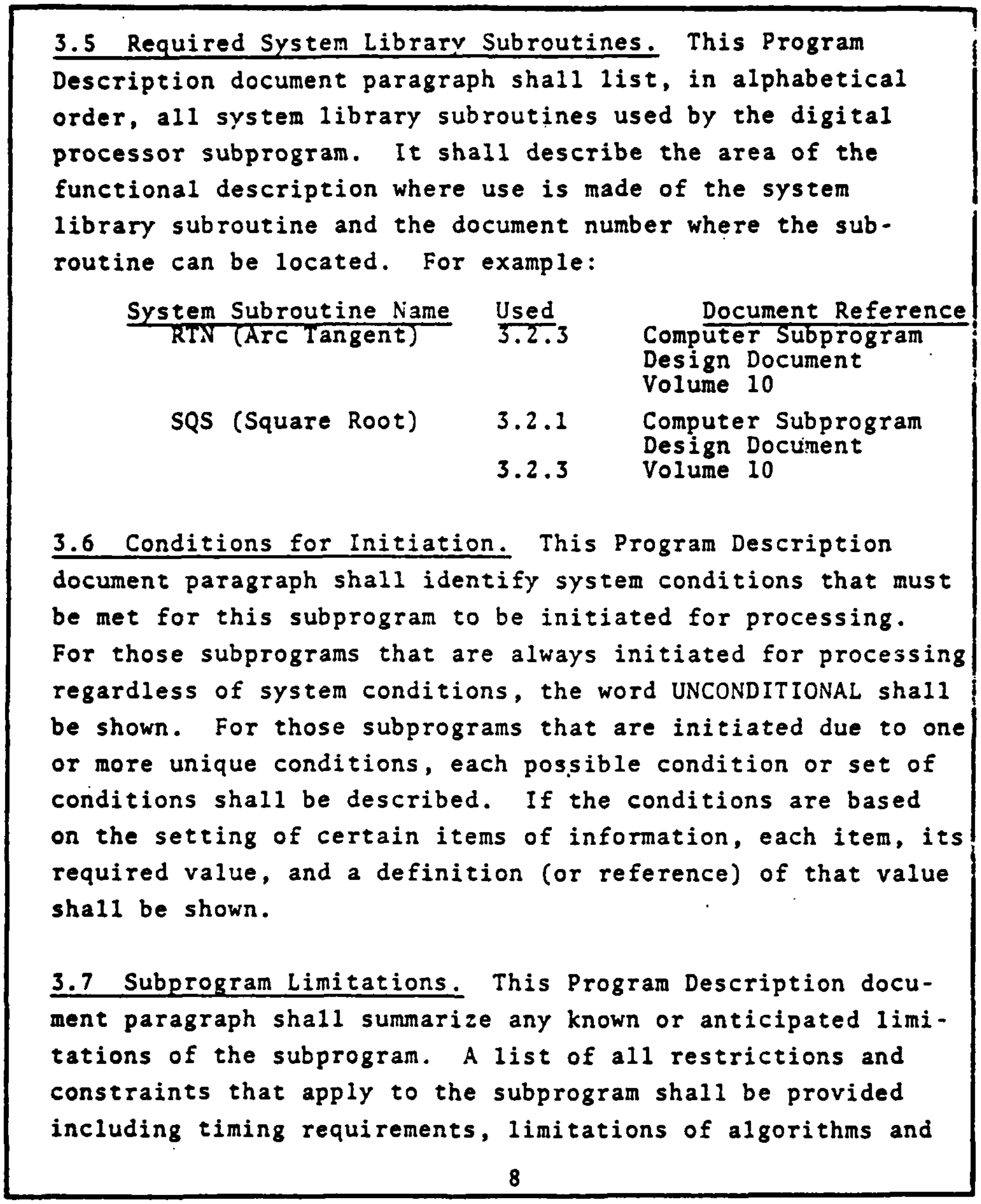
| 31 | 30 | 29 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| — | T<br>T | ELEVATION (SS) |  | —— |  | B<br>1 | B<br>2 | A<br>T | H<br>R | S<br>O | M<br>2 | M<br>3 | G<br>1 | T<br>1 | T<br>2 | T<br>E | F<br>A | — | G<br>T | F<br>T | R<br>R |

| FIELD | DESCRIPTION | UNITS | SCALING |
|---|---|---|---|
| TT | Test Target - Interpret as a non-tactical track |  |  |
| ELEVATION (SS) | A value expressing the elevation angle at which the radar is to conduct its Sector Search. Minimum value is 1 degree. Maximum value is 85 degrees. MSB = X, LSB = Y. | BAMS | 12 |
| B1 | Sector 1 Blanking - Interpret as first sector in which the radar is blanked during Horizon Search Mode. |  |  |
| B2 | Sector 2 Blanking - Interpret as second sector in which the radar is blanked during Horizon Search Mode. |  |  |
| AT | Alternate Air Target - Interpret as order to select alternate air fuzing for the appropriate missile type when the LS is assigned to the appropriate MR. |  |  |
| HR | Horizon Search Request - Interpret as order to alert the console associated with the appropriate MR to a Horizon Search Request. |  |  |
| SO | Sector search Order - Interpret as place appropriate MR in Sector Search Mode. Associated with Elevation (SS). |  |  |
| M2 | Missile Radar 2 - Interpret as a modifier. |  |  |
| M3 | Missile Radar 3 - Interpret as a modifier. |  |  |
| G1 | Gun Radar 1 - Interpret as a modifier. |  |  |
| T1 | TDT-1 - Interpret as a modifier to any data associated to indicate source of data. |  |  |
| T2 | TDT-2 - Interpret as a modifier to any data associated to indicate source of data. |  |  |
| TE | Terminate Engagement - Interpret as Break Track on associated MR/GR and proceed to any subsequent engagement requirements. Subject to legality checks. |  |  |
| FA | Fire Again - Fire again on appropriate track. Subject to legality checks. |  |  |
| GT | Gun Target - Interpret as a GR-1 function nd route status to GR-1. |  |  |
| FT | Fast Target - Interpret as associated with field HR with appropriate MR. Does not apply to GR-1. |  |  |
| RR | Release MR/GR - Interpret as Break Track with no further engagement requirements and return MR/GR to Air Ready mode. |  |  |

Figure 3-1 Sample Input/Output Word Format Description

7

Figure 2-8. Program Description Document (Page 11 of 16)



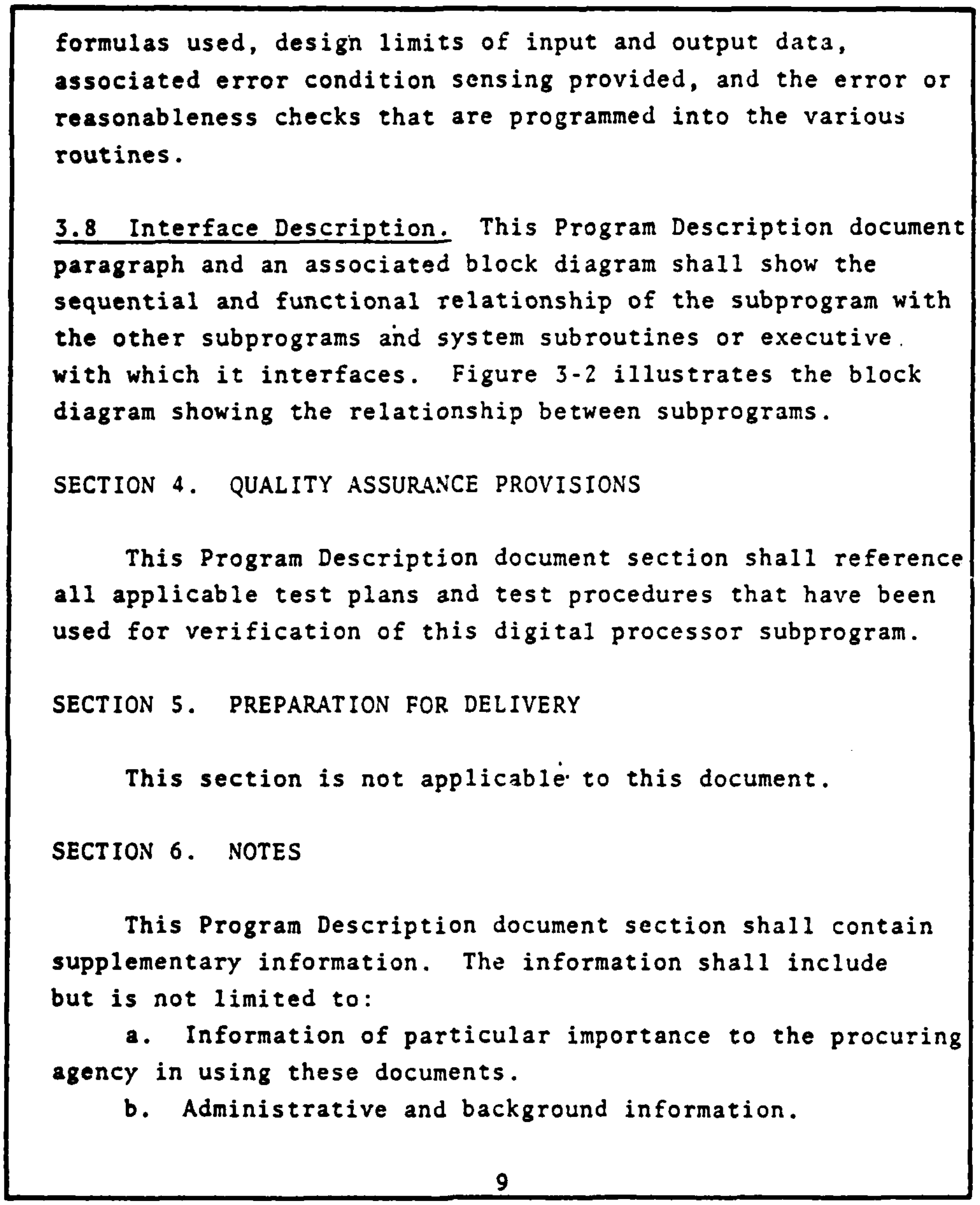
3.5 Required System Library Subroutines. This Program Description document paragraph shall list, in alphabetical order, all system library subroutines used by the digital processor subprogram. It shall describe the area of the functional description where use is made of the system library subroutine and the document number where the subroutine can be located. For example:

| System Subroutine Name | Used | Document Reference |
|---|---|---|
| RTN (Arc Tangent) | 3.2.3 | Computer Subprogram Design Document Volume 10 |
| SQS (Square Root) | 3.2.1<br>3.2.3 | Computer Subprogram Design Document Volume 10 |

3.6 Conditions for Initiation. This Program Description document paragraph shall identify system conditions that must be met for this subprogram to be initiated for processing. For those subprograms that are always initiated for processing regardless of system conditions, the word UNCONDITIONAL shall be shown. For those subprograms that are initiated due to one or more unique conditions, each possible condition or set of conditions shall be described. If the conditions are based on the setting of certain items of information, each item, its required value, and a definition (or reference) of that value shall be shown.

3.7 Subprogram Limitations. This Program Description document paragraph shall summarize any known or anticipated limitations of the subprogram. A list of all restrictions and constraints that apply to the subprogram shall be provided including timing requirements, limitations of algorithms and

Figure 2-8. Program Description Document (Page 12 of 16)



formulas used, design limits of input and output data, associated error condition sensing provided, and the error or reasonableness checks that are programmed into the various routines.

3.8 Interface Description. This Program Description document paragraph and an associated block diagram shall show the sequential and functional relationship of the subprogram with the other subprograms and system subroutines or executive with which it interfaces. Figure 3-2 illustrates the block diagram showing the relationship between subprograms.

SECTION 4. QUALITY ASSURANCE PROVISIONS

This Program Description document section shall reference all applicable test plans and test procedures that have been used for verification of this digital processor subprogram.

SECTION 5. PREPARATION FOR DELIVERY

This section is not applicable to this document.

SECTION 6. NOTES

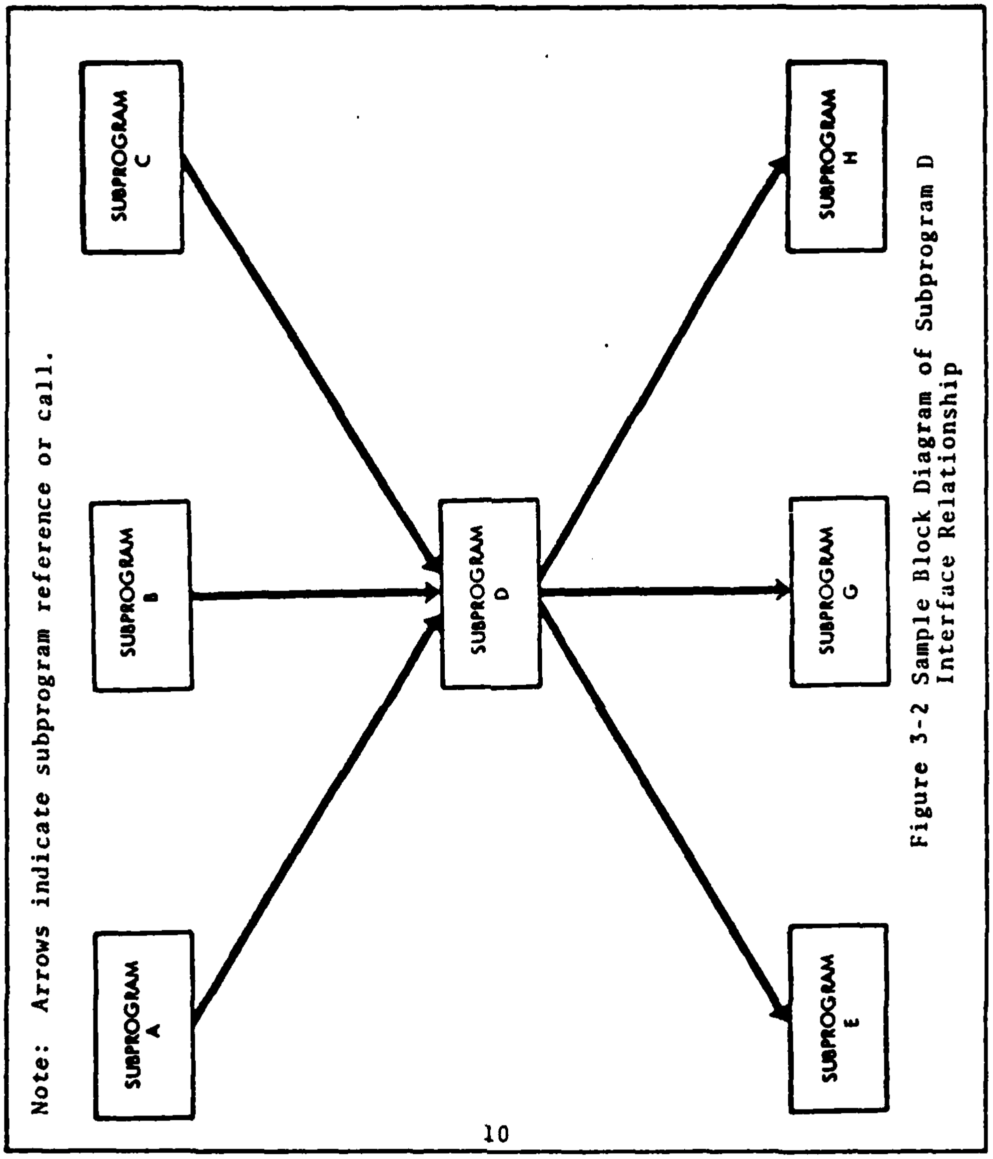
This Program Description document section shall contain supplementary information. The information shall include but is not limited to:

a. Information of particular importance to the procuring agency in using these documents.

b. Administrative and background information.

9

Figure 2-8. Program Description Document (Page 13 of 16)

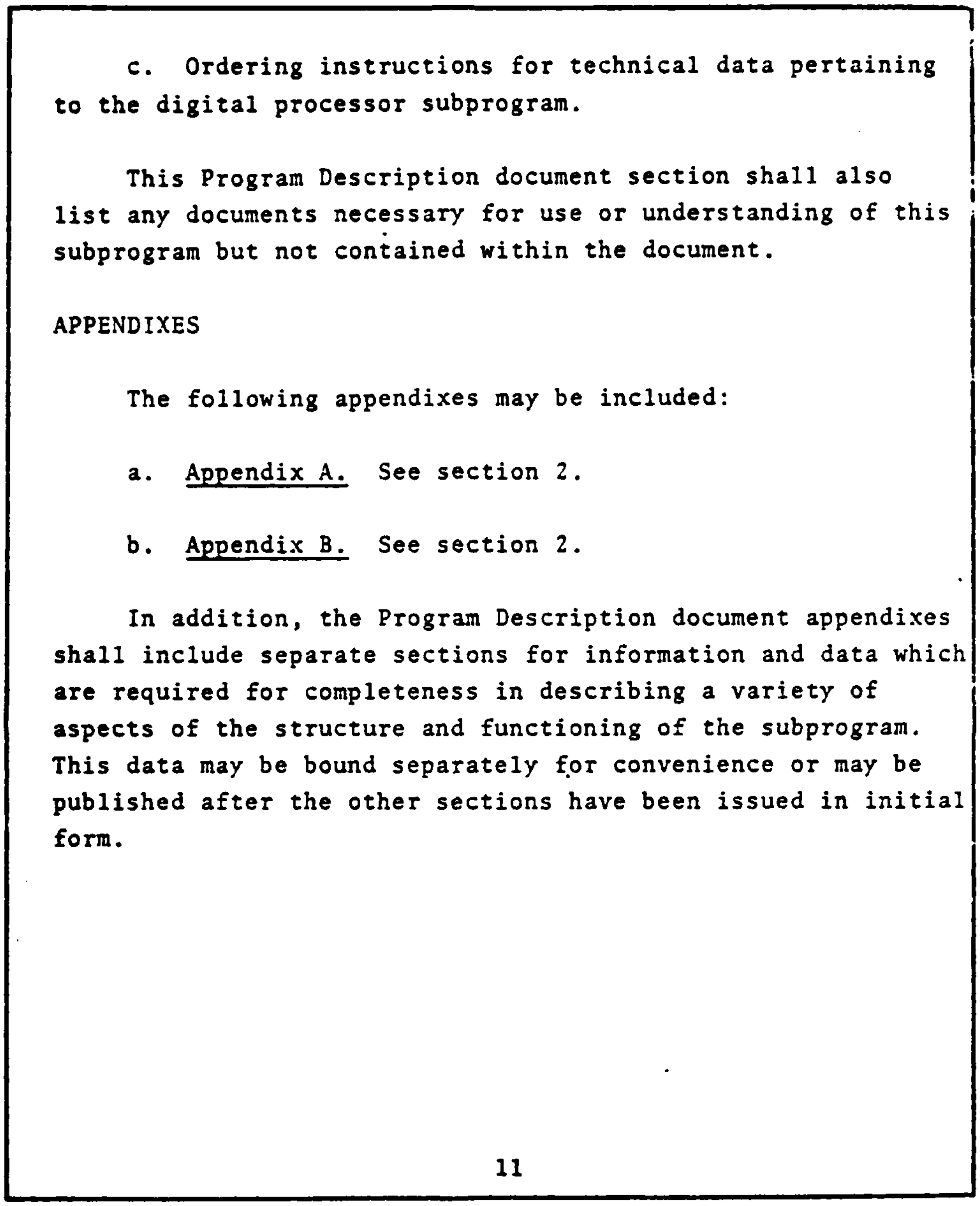


Note: Arrows indicate subprogram reference or call.

Figure 3-2 Sample Block Diagram of Subprogram D Interface Relationship

10

Figure 2-8. Program Description Document (Page 14 of 16)



c. Ordering instructions for technical data pertaining to the digital processor subprogram.

This Program Description document section shall also list any documents necessary for use or understanding of this subprogram but not contained within the document.

APPENDIXES

The following appendixes may be included:

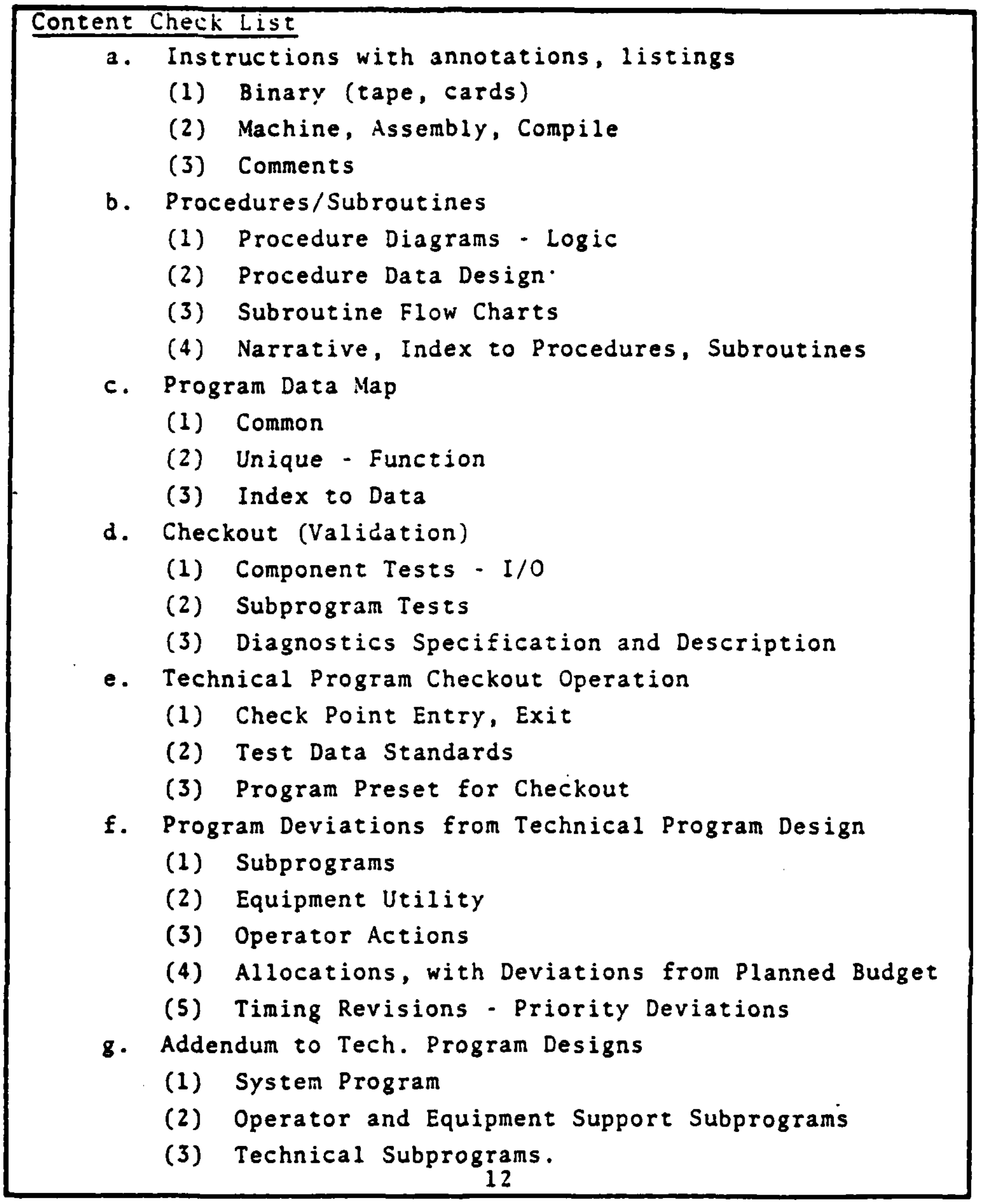
a. <u>Appendix A.</u> See section 2.

b. <u>Appendix B.</u> See section 2.

In addition, the Program Description document appendixes shall include separate sections for information and data which are required for completeness in describing a variety of aspects of the structure and functioning of the subprogram. This data may be bound separately for convenience or may be published after the other sections have been issued in initial form.

11

Figure 2-8. Program Description Document (Page 15 of 16)



Content Check List

a. Instructions with annotations, listings
   (1) Binary (tape, cards)
   (2) Machine, Assembly, Compile
   (3) Comments

b. Procedures/Subroutines
   (1) Procedure Diagrams - Logic
   (2) Procedure Data Design
   (3) Subroutine Flow Charts
   (4) Narrative, Index to Procedures, Subroutines

c. Program Data Map
   (1) Common
   (2) Unique - Function
   (3) Index to Data

d. Checkout (Validation)
   (1) Component Tests - I/O
   (2) Subprogram Tests
   (3) Diagnostics Specification and Description

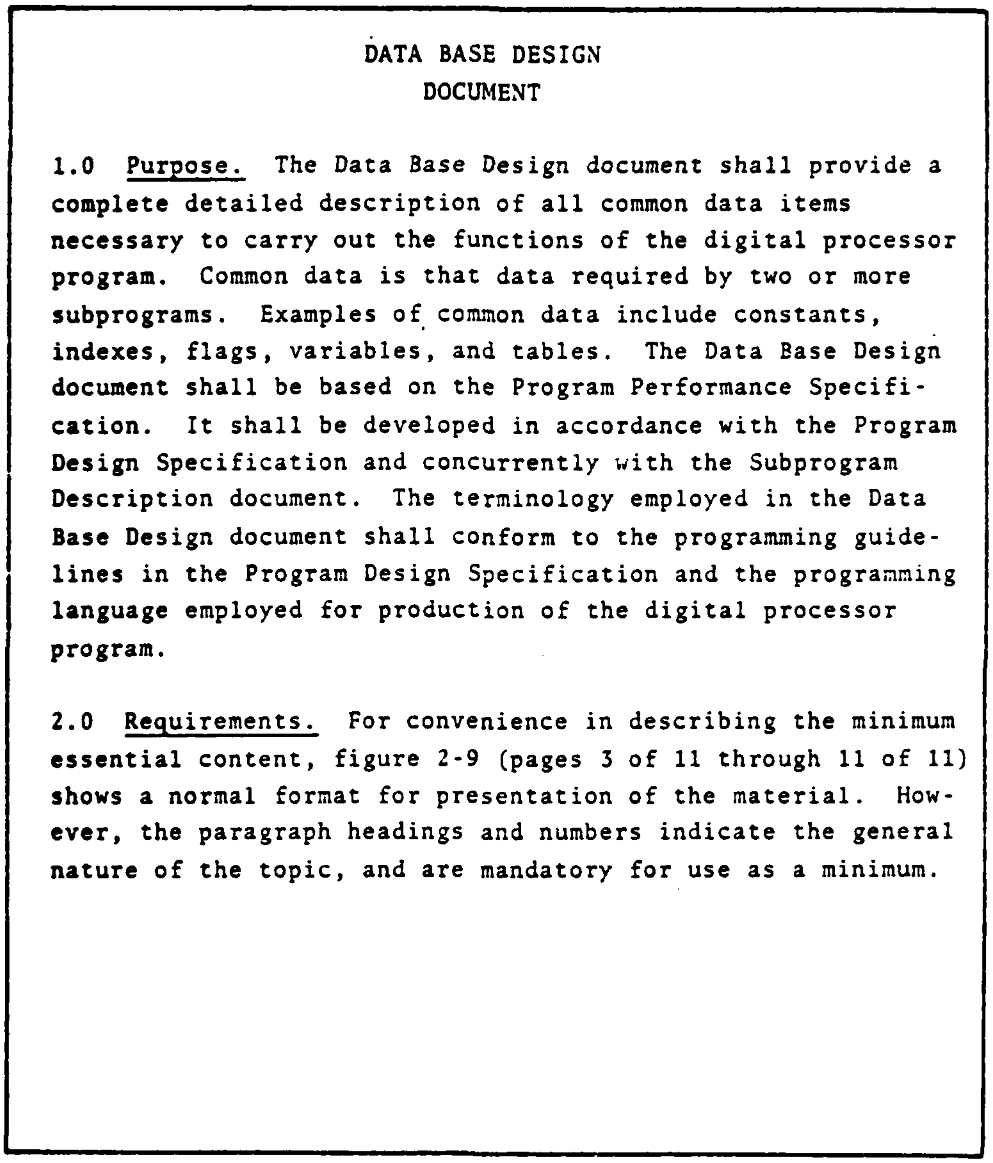
e. Technical Program Checkout Operation
   (1) Check Point Entry, Exit
   (2) Test Data Standards
   (3) Program Preset for Checkout

f. Program Deviations from Technical Program Design
   (1) Subprograms
   (2) Equipment Utility
   (3) Operator Actions
   (4) Allocations, with Deviations from Planned Budget
   (5) Timing Revisions - Priority Deviations

g. Addendum to Tech. Program Designs
   (1) System Program
   (2) Operator and Equipment Support Subprograms
   (3) Technical Subprograms.

12

Figure 2-8. Program Description Document (Page 16 of 16)



DATA BASE DESIGN
DOCUMENT

1.0 Purpose. The Data Base Design document shall provide a complete detailed description of all common data items necessary to carry out the functions of the digital processor program. Common data is that data required by two or more subprograms. Examples of common data include constants, indexes, flags, variables, and tables. The Data Base Design document shall be based on the Program Performance Specification. It shall be developed in accordance with the Program Design Specification and concurrently with the Subprogram Description document. The terminology employed in the Data Base Design document shall conform to the programming guidelines in the Program Design Specification and the programming language employed for production of the digital processor program.

2.0 Requirements. For convenience in describing the minimum essential content, figure 2-9 (pages 3 of 11 through 11 of 11) shows a normal format for presentation of the material. However, the paragraph headings and numbers indicate the general nature of the topic, and are mandatory for use as a minimum.

Figure 2-9. Data Base Design (Page 1 of 11)

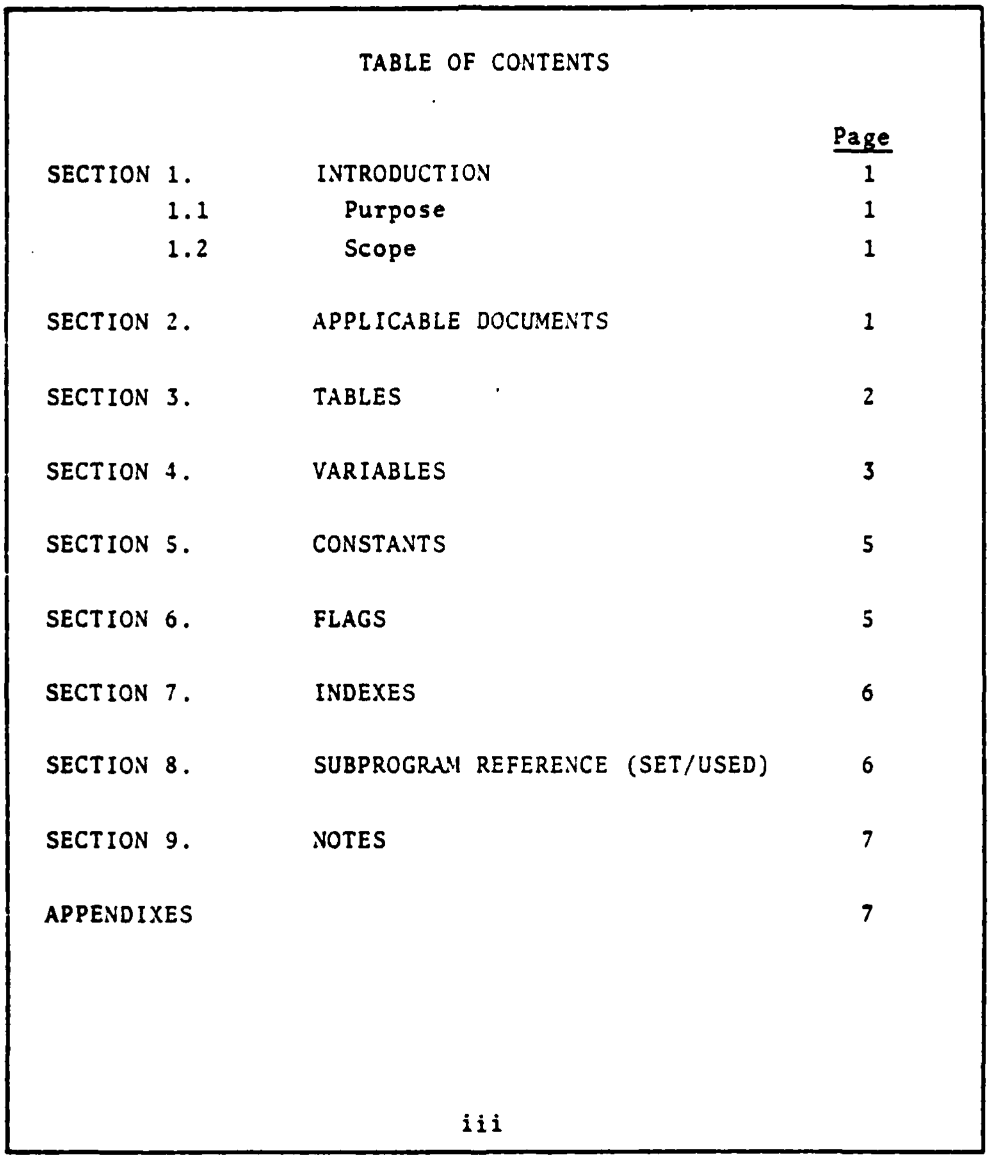
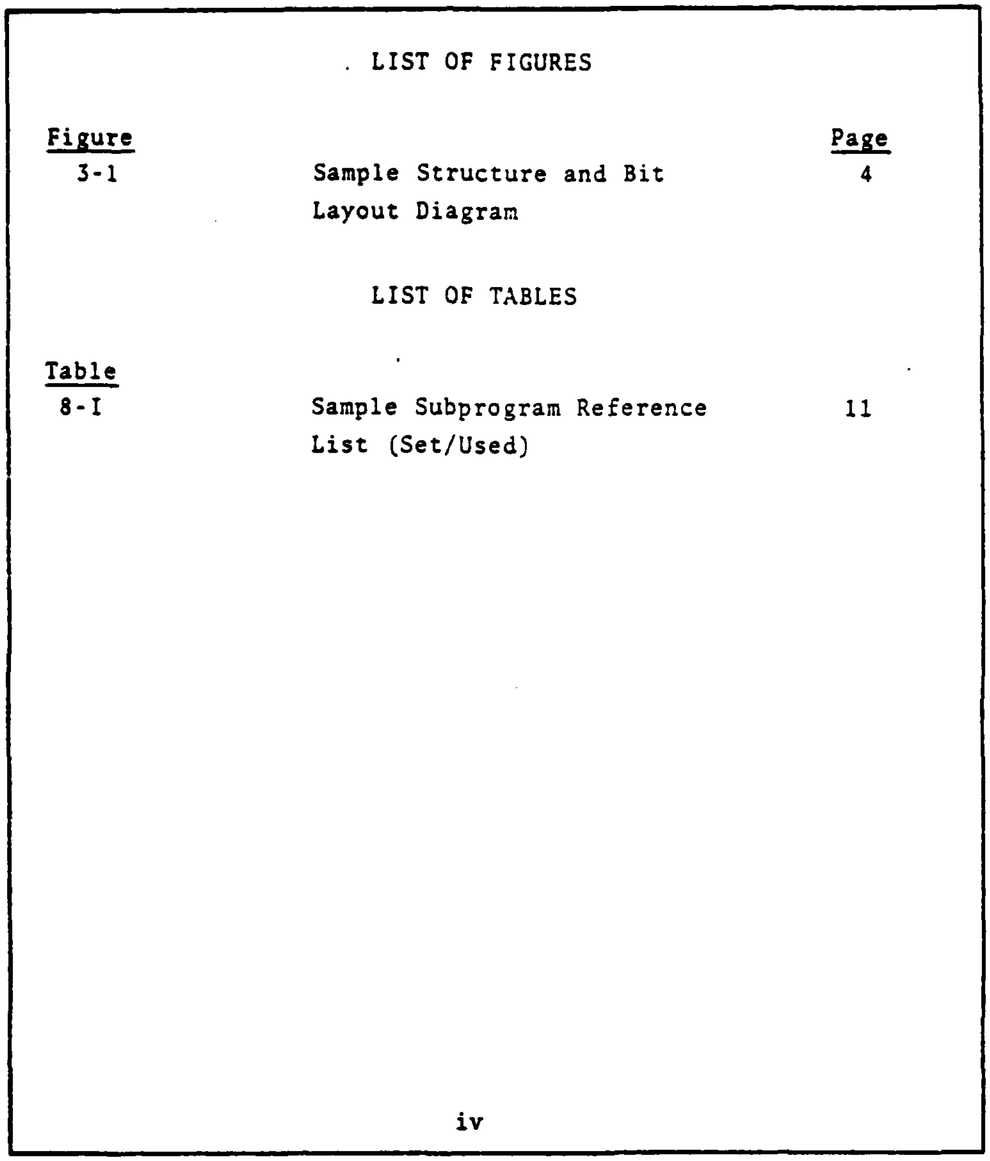


TABLE OF CONTENTS

iii

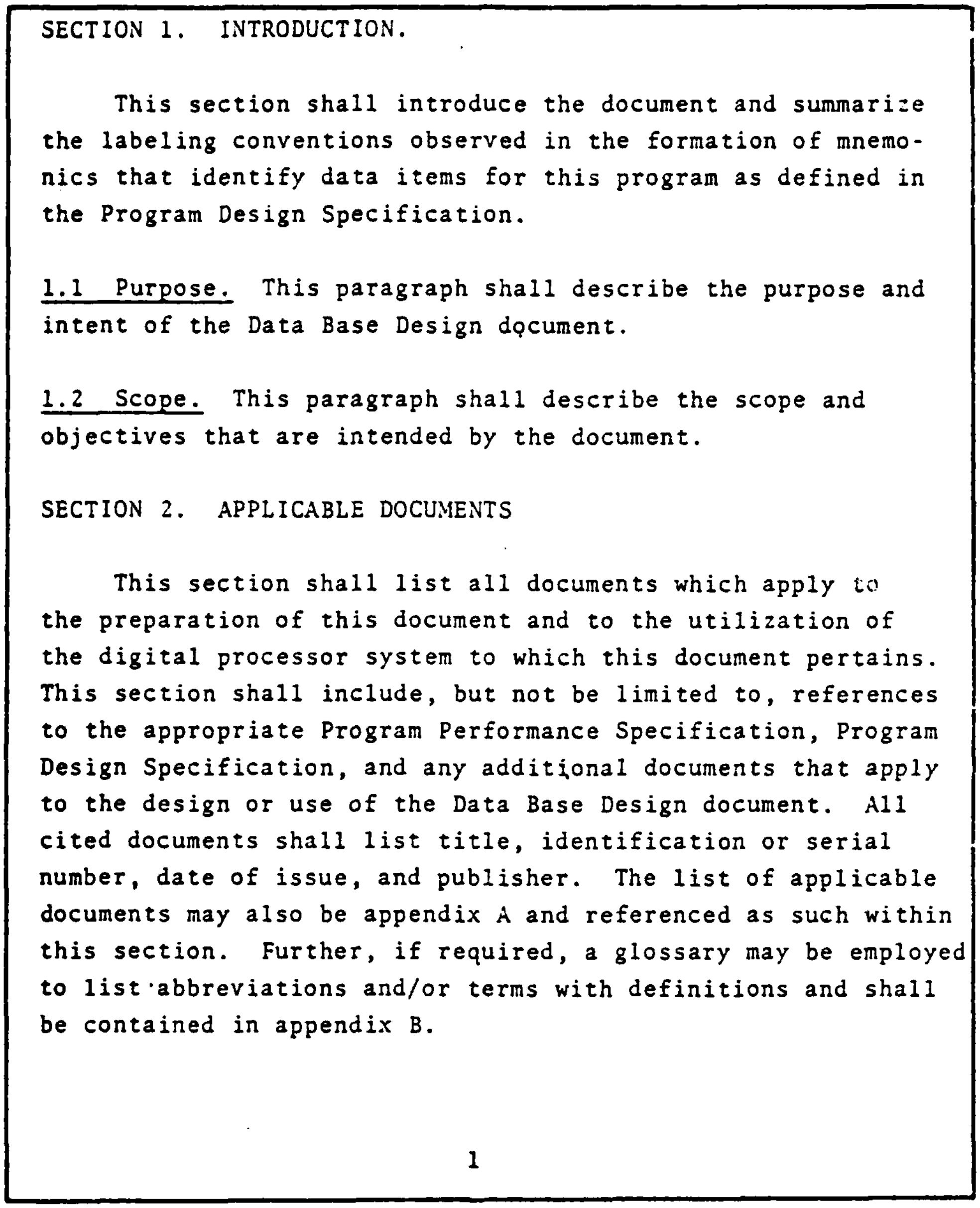
Figure 2-9. Data Base Design (Page 2 of 11)



## LIST OF FIGURES

## LIST OF TABLES

Figure 2-9. Data Base Design (Page 3 of 11)



## SECTION 1. INTRODUCTION.

This section shall introduce the document and summarize the labeling conventions observed in the formation of mnemonics that identify data items for this program as defined in the Program Design Specification.

1.1 Purpose. This paragraph shall describe the purpose and intent of the Data Base Design document.

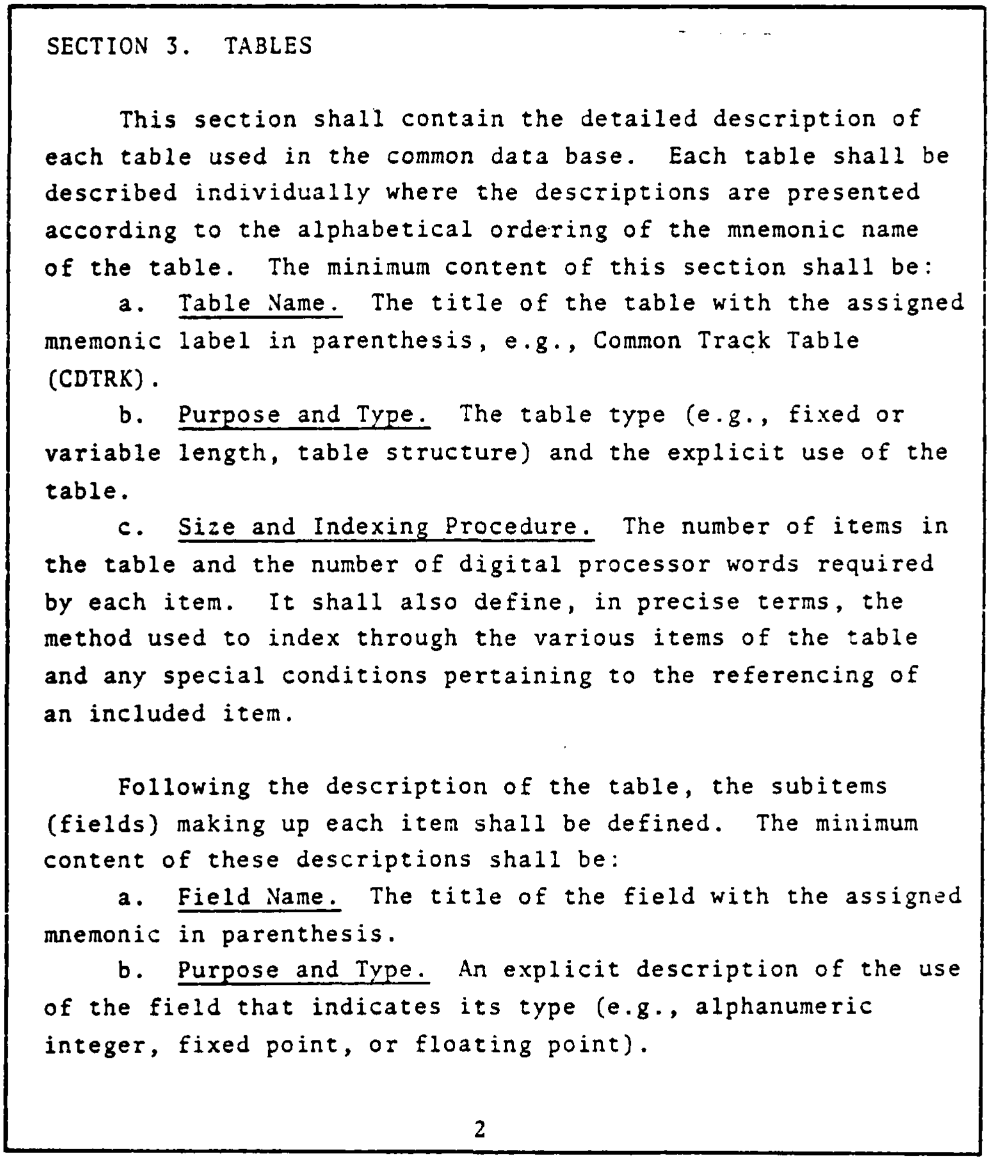
1.2 Scope. This paragraph shall describe the scope and objectives that are intended by the document.

## SECTION 2. APPLICABLE DOCUMENTS

This section shall list all documents which apply to the preparation of this document and to the utilization of the digital processor system to which this document pertains. This section shall include, but not be limited to, references to the appropriate Program Performance Specification, Program Design Specification, and any additional documents that apply to the design or use of the Data Base Design document. All cited documents shall list title, identification or serial number, date of issue, and publisher. The list of applicable documents may also be appendix A and referenced as such within this section. Further, if required, a glossary may be employed to list abbreviations and/or terms with definitions and shall be contained in appendix B.

1

Figure 2-9. Data Base Design (Page 4 of 11)



SECTION 3. TABLES

This section shall contain the detailed description of each table used in the common data base. Each table shall be described individually where the descriptions are presented according to the alphabetical ordering of the mnemonic name of the table. The minimum content of this section shall be:

a. Table Name. The title of the table with the assigned mnemonic label in parenthesis, e.g., Common Track Table (CDTRK).

b. Purpose and Type. The table type (e.g., fixed or variable length, table structure) and the explicit use of the table.

c. Size and Indexing Procedure. The number of items in the table and the number of digital processor words required by each item. It shall also define, in precise terms, the method used to index through the various items of the table and any special conditions pertaining to the referencing of an included item.

Following the description of the table, the subitems (fields) making up each item shall be defined. The minimum content of these descriptions shall be:

a. Field Name. The title of the field with the assigned mnemonic in parenthesis.

b. Purpose and Type. An explicit description of the use of the field that indicates its type (e.g., alphanumeric integer, fixed point, or floating point).

Figure 2-9. Data Base Design (Page 5 of 11)

c. Size. The size of the field in words or bits (if numeric) or number of characters (if alphabetic).

d. Binary Point. This information shall be included for all numeric type fields except floating point, and shall indicate the bit position of the binary point (scaling) of the variable.

e. Range of Values and Initial Condition. The minimum and maximum values that are valid for the field, and the initial condition of the field if it is preset. For alphanumeric types the data code (e.g., ASCII, BCD) shall also be given.

f. Static/Dynamic. The changeability nature of the field (e.g., unchanging value is static, changing field values are dynamic).

g. Structure and Bit Layout. A diagram for each digital processor word required by the field, as shown in figure 3-1.

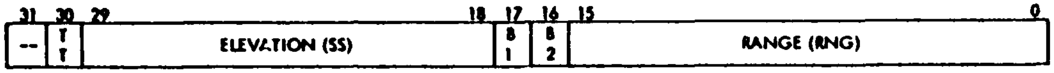
SECTION 4. VARIABLES

This section shall contain the detailed description of each variable included in the common data base. Each variable shall be described individually where the descriptions are presented according to the alphabetical ordering of the mnemonic names of the variables. The minimum content of this paragraph shall be the following information and shall be in accordance with the requirements defined in section 3 of this document:

a. Variable Name

b. Purpose and Type

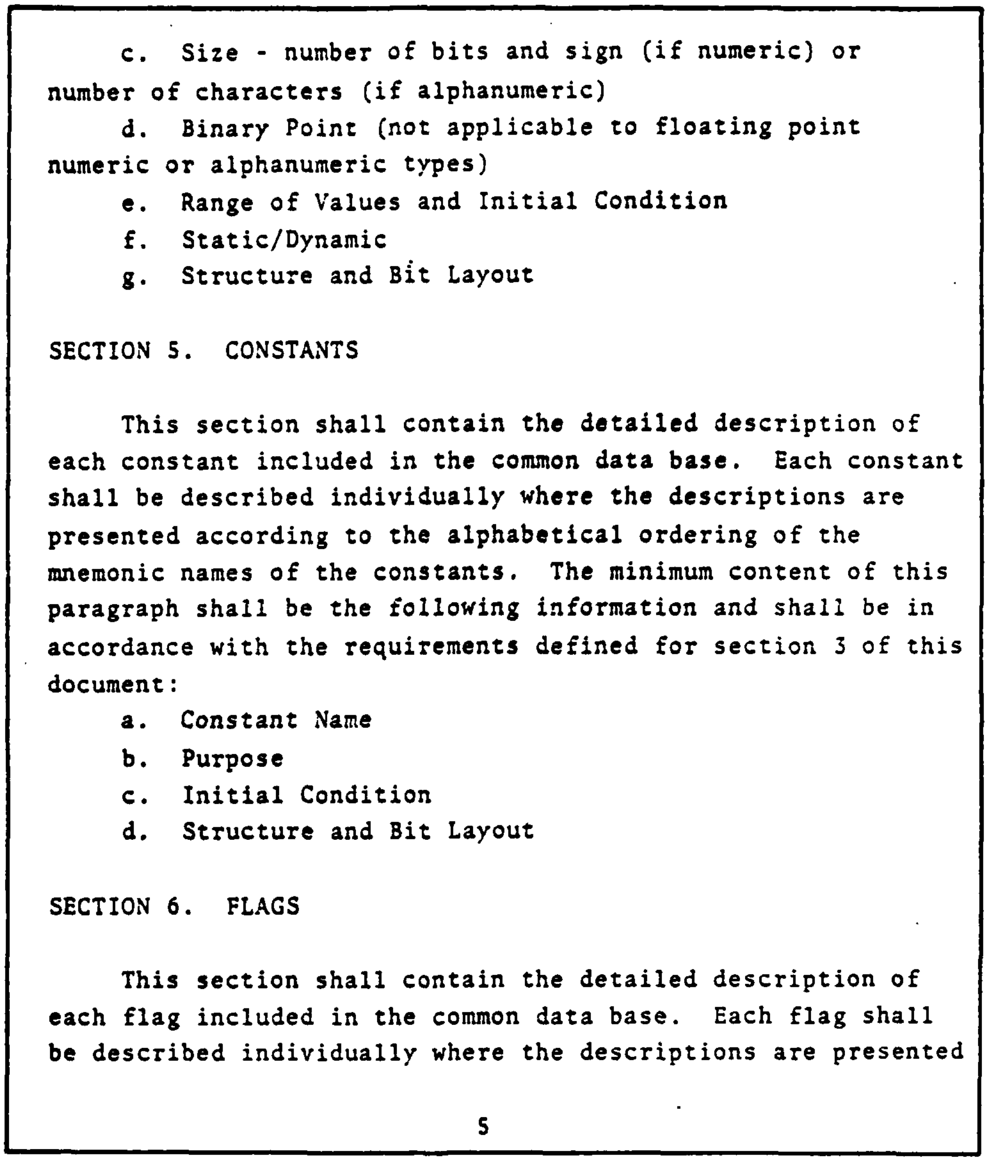
Figure 2-9. Data Base Design (Page 6 of 11)



| FIELD | | UNITS | SCALING |
|---|---|---|---|
| TT | Test Target - (BiT $2^{30}$ = 1) Interpret as a non-tactical track | | |
| ELEVATION (SS) | A value expressing the elevation angle at whicn the radar is to conduct its Sector Search. Minimum value is 1 degree. Maximum value is 85 degrees. MSB = X, LSB = Y. | BAMS | 12 |
| B1 | Sector 1 Blanking - (Bit $2^{17}$ = 1) Interpret as first sector in which the radar is blanked during Horizon Search Mode. | | |
| B2 | Sector 2 Blanking - (Bit $2^{16}$ = 1) Interpret as second sector in which the radar is blanked during Horizon Search Mode. | | |
| RANGE (RNG) | A value expressing the distance to the target nautical miles. Minimum value .25 nautical miles. Maximum value is 256.00 nautical miles. | n. miles | 6 |

Figure 3-1 Sample Structure and Bit Layout Diagram

Figure 2-9. Data Base Design (Page 7 of 11)



c. Size - number of bits and sign (if numeric) or number of characters (if alphanumeric)

d. Binary Point (not applicable to floating point numeric or alphanumeric types)

e. Range of Values and Initial Condition

f. Static/Dynamic

g. Structure and Bit Layout

SECTION 5. CONSTANTS

This section shall contain the detailed description of each constant included in the common data base. Each constant shall be described individually where the descriptions are presented according to the alphabetical ordering of the mnemonic names of the constants. The minimum content of this paragraph shall be the following information and shall be in accordance with the requirements defined for section 3 of this document:

a. Constant Name

b. Purpose

c. Initial Condition

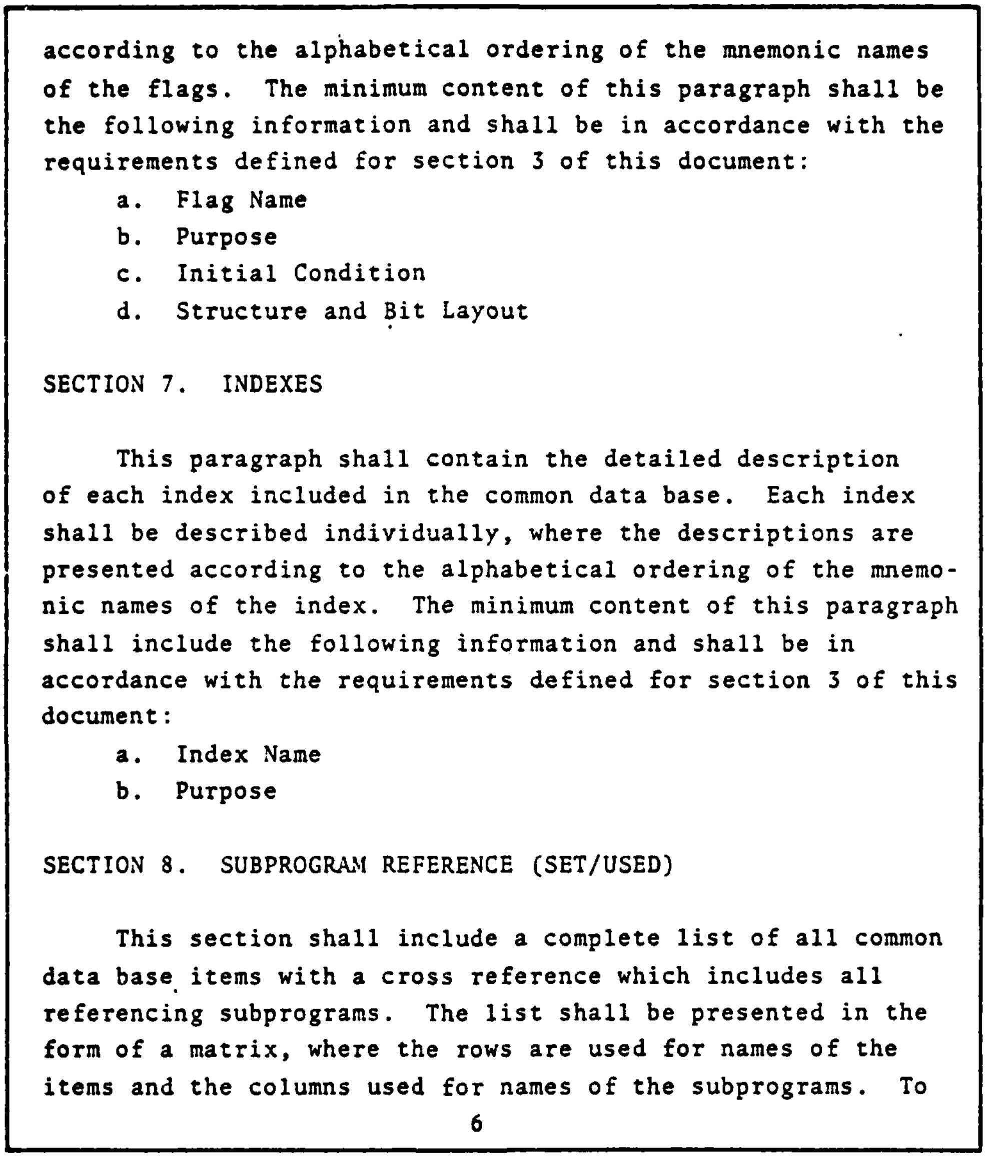
d. Structure and Bit Layout

SECTION 6. FLAGS

This section shall contain the detailed description of each flag included in the common data base. Each flag shall be described individually where the descriptions are presented

5

Figure 2-9. Data Base Design (Page 8 of 11)



according to the alphabetical ordering of the mnemonic names of the flags. The minimum content of this paragraph shall be the following information and shall be in accordance with the requirements defined for section 3 of this document:

a. Flag Name
b. Purpose
c. Initial Condition
d. Structure and Bit Layout

SECTION 7. INDEXES

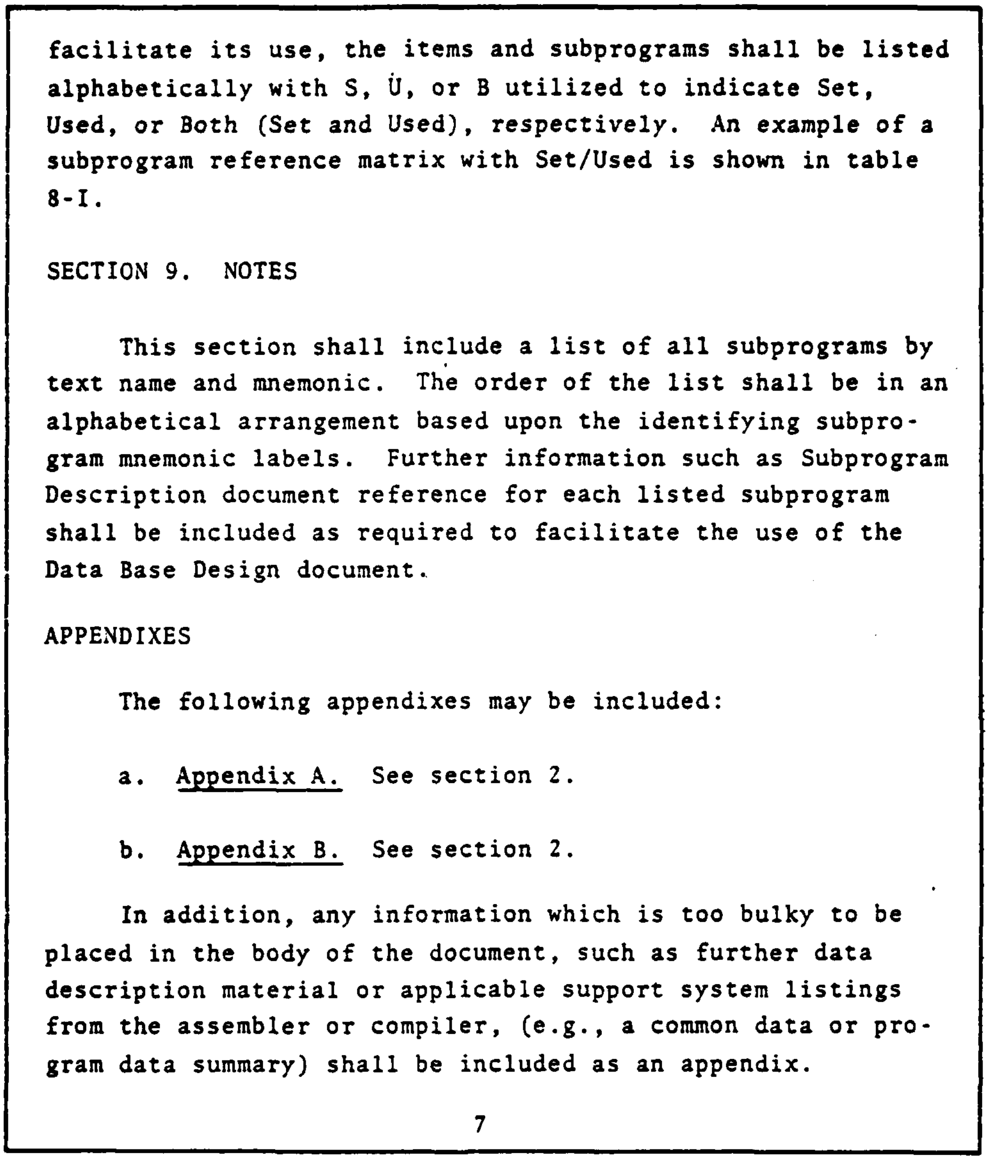
This paragraph shall contain the detailed description of each index included in the common data base. Each index shall be described individually, where the descriptions are presented according to the alphabetical ordering of the mnemonic names of the index. The minimum content of this paragraph shall include the following information and shall be in accordance with the requirements defined for section 3 of this document:

a. Index Name
b. Purpose

SECTION 8. SUBPROGRAM REFERENCE (SET/USED)

This section shall include a complete list of all common data base items with a cross reference which includes all referencing subprograms. The list shall be presented in the form of a matrix, where the rows are used for names of the items and the columns used for names of the subprograms. To

Figure 2-9. Data Base Design (Page 9 of 11)



facilitate its use, the items and subprograms shall be listed alphabetically with S, U, or B utilized to indicate Set, Used, or Both (Set and Used), respectively. An example of a subprogram reference matrix with Set/Used is shown in table 8-I.

## SECTION 9. NOTES

This section shall include a list of all subprograms by text name and mnemonic. The order of the list shall be in an alphabetical arrangement based upon the identifying subprogram mnemonic labels. Further information such as Subprogram Description document reference for each listed subprogram shall be included as required to facilitate the use of the Data Base Design document.

## APPENDIXES

The following appendixes may be included:

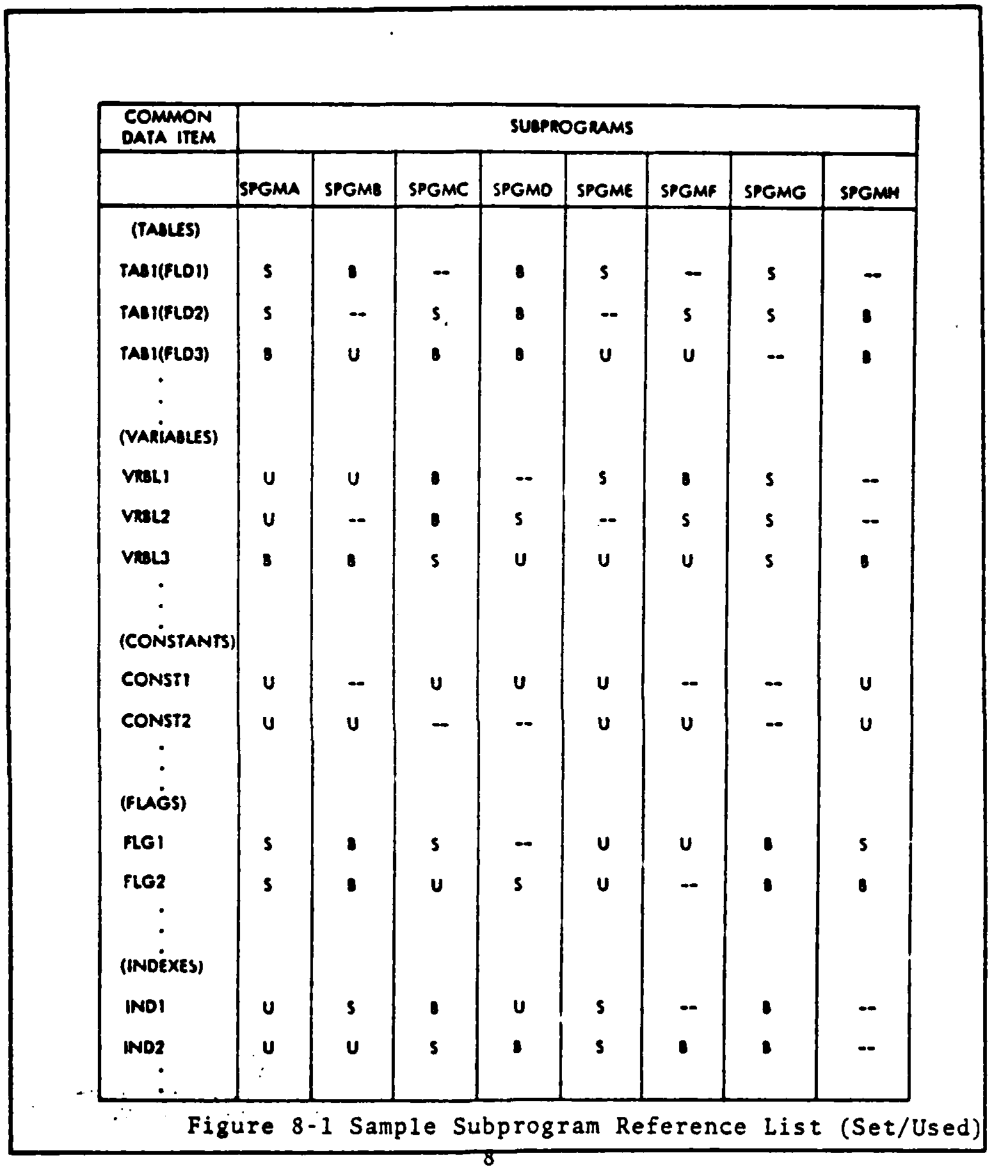
a. <u>Appendix A.</u> See section 2.

b. <u>Appendix B.</u> See section 2.

In addition, any information which is too bulky to be placed in the body of the document, such as further data description material or applicable support system listings from the assembler or compiler, (e.g., a common data or program data summary) shall be included as an appendix.

7

Figure 2-9. Data Base Design (Page 10 of 11)

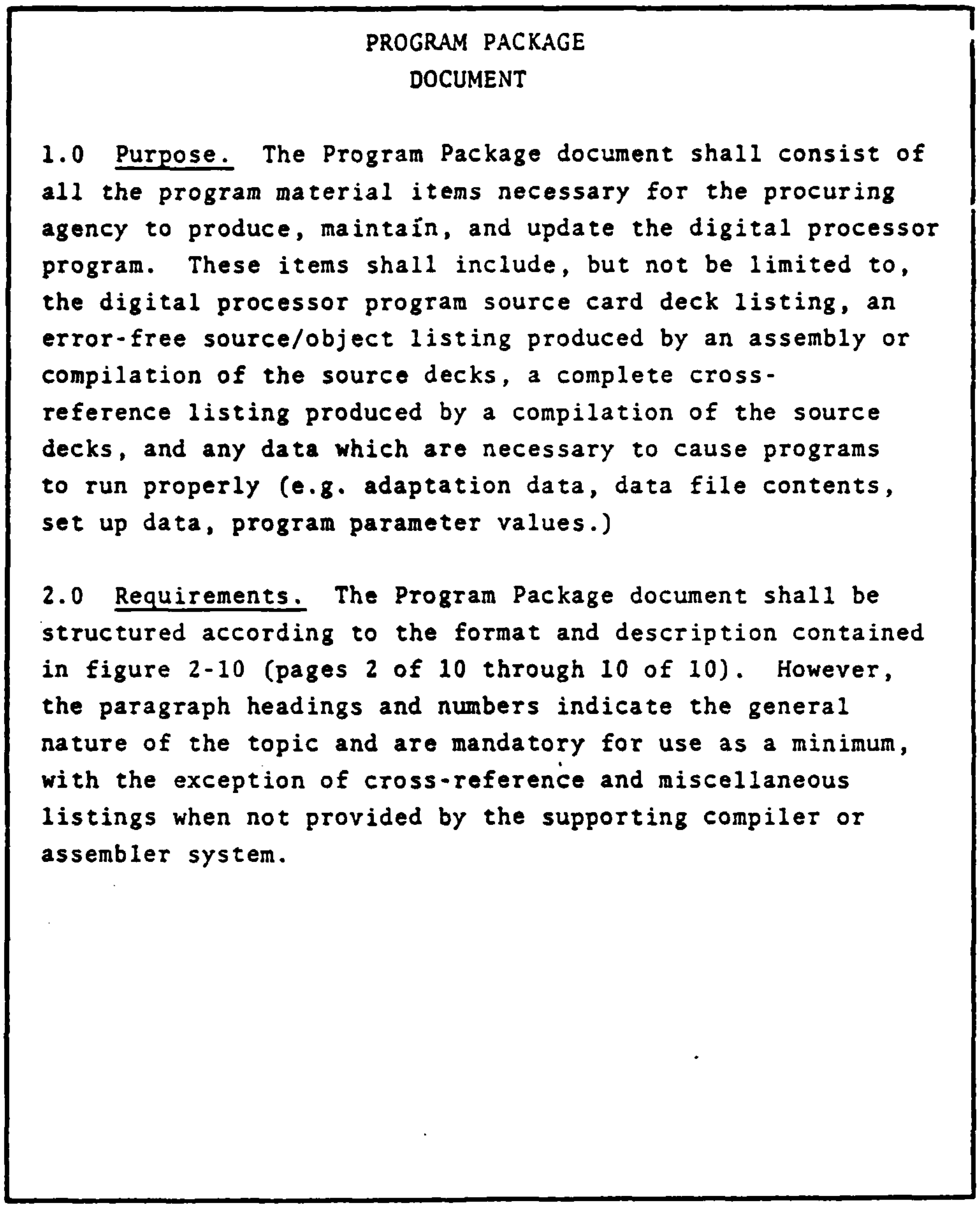


| COMMON DATA ITEM | SUBPROGRAMS | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | SPGMA | SPGMB | SPGMC | SPGMD | SPGME | SPGMF | SPGMG | SPGMH |
| (TABLES) | | | | | | | | |
| TAB1(FLD1) | S | B | -- | B | S | -- | S | -- |
| TAB1(FLD2) | S | -- | S | B | -- | S | S | B |
| TAB1(FLD3) | B | U | B | B | U | U | -- | B |
| ⋮ | | | | | | | | |
| (VARIABLES) | | | | | | | | |
| VRBL1 | U | U | B | -- | S | B | S | -- |
| VRBL2 | U | -- | B | S | -- | S | S | -- |
| VRBL3 | B | B | S | U | U | U | S | B |
| ⋮ | | | | | | | | |
| (CONSTANTS) | | | | | | | | |
| CONST1 | U | -- | U | U | U | -- | -- | U |
| CONST2 | U | U | -- | -- | U | U | -- | U |
| ⋮ | | | | | | | | |
| (FLAGS) | | | | | | | | |
| FLG1 | S | B | S | -- | U | U | B | S |
| FLG2 | S | B | U | S | U | -- | B | B |
| ⋮ | | | | | | | | |
| (INDEXES) | | | | | | | | |
| IND1 | U | S | B | U | S | -- | B | -- |
| IND2 | U | U | S | B | S | B | B | -- |
| ⋮ | | | | | | | | |

Figure 8-1 Sample Subprogram Reference List (Set/Used)

8

Figure 2-9. Data Base Design (Page 11 of 11)



PROGRAM PACKAGE
DOCUMENT

1.0 Purpose. The Program Package document shall consist of all the program material items necessary for the procuring agency to produce, maintain, and update the digital processor program. These items shall include, but not be limited to, the digital processor program source card deck listing, an error-free source/object listing produced by an assembly or compilation of the source decks, a complete cross-reference listing produced by a compilation of the source decks, and any data which are necessary to cause programs to run properly (e.g. adaptation data, data file contents, set up data, program parameter values.)

2.0 Requirements. The Program Package document shall be structured according to the format and description contained in figure 2-10 (pages 2 of 10 through 10 of 10). However, the paragraph headings and numbers indicate the general nature of the topic and are mandatory for use as a minimum, with the exception of cross-reference and miscellaneous listings when not provided by the supporting compiler or assembler system.

Figure 2-10. Program Package (Page 1 of 10)

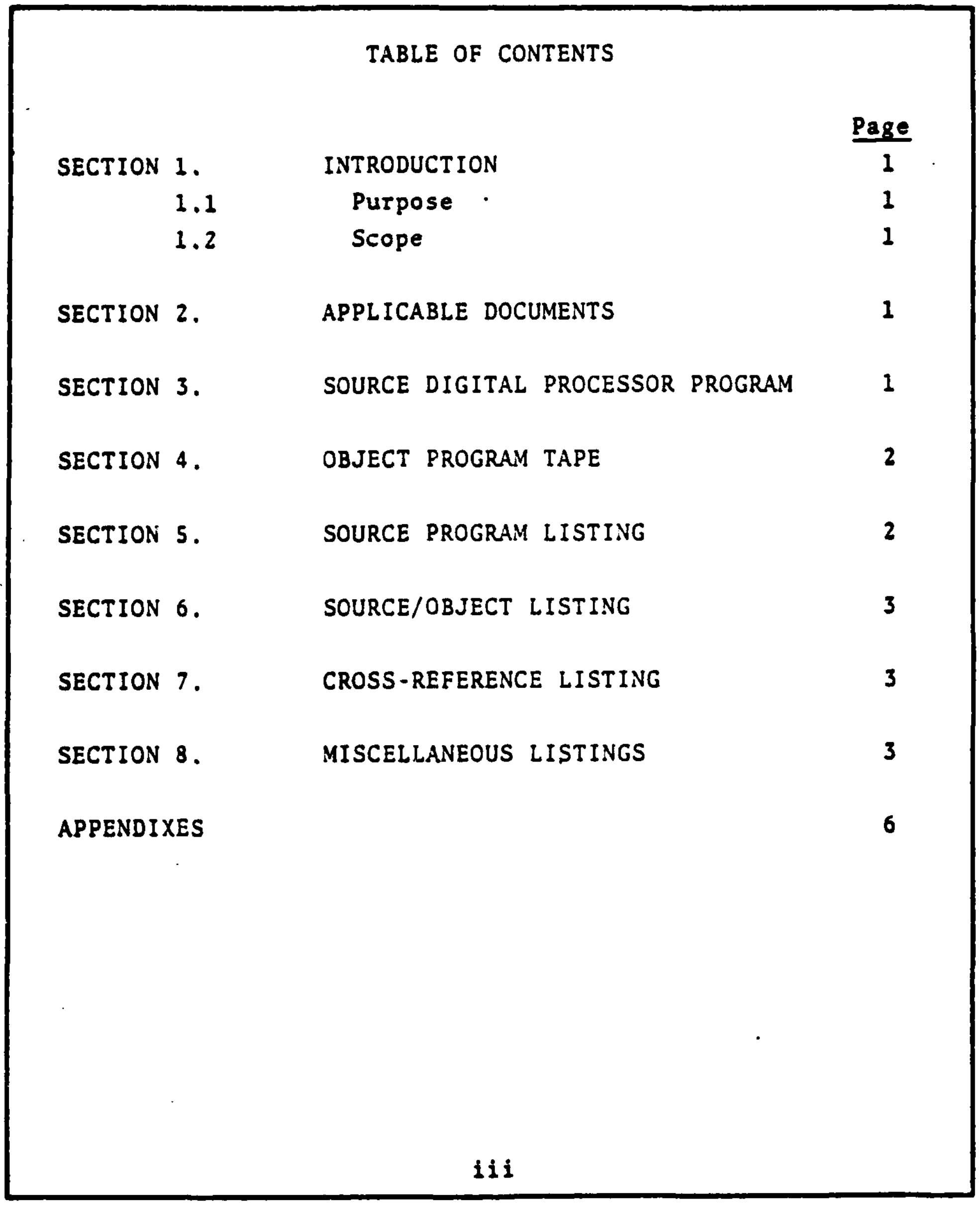
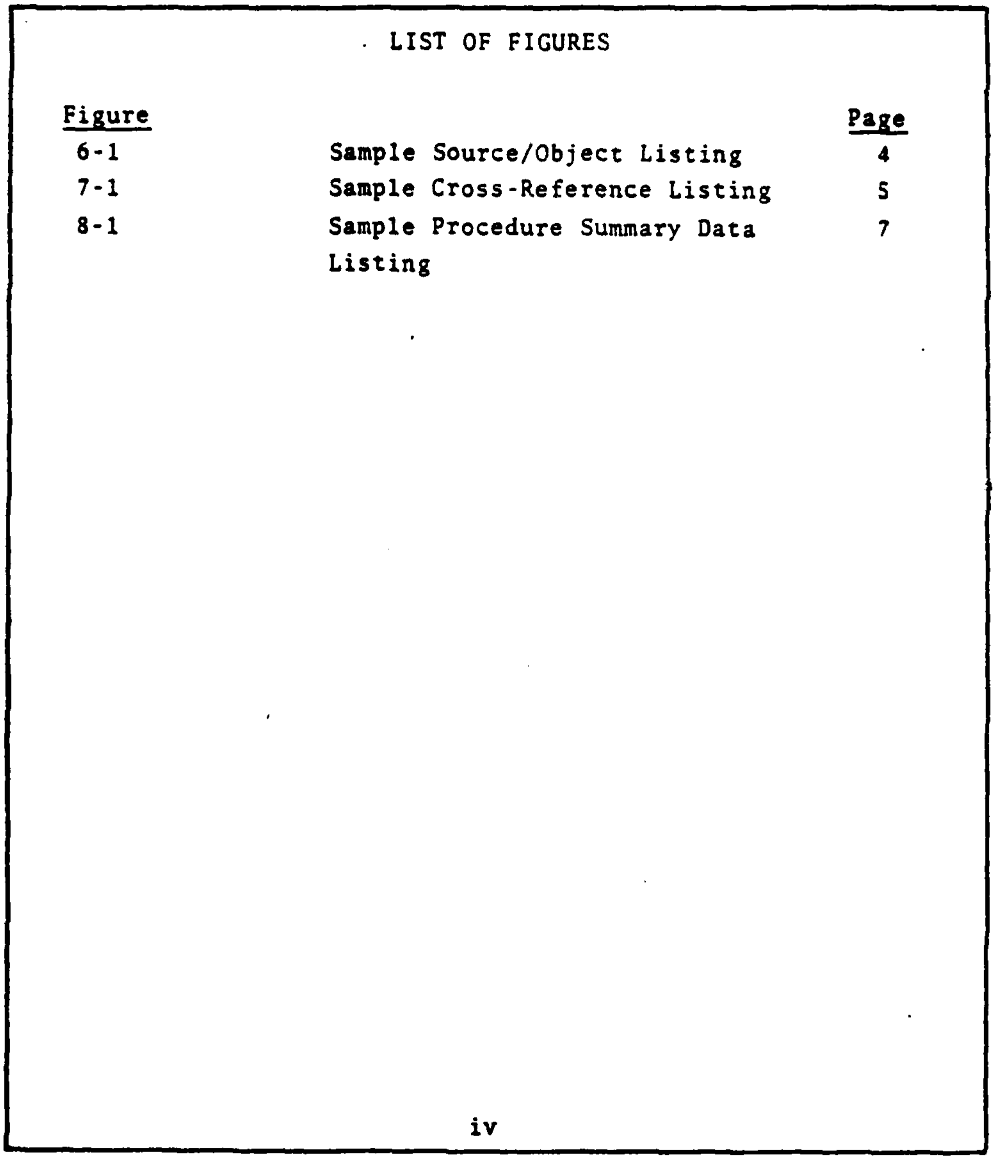


TABLE OF CONTENTS

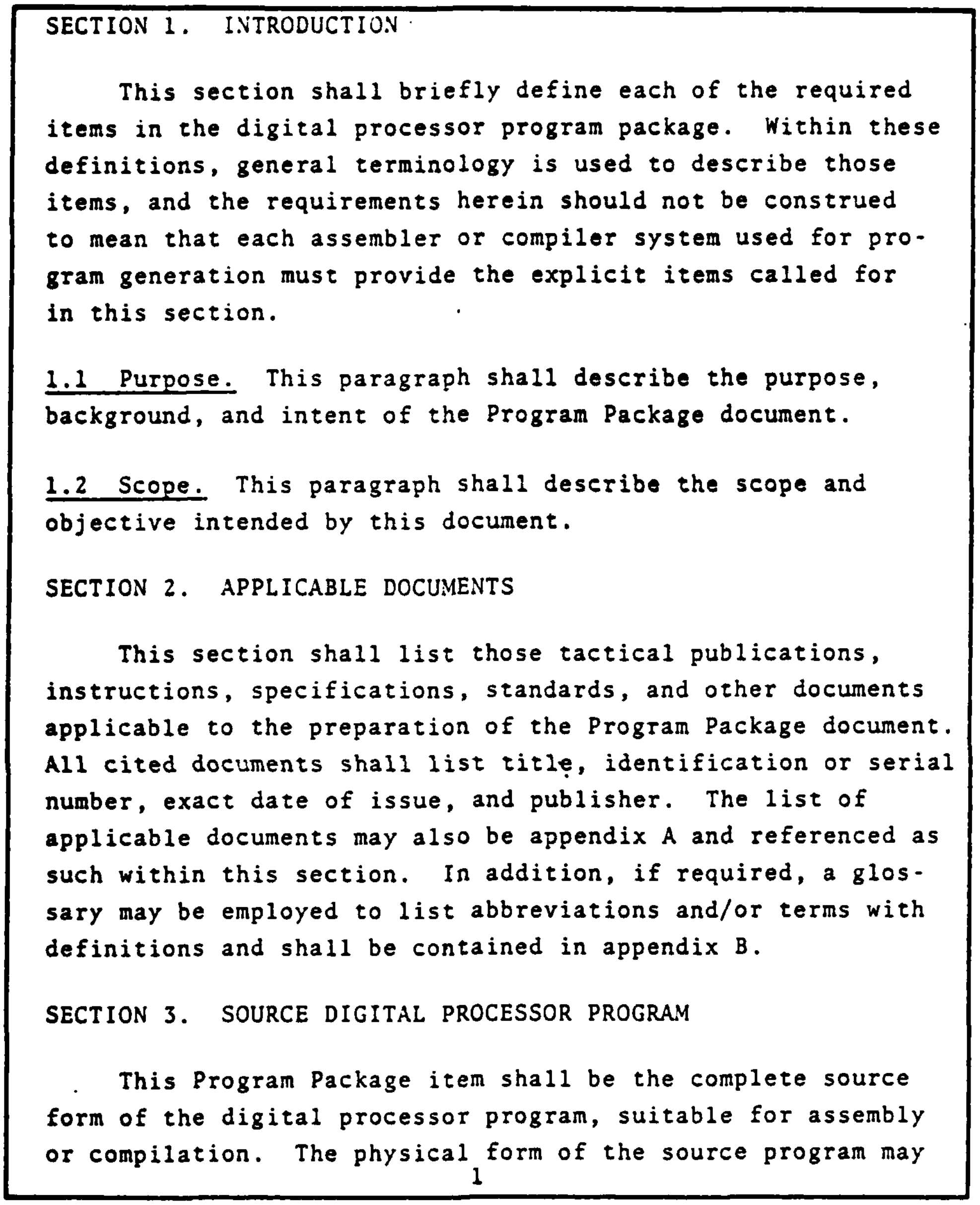
iii

Figure 2-20. Program Package (Page 2 of 10)



## LIST OF FIGURES

Figure 2-20. Program Package (Page 3 of 10)



SECTION 1. INTRODUCTION

This section shall briefly define each of the required items in the digital processor program package. Within these definitions, general terminology is used to describe those items, and the requirements herein should not be construed to mean that each assembler or compiler system used for program generation must provide the explicit items called for in this section.

1.1 Purpose. This paragraph shall describe the purpose, background, and intent of the Program Package document.

1.2 Scope. This paragraph shall describe the scope and objective intended by this document.

SECTION 2. APPLICABLE DOCUMENTS

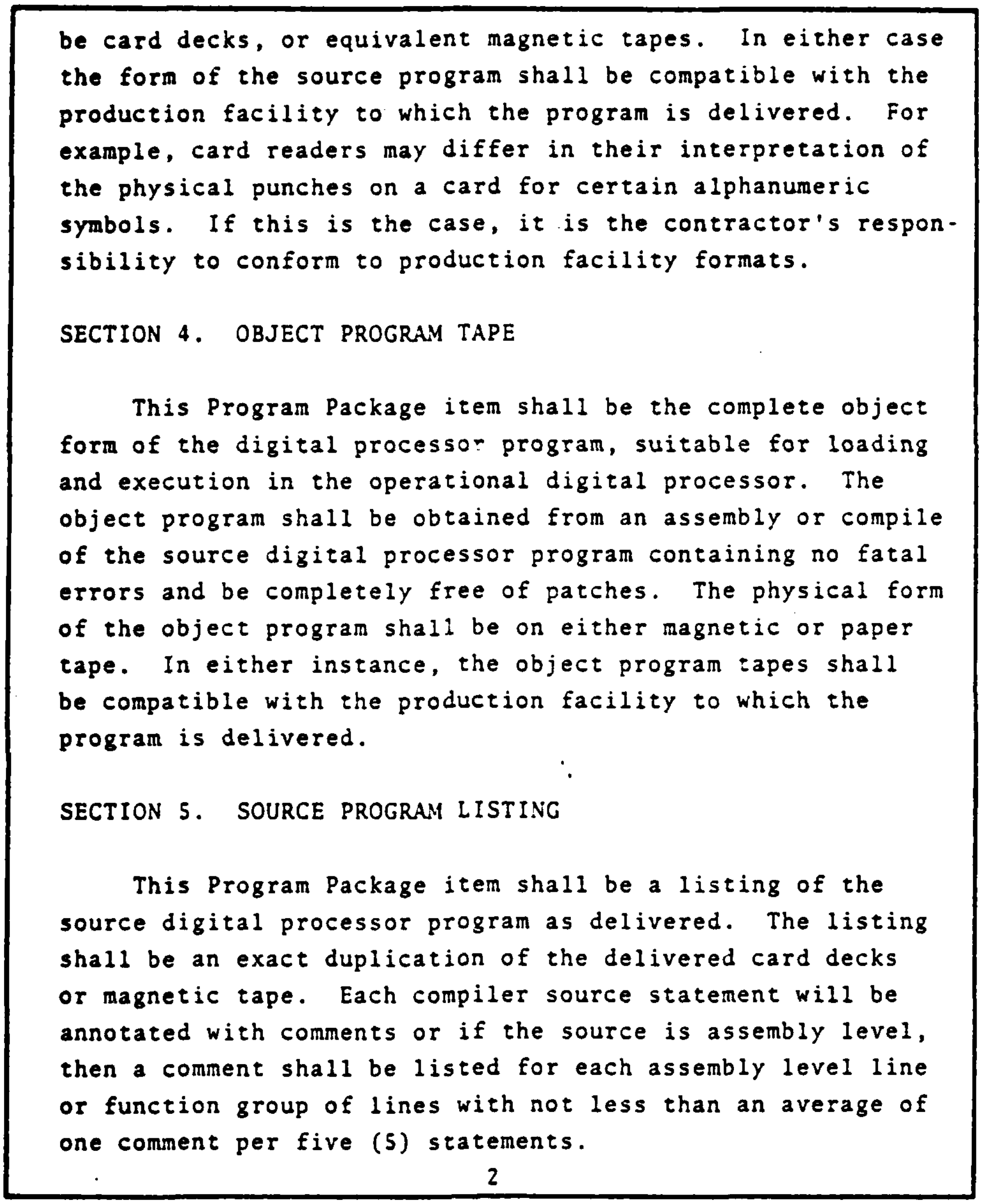
This section shall list those tactical publications, instructions, specifications, standards, and other documents applicable to the preparation of the Program Package document. All cited documents shall list title, identification or serial number, exact date of issue, and publisher. The list of applicable documents may also be appendix A and referenced as such within this section. In addition, if required, a glossary may be employed to list abbreviations and/or terms with definitions and shall be contained in appendix B.

SECTION 3. SOURCE DIGITAL PROCESSOR PROGRAM

This Program Package item shall be the complete source form of the digital processor program, suitable for assembly or compilation. The physical form of the source program may

1

Figure 2-20. Program Package (Page 4 of 10)



be card decks, or equivalent magnetic tapes. In either case the form of the source program shall be compatible with the production facility to which the program is delivered. For example, card readers may differ in their interpretation of the physical punches on a card for certain alphanumeric symbols. If this is the case, it is the contractor's responsibility to conform to production facility formats.

SECTION 4. OBJECT PROGRAM TAPE

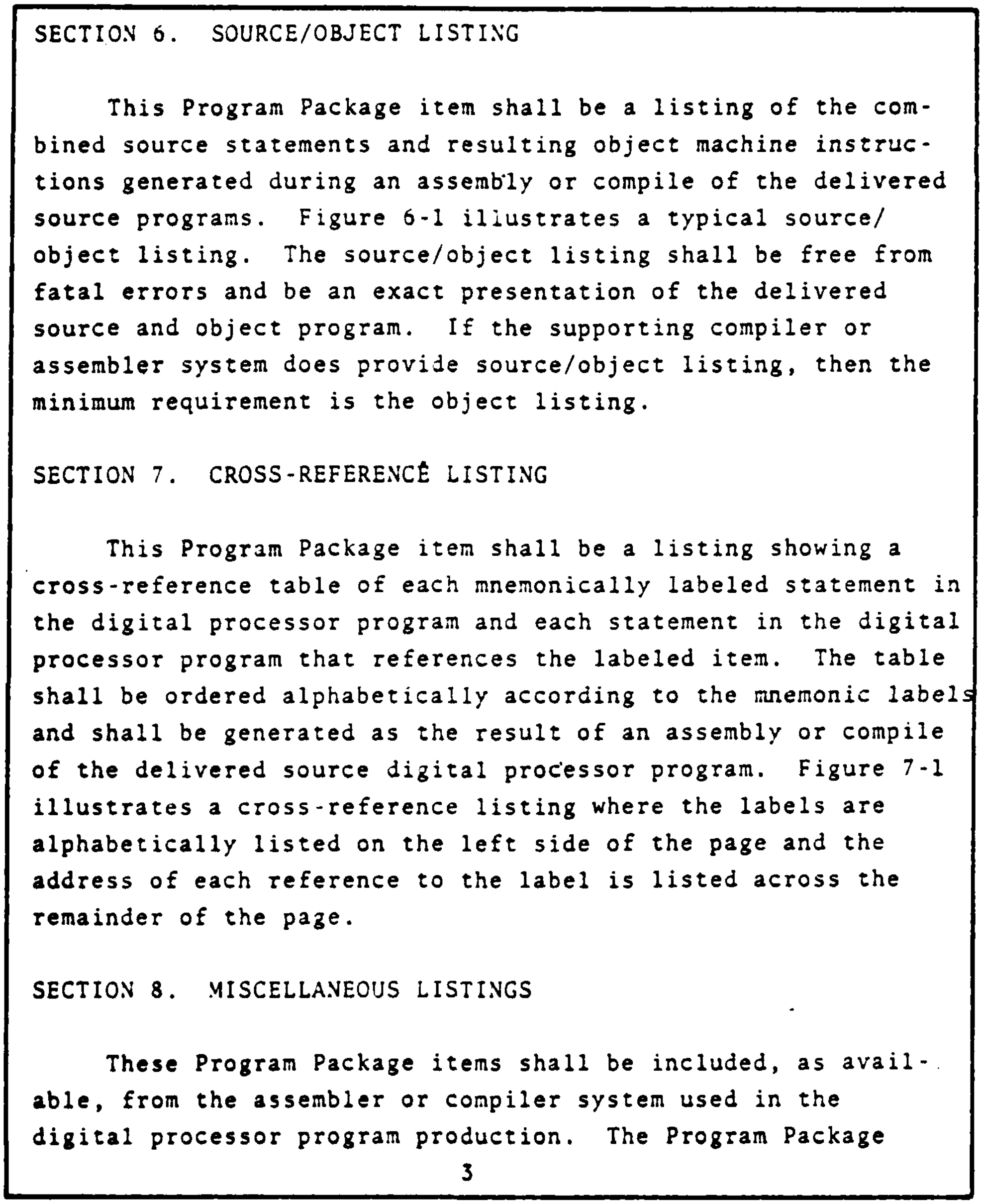
This Program Package item shall be the complete object form of the digital processor program, suitable for loading and execution in the operational digital processor. The object program shall be obtained from an assembly or compile of the source digital processor program containing no fatal errors and be completely free of patches. The physical form of the object program shall be on either magnetic or paper tape. In either instance, the object program tapes shall be compatible with the production facility to which the program is delivered.

SECTION 5. SOURCE PROGRAM LISTING

This Program Package item shall be a listing of the source digital processor program as delivered. The listing shall be an exact duplication of the delivered card decks or magnetic tape. Each compiler source statement will be annotated with comments or if the source is assembly level, then a comment shall be listed for each assembly level line or function group of lines with not less than an average of one comment per five (5) statements.

2

Figure 2-20. Program Package (Page 5 of 10)



SECTION 6. SOURCE/OBJECT LISTING

This Program Package item shall be a listing of the combined source statements and resulting object machine instructions generated during an assembly or compile of the delivered source programs. Figure 6-1 illustrates a typical source/object listing. The source/object listing shall be free from fatal errors and be an exact presentation of the delivered source and object program. If the supporting compiler or assembler system does provide source/object listing, then the minimum requirement is the object listing.

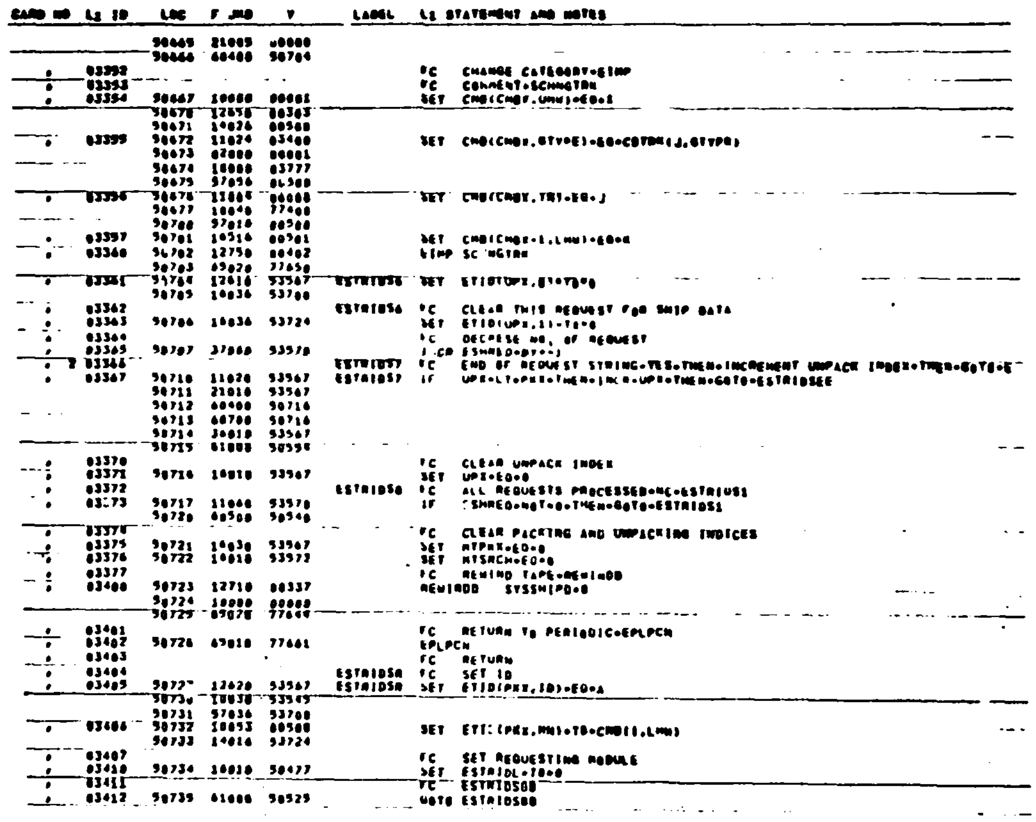
SECTION 7. CROSS-REFERENCE LISTING

This Program Package item shall be a listing showing a cross-reference table of each mnemonically labeled statement in the digital processor program and each statement in the digital processor program that references the labeled item. The table shall be ordered alphabetically according to the mnemonic labels and shall be generated as the result of an assembly or compile of the delivered source digital processor program. Figure 7-1 illustrates a cross-reference listing where the labels are alphabetically listed on the left side of the page and the address of each reference to the label is listed across the remainder of the page.

SECTION 8. MISCELLANEOUS LISTINGS

These Program Package items shall be included, as available, from the assembler or compiler system used in the digital processor program production. The Program Package

Figure 2-20. Program Package (Page 6 of 10)



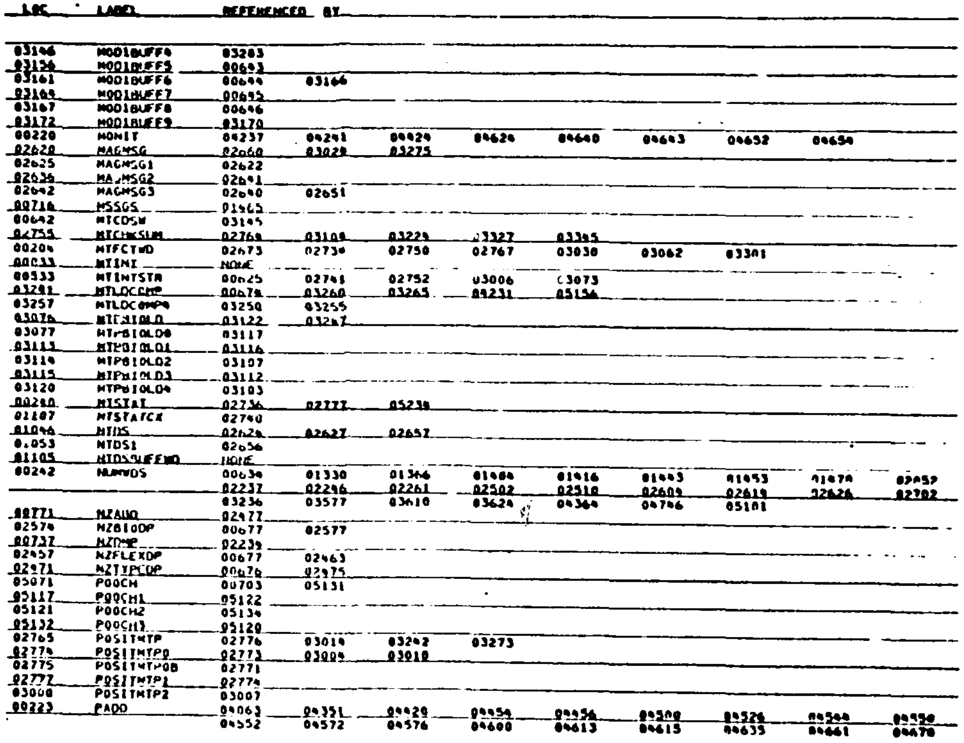
```
CARD NO  L1 ID   LOC    F JMD   Y       LABEL     L1 STATEMENT AND NOTES

                 90665  21005  00000
                 90666  60400  90704
   •     03352                                    FC   CHANGE CATEGORY.EIMP
   •     03353                                    FC   COMMENT.SCHNGTRN
   •     03354   90667  10000  00001              SET  CMD(CMDF,UNW).EQ.1
                 90670  12650  00303
                 90671  14026  00500
   •     03355   90672  11024  03400              SET  CMD(CMDX,QTYPE).EQ.CQTDR(J,QTYPR)
                 90673  07000  00001
                 90674  10000  03777
                 90675  57056  06300
   •     03356   90676  11005  00000              SET  CMD(CMDX,TRI.EQ.J
                 90677  10040  77400
                 90700  57010  00500
   •     03357   90701  10516  00701              SET  CMD(CMDX-1,LMN).EQ.0
   •     03360   90702  12750  00402              EIMP SC NGTRN
                 90703  65020  77650
   •     03361   90704  12010  53567  ESTRIDS6    SET  ETID(UPI,ID).TO.0
                 90705  10036  53700
   •     03362                        ESTRIDS6    FC   CLEAR THIS REQUEST FOR SHIP DATA
   •     03363   90706  10036  53724              SET  ETID(UPI,1).TO.0
   •     03364                                    FC   DECREASE NO. OF REQUEST
   •     03365   90707  37060  53570              J.CD ESHRED.BY.1
   •   2 03366                        ESTRIDS7    FC   END OF REQUEST STRING.YES.THEN.INCREMENT UNPACK INDEX.THEN.GOTO.E
   •     03367   90710  11020  53567  ESTRIDS7    IF   UPI.LT.PKI.THEN.INCR.UPI.THEN.GOTO.ESTRIDSEE
                 90711  21010  53567
                 90712  60400  90716
                 90713  60700  90716
                 90714  30010  53567
                 90715  61000  90550
   •     03370                                    FC   CLEAR UNPACK INDEX
   •     03371   90716  10010  53567              SET  UPI.TO.0
   •     03372                        ESTRIDS0    FC   ALL REQUESTS PROCESSED.NC.ESTRIDS1
   •     03373   90717  11040  53570              IF   ESHRED.NQT.0.THEN.GOTO.ESTRIDS1
                 90720  60500  90540
   •     03374                                    FC   CLEAR PACKING AND UNPACKING INDICES
   •     03375   90721  14030  53567              SET  NTPNX.TO.0
   •     03376   90722  10010  53572              SET  NTSRCH.TO.0
   •     03377                                    FC   REWIND TAPE.REWINDD
   •     03400   90723  12710  00337              REWINDD   SYSSHIPD.0
                 90724  10000  00000
                 90725  65070  77644
   •     03401                                    FC   RETURN TO PERIODIC.EPLPCN
   •     03402   90726  67010  77661              EPLPCN
   •     03403                                    FC   RETURN
   •     03404                        ESTRIDSA    FC   SET ID
   •     03405   90727  12620  53567  ESTRIDSA    SET  ETID(PKI,ID).EQ.A
                 90730  10030  53545
                 90731  57036  53700
   •     03406   90732  10053  00500              SET  ETI:(PKI,MN).TO.CMD(I,LMN)
                 90733  14016  53724
   •     03407                                    FC   SET REQUESTING MODULE
   •     03410   90734  10010  50477              SET  ESTRIDL.TO.0
   •     03411                                    FC   ESTRIDS00
   •     03412   90735  61000  90529              GOTO ESTRIDS00
```

Figure 6-1 Sample Source/Object Listing

4

Figure 2-2Ø. Program Package (Page 7 of 1Ø)

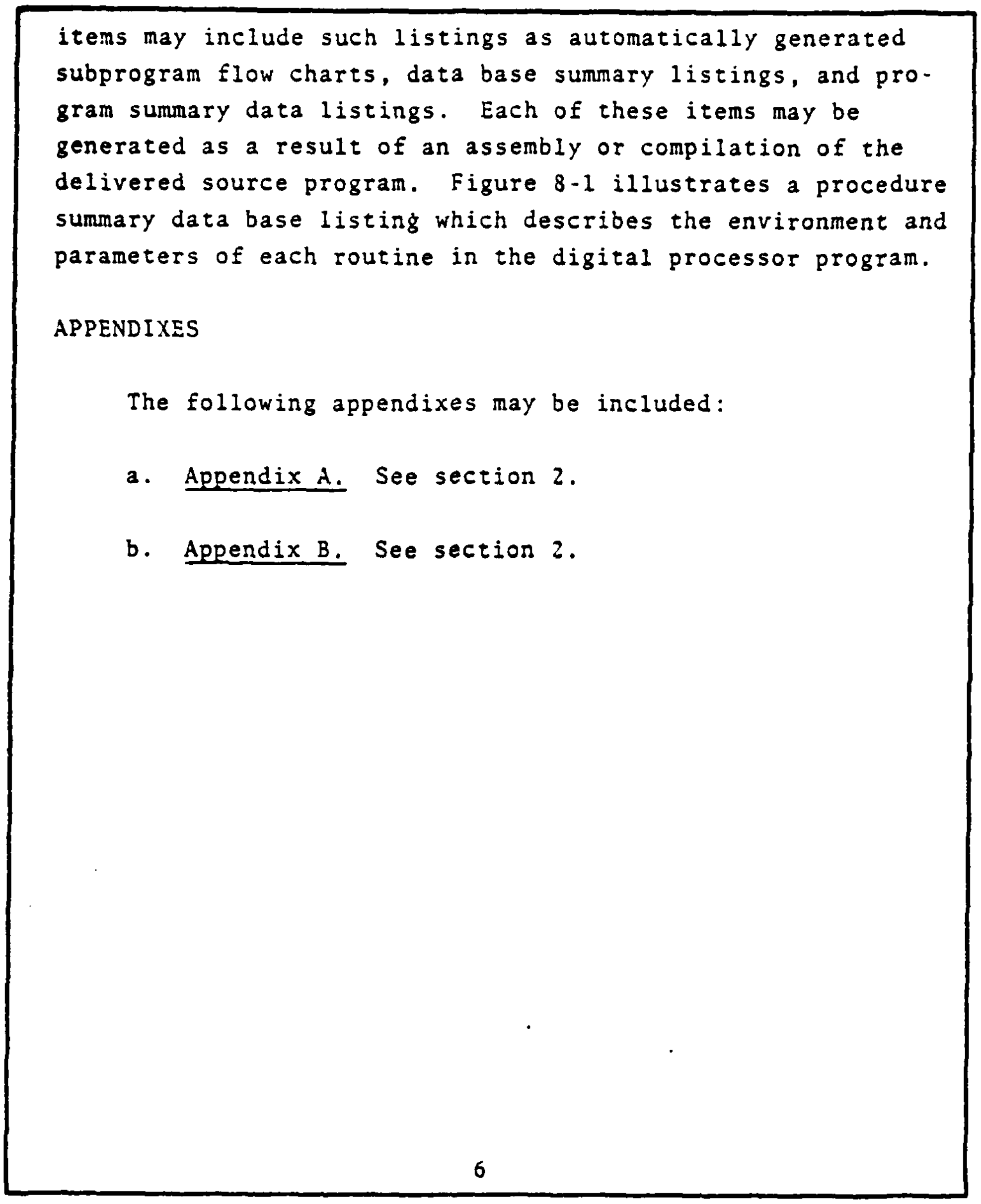


```
LOC     LABEL        REFERENCED BY

03146   MOD1BUFF4    03203
03156   MOD1BUFF5    00643
03161   MOD1BUFF6    00644    03164
03164   MOD1BUFF7    00645
03167   MOD1BUFF8    00646
03172   MOD1BUFF9    03170
00220   MONIT        04237    04241    04424    04624    04640    04643    04652    04654
02620   MAGMSG       02660    03024    03275
02625   MAGMSG1      02622
02636   MAGMSG2      02641
02642   MAGMSG3      02640    02651
00716   MSSGS        01465
00642   MTCDSW       03145
02755   MTCHKSUM     02764    03104    03224    03327    03345
00204   MTFCTWD      02673    02734    02750    02767    03030    03062    03301
00033   MTINI        NONE
00533   MTINTSTR     00625    02741    02752    03006    03073
03241   MTLOCCMP     00674    03260    03265    04231    05156
03257   MTLOCOMP4    03250    03255
03076   MTCSIOLD     03122    03267
03077   MTPBIOLD0    03117
03113   MTPBIOLD1    03116
03114   MTPBIOLD2    03107
03115   MTPBIOLD3    03112
03120   MTPBIOLD4    03103
00240   MTSTAT       02736    02777    05234
02107   MTSTATCK     02740
01046   MTDS         02624    02627    02657
01053   MTDS1        02656
01105   MTDSBUFEND   NONE
00242   NUMWDS       00634    01330    01366    01404    01416    01443    01453    01470    02057
                     02237    02246    02261    02502    02510    02604    02614    02626    02702
                     03236    03577    03610    03624    04364    04746    05101
00771   NZAUD        02477
02574   NZBIODP      00677    02577
00737   NZDMP        02234
02457   NZFLEXDP     00677    02463
02471   NZTYPCDP     00676    02475
05071   POOCH        00703    05131
05117   POOCH1       05122
05121   POOCH2       05134
05132   POOCH3       05120
02765   POSITMTP     02776    03014    03242    03273
02774   POSITMTP0    02773    03004    03010
02775   POSITMTP0B   02771
02772   POSITMTP1    02774
03000   POSITMTP2    03007
00223   PADD         04063    04351    04420    04454    04456    04500    04524    04544    04550
                     04552    04572    04576    04600    04613    04615    04635    04641    04670
```

Figure 7-1 Sample Cross-Reference Listing

Figure 2-20. Program Package (Page 8 of 10)



items may include such listings as automatically generated subprogram flow charts, data base summary listings, and program summary data listings. Each of these items may be generated as a result of an assembly or compilation of the delivered source program. Figure 8-1 illustrates a procedure summary data base listing which describes the environment and parameters of each routine in the digital processor program.

APPENDIXES

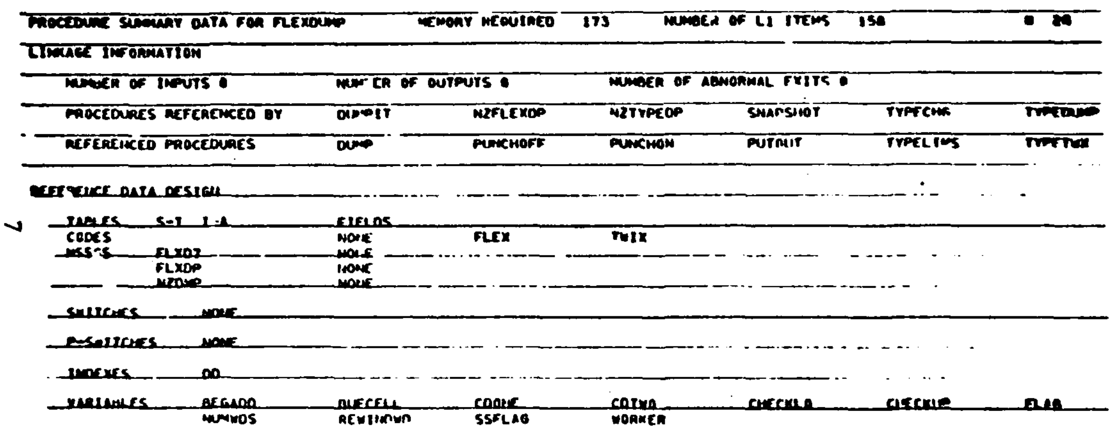
The following appendixes may be included:

a. Appendix A. See section 2.

b. Appendix B. See section 2.

6

Figure 2-2Ø. Program Package (Page 9 of 1Ø)



```
PROCEDURE SUMMARY DATA FOR FLEXDUMP        MEMORY REQUIRED   173      NUMBER OF L1 ITEMS   158                 0  20

LINKAGE INFORMATION

     NUMBER OF INPUTS 0                NUM ER OF OUTPUTS 0                NUMBER OF ABNORMAL EXITS 0

     PROCEDURES REFERENCED BY          DUMPIT        NZFLEXOP      NZTYPEOP      SNAPSHOT      TYPECHG       TYPEDUMP

     REFERENCED PROCEDURES             DUMP          PUNCHOFF      PUNCHON       PUTOUT        TYPELIMS      TYPETWX

REFERENCE DATA DESIGN

     TABLES    S-1  1-A                FIELDS
     CODES                             NONE          FLEX          TWIX
     MSS'S         FLXD2               NONE
                   FLXDP               NONE
                   NZDMP               NONE

     SWITCHES        NONE

     P-SWITCHES      NONE

     INDEXES         DO

     VARIABLES       BEGADD            DUECELL       CODNE         COTWA         CHECKLA       CHECKUP       FLAG
                     NUMWDS            REWINDWD      SSFLAG        WORKER
```

Figure 8-1 Sample Procedure Summary Data Listing

7

Figure 2-20. Program Package (Page 10 of 10)

# APPENDIX C - Standards and Conventions for Use of the CMS-2 Language

Developed by: Tactical System Programming Support Branch,
Marine Corps Tactical System Support Activity,
Camp Pendleton, California 92055

I. Background. While CMS-2 is not the most modern, state-of-the-art computer language in existence, it is nevertheless a powerful High Order Programming Language (HCL) which permits the development of well-designed, structured computer programs. When properly designed and coded, CMS-2 programs can be readily maintained. The purpose of this document is to provide guidance for the design and coding (programming) of CMS-2 programs. SECNAVINST 3560.1 (Tactical Digital Systems Documentation Standards) and MIL-STD-1679 (NAVY) (Weapon System Software Development), although excellent in many respects, provide little specific guidance with regard to the computer program itself. The computer program listing is the single most important tool for software maintenance. Since guidance for computer programs is highly language-dependent at the coding (or listing) level, this document provides guidance in terms of the CMS-2 language. These standards must be complied with. Use of the words "shall" and "must" mean strict adherence is required. Section II defines terms which are used throughout the document. Section III provides guidance on the design and structuring of CMS-2 programs. Section IV gives specific guidance on the standards and conventions for coding CMS-2 programs.

II. Definition of Terms. The purpose of this section is to define several programming terms in relation to specific CMS-2 constructs. This will serve to eliminate much of the semantical confusion which has prevailed. A module, as used in SECNAVINST 3560.1 and in this standard, shall be a SYS-PROC or collection of functionally related SYS-PROC's. Where possible, one module as defined in the Program Design Specification (PDS) shall be mapped into one SYS-PROC in the CMS-2 program. However, where size becomes large, a collection of functionally related SYS-PROC's may constitute a module. A routine, as used in SECNAVINST 3560.1 and in this standard, is a CMS-2 PROCEDURE or CMS-2 FUNCTION. All routines shall be PROCEDURES or FUNCTIONS; there shall be a one-to-one correspondence between them. The use of non-called, "in-line" routines is prohibited. A prologue is defined as the lengthy set of comments found at the beginning of each PROCEDURE or FUNCTION. Section IV.D provides extensive guidance on prologues.

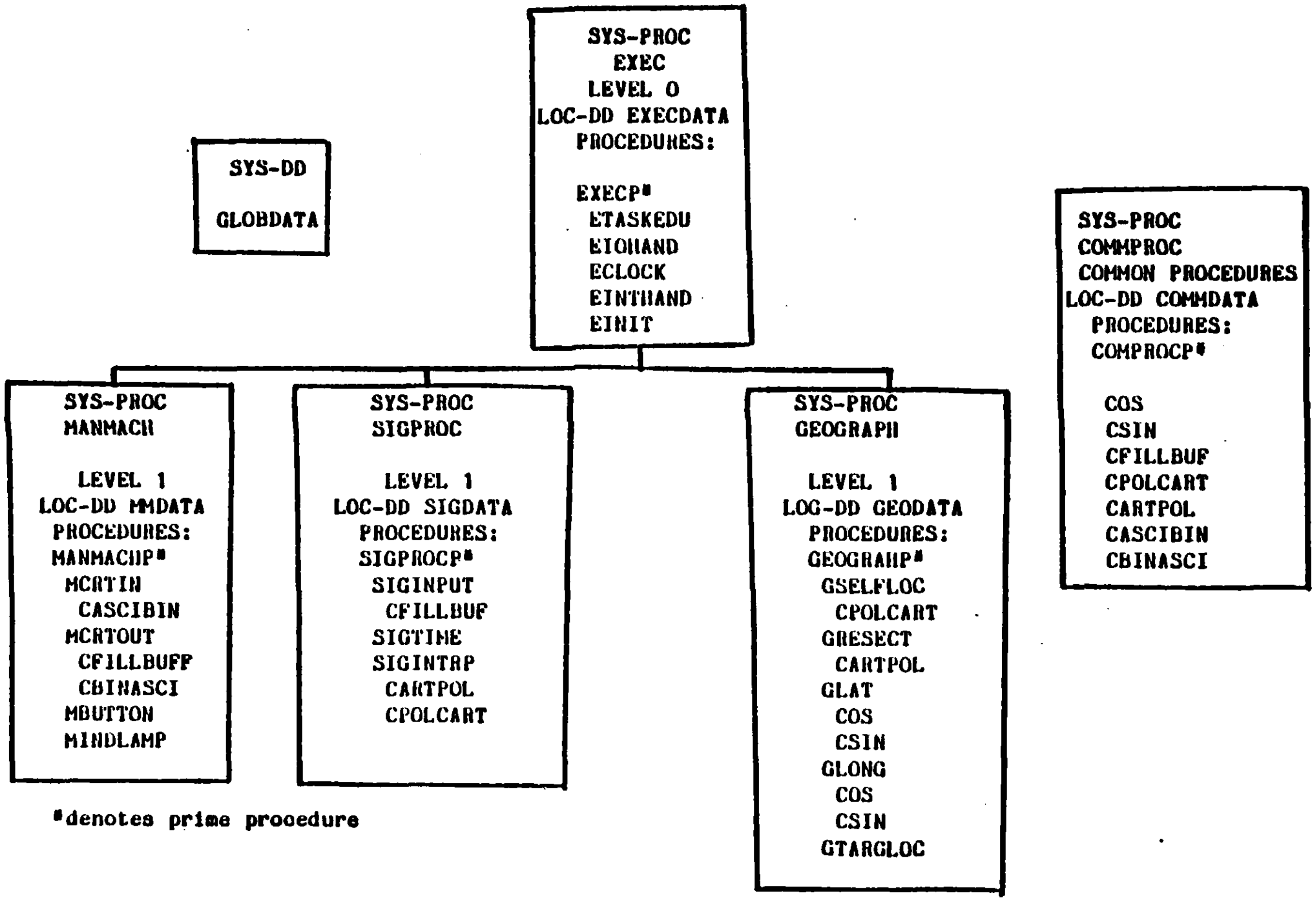
## III. Design and Structure of CMS-2 Programs.

A. From PPS to Program. The performance functional requirements described in the Program Performance Specification (PPS) shall be mapped into program modules which are documented in the Program Design Specification (PDS). The modules of the PDS are then mapped into SYS-PROC's (or logical groups of SYS-PROC's) of the CMS-2 program. These SYS-PROC's are further refined into individual PROCEDURE's or FUNCTION's using the top-down method. The SYS-PROC's and their subordinate PROCEDURES or FUNCTION's must then be documented in the Program Description Document (PDD). It is important that the PDD contain the English name as well as the CMS-2 mnemonic (or code name) of every SYS-PROC (module), PROCEDURE, and FUNCTION. Once this has been done, the computer program may be coded. The entire process is characterized as a number of successive refinements; moving from higher to lower (more detailed) levels of abstraction; going from the general to the specific; progressing from functional requirements to the modules to the manifestation of the requirements in SYS-PROC's, PROCEDURES and FUNCTION's.

B. Data Design Considerations. The global data base requirements of the computer program should reside in one SYS-DD. One SYS-DD should be used. However, if more than one SYS-DD is used, it must be for a logical design consideration such as regional data pools (for large programs) or COMPOOL's for efficient compilation reasons. Under no circumstances will SYS-DD's be allowed to proliferate as desired by individual programmers. Computer programs having n SYS-DD's for n programmers is prohibited. In an analogous manner, each SYS-PROC shall have only one LOC-DD to describe its regional (local) data. The documentation of data base information shall be done in the computer program listing. A Data Base Design document (DBD) is neither desired nor required. Guidance on how data base information is to be implemented in the program listing is given in Section IV.

C. Hierarchical Structure.

Hierarchical structure is important in a program. This structure must be documented by means of a hierarchy diagram which shows the structural relationship between parts of the program. The PDD shall show program structure within a module by means of a complete hierarchy diagram. The PDS shall show part of this structure by means of a hierarchy diagram which describes the program down to the module (SYS-PROC) level diagram. Figure B-1 is an example hierarchy diagram which illustrates a number of desirable attributes of CMS-2 program



*denotes prime procedure

FIGURE B-1 Program Hierarchy

design. There are five SYS-PROC's (EXEC, MANMACH, SIGPROC, GEOGRAPH, and COMMPROC) which comprise the major modules of the system. The hierarchical structure of the program is shown by physical location on the chart and by the designation of levels. In this example, the executive SYS-PROC, EXEC, is at the highest level of control and is at level 0. Only one module (SYS-PROC), the executive, should be at level 0. Only one SYS-PRCC should provide overall control. All other modules (applications modules) are subordinate and are at level 1 or below. Where standard executives such as SDEX-7 or SDEX-20 are used, they will be at level 0. The SYS-PROC's shown at level 1 are the applications modules of the CMS-2 program. MANMACH provides the man-machine interface and consists of the PROCEDURES MANMACHP (which is the prime PROCEDURE), MCRTIN, MCRTOUT, MBUTTON, and MINDLAMP. Notice that, within each SYS-PROC, the calling hierarchy is shown by indentation. For example, each prime PROCEDURE is to the left of all others; and in SYS-PRCC GEOGRAPH, for example, PROCEDURE CARTPOL is to the right of GRESECT. This shows that CARTPOL is subordinate to GRESECT. The following walkthrough is given for further clarification: SYS-PROC EXEC is at hierarchy level 0, SYS-PROC GEOGRAPH is at level 1, (PRIME) PROCEDURE GEOGRAPHP is at level 2, PROCEDURE GRESECT is at level 3, and PROCEDURE CARTPOL is at level 4. In a large program there would be even more levels. SYS-PROC's (modules) are at levels 0 and 1; PROCEDURES (and FUNCTION's) are at levels 2 or more. Although the CMS-2 language permits only two levels of hierarchy from an administrative or syntactical view, it is possible to achieve many structural levels as dictated by the program design by the use of a calling hierarchy.

Common PROCEDURES from the common SYS-PROC, COMMPROC, are called from MANMACH and are thus shown in the hierarchy diagram where they are called even though they actually exist in SYS-PROC COMMPROC. Using this convention, a common PROCEDURE may appear in several application SYS-PROC's where invoked. For example, CFILLBUF is shown in SYS-PROC MANMACH and SYS-PROC SIGPROC since it is invoked from both places. The actual location of CFILLBUF and all other common PROCEDURES is in SYS-PROC COMMPROC, which serves to administratively group the common PROCEDURES. From the total system viewpoint, COMMPROC can be considered to be part of the executive program, although functionally separate. Note that figure B-1 also shows the global data design, SYS-DD GLOBDATA, which contains all global data items in one place.

There shall be no direct calls between SYS-PROC's. Control between SYS-PROC's shall be passed through the executive module. PROCEDURES within a SYS-PROC shall not call PROCEDURES in another SYS-PROC except in the case of common PROCEDURES which shall be grouped in one SYS-PROC. PROCEDURES within the same SYS-PROC shall call only those PROCEDURES which are subordinate, e.g., a PROCEDURE at level 3 shall call only PROCEDURES at level 4, 5, 6 ... n.

## IV. Programming Standards and Conventions

A. General. The computer program listing is the most important tool for the maintenance programmer. The importance of this Section cannot be overemphasized. The primary purpose of this Section is to maximize the maintainability of CMS-2 program listings. Since maintainability is paramount, it is crucial to realize that clarity takes precedence over efficiency; readability takes precedence over writeability. The life-cycle of tactical computer program will see a large fraction of total system costs devoted to software maintenance. It is important that CMS-2 programs be clear, concise, structured, well-designed, modularized, and straightforward -- even at the expense of a few words of computer memory.

B. Program Organization

Figure B-2 illustrates the physical organization of a well-designed CMS-2 program. As required by the compiler, the MAJOR HEADER comes first. When only one MAJOR HEADER is required, all compile-time controls shall be located in this MAJOR HEADER. However, there are times when a program should be compiled several different ways to generate object code for different target computers. When this is required, MINOR HEADERS shall be used with each one containing different C-SWITCHES, MEANS, and EQUALS statements to generate different object programs. Then by use of the librarian, the desired object program may be generated at compile time. The next program element after the various headers is the SYS-DD. Where practicable, all global data items should be declared in one SYS-DD. The restrictions of paragraph III.B of this Enclosure apply. Next, the various SYS-PROC's of the CMS-2 program appear, and, of course, there will normally be many more than shown in Figure B-2. Each SYS-PROC should contain a LOC-DD (if required) which is physically located at the beginning of the SYS-PROC. After the LOC-DD, the various PROCEDURES of the SYS-PROC will appear, and each PROCEDURE shall contain LOC-INDEX'es (as required) at the physical beginning of the PROCEDURE, immediately after the prologue. Where prime PROCEDURES are used (and their use is

```
EXAMPLE SYSTEM
 MAJOR HEADER

 MINOR HEADER 1

 MINOR HEADER 2
      .
      .

 MINOR HEADER DOCUMENTATION
      .
        *
 SYS-DD


 SYS-PROC 1

     LOC-DD

     PROCEDURE 1A
       LOC-INDEX

     PROCEDURE 1B
       LOC-INDEX
           .
           .
           .
 SYS-PROC 2
     LOC-DD

     PROCEDURE 2A
       LOC-INDEX

     PROCEDURE 2B
       LOC-INDEX
           .
           .
           .

           .
           .
           .
END-SYSTEM EXAMPLE
```

*This MINOR HEADER contains the overall program description and prologue.

Figure B-2 CMS-2 Program Physical Organization encouraged), they shall be the first PROCEDURE in the SYS-PROC. The use of LOCREF to preclude the necessity for forward referencing requirements at compile time is encouraged. The LOCREF operator permits PROCEDURES to be physically laid out in the listing in a top-down order which corresponds to the program calling hierarchy. When CMS-2 FUNCTIONS are used, they should appear in a location analogous to PROCEDURES.

C. CSwitches and Headers

CSWITCHES are used to selectively vary object code generated at compile time. They are particularly useful when it is desirable to generate different object programs for different (but similar) target computer configurations. When this is done, the C-SWITCH control statements that control the turning on and off of CSWITCHES will be located in a separate MINOR HEADER, and all of these MINOR HEADERS will be included on the library tape. Of course, at compile time, those required will be selected by the librarian to generate object code for a desired target configuration. However, by placing all MINOR HEADERS on the library tape, all C-SWITCH settings will be available for inspection by maintenance programmers. Each CSWITCH setting in each MINOR HEADER will be well documented with a clear, detailed comment explaining the purpose of the switch, the conditions when it should be used, and all unique aspects of the target configuration it is used for. Then, in the body of the program, CSWITCH brackets will be highlighted by use of a blank line, a line of asterisks, a comment containing the CSWITCH title, another line of asterisks, and another blank line. For example:

```
'' ************************************************************** ''
'' CSWITCH TAOC IS USED TO GENERATE TARGET CODE FOR THE TACTICAL   ''
'' AIR OPERATIONS CENTER CONFIGURATION                             ''
'' ************************************************************** ''

CSWITCH TAOC $
. . . .
. . . . (program code)
. . . .

'' ************************************************************** ''
'' END TACTICAL AIR OPERATIONS CENTER CONFIGURATION CODE           ''
'' ************************************************************** ''

END-CSWITCH TACC $
```

The use of nested CSWITCHES, while not prohibited, is discouraged. When MEANS and EQUALS are used for parameterization and to achieve different target computer configurations, they will be included in separate MINOR HEADERS as appropriate. They will be physically grouped together within each header, not mixed with CSWITCH controls and other compiler options. Furthermore, every MEANS and EQUALS declaration will contain a comment which describes the purpose and use of the statement. For example:

```
'' IN THE TACC CONFIGURATION, THE MAG TAPE DRIVE IS CABLED TO      ''
'' CHANNEL 4.  THIS EQUALS STATEMENT IS USED IN CONJUNCTION WITH   ''
'' CSWITCH TAOC.  CHANGING THIS ONE STATEMENT WILL PERMIT THE      ''
'' PROGRAM TO INTERFACE WITH MAG TAPE DRIVES ON OTHER CHANNELS     ''
   MTCHAN EQUALS 4 $
```

Finally, headers should be logically organized so that compiler controls, CSWITCHES, MEANS statements, EQUAL statements, and other items are physically grouped together.

D. Prologues

Prologues, or narratives as they are sometimes called, are one of the most important aspects of computer program documentation. Good prologues are essential to the understanding of a program by maintenance programmers. They are defined as the lengthy set of comments found at the beginning of each PROCEDURE in a well-documented program. Prologues are required at the beginning of every element of a CMS-2 program. Every prologue shall be clearly delimited from executable code by use of lines of asterisks. A prologue is required at the beginning of the MAJOR HEADER, every MINOR HEADER, every SYS-DD, every SYS-PROC, every LOC-DD, every PROCEDURE, and every FUNCTION in a CMS-2 program. The larger and more complex the program element, the more extensive the prologue should be. In addition, there shall be a large MINOR HEADER which contains a prologue describing the purpose and function of the entire program located before the first SYS-DD (refer to Figure B-2). The program prologue shall describe the overall purpose and functioning of the program, the computer used for compilation, the target computer (or computers), the name of the chief programmer, the company responsible for the program's development, the date the program was delivered to the government, the nomenclature of the tactical system in which the program executes, applicable references and standards (such as the Program Performance Specification and standards which deal with data links, for example), and other pertinent data. In addition, each module of the program will be listed, a brief description of each module will be given and the functional relationships of the modules will be briefly stated. The order of execution, to include the sequence in which the modules are invoked, will be explained in general terms.

The MAJOR HEADER and each MINOR HEADER shall contain a prologue. Wherever different headers are used to generate different object code, the prologue will describe the purpose of the header and specifically identify the target computer and equipment configuration.

The SIS-DD (or SYS-DD's) of the program shall contain a prologue which describes the global data design to include a description of how the SYS-DD is organized. Specifically, MEANS and EQUALS declarations, TABLE declarations, and VRBL declarations shall be segregated and grouped according to type. This shall be explained in the SYS-DD prologue. As much as possible, the SYS-DD prologue shall function as an index to the SYS-DD . Special naming conventions beyond those described in this standard shall be explained in the prologue.

Each SYS-PROC in the computer program shall have an extensive prologue. If a program module consists of more than one SYS-PRCC, then there will be a prologue at the module level as well as one for each SYS-PROC within the module. This module level prologue shall describe how the module functions, shall be physically located at the top of the module, and shall list all SYS-PROC's which belong to the module. When a module is equivalent to a SYS-PROC, the module prologue requirement is satisfied by the SYS-PRCC prologue. In either case, module name, programmer(s), contractor, and delivery date shall be given first. The SYS-PRCC prologue shall contain an extensive, detailed description of the SYS-PROC's purpose and function. The sequence of processing shall be described in chronological order to include the calling sequence of control. The hierarchical structure of the SYS-PROC shall be described, with the name of every PROCEDURE and FUNCTION given. Finally, all inputs and outputs should be listed. The following example illustrates the structure of a good SYS-DD prologue:

```
MSMODULE SYS-PROC   $
COMMENT * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

MSMODULE - M-SERIES MESSAGE PROCESSING MODULE

PROGRAMMERS:  I.M. CODER, U. R. HACKER

CONTRACTOR:  SOFTWARE UNLIMITED, INC.

DELIVERY DATE:  30 MARCH 1980

PURPOSE:  TO PROVIDE THE JOINT SERVICE INTERAGENCY MESSAGE PROTOCOL
          REQUIRED OF THIS COMPUTER PROGRAM BY RESPONDING TO RECEIVED
          M-SERIES MESSAGES AND TRANSMITTING APPROPRIATE M-SERIES MESSAGES
          AS REQUIRED BY THE TECHNICAL INTERFACE DESIGN PLAN (TIDP).

LEVEL:  LEVEL ONE MODULE.

DETAILED DESCRIPTION:  (This portion of the prologue shall contain all of
the items discussed in the paragraph above.  In the case of large, complex
modules, it may extend for five or six pages, or more.  Processing should
be discussed in chronological order.)

SUPERORDINATE SYS-PROCS: _____________________
                         _____________________ (etc.)

SUBORDINATE PROCEDURES: ______________________
                        ______________________ (etc.)

FUNCTIONS: ___________________________________
           ___________________________________ (etc.)

INPUTS: ______________________________________
        ______________________________________ (etc.)

OUTPUTS: _____________________________________
         _____________________________________ (etc.)

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * $
```

The prologue for each PROCEDURE and FUNCTION shall be similar to that for each SYS-PROC except that these prologues will deal with the particular PROCEDURE or FUNCTION.

Each LOC-DD and LOC-INDEX in the program shall have a brief prologue describing the purpose and organization (if necessary) of these data design elements. The use of asterisks and single quote marks to highlight key comments is encouraged.

E. Data Declarations.

As specified in this standard, the Data Base Design (DBD) requirements of SECNAVINST 3560.1 and MIL-STD-1679 are to be met in the computer program listing. Consequently, it is very important that the data design elements of a CMS-2 program, the SYS-DD's, LOC-DD's, and LOC-INDEX's, contain the information found in the DBD.

Where possible, all global data elements should be contained in one SYS-DD. The use of EXTREF and EXTDEF for variables and tables should be avoided. If these elements are global, they should be in the SYS-DD. If the SYS-DD becomes too large, in terms of CMS-2 symbol table capacity, then some use of COMPOOLS may be required. Local data elements belong in a LOC-DD, and not in a SYS-DD. The SYS-DD should be organized to contain first the prologue described in paragraph III.D, then all MEANS and EQUALS declarations (logically grouped), all VRBL declarations (logically grouped), all TABLE (and array) declarations (logically grouped), and all P-SWITCH declarations.

All MEANS and EQUALS declarations should be contained in the SYS-DD unless it is necessary to place some of them in MINOR HEADERS so that the program may be compiled differently for different equipment configurations. The use of MEANS and EQUALS declarations in locations other than MINOR HEADERS or SYS-DD's is prohibited. The use of the EXCHANGE primitive is forbidden. The use of MEANS and EQUALS declarations to increase readability of the program is encouraged. For example, the statements

```
TRUE MEANS 1  $
FALSE MEANS 0 $
```

increase the readability of the program. The use of MEANS and EQUALS primitives to reduce typing work, such as

```
PROC MEANS PROCEDURE  $
```

is forbidden. The use of MEANS and EQUALS primitives to corrupt the CMS-2 language such as,

```
REPEAT MEANS GOTO $
```

is forbidden. The purpose of each MEANS or EQUALS declaration shall be documented with a meaningful comment as shown in paragraph IV C:

VRBL declarations shall contain meaningful comments which describe the purpose, initial value, range, and related data structures of the VRBL. The use of short, cryptic comments is forbidden. Every VRBL, no matter how simple, must have the above attributes explained. An example of a good VRBL declaration is:

```
'' MSGQPTR IS THE MESSAGE QUEUE POINTER WHICH ALWAYS POINTS TO   ''
'' THE LAST MESSAGE WHICH HAS BEEN INSERTED INTO TABLE MSGQUEUE. ''
'' IT IS INITIALIZED TO ZERO (WHEN THE MESSAGE QUEUE IS EMPTY)   ''
'' AND ITS RANGE IS FROM 0 TO 25 (WHEN THE MESSAGE QUEUE IS      ''
'' FULL).  IF IT IS EVER GREATER THAN 25, AN ERROR CONDITION     ''
'' (QUEUE OVERFLOW) WILL RESULT, AND THE QUEUE WILL BE FLUSHED   ''
'' WITH MSGQPTR RESET TO 0.                                      ''
VRBL MSGQPTR I 16 U P 0 $
```

TABLE declarations are similar to VRBL declarations when it comes to documentation requirements. Because TABLES can be very complex data structures, they must be explained in detail. Each TABLE, SUB-TABLE, LIKE-TABLE, and FIELD will be described as to purpose, initial value, range, and related data structures, if any. The following example illustrates these concepts:

```
COMMENT
    TABLE ACCOUNTS IS USED TO STORE INFORMATION ON 400 BANK ACCOUNTS.
    EACH ITEM (OR ACCOUNT) CONTAINS AN ACCOUNT NAME (FIELD ACCTNAME) WHICH
    CAN CONTAIN UP TO 40 ASC11 CHARACTERS, AN ACCOUNT NUMBER (FIELD
    ACCTNR) WHICH CAN RANGE FROM ZERO TO 9999, A BALANCE WHICH CAN RANGE
    FROM -9999.99 DOLLARS TO +9999.99 DOLLARS, AND AN ACTIVE/NON-ACTIVE
    FLAG (BOOLEAN FIELD ACTIVE) WHICH WHEN TRUE (=1) MEANS ACTIVE AND
    NON-ACTIVE WHEN FALSE (=0).  AT PROGRAM INITIALIZATION TIME, THE
    ENTIRE TABLE IS FLUSHED (SET TO ZEROES).  INDICES (OR POINTERS)
    RELATED TO THIS TABLE ARE VRBLS LASTACCT, NEXTACCT, AND NEWACCT.    $

TABLE ACCOUNTS V DENSE   400 $
  FIELD ACCTNAME H 20        $
  FIELD ACCTNR   I 14 U      $
  FIELD BALANCE  A 22 S 7    $
  FIELD ACTIVE   B           $
END-TABLE ACCOUNTS           $
```

Note that the FIELD declarations are indented two columns in from the TABLE declaration to show subordination. Also, that H, I, A, and B and 20, 14 and 22 are vertically aligned. Where possible, TABLES and VRBLS shall be declared in alphabetical order.

Local data items found in LOC-DD's and LOC-INDEX's shall be grouped and commented as shown above for SYS-DD's. The importance of placing data elements which are required by only one SYS-PROC into the LOC-DD cannot be overemphasized. This practice promotes information hiding and permits different programmers to work on different SYS-PROC's without concerning themselves with the names and details of other SYS-PROC's.

P-SWITCH's shall be declared in the SYS-DD if the PROCEDURE's used are global in scope. P-SWITCH's shall be declared in a LOC-DD if the PROCEDURE's used are of local scope. The declaration of a P-SWITCH outside a SYS-DD or LOC-DD is forbidden. They shall be well-commented as shown in the example below:

```
COMMENT
    BASED ON THE VALUE OF GLOBAL VARIABLE TRIGINDX (RANGE: 0-5), THIS
    P-SWITCH WILL CALL THE APPROPRIATE PROCEDURE WHICH WILL RETURN THE
    VALUE FOR ONE OF THE SIX TRIGONOMETRIC FUNCTIONS: SINE, COSINE,
    TANGENT; COTANGENT, COSECANT, OR SECANT.  THE INPUT ANGLE MUST BE AN
    ANGLE BETWEEN PLUS OR MINUS 360 DEGREES, AN A-TYPE VRBL (A 24 S 14)
    WITH FRACTIONAL ACCURACY TO ONE PART IN 16,384.  OUTPUT TRIGANS
    RETURNS AN ARITHMETIC VALUE IN THE RANGE PLUS OR MINUS 262,144 WITH
    FRACTIONAL ACCURACY TO ONE PART IN 8,192 (A 32 S 13).  CERTAIN
    TRIGONOMETRIC FUNCTIONS, SUCH AS TANGENT (90 DEGREES) HAVE INFINITE
    VALUE.  IN THESE CASES, A VALUE OF 262,144 IS RETURNED.              $

P-SWITCH TRIGFUNC INPUT ANGLE OUTPUT TRIGANS $
    SINE          ''        CASE 0        ''          $
    COSINE        ''        CASE 1        ''          $
    TANGENT       ''        CASE 2        ''          $
    COTANGNT      ''        CASE 3        ''          $
    COSECANT      ''        CASE 4        ''          $
    SECANT        ''        CASE 5        ''          $
END-SWITCH TRIGFUNC $
```

The use of the P-SWITCH operator for multipath branching is preferred over the use of the FOR operator in most cases. However, there are instances when the FOR operator is preferable; for example, when two or more values cause branching to the same procedure or when the range of values is not sequential. In the latter case, the FOR statement avoids the need for dummy procedures. In other computer languages, FOR is used for iterative looping. Only in CMS-2 is it used for multipath branching. Since P-SWITCH declarations are physically separated from their invocation, a meaningful comment at the point of invocation shall be provided for clarity.

F. Size of Elements.

There is no limit (other than those imposed by the compiler) to the size of a SYS-DD or LOC-DD. PROCEDURE's and FUNCTIONS's are limited to 100 lines of CMS-2 source code, exclusive of comments. This is an absolute limit which may be exceeded only upon prior approval by the government on a case-by-case basis. Where PROCEDURE's and FUNCTION's contain direct code, they are limited to 50 lines of code, exclusive of comments. The average size of all PROCEDURE's shall be 50 lines. Exceptions to these size restrictions are not permitted. Programs with overly large PROCEDURE's indicate poor design and a lack of partitioning the program into functionally independent parts of manageable, maintainable size. The use of "in-line routines" is expressly forbidden.

Every procedure shall have one and only one entry point. This is an absolute restriction. Every procedure should have only one RETURN or exit point, although this is not an absolute requirement.

G. Naming Conventions.

In the naming of program elements such SYS-PROC's, VRBL's, TABLE's, and PROCEDURE's, the CMS-2 language leaves much to be desired. Names are limited to eight characters and the underscore character is not permitted. This inhibits the readability of names. However, within the constraints of the compiler, much can be done to enhance readability and maintainability, which is the subject of this section.

Every module, or SYS-PROC, in a CMS-2 program shall have a unique prefix consisting of one or two characters. If less than 26 modules comprise the program, then one letter will suffice as the module prefix. If more than 26 modules are used, or if the program designer believes that it will enhance maintainability, then two characters shall be used. These two characters shall be two letters or a letter followed by a number. Examples of one-letter prefixes are U for UTILMOD, a utilities module and M for MMIMOD, a man-machine interface module. Examples of two-letter prefixes are M3 for M3MODULE and IO for IOMODULE.

Once a prefix has been established for each SYS-PROC (module), then every subordinate element of that module shall use the module prefix as the first one or two characters of every name. For example, IOMODULE might have as subordinates PROCEDURE's IOMTAPE (a magnetic tape handler), IOTTY (the teletype handler), and IOCRT (the computer-CRT interface). Every PROCEDURE, VRBL, FUNCTION, TABLE, etc. of a module shall contain its prefix as an identifying mark. Common (global) data elements are not subject to these restrictions, but will be named with a prefix starting with the letter C.

All names within a CMS-2 program shall be descriptive. They shall attempt to describe the item they represent. Names such as IOBUFFER, USINE, and MSGFLAG have inherent meaning and are easier for a maintenance programmer to remember while tracing through a program. Names such as A, X, N, or BX are meaningless, and their use is forbidden. Related data elements should have related names which show their interrelationship. For example, a TABLE called IOBUFFER might logically have an index or pointer which is called IOBUFPTR. Applying the above rules and common sense will increase the maintainability of a CMS-2 program.

H. Commenting.

Without good commenting, even a well-designed program can be extremely difficult to maintain. The use of meaningful comments to increase the understandability of a program cannot be overemphasized. Additionally, it is almost impossible to overcomment. It is better to overcomment than to undercomment. This section deals with in-line comments which serve to explain and supplement source code rather than PROCEDURE and module prologues which are discussed in section D. There are three kinds of comments: stand alone, which are on a separate line from any source code; terminating, which follows source code on the same line; and embedded, which are embedded within a source code line. More will be said about these three types later. For consistency, all stand alone comments shall precede the code they explain.

Comments should explain, amplify, and supplement source code rather than echo the code. For example the statement and comment

```
SET N TO N + 1 '' INCREMENT N '' $
```

does nothing to explain why N is being incremented. It is also an example of a terminating comment. Terminating comments are prohibited, except with direct code and to amplify data declarations. A better method of commenting would be:

```
'' A MESSAGE HAS JUST BEEN INSERTED IN MSGQUEUE.  INCREMENT       ''
'' MSGQPTR SO THAT IT POINTS TO THE LOCATION WHERE THE NEXT MSG  ''
'' MAY BE INSERTED.                                               ''
SET MSGQPTR TO MSGQPTR +1 $
```

Another example of an illuminating comment is:

```
'' THE MESSAGE QUEUE CAN ONLY HOLD 25 MSGS. THUS, IF MSGQPTR GT ''
'' 25 OVERFLOW HAS RESULTED--FLUSH THE MESSAGE QUEUE. ''
IF MSGQPTR GT 25 THEN FLUSHQ $
```

In CMS-2, there should be, on the average, no less than one line of commenting for every two lines of source code. In direct code, there should be, on the average, no less than one comment for every line of direct code. These averages pertain to amplifying comments, exclusive of prologue comments. These averages are minimum requirements. The use of more comments is encouraged.

The following example illustrates good terminating comments for direct code:

```
          L R3,CQPTR                         .CQPTR POINTS TO ITEMS IN
          LK R4,6                            .A CIRCULAR QUEUE OF SIZE 7
                                             .AND SHOULD RANGE FROM 0 TO 6
                                             .IN VALUE - SO INCREMENT IT OR
                                             .ZERO IT DEPENDING ON ITS VALUE
                                             .COMPARED TO 6
          CR R3, R4                          .IF CQPTR LESS THAN 6 THEN
          JLS INCRMT                         .GO TO INCREMENT
          LL R3, 0                           .ELSE SET CQPTR TO ZERO
          S R3, CQPTR                        .AND SAVE IT
          J BYPASS                           .BYPASS INCREMENT CODE
INCRMT.   IROR R                             .SET CQPTR=CQPTR+1
          S R3, CQPTR                        .AND SAVE IT
BYPASS.   __________________                 .CONTINUE
```

The above comments do not echo the code, they explain it. The comments, in effect, translate the assembly language into high level code. Contrast this with the following comments that merely echo the code:

```
          L R3, CQPTR                        .PUT CQPTR IN REG 3
          LK R4, 6                           .PUT 6 IN REG 4
          CR R3, R4                          .COMPARE REGS 3 AND 4
```

These comments are worse than none at all, for they insult the maintenance programmer by insinuating that he does not know the assembly language instruction set.

In addition to echoing the code, there are several other pitfalls that some commenters fall into. One of these is the "80 column mentality" where the programmer crowds terminating comments into the same line as the code at the expense of abbreviating the comment into an incomprehensible line of garble. For example the statement and comment

```
SET MSGQPTR TO MSGQPTR+1 '' INCR MSGQPTR PT NXT MSG '' $
```

would have been better as,

```
    '' INCREMENT MSGQPTR TO POINT TO THE NEXT MESSAGE IN THE QUEUE   ''
      SET MSGQPTR TO MSGQPTR +1 $
```

Another common pitfall is the embedded comment. For example the statement

```
IF '' THE MSG QPTR '' MSGQPTR GT 25
'' MAX SIZE OF THE QUEUE '' THEN
'' FLUSH THE QUEUE '' FLUSHQ $
```

embeds so many comments into the code, it is difficult to distinguish between the code and the comments. Embedded comments are prohibited. The preferred method is to place comments on separate lines, and, where appropriate, separate them from the code by indenting, using blank lines, and blocking comments with asterisks.

### I. Physical Layout

Good physical layout is defined as that property of a computer program listing which makes it capable of being read and understood by a programmer not familiar with the program. Good physical layout implies ease of understanding and good readibility. Good readability may be achieved by a variety of techniques, some of which are separation of logical elements of code, separation of comments and code, blocking (by using lines of asterisks) lengthy comments or prologues, the appropriate use of blank lines, logical indentation, and the lining up of BEGIN-END and IF-ELSE pairs.

Separation of logical elements and the use of blank lines go hand in hand. The practice of beginning PROCEDURES on a new page serves to separate these logical elements and promote readability. The use of blank lines to separate prologues and lengthy comments from executable code also promotes readability. Prologues and lengthy comments should be boxed by asterisks to make them stand out and be separated from the code. Blank lines should be used freely to prevent crowding and to separate logical entities.

Indentation is a key part of physical layout. Indentation is defined as the physical indenting of logically subordinate and nested program constructs. A truly structured program is structured in two ways. First, it is structured with regard to the flow of control of the program. Second, it is physically structured by the use of indentation. Indentation shall be used so that program logical pairs are lined up and stand out. Every BEGIN shall be physically indented to line up with its corresponding END. The nested level of the BEGIN-END block shall be denoted by a number in a terminating comment. The following example illustrates the good use of indentation to achieve readability.

```
BEGIN '' 1 '' $
  _______________
  _______________
  IF ____________ THEN
    BEGIN ''2'' $
       ___________
       ___________
       IF _________ THEN
          BEGIN ''3'' $
            _____________
            _____________
            _____________
          END ''3'' $
       ELSE
          BEGIN ''4'' $
            _____________
            _____________
          END ''4'' $
    END ''2'' $
END ''1'' $
```

In the above example, it is clear which BEGIN belongs with which END. The practice of "hiding" BEGIN's as follows

```
      IF ______________ THEN BEGIN $
```

is prohibited.

CMS-2 has two drawbacks which make indenting difficult. First, the code must begin in column 11 or later; Columns 1-10 are not available for indenting. Second, the fact (in CMS-2Y at least) that side-by-side object code begins in column 28 complicates the problem. If the programmer indents too much, the source CMS-2 code gets mixed up with the generated object code. The situation calls for case-by-case judgements on the part of the programmer. As a rule, two columns per indentation level is preferred when there are eight or less levels of indentation. When more than eight levels of indentation or nesting occur, the programmer should use one column of indentation per level to avoid mixing the source and object code.

A final note on readability: All PROCEDURES shall begin at the top of a new page by use of the page eject function. (SYS-PROC's and SYS-DD's are placed at the top of the page automatically by the compiler.)

## J. Direct Code

Direct code should be used only to achieve input or output, work around compiler problems, or to optimize frequently executed code. Optimization will be done only after testing of the fully loaded running system proves that optimization is required. The latter reason for using direct code is permitted only when prior approval is given by the cognizant government agency. This will be done on a case-by-case basis. Direct code shall be used to work around compiler problems only when it is not possible to work around them in high-level code. Whenever direct code is used, it shall be clearly separated from the high level code by the use of blank lines, lines of asterisks, and a prologue, similar to the prologue required at the beginning of each procedure. This prologue shall describe the reason for the section of direct code. Within the section of direct code, the use of comments is important (see Section H on commenting direct code).

K. IF Clauses

The use of complex IF clauses can cause logical problems with the flow of control of a CMS-2 program. IF clauses should be simple, such as

```
IF IOFLAG EQ 10 THEN ...
```

Complex IF clauses are difficult to understand and lead to logic flaws. The use of more than one AND or one OR per IF clause is discouraged. Where complex IF statements are used, they shall be generously commented. The use of the CCMP operator is forbidden.

APPENDIX D - Program Planning Summary

Available from: Defense Technical Information Center

RESEARCH AND DEVELOPMENT PLANNING SUMMARY

1. AGENCY ACCESSION

2. DATE OF SUMMARY MAY 79

REPORT CONTROL SYMBOL

3. KIND OF SUMMARY D-CHANGE

4. LEVEL OF SUMMARY TASK AREA

5. SUMMARY SECURITY U

6. REGRADING N/A

7. WORK SECURITY U

8a. PROGRAM ELEMENT/PROJECT/TASK AREA NUMBER 62721N ZF21-242-001

8b. FORMER PROGRAM ELEMENT/PROJECT/TASK AREA NUMBER RF21-242-401

9. TITLE (Precede with Security Classification Code)

(U) SOFTWARE COST REDUCTION

10. RESPONSIBLE DOD ORGANIZATION

NAME NAVAL RESEARCH LABORATORY
ADDRESS WASHINGTON, DC 20375

RESP. IND. DR. BRUCE WALD CODE 7500
TELEPHONE NO. 202-767-2903

11. START DATE 1 OCT 78

12. COMPLETION DATE 1 OCT 82

| 13. RESOURCES ESTIMATE | | a. RDT&E FUNDS (Thousands) | b. % OF FUNDS ON CONTRACT/GRANT PROGRAM |
|---|---|---|---|
| CURRENT FY | 79 | 114 | |
| BUDGET FY | 80 | 380 | 15 |
| BUDGET FY | 81 | 550 | 40 |

14. PARTICIPATION 004200 Computers; 019700 Computers and related programming (Control, guidance, and navigation)

15. MISSION OBJECTIVE

16. SCIENTIFIC/TECHNICAL AREA

17 THROUGH 19.

17. (U) OBJECTIVE AND APPROACH: Reduce the life cycle cost of Naval software by conducting a critical experiment to assess the value of software engineering (SE) innovations to assure that a) technology base funds are spent only on potentially useful techniques, and b) software acquisition managers are made aware of the value of these techniques. In the experiment, an existing flight software package for the A-7 aircraft is being redesigned in accordance with new SE principles and the efficiency, real-time performance and flexibility of the new software will be compared with the performance of software produced by more conventional methods.

18. (U) PLANS. FY 80: Initiate redevelopment of A7 Onboard Flight Program (OFP) in accordance with the following software engineering techniques: Information Hiding Modules, Abstract Interfaces, Cooperating Sequential Processes, Process Synchronization Primitives, Uses Hierarchy, Resource Monitor Modules, Formal Specifications, Disciplined Programming and Program Verification.
FY80: Continue redevelopment and begin to assess advantages and costs of these techniques. FY81 milestones: Complete design documentation, Dec 79; complete implementation of a kernel of software to perform a selected subset of functions, June 80.

19. (U) PROGRESS AND ACCOMPLISHMENTS. This project was initiated with NRL Technology Base funding; a Software Requirements document was produced under that project. The document has been reviewed by NWC personnel for accuracy and sufficiency. It describes the principal interfaces between the software and the other system components and all the functions to be performed by the software. This document will serve as a reference for the remainder of the project, and is being used by NWC for other purposes. A paper has been published about the techniques developed to document software requirements. The major software modules and patterns of interaction have been identified and described.

# LIST OF REFERENCES

1---"Computer Snafu Falsely Signals Soviet Attack"; Monterey Herald; Nov 10, 1979.

2---Greve, F. "Pentagon Calls Its Computer 'A Disaster'": S. F. Sunday Examiner & Chronicle; Nov 4, 1979.

3---Defense Science Board; Report of the Task Force on Technology Base Strategy; p. 41; October 1976. (DLSIE Accession No.: LD 38154A)

4---Coppola, A. and Sukert, A. N.; Reliability and Maintainability Management Manual; Rome Air Development Center Report RADC-TR-79-200; pp.127-151; July 1979.

5---Myers, G. J.; Software Reliability Principles and Practices; John Wiley & Sons; 1976.

6---De Roze, B .C.; Special Presentation, Proceedings of the "Managing the Development of Weapons System Software" Conference; pp.4-2 - 4-12; May 1976.

7---Mills, H. D.; "Software Development"; IEEE Transactions on Software Engineering; December 1976.

8---Van Tassel, D.: Program Style, Design, Efficiency, Debugging and Testing; Prentice-Hall; 1978.

9---Boehm, B.; "Software Engineering Education, Some Industry Needs", Software Engineering Education; Wasserman, A. and Freeman, P. (Editors); Springer-Verlag; New York, 1967.

10--Daly, E. B.; "Management of Software Development"; IEEE Transactions on Software Engineering; pp. 229-242; May 1977.

11--Oxman, S. W.; "The Testing of the TRIDENT Command and Control System"; Digest for the Workshop on Software Testing and Test Documentation; pp. 284-295; December 1978.

12--Tausworthe, R. C.; Standardized Development of Computer Software (Part I, Methods; Part II, Standards); Jet Propulsion Laboratory, California Institute of Technology; Part I, 1976: Part II 1978.

13--McCall, J. A.; "The Utility of Software Quality Metrics in Large-Scale Software System Developments." Second Software Life Cycle Management Workshop; pp. 191-194; August 21-22, 1978.

14--Stewart, S. L. (Editor); Concepts in Quality Software Design ; NBS Technical Note 842; U.S. Government Printing Office; 1974.

15--Swanson, E. B.; "The Dimensions of Maintenance"; Proceedings 2nd International Conference on Software Engineering; pp. 492-497; 1976.

16--Canning, R. G., (Editor); "That Maintenance 'Iceberg'"; EDP Analyzer; October 1972.

17--Kline, M. B. "Software & Hardware R&M: What are the Differences?"; Proceedings Annual Reliability and Maintainability Symposium; IEEE; pp. 179-185; 1980.

18--Manley, J. H.; "Software Life Cycle Management: Dynamics Theory"; Second Software Life Cycle Management Workshop; pp. 7-20; August 21-22, 1978.

19--Brown, J. R.; "Modeling, Measuring and Managing Software Cost"; Second Software Life Cycle Management Workshop; pp. 47-51; August 21-22, 1978.

20--McHenry, R. C. and Walston, C. E.; "Software Life Cycle Management: Weapons Process Developer"; IEEE Transactions on Software Engineering; pp. 334-344; July 1978.

21--U.S. General Accounting Office; Report to the Congress; Problems in Developing the Advanced Logistics System. Report Number LCD-75-101; 17 June 1976.

22--Cave, W. C. and Salisbury, A. B.: "Controlling the Software Life Cycle - The Project Management Task"; IEEE Transactions on Software Engineering; pp. 326-334; July 1978.

23--Cooper, J. D.; "Corporate Level Software Management"; IEEE Transactions on Software Engineering; pp. 319-3255; July 1978.

24--MITRE Corporation. DoD Weapons Systems Software Acquisition and Management Study ; MTR-6908; Vo. I; June 1975. (DLSIE Accession No.: LD 38652A)

25--Kossiakoff, A., etal.; DoD Weapon System Software Management Study ; Applied Physics Laboratory, The Johns Hopkins University; Report SR-75-3; June 1975. (DTIC Accession Number: AD-A022160)

26--Assistant Secretary of Defense; Defense System Software Management Plan ; Mar 1976. (DTIC Accession No.: AD A022558)

27--Stanfield, J. R. and Skrukrud, A. M.; Software Acquisition Management Guidebook, Software Maintenance Volume ; Systems Development Corp.; TM-5772/004/02; Nov 77. (DTIC Accession Number: AD-A053040)

28--Bersoff, E. H.; Henderson, V. D.; and Siegel, S. G.; "Software Configuration Management: A Tutorial"; Computer; pp. 6-14; January 1979.

29--De Roze, B. C. and Nyman, T. H.; "The Software Life Cycle: A Management and Technological Challenge in the Department of Defense"; IEEE Transactions on Software Engineering; Vol SE-4, No. 4; pp. 309-318; July 1978.

30--Schneidewind, N. F.; "The Applicability of Hardware Reliability Principles to Computer Software"; Software Quality Management; Petrocelli Books; pp. 171-181; 1979.

31--Fein, R.; Survey of Software Development Technology at the Naval Surface Weapons Center; Dahlgren Laboratory, Dahlgren, Va.; July 1976. (DTIC Accession No.: AD A027451)

32--Pariseau, R. J.; Improved Software Productivity for Military Computer Systems Through Structured Programming ; Report NADC-76044-50; Naval Air Development Center; 12 March 1976.

33--Dijkstra, E. W.; "Programming Considered as a Human Activity"; Proceedings of the IFIP Congress; pp. 213-217; 1965.

34--Dijkstra, E. W.; A Discipline of Programming; Prentice-Hall; 1976.

35--Warnier, J. D.; Logical Construction of Programs (L.C.P.); Van Nostrand Reinhold Co.; 1974.

36--Jackson, M. A.; Principles of Program Design; Academic Press; 1975.

37--Yourdon, E.; Techniques of Program Structure and Design; Prentice-Hall; 1975.

38--Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R.; Structured Programming; Academic Press; 1972.

39--McGowan, C. L. and Kelley, J. R.; Top-Down Structured Programming Techniques; Petrocelli/Charter; 1975.

40--Jensen, R. W. and Tonies, C. C.; Software Engineering; Prentice-Hall, 1979.

41--Wirth, N.; "On the Composition of Well-Structured Programs"; ACM Computing Surveys; December 1974.

42--McHenry, R. C. and Rand, J. A.; "Software Technology and System Integration"; 2nd Software Life Cycle Management Workshop; pp. 77-80; 20-22 August 1978.

43--McHenry, R. C. and Rand, J. A.; Integration Engineering: An Approach to Rapid System Deployment ; FSD 77-0179; IBM; 1977.

44--McHenry, R. C. and Rand, J. A.; Software Technology and Integration Engineering ; FSD 78-0034; IBM; 1977.

45--Meyers, G. J.; Composite Design: The Design of Modular Programs; Technical Report TR00.2406; IBM; January 29, 1973.

46--McGowan, C. L.; and McHenry, R. C.; "Software Management"; Research Directions in Software Technology; MIT Press; pp. 207-253; 1979.

47--Bohm, C. and Jacopini, G.; "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules"; Communications of the ACM; May 1966.

48--Buxton, J. N. and Randel B. (Editors); Software Engineering Techniques; Report on a Conference Sponsored by the Nato Science Committee, Rome, Italy; 27-31 October 1969.

49--Glass, R. L.; Software Reliability Guidebook; Prentice-Hall; 1979.

50--Fisher, D. A.; "The Interaction Between the Preliminary Designs and the Technical Requirements for the DoD Common High Order Language"; Proceedings of 3rd International Conference on Software Engineering; pp. 82-83; 10-12 May 1978.

51--Glass, R. L.; "From Pascal to Pebbleman and Beyond"; Datamation; pp. 146-150; July 1979.

52--Dijkstra, E. W.; "On the Green Language Submitted to the DoD"; SIGPLAN NOTICES; pp.16-21; October 1978.

53--Hurwitz, J. and Kinucan, P.; "ADA"; Mini-Micro Systems; December 1979.

54--Bowen, J. B.; "A Survey of Standards and Proposed Metrics for Software Quality Testing"; Computer; pp. 37-42; August 1979.

55--Canning, R. G., (Editor); "The Production of Better Software"; EDP Analyzer; February 1979.

56--Miyamoto, I.; "Reliability Evaluation and Management for an Entire Software Life Cycle"; Second Software Life Cycle Management Workshop; pp. 195-208; August 21-22, 1978.

57--Glasser, A. L.; "The Evolution of a Source Code Control System"; Proceedings of the Software Quality and Assurance Workshop; ACM; pp. 122-125; 1978.

58--Josephs, W. H.; "A Mini-Computer Based Library Control System"; Proceedings of the Software Quality and Assurance Workshop; ACM; pp. 126-132; 1978.

59--IBM Federal Systems Center; "Documentation Standards"; Structured Programming Series; Vol. VII ; USAF RADC; July 1975. (DTIC Accession Numbers: AD-A008639 and AD-A016414)

60--Chapin; "Flow Charting with the ANSI Standard: A Tutorial," ACM Computing Surveys; June 1970.

61--Brooks, F. P. Jr; The Mythical Man-Month ; Addison-Wesley; 1975.

62--Aron, J.; The Program Development Process. The Individual Programmer; Addison-Wesley; 1974.

63--Weinberg, G. M.; The Psychology of Computer Programming; Van Nostrand Reinhold Co.; 1971.

64--Schneiderman, Mayer, McKay and Heller; "Experimental Investigation of the Utility of Detailed Flow Charts in Programming"; Communications of the ACM; June 1977.

65--IBM Corp.; HIPO - A Design Aid and Documentation Technique; GC20-1851-1; 1974.

66--Anderson, G. E. and Shumate, K. C.; "Documentation Study Proves Utility of Program Listings"; Computerworld; May 21, 1979.

67--Pooch, U. W.; "Translation of Decision Tables"; ACM Computing Surveys; pp. 125-151; June 1974.

68--Keller, J. F. and Roesch, R. W., Jr.; A Decision Logic Table Preprocessor ; Masters Thesis, Naval Postgraduate School, Monterey California; June 1977.

69--Fisher, D. L.; "Data Documentation and Decision Tables"; Communications of the ACM; pp. 26-31; January 1966.

70--Yoder, C. M. and Schrag, M. L.; "Nassi-Shneiderman Charts - An Alternative to Flowcharts for Design"; Proceedings of the Software Quality and Assurance Workshop; ACM; pp. 79-86; 1978.

71--Knuth, D. E.; "Computer Programming As an Art"; Communications of the ACM; pp. 667-673; December 1974.

72--Wegner, P.; "Introduction and Overview"; Research Directions in Software Technology; MIT Press; pp. 1-36; 1979.

73--Lientz, B. P. and Swanson, E. B.; Software Maintenance Management; Addison-Wesley; 1980.

# INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22314

2. Library, Code 0142 2
Naval Postgraduate School
Monterey, California 93940

3. Department Chairman, Code 52 2
Computer Science Department
Naval Postgraduate School
Monterey, California 93940

4. Professor Norman F. Schneidewind, Code 54Ss 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93940

5. Professor Melvin B. Kline, Code 54Kx 1
Administrative Sciences Department
Naval Postgraduate School
Monterey, California 93940

6. Associate Professor B. J. Carey, Code 52Ck 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93940

7. Assistant Professor L. Cox, Code 52C1 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93940

8. Major Russell D. Pilcher, USMC 5
128 South 2nd East
Kaysville, Utah 84037

9. Lieutenant Mark Moranville, USN 1
Naval Electronics Systems Engineering Center
San Diego, California 92101

FILM