

DOD-STD-1703 (NS)

12 FEBRUARY 1987

SUPERSEDING

NSAM 81-3

26 JULY 1979

MILITARY STANDARD

SOFTWARE PRODUCT STANDARDS



AMSC NO. G4044

AREA MCCR

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.


NSA/CSS Manual 81-3/DOD-STD1703(NS)
15 April 1987

FOREWORD

This manual is a companion volume to NSA Manual 81-2, the NSA/CSS Software Acquisition Manual. For in-house software development, it is called NSA Manual 81-3, NSA/CSS Software Product Standards Manual. For contracted software acquisition, the document is called DOD-STD-1703(NS), Software Product Standards. It provides outlines of required documents, programming standards, and descriptions of recommended software design methodologies. It also identifies Data Item Descriptions recommended for use in contracted software acquisition.

This second edition of the Manual has been designed to be more useful and usable to software managers and software developers. Guidance has been added to better explain the development activities associated with preparation of the required documents. Formats for several additional optional documents have also been added.

This manual is effective upon publication. It supersedes the first edition of NSA/CSS Manual 81-3, dated 26 July 1979. Changes to the first edition are printed in bold type. Comments and recommendations for improvements should be referred to the Deputy Director for Telecommunications and Computer Services.


GEORGE R. COTTER
Deputy Director
for

Telecommunications and Computer Services

DISTRIBUTION II

Plus DISTRIBUTION III FOR T&R

OPI T303 (963-3227s, 688-7691b)

TABLE OF CONTENTS

	<u>PAGE</u>
PART I - GENERAL	
1.1 INTRODUCTION	1-3
1.2 DEFINITIONS	1-5
1.2.1 SOFTWARE	1-5
1.2.2 FIRMWARE	1-5
1.2.3 MANAGEMENT AUTHORITIES, ROLES, AND RESPONSIBILITIES	1-5
1.2.4 SYSTEM AND SOFTWARE	1-6
1.2.5 CONFIGURATION MANAGEMENT BASELINES	1-7
1.2.6 DATA DICTIONARY	1-8
1.3 TAILORING	1-8
1.4 SOFTWARE TERMINOLOGY AND THE SOFTWARE DEVELOPMENT PROCESS	1-9
1.4.1 SOFTWARE REQUIREMENTS DEFINITION	1-9
1.4.2 PRELIMINARY DESIGN	1-10
1.4.3 DETAILED DESIGN	1-11
1.4.4 REVIEWS	1-11
1.4.5 SUMMARY	1-11
1.5 REFERENCES	1-15
1.6 LIST OF DATA ITEM DESCRIPTIONS	1-16
PART II - SOFTWARE PRODUCT MANAGEMENT	
2.1 SOFTWARE DEVELOPMENT PLAN	2-3
2.1.1 POLICY AND REQUIREMENTS SUMMARY	2-3
2.1.2 GUIDANCE	2-3
2.1.2.1 GENERAL	2-3
2.1.2.2 INCREMENTAL DEVELOPMENT	2-3
2.1.2.3 SOFTWARE STANDARDS AND PRACTICES MANUAL	2-5
2.1.3 FORMAT FOR THE SOFTWARE DEVELOPMENT PLAN	2-7
2.1.4 FORMAT FOR THE SOFTWARE STANDARDS AND PRACTICES MANUAL	2-12
2.2 SOFTWARE QUALITY ASSURANCE	2-15
2.2.1 POLICY AND REQUIREMENTS SUMMARY	2-15
2.2.2 GUIDANCE	2-15
2.2.2.1 GENERAL	2-15
2.2.2.2 DETERMINING THE SIZE OF THE SOFTWARE QUALITY ASSURANCE EFFORT	2-15
2.2.3 FORMAT FOR THE SOFTWARE QUALITY ASSURANCE PLAN	2-17

	<u>PAGE</u>
2.3 SOFTWARE CONFIGURATION MANAGEMENT	2-21
2.3.1 POLICY AND REQUIREMENTS SUMMARY	2-21
2.3.2 GUIDANCE	2-21
2.3.3 FORMAT FOR THE SOFTWARE CONFIGURATION MANAGEMENT PLAN	2-25
 PART III - SOFTWARE DEVELOPMENT SPECIFICATIONS	
3.1 SOFTWARE REQUIREMENTS SPECIFICATION	3-3
3.1.1 POLICY AND REQUIREMENTS SUMMARY	3-3
3.1.2 GUIDANCE	3-3
3.1.2.1 GENERAL	3-3
3.1.2.2 IDENTIFICATION AND SELECTION OF COMPUTER PROGRAMS	3-5
3.1.2.3 METHODS OF VERIFYING THE SATISFACTION OF SOFTWARE REQUIREMENTS	3-8
3.1.2.4 COMPUTER SECURITY REQUIREMENTS	3-9
3.1.3 FORMAT FOR THE SOFTWARE REQUIREMENTS SPECIFICATION	3-11
3.2 SOFTWARE SYSTEM/SUBSYSTEM SPECIFICATION	3-25
3.2.1 POLICY AND REQUIREMENTS SUMMARY	3-25
3.2.2 GUIDANCE	3-25
3.2.2.1 INTERFACE CONTROL DOCUMENTATION	3-25
3.2.3 FORMAT FOR THE SOFTWARE SYSTEM/SUBSYSTEM SPECIFICATION	3-27
3.2.4 FORMAT FOR INTERFACE CONTROL DOCUMENT	3-43
3.3 SOFTWARE PROGRAM SPECIFICATION	3-47
3.3.1 POLICY AND REQUIREMENTS SUMMARY	3-47
3.3.2 GUIDANCE	3-47
3.3.3 FORMAT FOR SOFTWARE PROGRAM SPECIFICATION	3-49
3.4 SEPARATE DATA DOCUMENTATION	3-59
3.4.1 INTRODUCTION	3-59
3.4.2 GUIDANCE	3-60
3.4.2.1 SELECTION CRITERIA FOR DATA DOCUMENTATION METHODS	3-60
3.4.2.2 PREPARATION OF THE DATA DICTIONARY DOCUMENT	3-61
3.4.2.3 STANDARD DATA ELEMENT NAMES	3-63
3.4.2.4 SAMPLE DATA DICTIONARY DOCUMENT ENTRIES	3-63
3.4.3 FORMAT FOR THE DATA DICTIONARY DOCUMENT	3-69
 PART IV - SOFTWARE DEVELOPMENT PRACTICES	
4.1 SOFTWARE ANALYSIS AND DESIGN	4-3
4.1.1 POLICY AND REQUIREMENTS SUMMARY	4-3
4.1.2 INTRODUCTION	4-3
4.1.3 ANALYSIS AND DESIGN METHODOLOGIES	4-5
4.2 UNIT DEVELOPMENT FOLDERS	4-43
4.2.1 POLICY AND REQUIREMENTS SUMMARY	4-43
4.2.2 GUIDANCE	4-43
4.2.3 FORMAT FOR UNIT DEVELOPMENT FOLDERS	4-45

		<u>PAGE</u>
4.3	SOFTWARE DESIGN AND CODE INSPECTIONS	4-53
4.3.1	POLICY AND REQUIREMENTS SUMMARY	4-53
4.3.2	INTRODUCTION	4-53
4.3.3	THE INSPECTION PROCESS	4-53
4.3.3.1	PARTICIPANTS IN THE INSPECTION PROCESS	4-54
4.3.3.2	TYPES OF INSPECTIONS	4-54
4.3.3.2.1	DETAILED DESIGN INSPECTIONS	4-54
4.3.3.2.2	CODE INSPECTIONS	4-57
4.3.3.2.3	OTHER INSPECTIONS	4-57
4.3.3.3	FORMAL INSPECTION STEPS	4-57
4.3.4	PERSONNEL CONSIDERATIONS WHEN USING INSPECTIONS	4-63
4.4	PROGRAMMING STANDARDS	4-65
4.4.1	POLICY AND REQUIREMENTS SUMMARY	4-65
4.4.2	GUIDANCE	4-65
4.4.3	PROGRAMMING STANDARDS AND GUIDELINES	4-67

PART V - SOFTWARE TEST AND OPERATIONS

5.1	SOFTWARE GENERAL UNIT TEST PLAN	5-3
5.1.1	POLICY AND REQUIREMENTS SUMMARY	5-3
5.1.2	GUIDANCE	5-3
5.1.3	FORMAT FOR THE GENERAL UNIT TEST PLAN	5-5
5.2	SOFTWARE SYSTEM INTEGRATION AND TEST PLAN	5-9
5.2.1	POLICY AND REQUIREMENTS SUMMARY	5-9
5.2.2	GUIDANCE	5-9
5.2.3	FORMAT FOR THE SOFTWARE SYSTEM INTEGRATION AND TEST PLAN	5-11
5.3	SOFTWARE SYSTEM DEVELOPMENT TEST AND EVALUATION (DT&E) PLAN	5-19
5.3.1	POLICY AND REQUIREMENTS SUMMARY	5-19
5.3.2	GUIDANCE	5-19
5.3.3	FORMAT FOR SOFTWARE SYSTEM DT&E TEST PLAN	5-21
5.4	OPTIONAL TEST AND BUILD DELIVERY DOCUMENTATION	5-27
5.4.1	POLICY AND REQUIREMENTS SUMMARY	5-27
5.4.2	GUIDANCE	5-27
5.4.2.1	SOFTWARE TEST PROCEDURES	5-27
5.4.2.2	SOFTWARE TEST REPORT	5-28
5.4.2.3	BUILD DESCRIPTION DOCUMENT	5-28
5.4.3	FORMAT FOR SOFTWARE TEST PROCEDURES	5-29
5.4.4	FORMAT FOR SOFTWARE TEST REPORT	5-33
5.4.5	FORMAT FOR BUILD DESCRIPTION DOCUMENT	5-35

		<u>PAGE</u>
5.5	SOFTWARE MANUALS	5-39
5.5.1	POLICY AND REQUIREMENTS SUMMARY	5-39
5.5.2	GUIDANCE	5-39
5.5.3	DESIGN REQUIREMENTS FOR SOFTWARE MANUALS	5-40
5.5.4	FORMAT FOR USER'S MANUAL	5-43
5.5.5	FORMAT FOR SOFTWARE SYSTEM USER'S MANUAL	5-49
5.5.6	FORMAT FOR POSITIONAL HANDBOOKS	5-55
5.5.7	FORMAT FOR COMPUTER OPERATION MANUAL	5-59
5.5.8	FORMAT FOR PROGRAM MAINTENANCE MANUAL	5-63
5.5.9	FORMAT FOR FIRMWARE SUPPORT MANUAL	5-69
PART VI - SOFTWARE PRODUCT ACCEPTANCE		
6.1	SOFTWARE END-PRODUCT ACCEPTANCE PLAN	6-3
6.1.1	POLICY AND REQUIREMENTS SUMMARY	6-3
6.1.2	FORMAT FOR SOFTWARE END-PRODUCT ACCEPTANCE PLAN	6-5

LIST OF FIGURES

		<u>PAGE</u>
FIGURE 1-1	SOFTWARE DEVELOPMENT SEQUENCE	1-4
FIGURE 1-2	SYSTEM ENGINEERING ACTIVITY	1-13
FIGURE 1-3	SOFTWARE REQUIREMENTS ANALYSIS ACTIVITY	1-13
FIGURE 1-4	SOFTWARE PRELIMINARY DESIGN ACTIVITY	1-13
FIGURE 1-5	SOFTWARE DETAILED DESIGN ACTIVITY	1-14
FIGURE 3-1	EXAMPLE OF A TEST AND QUALIFICATION CROSS-REFERENCE INDEX	3-22
FIGURE 3-2	EXAMPLE OF A TABLE THAT IDENTIFIES COMPUTER PROGRAM COMPONENTS	3-31
FIGURE 3-3	EXAMPLE OF A FUNCTIONAL REQUIREMENTS MATRIX	3-33
FIGURE 3-4	EXAMPLE OF A TABLE ALLOCATING REQUIREMENTS TO SOFTWARE UNITS	3-40
FIGURE 3-5	RELATIONSHIPS BETWEEN DATA STORES, DATA STRUCTURES, AND DATA ELEMENTS	3-65
FIGURE 3-6	EXAMPLE OF A DATA STORE SET-USE MATRIX	3-76
FIGURE 3-7	EXAMPLE OF A DATA STRUCTURE SET-USE MATRIX	3-76
FIGURE 4-1	LEVEL 0 DATA FLOW DIAGRAM	4-8
FIGURE 4-2	PROCESS DECOMPOSITION	4-10
FIGURE 4-3	STRUCTURE CHART	4-13
FIGURE 4-4	BASIC N ² CHART	4-19
FIGURE 4-5	N ² CHART - CIRCLE FORMAT	4-20
FIGURE 4-6	DECISION TABLE PARTS	4-37
FIGURE 4-7	LIMITED ENTRY DECISION TABLE	4-37
FIGURE 4-8	EXTENDED ENTRY DECISION TABLE	4-38
FIGURE 4-9	MIXED ENTRY DECISION TABLE	4-38
FIGURE 4-10	EXAMPLE OF UDF COVER SHEET	4-46
FIGURE 4-11	EXAMPLE OF UDF CHANGE LOG	4-47
FIGURE 4-12	EXAMPLE OF REFERENCE LOG FOR MATERIAL IN A SEPARATE LOCATION	4-48
FIGURE 4-13	UNIT TEST CASE/REQUIREMENTS/FUNCTIONAL CAPABILITIES LIST MATRIX	4-51
FIGURE 4-14	SAMPLE DETAILED DESIGN INSPECTION CHECKLIST	4-55
FIGURE 4-15	GENERAL CODE INSPECTION CHECKLIST	4-58
FIGURE 4-16	SAMPLE DETAILED DESIGN INSPECTION PROBLEM LIST	4-61
FIGURE 4-17	SAMPLE CODE INSPECTION PROBLEM LIST	4-62
FIGURE 5-1	EXAMPLE OF A SOFTWARE REQUIREMENTS/TEST CASE MATRIX	5-24
FIGURE 5-2	TEST PROCEDURE TABLE	5-31

PART I - GENERAL

1.1 INTRODUCTION

This is the second edition of NSAM 81-3. It describes a set of documents and activities that comply with the policies of the NSA/CSS Software Acquisition Manual (NSAM 81-2). Figure 1-1 depicts the sequence of software development covered by the NSA/CSS Software Acquisition Manual. It also identifies documents required by the Manual and shows the sequence in which they shall be prepared and reviewed.

Since they were published in 1978 and 1979, these manuals have served as the standard for the acquisition and development of software systems for the National Security Agency/Central Security Service. The new editions of NSAM 81-2 and NSAM 81-3 take advantage of what has been learned in the seven years since they were originally published. Changes to the manuals are not radical. Formats for several optional documents have been added. Other plans and specifications have been modified to emphasize important activities or to make them easier to understand and use. A policy on Software Inspections has been added to encourage more use of systematic peer reviews to detect defects and errors in software products. In several places, the manuals have been changed to encourage, but not mandate, incremental development of software systems. Changes to the original manuals are typed in bold print.

This manual is divided into six parts. Part I provides general information relating to the other five parts. Parts II through VI are subdivided into seventeen sections relating to specific policies of NSAM 81-2. Twelve of the sections provide formats and contents of documents required by NSAM 81-2. Two other sections contain formats for separate data base documentation and optional test documentation. The other three sections contain programming standards, a description of recommended design and analysis methodologies, and guidance on how to conduct software inspections. Each section contains a summary of the applicable policy of NSAM 81-2. Most sections also have additional guidance to aid in the preparation and use of the documents.

Software Acquisition Managers and Agency Software Development Managers may choose to develop project-specific documents that differ from the documents described in this manual. Any documents or specifications which differ, however, must contain the information necessary to comply with the policies of NSAM 81-2. Use of the document outlines contained in this manual also does not guarantee that the document satisfies all of the requirements of the NSA/CSS Software Acquisition Manual. Many of the policies have requirements that cannot be directly represented in a description of format and contents of a document. Software developers should periodically compare evolving documents and ongoing activities with their associated policies to ensure that they satisfy the provisions of the NSA/CSS Software Acquisition Manual.

Proposed changes to this manual must be submitted in writing to the Deputy Director for Telecommunications and Computer Services (DDT). The Software Development Policy Change Control Board (See NSAM 81-2, Section 1.6) will review proposals for changing both NSAM 81-2 and 81-3 and recommend action for DDT approval.

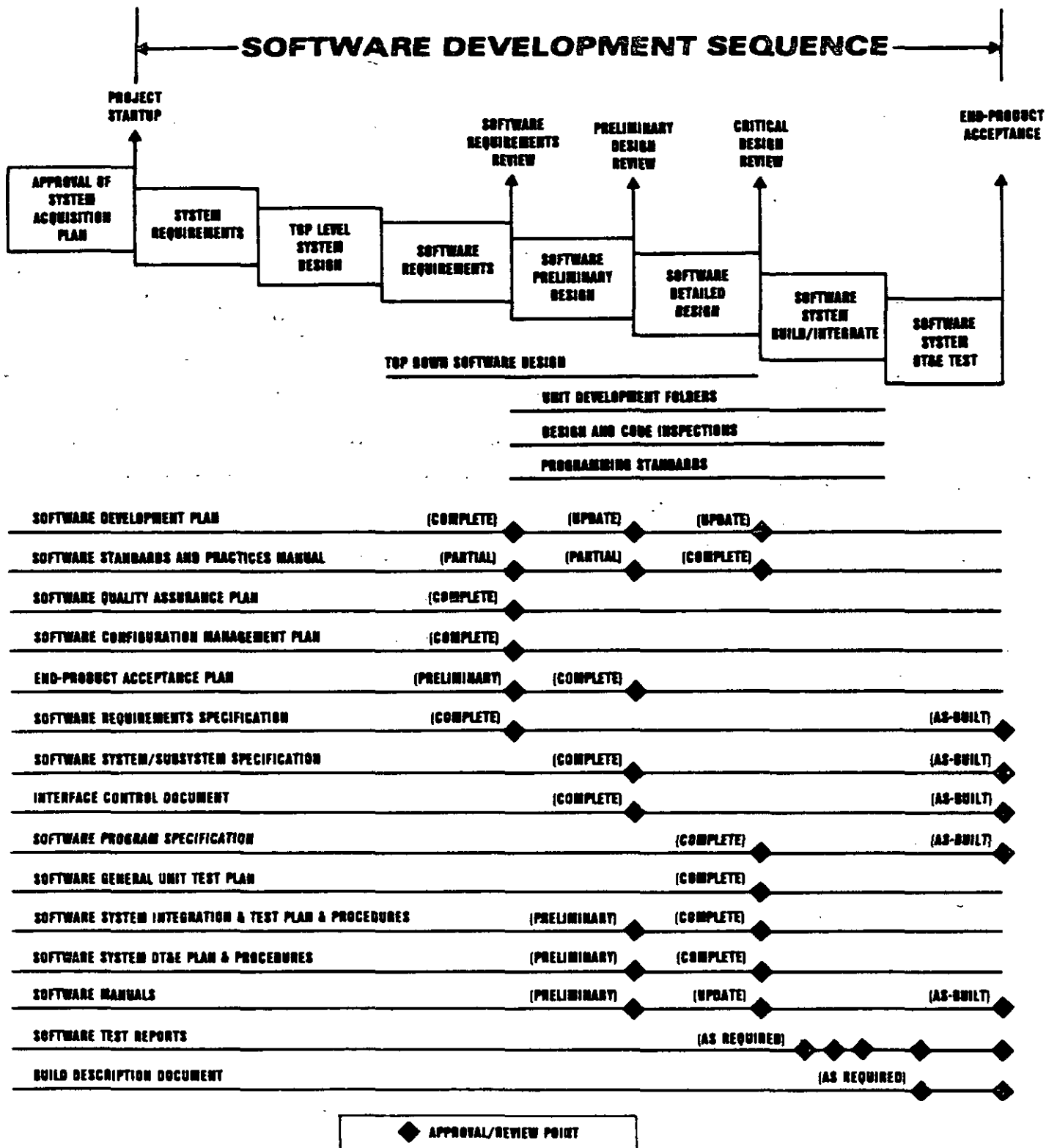


FIGURE 1-1

1.2 DEFINITIONS

1.2.1 SOFTWARE

As used in this manual, software covers the full range of computer programs, firmware, microcode, and data definitions including operating systems, standard utilities, file management systems, and applications programs written to implement the functions of a specific computer-based system.

1.2.2 FIRMWARE

The combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device. The software cannot be readily modified under program control.

1.2.3 MANAGEMENT AUTHORITIES, ROLES, AND RESPONSIBILITIES

- a. Acquisition Decision Authority - Senior NSA/CSS executives (e.g., DDR, DDI, DDT, DDO) or their designees (e.g., ADDR, Group Chief) who, with concurrence of the NSA/CSS System Acquisition Executive (DDR), have been designated to be responsible for exercising management control and decision and approval authority over projects and systems during the system acquisition planning approval and acquisition phases of the systems management process.
- b. Software Acquisition Manager - The person responsible to the System Acquisition Manager for managing the planning and acquisition of the software segment(s) of a system acquisition. If no Software Acquisition Manager is identified, the System Acquisition Manager shall assume the responsibilities of the Software Acquisition Manager.
- c. Software Development Manager - the individual in the Software Development Organization responsible for managing a software development project.
- d. Software Development Organization (also referred to in this manual as the "developer") - The organization that designs, builds, and integrates software to satisfy the System Acquisition Plan and, as applicable, procurement documents.
- e. Software Quality Assurance Manager - The individual in the Software Development Organization responsible for the Quality Assurance function.
- f. System Acquisition Manager - The individual appointed by the Acquisition Decision Authority during acquisition planning who is responsible for managing the acquisition steps of System Design, System Building and Integration, and DT&E.
- g. System Acquisition Organization (also referred to in this manual as the "customer") - The organization responsible for planning the Acquisition Phase of the system and for insuring that the developer builds a system that

satisfies system development requirements identified in the System Acquisition Plan. The System Acquisition Manager is responsible for activities performed in this organization.

h. System Operations Customer (also referred to in this manual as the user) - The principal representative of the operations group(s) with responsibility for reviewing and controlling changes to the operational requirements and participating during the entire Acquisition Phase. This individual is identified during the System Concept Phase.

i. System Support Authority - The Chief of the organization responsible for managing life-cycle support planning functions (during the acquisition phase) and for providing life cycle support (during the operational phase).

j. Technical Review Board - The people appointed by the System Acquisition Manager to participate in reviews and assist in evaluating the adequacy of the developer's approach to satisfying system development requirements. The Board includes representatives of the Systems Operations Customer, System Support Authority, and other members with necessary technical skills and experience.

1.2.4 SYSTEM AND SOFTWARE

The following definitions identify elements that apply to the computer software portion of systems. They are particularly relevant to the Configuration Management Plan in Part II and to the Software Specifications in Part III.

a. System - A system is a composite of equipment, skills, and techniques capable of performing or supporting an operational role. A complete system includes all of the associated equipment, facilities, material, computer programs, firmware, technical documentation, services, and personnel required for operations and support to the degree necessary for self-sufficient use in its intended operational environment.

b. Computer Program Function - A system function whose implementation has been formally allocated to software.

c. Software System/Subsystem - The portion of a system or subsystem that consists of one or more computer programs.

d. Computer Program - A series of instructions or statements and data definitions in a form acceptable to a computer, designed to cause the computer to execute an operation or operations. Computer programs include operating systems, job-control language procedures, assemblers, compilers, interpreters, data maintenance/diagnostic programs, as well as applications programs. A computer program normally satisfies an end-use function and is designated for configuration management purposes as a configuration item. It is a deliverable entity which implements a specified grouping of computer program functions.

e. Computer Program Component - A functionally or logically distinct part of a computer program. It is distinguished in design and development as an assembly of subordinate elements (units) of a computer program. It is immediately subordinate to a computer program and is the lowest level to which allocation of a computer program function must be displayed.

f. Unit - An aggregate of appropriately-sized software to which the satisfaction of requirements can be traced. It is one of a hierarchy of entities into which a computer program component is decomposed and may be at any level of the software hierarchy. Typical units are between 100 and 1000 higher-order language statements. Typical criteria for selecting units are:

- (1) The unit performs a well-defined function;
- (2) The unit is amenable to development by one person within the assigned schedule;
- (3) The unit is an aggregate of software to which the satisfaction of requirements can be traced;
- (4) The unit is amenable to thorough testing;
- (5) The unit has a cyclomatic complexity of 10 or less. (for an explanation of this concept, see page 4-71).

g. Routine - A set of instructions and statements that exists as an identifiable entity and carries out some well-defined operation or set of operations. It is usually the smallest compilable element of a software system. A unit may consist of one routine or it may have several routines.

1.2.5 CONFIGURATION MANAGEMENT BASELINES

The NSA/CSS Systems Acquisition Manual (NSA Manual 81-1) identifies configuration baselines which are normally defined during the system life cycle. The following definitions identify the configuration baselines as they apply to software development.

a. Software Functional Baseline - This baseline is established after top-level system design activities have allocated development requirements to the Software System or Subsystem. It is established with publication and approval of the Software Requirements Specification. This approval takes place at the Software Requirements Review.

b. Allocated Baseline - This baseline is established with the specification of the detailed functional design and the performance of each functional element sufficient to begin the detail design process. It is normally established at a Preliminary Design Review during the Acquisition Phase.

c. Product Baseline - This baseline describes the "as built" software system in terms of its function, performance, and operational characteristics. It is first established at the beginning of the system integration and continues in effect until completion of the Operational Test and Evaluation Phase.

1.2.6 DATA DICTIONARY

A manual or automated file holding data element definitions and information on how data is shared and used in a system. The dictionary contains the full definition of each data element. Directories show how the data elements are used in files/data bases/interface formats, record types, data fields, or software. The basic function is to produce consistent documentation of the whole system and to encourage standardization, e.g., of naming conventions. Many additional functions can be added, including the building of general purpose software such as data validation programs using data configurations stored in the Data Dictionary.

1.3 TAILORING

The software development policies in the NSA/CSS Software Acquisition Manual (NSAM 81-2) and the documents and activities described in this Manual (NSAM 81-3) identify a sound methodology for acquiring and developing software systems. There are, however, many ways to apply the methodology to different types of software projects and still satisfy the requirements of NSAM 81-2.

When considering how to tailor the standards, software managers should consider their own needs as managers and the needs of the ultimate users of the software. Many factors affect these needs. Some of them, stated as questions here, need to be answered before any tailoring decisions are made. They are as follows:

- a. How large and how complex is the software product?
- b. Who will use the software product?
- c. How long will it be used?
- d. Will someone be required to provide life-cycle support for the product after it is developed?
- e. During development, which of the following does the customer consider most important: cost control, schedule control, or functional capability?
- f. What are the development risks?
- g. How large is the development staff?

Based upon the answers to these questions, software managers should determine the activities, documents, and products that developers, customers, and users need to have a high-quality software product. If activities prescribed by NSAM 81-2 are not necessary, Section 1.5 (pages 7 - 8) of NSAM 81-2 describes the method for requesting exceptions to the required activities. If the software project is not large enough to warrant separate activities and plans, they may be combined, reduced, or amplified as appropriate. If sections within document formats are not relevant, they may be omitted with the notation that they are not applicable.

The important point is that tailoring of activities and documents is both acceptable and expected. There is, however, no quick, easy formula for generalized tailoring; software managers must adapt the standards to meet the needs of their projects. When considering how to tailor the use of standards on a project, software managers should also remember the following axioms of software development:

- a. To be successful, a software project must be well-managed.
- b. Good management requires planning, attention to detail, and discipline.
- c. Documentation is a primary vehicle with which managers manage.
- d. Until a software product is tested, no one will know its quality.
- e. Both customers and users have a vital role to play in any software development effort; documentation gives them the information necessary to perform their role.

Tailoring must never eliminate activities or documentation that will cause these axioms to be compromised.

1.4 SOFTWARE TERMINOLOGY AND THE SOFTWARE DEVELOPMENT PROCESS

NSAM 81-2 and this manual assume that system-level activities required by NSA/CSS Circular 25-5, Systems Acquisition Management, result in a thorough system engineering exercise in which the system objectives are decomposed into a hierarchy of solution-independent, system-level functions. At this point of development, each system function is defined only in general terms. These manuals further assume that the solution to each system function has been formally allocated to hardware, software, or manual systems or subsystems. System functions allocated to software systems then come under the jurisdiction of the software management policies of NSAM 81-2 and become known as computer program functions.

At this point, the software development process begins. NSAM 81-2 and this manual describe a three-step process for designing software systems. The first is Software Requirements Definition. The second is Preliminary (or Architectural) Design, and the third is Detailed Design. The product of each step is a Specification that documents the software as it evolves from a concept to a functioning system.

1.4.1 SOFTWARE REQUIREMENTS DEFINITION

After the software functions are identified, Software Requirements Definition begins. The Software Manager analyzes the software functions (also called software requirements) to determine the extent to which they should be grouped into composite entities for acquisition purposes. Each such grouping of software functions is called a computer program and is identified, defined,

and described in the Software Requirements Specification. Each computer program is a software entity that must be built and delivered. The size and components of the computer programs, however, are not yet known. The Software Requirements Definition step is completed when requirements for all computer programs are defined, the Software Requirements Specification is completed, and the customer agrees that the requirements can be baselined.

1.4.2 PRELIMINARY DESIGN

Once the functions of a computer program are defined, preliminary design begins. Software design is basically a decision-making process. First the designer must identify what decisions must be made, then generate alternative ways of making the decisions, then evaluate the alternatives. Finally, the designer must select one alternative as the best design approach.

As a part of selecting the best design approach, the designer must establish the architecture of the software system that will be built. This means spelling out in general terms how the software system will look--what functions it will perform--how it will be built--and what major algorithms it will use. This establishes the primary elements of the system and identifies their relationships with each other. The chosen alternative and the selected architecture is the preliminary design documented in the Software/Subsystem Specification.

During preliminary design, the individual computer program functions may lose their identity in the software structure. To ensure that each function is implemented, this manual requires that each function of software be allocated and be traceable to at least one level lower than the computer program level. This requires that each computer program be broken into components. A single function will not be allocated to more than one computer program; but within a computer program, functions may be allocated to more than one component.

Computer Program Components may likewise be successively decomposed into increasingly more detailed entities. The number of decompositions will vary with the size and complexity of the Computer Program Component. Since the exact number cannot be specified in advance, the products of component decomposition are given the generic name "units." A Computer Program Component always consists of at least one unit and may consist of a hierarchy of many units.

Software units are the basic building blocks for which code is developed and integrated to form the completed software system. By decomposing the software system into small modular elements, the software is made easier to understand, develop, maintain, and modify. Development of units may also be easily tracked and responsibilities for development easily defined, thus increasing management visibility into the low-level development process.

Initially, the software units are abstract functional entities. Designers begin with functional decomposition and proceed to more detailed functional definition, but they must also design the control architecture

along with interfaces and data requirements. Software units must be identified to manage the sequence of actions performed by the processing units. Interfaces among units must be identified. The flow of control and of data through the software structure must be documented. Preliminary design is completed when the total software structure is decomposed into software units that will perform the routine algorithmic and data processing functions necessary to implement all of the input-to-output paths of the requirements.

1.4.3 DETAILED DESIGN

During preliminary design, software developers determine what software units will do and how they relate to each other. Criteria for selecting units, however, are subjective because the unit has not yet been built. During detailed design, developers design the logic of each software unit. This includes calling sequences, error exits, inputs, outputs, algorithms, and processing flow. As they design the logic, developers may find that further decomposition is needed. Lines of code estimates may have increased past the threshold of acceptable size, or complexity may have increased more than anticipated.

To allow for further decomposition during detailed design, this manual allows units to be broken into several routines. Routines are the smallest compilable or interpretable software entities. They are the building blocks of units. The detailed design of each unit, including routines that make up units, is documented in the Software Program Specification.

The result of the decomposition process from computer programs to computer program components to units to routines is a rigorous hierarchy of definitions schematically depicted in Figures 1-2, 1-3, 1-4, and 1-5.

1.4.4 REVIEWS

The Software Specifications defined in Part III of this manual assume a software acquisition process that calls for Reviews at the completion of the three major steps:

- a. Software Requirements Review - Definition of deliverable software entities and allocation of system functions to computer programs.
- b. Preliminary Design Review - Specification of the complete structure of all deliverable computer programs, including identification of all software units and their relations with each other.
- c. Critical Design Review - Detailed design of all software units including all routines identified during detailed design.

1.4.5 SUMMARY

At the first milestone, software is designed to the computer program level and is documented in the Software Requirements Specification. The second milestone defines the computer program components and identifies the

units in the software structure (but not the detailed design of the various entities comprising the structure). This marks the completion of the preliminary design and is documented in the Software System/Subsystem Specification. The third milestone represents a detailed design of the software structure, including the design of all routines within units, and is documented in the Software Program Specification.

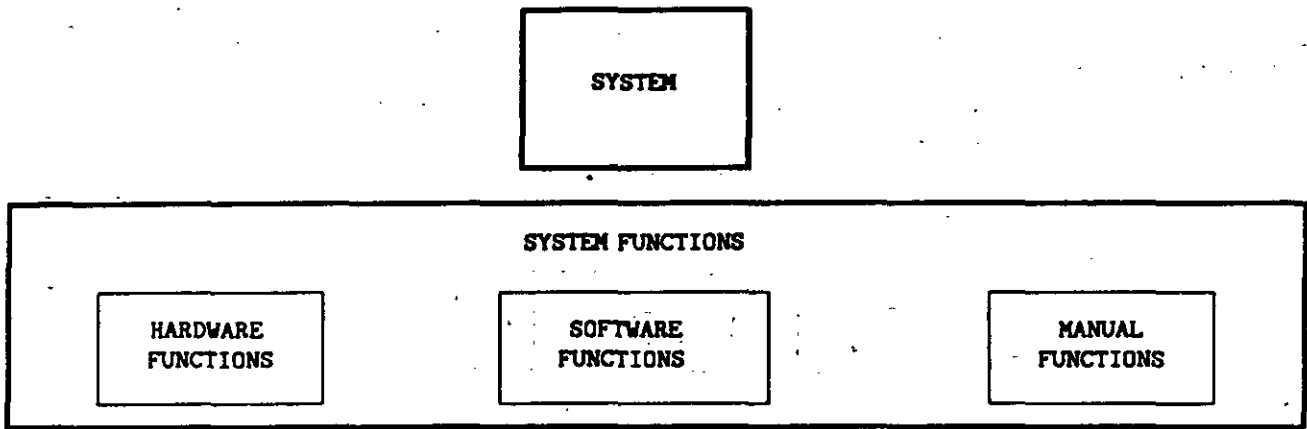


FIG. 1-2 SYSTEM ENGINEERING ACTIVITY

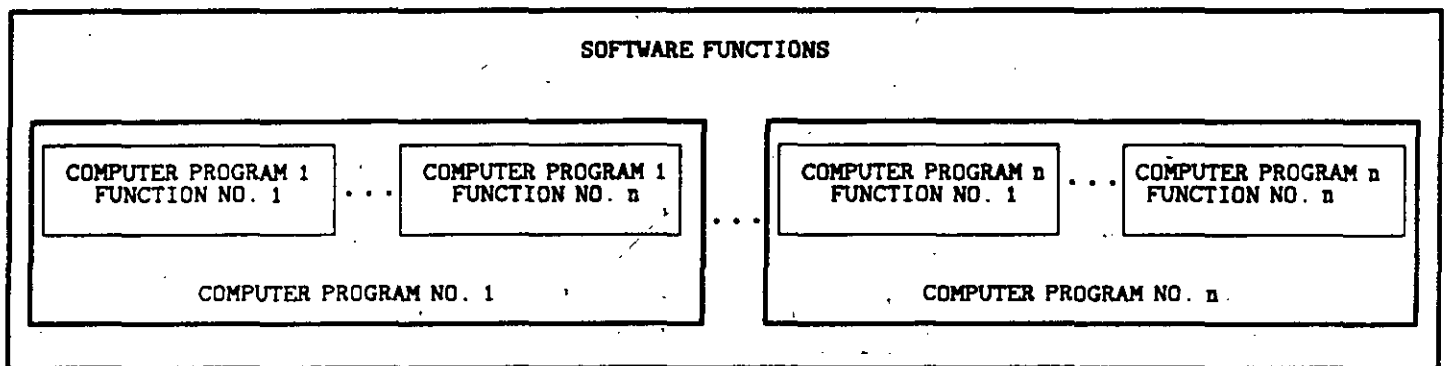


FIG. 1-3 SOFTWARE REQUIREMENTS ANALYSIS ACTIVITY

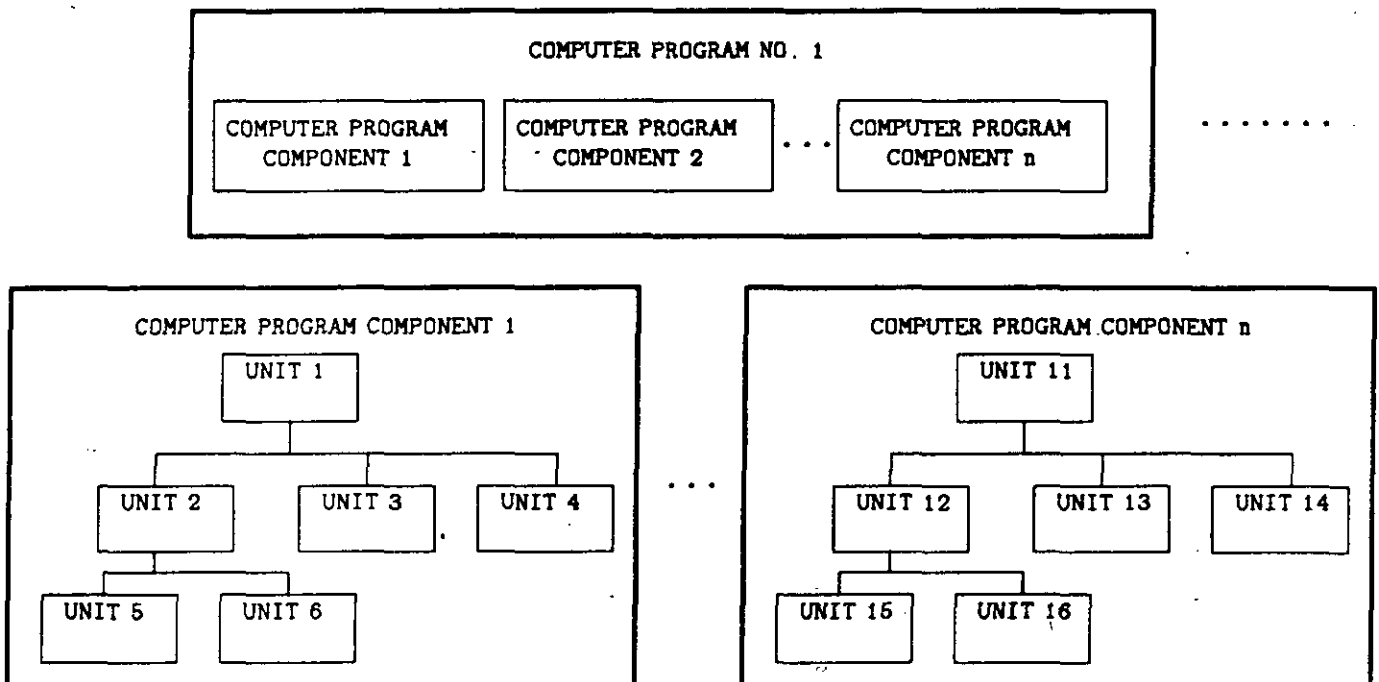


FIG. 1-4 SOFTWARE PRELIMINARY DESIGN ACTIVITY

COMPUTER PROGRAM COMPONENT 1

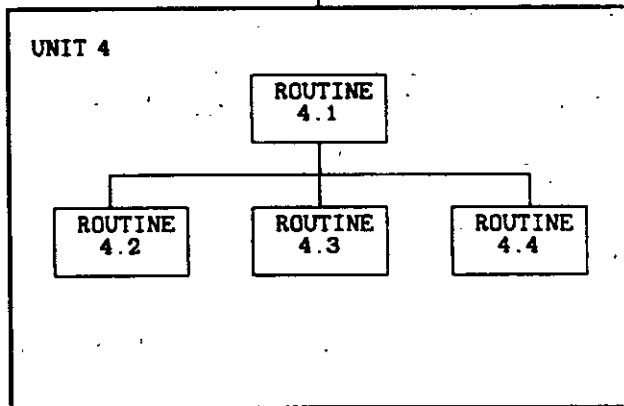
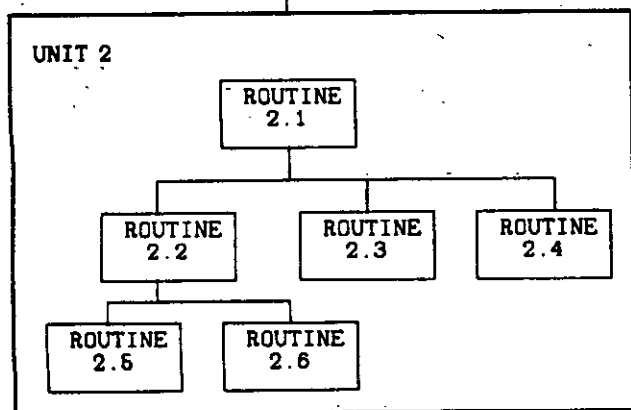
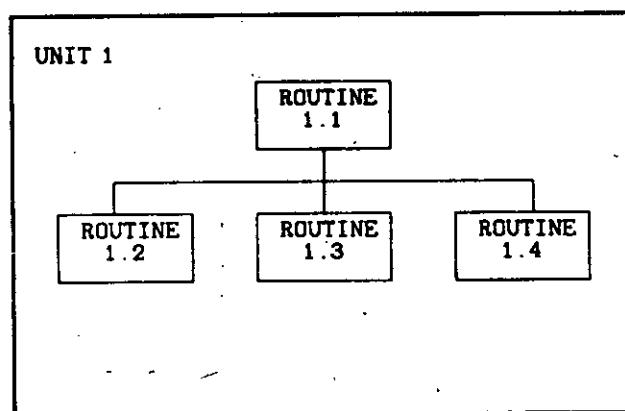


FIG. 1-5 SOFTWARE DETAILED DESIGN ACTIVITY

1.5 REFERENCES

This manual was developed from material contained in a number of reference documents which are listed below:

- a. DoD STD 7935, Department of Defense Standard, Automated Data Systems (ADS) Documentation; dated 15 February 1983.
- b. DoD-STD-2167, Military Standard, Defense System Software Development, dated 4 June 1985.
- c. NSA Manual 81-1, NSA/CSS Systems Acquisition Manual; dated 1 June 1984.
- d. Software Acquisition Management Guidebooks, Electronics Systems Division, Hanscom AFB, Massachusetts, published between 1975 and 1978.
- e. NSA Manual 81-3 (original version), NSA/CSS Software Product Standards Manual; dated 26 July 1979.

1.6 LIST OF DATA ITEM DESCRIPTIONS

This section identifies the Data Item Descriptions that apply to documents described in this manual. When this manual is used for contracted software acquisition, it should be referred to as DOD-STD-1703(NS), Software Products Standards. The term "Data Item Description" is used to refer to a formal collection of information (data) acquired during the contract software acquisition process to support the management and technical objectives of the project. Such collections of information are identified in the Contract Data Requirements List (DD 1423). The specific content and organization of each is defined in the following Data Item Descriptions:

DI-MCCR-80297	Software Development Plan
DI-MCCR-80298	Software Standards and Practices Manual
DI-MCCR-80300	Software Configuration Management Plan
DI-MCCR-80299	Software Quality Assurance Plan
DI-MCCR-80319	Software End-Product Acceptance Plan
DI-MCCR-80301	Software Requirements Specification
DI-MCCR-80302	Software System/Subsystem Specification
DI-MCCR-80303	Interface Control Document
DI-MCCR-80304	Software Program Specification
DI-MCCR-80305	Data Dictionary Document
DI-MCCR-80306	Unit Development Folders
DI-MCCR-80307	General Unit Test Plan
DI-MCCR-80308	Software System Integration and Test Plan
DI-MCCR-80309	Software System Development Test and Evaluation (DT&E) Plan
DI-MCCR-80310	Software Test Procedures
DI-MCCR-80311	Software Test Report
DI-MCCR-80312	Build Description Document
DI-MCCR-80313	User's Manual
DI-MCCR-80314	Software Systems Users Manual
DI-MCCR-80315	Positional Handbooks
DI-MCCR-80316	Computer Operation Manual
DI-MCCR-80317	Program Maintenance Manual
DI-MCCR-80318	Firmware Support Manual

PART II - SOFTWARE PRODUCT MANAGEMENT

2.1 SOFTWARE DEVELOPMENT PLAN

2.1.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 2.1).

Software projects shall be developed according to a Software Development Plan. The plan shall be prepared by the developer prior to the Software Requirements Review. This plan will provide a statement of the developer's plans for producing and controlling the development of software.

2.1.2 GUIDANCE

2.1.2.1 GENERAL

A primary function of the Software Development Plan is to ensure that organizations developing software consider both the technical and management aspects of software development. It also provides a means of improving visibility into the software development process. It should identify the products to be developed; development resources required; schedules for formal reviews, internal developer milestones, and deliveries of products; organizational relationships; and reporting and control procedures.

A Software Development Plan should be prepared by each organization developing software for a system. When more than one organization develops software, the System Acquisition Organization should prepare a higher-level Software Development Plan to consolidate and integrate software development and testing activities. This should help to coordinate interrelated activities of different developers and also aid in management of support facilities and products delivered by one developer for subsequent use or integration by another.

For software developed on contract, the Software Development Plan may be used as a source selection criterion. This requires that the Plan be identified in the Request for Proposal as a specific part of the contractor's proposal. This emphasizes the importance of software and its proper development and management throughout the system acquisition phase. It also establishes the framework for the Software Development Plan to become a contractual document. When it becomes a contractual document, the contractor is obligated to follow the procedures, controls and methods described in it.

2.1.2.2 INCREMENTAL DEVELOPMENT

After Preliminary Design, there are two ways to continue software development: through a single development effort and a single delivery of the total software capability; or by sub-sets of the software's functional capability with separate, sequential (or near sequential) development efforts for each subset. Some call the first method turnkey system development; others call it the "big bang" approach. The second method is called incremental development or the build approach.

There are advantages and disadvantages to either technique. Incremental development allows software developers to demonstrate and deliver limited functional capabilities much sooner than turnkey development. Users are able to use the system sooner and provide feedback to the developers. Developers can learn from their experiences on early builds and apply what they have learned to later builds. Incremental development also requires that the software be tested sooner, longer, and more thoroughly.

Software systems that are developed in increments may take longer to complete, but they can be developed with fewer people. Planning is more complicated because the order in which functional capabilities are selected for implementation is more critical. Fewer people are needed because more time is available for them to complete the same number of development tasks.

If planning for turnkey development is easier than planning for incremental development, managing the implementation is much more difficult. Schedules are shorter, more tasks have to be developed concurrently, more people are needed to complete work in less time, and there is less margin for error during development. Thus, problems are more likely to seriously affect the development schedule.

The point is not to argue the advantages and disadvantages of either development approach. Whether incremental development actually takes longer than turnkey development is debatable. Because incremental development allows limited functional capabilities to be used sooner and tested more thoroughly, the total system, when completed, should be more mature and reliable because it has fewer remaining defects. Whether there are formal intermediate deliveries or not, any large system should be developed in increments.

For either type of development approach, software managers should decide early in the development effort which approach their projects will follow. In most cases, the advantages of incremental development will outweigh the advantages of turnkey development. Only when the following characteristics are present should turnkey development be selected:

- a. The entire software system can be developed in less than twelve months.
- b. The software cannot practically be broken into builds.
- c. The added cost of developing support software (emulation, simulation, test) to test and integrate builds is more than 20% of the total development cost.
- d. The need for delivery of the total software capability is so great that managers are willing to accept the added risks of turnkey development.

Planning for incremental development should begin during the Requirements Definition step and continue during Preliminary Design. As soon as the decision is made to build the software in increments, developers should begin to identify the builds and plan for their implementation. They begin by deciding which subsets of functional capability can be grouped into builds.

A build may consist of complete computer programs (as defined in the Software Requirements Specification) or it may cut across several computer programs. From the standpoint of management control, it is better that complete computer programs be selected for builds; but other factors must also be considered. High-priority user requirements should be allocated to early builds. High-risk, but well understood requirements should also be satisfied in early builds. Software that interfaces with other systems, other devices, or communications lines should be scheduled in builds that will be completed close to the time that the external interfaces are available. Poorly defined or less well-understood requirements should be implemented in later builds. Functions in a build should also be compatible with each other.

Builds therefore can and should be tentatively selected during the Requirements Definition phase, but they should not be formally established until the Preliminary Design is complete. This is because builds should satisfy several additional criteria that will not be known until the software architecture is established. Builds should represent logical divisions of the software architecture. Interfaces between builds should be kept simple. Once a build is implemented, changes to existing build software should be kept to a minimum. A common problem of incremental development is caused when it is belatedly discovered that software in an early build is not compatible with a later build. This problem can be minimized by considering the build software as an integral part of the complete software architecture established during preliminary design. Interfaces between builds should be examined and localized in as few software units as possible. Complete software requirements should always be implemented in individual builds.

After Preliminary Design Review, build implementation can begin. Separate Critical Design Reviews for each build should be conducted as the project is now broken into several smaller projects. Schedules for each build have to be closely coordinated. A build should take from four to six months from beginning of detailed design through completion of Build DT&E.

2.1.2.3 SOFTWARE STANDARDS AND PRACTICES MANUAL

The Software Standards and Practices Manual establishes the standards, procedures, guidelines, and restrictions that software developers will follow to develop software for a project. It tells developers the things they are supposed to do and how they are supposed to do them. Every project member, as well as members of the acquisition team, should have a copy.

Rather than establish detailed standards and practices for all development activities at the beginning of a project, software developers should plan to provide particular sections of the manual as they are needed. By Critical Design Review, all sections should be complete. A three-ring binder will make additions and updates easy.

The Software Development Manager should assign someone, such as the Software Quality Assurance Manager or the Lead Software Engineer, responsibility for the project's standards and practices. This person should be responsible for maintaining the manual, for assuring that project members know how to follow the rules, and for ensuring that the rules are being followed.

2.1.3 FORMAT FOR THE SOFTWARE DEVELOPMENT PLAN

SOFTWARE DEVELOPMENT PLAN

TABLE OF CONTENTS

Section 1.	Introduction
Section 2.	Organization and Responsibility
Section 3.	Management and Technical Controls
Section 4.	Resources
4.1	Personnel
4.2	Training
4.3	Data Processing Equipment
Section 5.	Software Development Schedule
Section 6.	Risk Areas
Section 7.	Monitoring and Reporting
Section 8.	Documentation
Section 9.	Development Approach
Section 10.	Use of Existing Software
10.1	Commercially-Available Software
10.2	Existing Applications Software
Section 11.	Development and Test Tools
Section 12.	Security Controls and Requirements

Section 1. Introduction. This section shall describe the scope, purpose, application, and authority of the development effort. This should include a brief overview of the management philosophy and methodology that will be used on the project.

Section 2. Organization and Responsibility. This section shall describe the organization, responsibilities, and structure of the groups that will be designing, producing, and testing all segments of the software system. It shall also identify the name and management position of each supervisor.

Section 3. Management and Technical Controls. This section shall describe the management and technical controls that will be used during development, including controls for insuring that all performance and design requirements have been identified and implemented.

Section 4. Resources.

4.1 Personnel. This section shall identify the level of manpower that will be allocated to each task shown in the development schedule, including numbers, duration of assignment, and required skills. This includes administrative and logistic support personnel. It shall also include a description of the basis for estimating required personnel resources, including data from parametric estimating models.

When known, personnel assigned to software development tasks shall be listed by name. This section shall also identify security clearance requirements and plans for obtaining the necessary security clearances for personnel working on the software system (if applicable).

4.2 Training. This section shall identify training required for people working on the project and dates by which the training must be completed.

4.3 Data Processing Equipment. This section shall identify requirements for the use of data processing equipment to support the development of computer programs and their subsequent testing. It shall also describe the plan for assuring that the necessary hardware is available at the appropriate times.

Section 5. Software Development Schedule. This section shall present a graphic and narrative description of the scheduled events and milestones of the software development effort. The description shall explain the methods used to develop the schedule and include data used in parametric estimating models. The schedule shall be updated to reflect additional detail as the

project moves through successive phases of the development cycle. By Preliminary Design Review, this section shall include a development schedule for each computer program and data base. The graphic description shall be a chart identifying schedules for the following:

- a. all deliverables;
- b. preparation of management and test plans;
- c. all levels of testing;
- d. reviews, including major reviews and other internal milestones;
- e. transition to life-cycle support activity.

The chart should illustrate a relationship with project schedules. Critical paths shall also be identified.

Section 6. Risk Areas. This section shall identify any high risk areas or issues in the software development effort. It shall also describe actions that will be taken to minimize the risks.

Section 7. Monitoring and Reporting. This section shall describe the procedure for measuring and reporting the status of program development. It shall also describe the manner in which problems, risk areas, and recommended solutions to problems will be reported.

Section 8. Documentation. This section shall describe the approach for developing computer program documentation and will identify the documentation that will be produced. This shall include the plan for developing test-planning documentation, the Software Requirements Specification, the System/Subsystem Specification, the Program Specification, Software Manuals, and any other documentation.

Section 9. Development Approach. This section shall describe the approach that will be taken to design and implement the software system. It shall identify plans for prototyping and describe how the prototyping efforts fit into the overall development effort. It shall also identify whether the software will be developed as a complete system or whether it will be developed in increments and explain the rationale for the selected approach. If the software is to be developed in increments (builds), the method, plan, and schedule for selecting deliverable increments shall be described.

Section 10. Use of Existing Software.

10.1 Commercially-available Software. This section shall describe plans for using commercially-available software to satisfy requirements of the system. It shall identify each software package and provide the rationale for its use. For each package, it shall also identify the following:

- a. data rights;
- b. documentation that will be provided;
- c. plans for certification.

10.2 Existing Applications Software. This section shall describe plans for using existing applications software to satisfy requirements of the system. It shall also identify who owns the software and describe plans for acquiring it. The description shall also include the following information:

- a. data rights;
- b. documentation that will be provided;
- c. plans for determining whether the software performs as expected.

Section 11. Development and Test Tools. This section shall identify the special tools and techniques that will be used during development and testing of the computer programs. Some examples are as follows:

- a. Special simulation;
- b. Data reduction;
- c. Code optimizers;
- d. Code auditors;
- e. Special utility programs;
- f. Software security test tools.

Section 12. Security Controls and Requirements. This section shall identify security controls that will be used during software development (e.g., physical security, document access controls, computer access controls, etc.). It shall also describe the method of implementing and maintaining the security controls. It shall also identify any unique security problems and installation security requirements.

2.1.4 FORMAT FOR THE SOFTWARE STANDARDS AND PRACTICES MANUAL

SOFTWARE STANDARDS AND PRACTICES MANUAL

TABLE OF CONTENTS

Section 1.	Scope
Section 2.	Introduction
Section 3.	Requirements Analysis
Section 4.	Design
Section 5.	Data Base Definition and Control
Section 6.	Implementation
Section 7.	Inspections
Section 8.	Program Support Library

Section 1. Scope. This section shall describe the purpose of the Software Standards and Practices Manual. It shall also identify the software project to which it applies and the person responsible for the manual.

Section 2. Introduction. This section shall contain a summary of the contents of this manual. It shall also identify when and where particular standards are applicable.

Section 3. Requirements Analysis. This section shall specify the analysis techniques and any supporting tools that will be used to define the requirements of the software. It shall also prescribe rules and conventions for applying the techniques and tools.

Section 4. Design. This section shall describe the project's software design methodology. It shall prescribe standards for design representation, procedures for use of a Program Design Language (if applicable), and other rules and conventions for using the design methodology.

Section 5. Data Base Definition and Control. This section shall identify standards and conventions for the definition, design, management, and control of data bases.

Section 6. Implementation. This section shall identify standards and practices to be followed during the code and unit test phases of the project. As a minimum, it shall include standards and conventions for the following:

- 6.1 Interface Definition and Control
- 6.2 Naming Conventions
- 6.3 Unit and Routine Construction Standards
- 6.4 Coding Standards
- 6.5 Prologue and Comment Standards
- 6.6 Unit Development Folders
- 6.7 Unit Testing

Section 7. Inspections. This section shall prescribe procedures and rules for conducting design and code inspections.

Section 8. Program Support Library. This section shall describe the functions of the Program Support Library and identify how it will be used to manage and control the flow of data in the project.

2.2 SOFTWARE QUALITY ASSURANCE

2.2.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 2.3)

Software developers shall develop and implement an independent Quality Assurance program to assure that the prescribed software product standards and individual project quality requirements are met during the software development process. This program shall be described in a Quality Assurance Plan and shall provide for detection, reporting, analysis, and correction of project deficiencies. The degree of formality and control employed should be influenced by the quality specification in the contract, the size and complexity of the project, the significance of the product, and the development risks.

The Quality Assurance Plan shall state the quality objectives of the project as conditioned by the product requirements and the significance of the intended application. This Plan shall be prepared by the developer before the Software Requirements Review.

2.2.2 GUIDANCE

2.2.2.1 GENERAL

A primary role of the Software Quality Assurance Program is to identify problems before they become serious. Problems may be defects in software products or deficiencies in processes that affect the development of software products. A second function is to assure that defects and deficiencies are corrected. A third function is to provide education.

There are many ways to perform these functions. Large or medium-sized, high-risk projects may require a full-time Software Quality Assurance Manager and a supporting staff to perform the required functions. Managers of smaller, lower-risk projects may use different means to accomplish the quality assurance functions. The Software Development Manager and other senior project members may assume responsibilities that a full-time Software Quality Assurance Manager would normally have. Internal reviews, audits, walkthroughs, and inspections may take on added significance.

The important point is that someone must perform the three functions identified in the first paragraph. Standards and plans must be established and followed; project members must understand the standards and plans; someone must have responsibility for discovering defects and deficiencies; and there must be a procedure for tracking the resolution of problems. If these things are not done, it can almost be guaranteed that the software project will not be successful and that its products will not be of high quality.

2.2.2.2 DETERMINING THE SIZE OF THE SOFTWARE QUALITY ASSURANCE EFFORT*

Several approaches are useful in determining the need for an increased or decreased software Quality Assurance effort. One method is to perform a risk

analysis of the impact of the software on the overall system. Whenever risk is great, an intensive software quality assurance program is warranted. Another approach is to analyze the types and thoroughness of software testing available on a project. It is often difficult to design a thorough and realistic test program which provides confidence that the software will perform properly in its system operational environment. That is particularly true when multiple capabilities are to be tested or when complicated interactive environments are required. Other risk factors which serve as criteria for increased emphasis on the Quality Assurance program include:

- a. Complexity of software applications (e.g., real-time constraints, complex algorithms, multiple processes).
- b. Amount of software; potential for error increases greatly with size.
- c. Instability of requirements.
- d. Uniqueness of application; i.e., has this ever been done before?
- e. Lack of experienced personnel.
- f. Rushed development schedules.
- g. Mission criticality of software.
- h. Lack of interface confidence; i.e., are the interfaces with computer programs and other system components incompletely defined or likely to change?
- i. Instability or unavailability of computer hardware and support software.
- j. Unsuitability of the computer and the programming language to the application.
- k. Unavailability of a realistic test environment.

If the Quality Assurance effort must be limited, emphasis should be placed on the quality of the development specifications and of the interface definitions between computer programs and other system components.

*This section was adapted from the Air Force Guidebook, "Software Quality Assurance," written by George Neil and Harvey Gold of System Development Corporation, for the Electronic System Division, Hanscom Air Force Base, Massachusetts.

2.2.3 FORMAT FOR THE SOFTWARE QUALITY ASSURANCE PLAN

SOFTWARE QUALITY ASSURANCE PLAN

TABLE OF CONTENTS

Section 1. Introduction

- 1.1 Purpose
- 1.2 Scope
- 1.3 Applicable Documents
 - 1.3.1 Customer Documents
 - 1.3.2 Developer Documents
 - 1.3.3 Project-Specific Documents
- 1.4 QA Plan Maintenance
- 1.5 Project Software Development Cycle

Section 2. Quality Assurance Organization

- 2.1 QA Operational Responsibilities
- 2.2 QA Program Responsibilities
- 2.3 QA Reports

Section 3. Quality Assurance Functions

- 3.1 Quality Standards and Procedures
- 3.2 Audits
- 3.3 QA Participation in Reviews, Audits, Control Boards
- 3.4 Test Monitoring
- 3.5 Discrepancy Control Monitoring and Review
- 3.6 Tools, Techniques, and Methodologies

Section 4. Quality Assurance Application Areas

- 4.1 Work Tasking and Authorization
- 4.2 Configuration Management
- 4.3 Testing
- 4.4 Computer Program Design
- 4.5 Computer Program Development
- 4.6 Software Documentation
- 4.7 Library Controls

Section 1. Introduction

1.1 Purpose. This paragraph shall state the purpose of the plan and its relation to other plans. It shall define who in the project is responsible for tasks affecting software quality. It shall describe the relationship between the project QA organization and the development and test organizations.

1.2 Scope. This paragraph shall define the specific scope and range of QA activities. It shall describe the quality objectives of the project, explain how the objectives relate to software end-product requirements, and establish the relative priorities of the quality objectives.

1.3 Applicable Documents. This section shall identify the documents that apply to the observance of the project's QA activities. These documents may include contractual requirements, customer specifications and standards, developer specifications and standards, military specifications and standards, etc.

1.3.1 Customer Documents

1.3.2 Developer Documents

1.3.3 Project-Specific Documents

1.4 QA Plan Maintenance. This paragraph shall describe the procedures for updating the QA Plan and keeping it current.

1.5 Project Software Development Cycle. This paragraph shall describe or reference a document (e.g., Software Development Plan) that describes the project's development cycle, including: activity phases, baseline events, audits, reviews, deployment, etc.

Section 2. Quality Assurance Organization

2.1 QA Operational Responsibilities. This paragraph shall identify the person responsible for the QA program described in this plan. It shall describe the person's operational relationship to the Software Development Manager.

2.2 QA Program Responsibilities. This section shall describe the major quality-related responsibilities of each organization whose functions affect product quality (e.g., configuration management, test).

2.3 QA Reports. This section shall identify the types of QA reports that will be prepared, including their frequency and to whom the reports will be given. It shall also identify how the resolution of problem areas will be assured.

Section 3. Quality Assurance Functions. This introductory section shall summarize the software QA approach for assuring that the quality objectives defined in Section 1.2 will be achieved. Each subsection should state how QA will help to achieve the quality objectives.

3.1. Quality Standards and Procedures. This section shall identify the quality standards and procedures that the developer will prepare and maintain and QA will monitor. Standards may include design standards, programming standards, and documentation standards.

3.2 Audits. This section shall describe or reference the procedures for preparation and execution of reviews and audits at key points in the development cycle. It shall identify the QA measures to be employed to ensure that the reviews and audits are conducted in accordance with the prescribed procedures. It shall also describe the approach for reporting the results of QA audits and the approach for responding to QA recommendations resulting from the audits.

3.3 QA Participation in Reviews, Audits, Control Boards. This section shall describe the planned participation of QA personnel in formal project reviews, audits, and control boards. It shall also define the role of QA personnel in activities relating to delivery of documentation and software end items.

3.4 Test Monitoring. The section shall describe plans for independent monitoring of formal test activities by software QA personnel to ensure that test requirements documented in test plans, test procedures, and the Software Requirements Specification are satisfied. It shall also include plans for reviewing test reporting to assure that actual test conditions and results are reported.

3.5 Discrepancy Control Monitoring and Review. This section shall describe how software QA personnel will assure that the software discrepancy (or problem) reporting system supports the software change control process and forms a data base for identifying that problems have been resolved. It shall also describe QA plans for review of discrepancy reports and how recommendations for corrective action will be reported to the Software Development Manager.

3.6 Tools, Techniques, and Methodologies. This section shall identify the tools, techniques, and methodologies that will be employed to support quality objectives and will describe how their use will help to satisfy these objectives.

Section 4. Quality Assurance Application Areas. This section shall explain how the QA functions described in Section 3 will be applied to the project's specific activities and products. For each project activity or project area described, there should be an explanation of how the QA functions will help to achieve quality objectives.

4.1 Work Tasking and Authorization. This section shall describe how software QA personnel will monitor work tasking and authorization procedures to assure that they are being followed. This section shall also describe procedures to track progress of work approved schedules and resource allocations.

4.2 Configuration Management. This section shall describe the QA measures to be applied to Configuration Management activities. It will include procedures to ensure that the objectives of the Configuration Management program are being attained.

4.3 Testing. This section shall identify QA measures relative to software testing. Software testing QA measures shall include:

- a. Analysis of software requirements to determine testability;
- b. Review of test plans and procedures for compliance with appropriate standards and development requirements;
- c. Review of test requirements and criteria for adequacy, feasibility, and satisfaction of requirements;
- d. Monitoring of tests and certification that test results are the actual findings of the tests;
- e. Review and certification of test reports;
- f. Ensuring that test-related documentation is maintained to allow repeatability of tests.

4.4 Computer Program Design. This section shall describe the procedures by which design documentation is reviewed to evaluate the logic of design, fulfillment of requirements, completeness, and compliance with specified standards.

4.5 Computer Program Development. This section shall establish the procedure, frequency, and responsibility for QA auditing of Unit Development Folders to assess developer compliance with project standards and procedures.

4.6 Software Documentation. This section shall state or reference documentation standards to be used for all deliverable software. It shall describe the QA measures to be applied to ensure delivery of correct documentation and change information.

4.7 Library Controls. This section shall identify the QA procedures for reviewing the procedures and controls for handling of source code and their object code from the time of their initial approval or acceptance until they have been incorporated into the final deliverable media.

2.3 SOFTWARE CONFIGURATION MANAGEMENT

2.3.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 2.4)

Software projects shall perform configuration management functions which shall be described in a Software Configuration Management Plan. This Plan shall establish a series of baselines and methods of controlling changes to these baselines. The degree of formality and control employed and manpower used should be conditioned by the size and complexity of the project, the significance of the product, and the development risks.

The Plan shall be prepared and approved by the Software Requirements Review.

2.3.2 GUIDANCE*

Configuration Management is defined in the current Joint Services Regulation, NSA/CSS 80-14, as:

"a discipline applying technical and administrative direction and surveillance to (1) identify and document the functional and physical characteristics of a configuration item, (2) control changes to those characteristics, and (3) record and report change processing and implementation status."

This definition means that configuration management is essentially a support function which interacts closely with (and depends upon the proper conduct of) engineering, design, test, and other management disciplines necessary for acquisition management. These interrelationships impose certain restrictions on the configuration management function which must be considered when developing a configuration management program. Two important restrictions are as follows:

- a. Configuration Management is concerned with system elements (e.g., computer programs, equipment, facilities) which are designated as configuration items;
- b. Configuration Management authority with system elements designated as configuration items is limited to certain special, formalized aspects of their management control.

The disciplines and restrictions historically associated with configuration management were established before software in systems became prominent. Procedures for managing hardware configuration items do not automatically apply to software, and there have been problems in applying hardware configuration management principles to computer programs.

*Material in this section was adapted from "An Air Force Guidebook to Computer Program Configuration Management," dated August 1977. This Guidebook was written by Lloyd Searle of the System Development Corporation for the Electronic Systems Division (ESD) of the Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

Computer programs are system elements which are neither quite the same as nor totally different from other system elements. They are intrinsically items of data and are written, recorded, translated, and reproduced in ways that are characteristic of data as opposed to equipment. Their role as elements of an operational system, however, are more like that of equipments; and there are reasons to manage their development with techniques similar to those used for equipment, particularly with respect to specifications, configuration control, interface control, reviews, and testing. For managing computer program development, therefore, procedures for managing equipment development must be tailored to take into account the unique characteristics of computer programs.

For purposes of managing their acquisition as elements of systems, however, computer programs should be designated as configuration items. The configuration of an item refers to the totality of its functional and physical properties. For equipment and computer programs, these properties are defined and documented in the form of specifications. Thus, specifications serve as the principal documentary instruments for configuration management. For software, NSAM 81-2 requires the preparation of three primary specifications: Software Requirements Specification, System/Subsystem Specification and Program Specification. The actual content of each specification results from the primary engineering efforts of technical analysis, design, and development. Configuration Management control begins by formally establishing a completed specification as an approved and accountable document. Through that action, the configuration described in the specification becomes an explicit point of departure (or baseline configuration) against which changes can be proposed and evaluated.

The process of formulating and implementing changes to an approved specification and its baselined configuration is also primarily a technical engineering effort. Configuration Management controls must be applied during the change process to: (1) assure that each proposed change is evaluated in relation to all relevant technical, schedule, and cost factors; and (2) assure that a change that is approved and implemented is reflected in a corresponding change to the specification. That way, the specification will continue to define the current approved configuration of the system or item.

Another important part of the Configuration Management process is the responsibility of those involved in Configuration Management to inform all participating technical and management activities (both in the System Acquisition Organization and the System Development Organization) of the configuration status of the system. This responsibility is accomplished by disseminating and controlling specifications, change proposals, actual changes, and periodic status reports to all appropriate activities.

The term "baseline management" is often used to describe the process of control described above. The baselines and attendant Configuration Management controls expand in discrete steps as specifications are completed and approved successively at the functional, allocated, and product levels (see definitions of these baselines in the Definitions section).

Before the specifications are approved, however, the technical documentation which leads to the completed specification evolves through many levels, forms, and iterations. Many software managers have adopted the techniques of baseline management and applied them to exercise systematic control over the development process which will eventually lead to the formal configuration control of the approved specification. As applied to the development process, the initial design at each level is documented; each proposed expansion, refinement, or other alteration is examined for its impact; the working documents are modified to reflect all approved refinements; the current status of approved design is made known to all participants; and records are kept to provide an "audit trail" as the design evolves. This sequence is likely to occur on an active and continuing basis as the design of the software system is developed at successively more detailed levels.

Use of the term "Configuration Management" to describe techniques for this type of control during development is common. This is often a source of confusion to software managers involved in a system development effort. Such control measures do constitute management controls; they are in fact dealing with the item's configuration; and some aspects of the controls should involve those with Configuration Management responsibility. To distinguish between Acquisition Management disciplines, however, the primary consideration is that technical managers must control the process until the specification is completed and approved. After the specification is approved, it is controlled by the official Configuration Management discipline described in the Software Configuration Management Plan. The techniques of management control prior to the approval of each specification should be described in the Software Development Plan.

Configuration Management disciplines must be applied to technical documents other than specifications (e.g., plans, manuals). They are also affected by changes that are made to specifications. Software managers responsible for development and test functions normally retain control of those documents, but responsibility for tracking and reporting their status is an integral part of Configuration Management. This is necessary to control the effects on other documents of changes made to specifications.

2.3.3 FORMAT FOR THE SOFTWARE CONFIGURATION MANAGEMENT PLAN

SOFTWARE CONFIGURATION MANAGEMENT PLAN

TABLE OF CONTENTS

- Section 1. Introduction
- Section 2. Organization and Responsibilities
- Section 3. Configuration Identification
 - 3.1 Baselines
 - 3.2 Rules for Configuration Identification
- Section 4. Configuration Control
 - 4.1 Software Configuration Control Procedures
 - 4.2 Storage and Release
- Section 5. Configuration Status Accounting
- Section 6. Associate and Subordinate Developer Configuration Management
- Section 7. Program Implementation Schedule

Section 1. Introduction. This section shall define the scope of the project's software configuration management activities, including the approach used to accomplish software configuration management as described in this plan. It shall also identify policies and directives relating to software configuration management for this project.

Section 2. Organization and Responsibilities. This section shall identify the organizational element that will perform the software configuration management function. It shall describe the responsibilities of the element and describe its interfaces with other organizations within and external to the project, including other configuration management organizations associated with the project and the project management office. It shall also identify the people responsible for the project's configuration management program.

Section 3. Configuration Identification. This section shall define the project baselines and the project configuration identification ground rules.

3.1 Baselines. This section shall identify the baselines to be used by the project, including as a minimum the following three:

- a. Functional (Requirements);
- b. Allocated (Design);
- c. Product ("as-built" software system).

For each baseline, the following shall be defined:

- a. Products (e.g., Software Requirements Specification, System/Subsystem Specification, Program Specification, deliverable computer programs, Interface Control Documents, Data Dictionary Documents, etc.);
- b. Review and approval events;
- c. Method of establishing the baseline;
- d. Contractual significance (if applicable).

3.2 Rules for Configuration Identification. This section shall indicate procedures for selecting and identifying software configuration items and any additional items considered necessary by the developer. This includes the rules for naming, marking, or identifying the items (e.g., design hierarchy names, unit and routine naming, version-identifying designators). In addition, this section shall:

- a. Define and depict (as appropriate) the software entities or levels of software hierarchy;

- b. Describe and depict (as appropriate) the documentation which will be produced and delivered. This description shall relate the software hierarchy and the documentation hierarchy.

Section 4. Configuration Control

4.1 Software Configuration Control Procedures. This section shall identify each configuration control procedure used to maintain, or control changes to, each baselined product. For each control procedure, it shall also describe:

- a. Products subject to this procedure;
- b. Change review and approval sequence;
- c. Review and approval authorities.

This section shall also relate events in the development cycle to the development of software products and show the following for each controlled product:

- a. Period of control;
- b. Degree of control applied during each period;
- c. Delivery events.

4.2 Storage and Release. This section shall describe the method of controlling the storage and release of software master tapes and copies of master documents. It shall also describe the establishment and operation of the project's program development library (or functional equivalent) and show its relation to other software configuration management activities.

Section 5. Configuration Status Accounting. This section shall specify procedures for collecting, recording, processing and maintaining data necessary for producing configuration status accounting reports. This includes the records and reports required to trace changes to controlled products during development. This section shall also describe the problem reporting system and its relationship to the change control process. In addition, this section shall identify the formats and contents of each software configuration management status account record and report.

Section 6. Associate and Subordinate Developer Configuration Management. This section shall describe the system for controlling associate and subordinate developers. It shall describe how associate and subordinate developers will support the requirements of software configuration management.

Section 7. Program Implementation Schedule. This section shall establish the major milestones for implementation of software configuration management. These shall at least include the following:

- a. Establishment of the software configuration control board;
- b. Phasing for program implementation baselines;
- c. Establishment of each of the configuration identifications;
- d. Establishment of interface control agreements with associate developers;
- e. Establishment of status accounting procedures.

PART III - SOFTWARE DEVELOPMENT SPECIFICATIONS

3.1 SOFTWARE REQUIREMENTS SPECIFICATION

3.1.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 3.1)

All system acquisitions in which software is to be acquired shall have a written Software Requirements Specification to provide a controlled statement of the functional, performance, design, and interface requirements for the software end products. If the System Acquisition Plan does not include a complete Software Requirements Specification, the developer shall produce such a specification prior to the Software Requirements Review.

3.1.2 GUIDANCE

3.1.2.1 GENERAL

The Software Requirements Specification identifies the computer software which will be developed to satisfy functional and performance requirements of a system or subsystem. If computer software is developed in several subsystems, a Software Requirements Specification should be prepared for each software subsystem.

The first step in constructing software is to define a set of governing requirements that will satisfy the operational goals of the system. Furthermore, these governing requirements must interpret and reexpress the system operational goals in "shall-statements" which reflect the first steps in design. The system user, who may not be well versed in the best technologies to implement the system, must understand how the goals are to be achieved at this lower level of functional specification. The designer, who may not deeply understand the user's mission, must understand how and why the design is motivated by system operational goals. Requirements-oriented "shall-statements" in the Software Requirements Specification are the best format to document these top-level specifications in terms commonly understood by both users and designers. When this step is completed, the succeeding steps of design and test can take place.

Requirements analysis implies that requirements must exist in some form (preferably written) before analysis can begin. Initially, requirements may be in any of several forms. They can be raw statements of functional capabilities that a user wants; they can be general statements or needs to solve some specific type of problem; or they can be more refined statements contained in a system-level specification or a concept of operations document. The important point is that requirements should not be accepted at face value; they must be analyzed to ensure that the developer knows the answers to the following questions before building the software system:

- a. What has to be done?
- b. Why does it have to be done?
- c. What are alternate ways of doing it?
- d. What functions should be allocated to various parts of the system?
- e. Is it technically feasible to implement each of the functions?
- f. How much will it cost (time, effort, dollars) to satisfy the required functions?
- g. Can tests be constructed to verify that functions and requirements have been satisfied?
- h. Will verification demonstrate that the system will solve the correct problem?

Requirements analysis includes many activities: developing prototypes and computer simulations of the system; conducting trade-off analyses; identifying major risk areas; allocating requirements to computer programs; selecting computer systems and system software; and finally, documenting the results of these analyses in the Software Requirements Specification. First, however, developers should look at the words of requirements to ensure that they express not just what the users say, but what they mean.

Requirements statements should be specific. They should be expressed in quantitative, measurable terms. They should state precisely what the software is required to do and be written in active voice. Too often, words and phrases such as "will," "a minimum of...", "where possible," "should," "as applicable," "flexible," "instantaneous," "timely," and "generally" are used to express requirements. General and imprecise words like these should not be used to document requirements.

After developers and users have a clear understanding of the words of requirements, developers should begin to relate the words to the first steps of system design. Functions need to be decomposed into more detailed requirements. Derived or previously unstated requirements have to be identified and documented. Interfaces and relationships, both internal and external, must be analyzed. Data flows and processing sequences must be identified as must data relationships and data transformation. Through these types of analysis, software developers should go as far into system design as necessary to answer the questions identified earlier. Software requirements can then be documented in the Software Requirements Specification and reviewed at the Software Requirements Review.

The generation of a high quality requirements specification is hard work, and the benefits are not always immediately apparent. Only when the design begins to emerge is it possible to make an accurate judgement of the real worth of the negotiated requirements and of the effort expended to come to that agreement. The benefits, however, are that top-down design will be possible, rigorous testing will be possible, the user will be involved, and management will be in control of the project.

3.1.2.2 IDENTIFICATION AND SELECTION OF COMPUTER PROGRAMS*

1. Identification and selection of computer programs is one of the most important activities required for preparation of the Software Requirements Specification. A computer program is generally understood to be a sequence of coded instructions, including coded values for fixed elements of data, designed to cause an assembly of computing equipment to execute an operation or set of operations. While some uses of the term imply an instruction sequence of limited size or complexity, its use in this manual implies no such limitation. The term refers to any set of instructions (presumably coherent) of whatever size or complexity. A computer program may be very large or very small, depending more on management than on technical considerations. That is, the determination that a given assembly of code constitutes a computer program is based heavily on such factors as source, schedule, and eventual use and control.

2. The identification of computer programs normally occurs during the top-level design of a system. Top-level system design is the process by which the complete set of equipment, computer programs, and facilities elements contemplated for a system are separated into individually-identified subsets. The individually-identified subsets are regarded as a level of management. Specifically it is the level:

- a. at which the System Acquisition Organization accepts individual parts of the system;
- b. below which the developer is responsible for management of the development and assembly of item components;

*Material in this section was adapted from "An Air Force Guide to Computer Program Configuration Management," dated August 1977. This Guidebook was written by Lloyd V. Searle of the System Development Corporation for the Electronic Systems Division (ESD) of the Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

- c. above which the System Acquisition Organization retains responsibility for interfaces, integration, and system performance.

3. The process of selecting and identifying computer programs is not subject to "stylized" rules. Decisions should be based on experience, knowledge of the principles and their implications, knowledge of the system functions, and both technical and administrative considerations. The identification of a given assembly of computer instructions and coded data as a computer program is basically a technical product of the system engineering process. Although accomplished at an early stage of development, it represents a design decision resulting from the steps of: (a) functional analysis and definition of system performance requirements, and (b) system design during which the defined functional and performance requirements are allocated among planned assemblies of system physical elements. Designers must analyze and study computer program design at the system or system segment level to assure that computer programs are technically sound and feasible. At that early stage, the designation of computer programs constitutes a commitment to develop a deliverable end item, e.g., in the form of a tape or a disk pack, which will perform its allocated functions when eventually assembled into the system.

4. Selection of computer programs, however, should not be based solely on technical considerations. A significant responsibility of Software Managers is to plan and direct the technical analysis and design effort in such a way that the proposed computer programs and computer program components meet established criteria for their subsequent management. They are the level at which Software Acquisition Managers exercise formal management control over developers in the areas of configuration management, contracting, program control, and monitoring technical progress. In identifying computer programs, developers should consider that the following activities apply to each computer program in the software system or subsystem:

- a. Preparation of specifications; e.g., Software Requirements Specification, Software System/Subsystem Specification, Program Specification.
- b. Proposed engineering changes and reports of change implementation.
- c. Management information reporting against the work breakdown structure.
- d. Technical reviews; e.g., Software Requirements Review, Preliminary Design Review, Critical Design Review.

- e. The preparation of user manuals.
- f. Formal test programs.
- g. Formal acceptance by the System Acquisition Manager.

5. Criteria listed in the next paragraph for identification and selection of computer programs should be regarded as a "shopping list." The importance and applicability of the considerations vary widely among systems. Because the criteria are not independent of each other, all relevant factors which apply to each computer program should be carefully considered.

6. It is generally desirable to avoid identifying more computer programs than necessary. A productive approach is to begin with the tentative assumption of a single computer program for a system or subsystem. The single computer program should then only be broken down into separate computer programs when fully justified by an applicable criterion as follows:

a. **Separate Computers.** Computer programs to be designed for operation in different types or models of computers must be separate computer programs. Separate computer programs may also be identified when a given installation uses a number of computers of the same type/model, each performing different functions in the system as a whole and having different sets of interfaces with other system elements.

b. **Separate Schedules.** Computer programs scheduled for development, testing, and incremental delivery at different times may be separate computer programs.

c. **Different System Functions and Uses.** In general, separate computer programs should be selected for mission, support, and diagnostic functions. Consider: intended locations of use; expected change cycles; user personnel directly concerned with their functional and performance characteristics; and responsibilities for control after deployment.

7. Although there may not be a single "right" solution, reasonable care and attention to the considerations outlined above should yield sound results. If these important considerations are ignored, the system development process may fail. Computer programs have, at times, been identified on the basis of system functions alone without adequate system design to verify their feasibility or cost-effective development as separate computer programs. It is incorrect to assume that breaking down a complex of data processing functions into many separate computer programs will make the elements more manageable and more visible. Neither size nor visibility is consistent with the accepted criteria for selecting computer programs. While one small computer program is generally easier to manage than one large one, the total management task may

be increased if the large one is artificially broken down into several smaller computer programs. Undesirable results may include: (1) increased paperwork; (2) more difficult interface control; and, (3) increased development time and costs. The result may be that the increased number of computer programs hampers visibility and management control rather than improves it.

3.1.2.3 METHODS OF VERIFYING THE SATISFACTION OF SOFTWARE REQUIREMENTS

Section 5.1 of the Software Requirements Specification identifies four basic methods or techniques for verifying the satisfaction of functional and performance requirements. These are: inspection, review of analytical data, review of test data, and demonstration. These methods may be used individually or be combined with other manual or automated techniques. They may be applicable to unit testing, integration testing or software system testing. The four methods are explained as follows:

a. **INSPECTION** - Formal verification of compliance with a performance or design requirement by examination of the compiler listing of the computer program and its design documentation at the time and place of formal testing. Inspection is the principal method by which design requirements specified in Section 4.6 are verified. It may also apply to selected requirements in other areas such as: verification of adherence to adopted standards for software development (e.g., coding conventions, formats, etc.; verification of data base requirements by comparing data base documentation with a system tape listing). Inspection is not often specified as a formal means of verification for a requirement.

b. **ANALYSIS** - Formal verification by examination and study of the computer program design and coding. This method includes comparison of program output data with data derived by independent computation through one of the following processes:

- (1) Independent but similar programs;
- (2) Hand calculations using the algorithms identified in software specifications;
- (3) Hand calculations using alternate equations or methods equivalent to those specified in the program being tested.

Analysis is most appropriate for programs that are primarily computational in nature, such as orbit determination programs or coordinate conversion equations. This method is typically tedious and time-consuming.

c. **DEMONSTRATION** - Formal verification consists of tests performed in a specified environment in which predesignated inputs produce known and predictable outputs that can be readily observed. This is the basic method by which satisfaction of requirements is demonstrated. Examples are: verification that displays are in the format necessary to satisfy human performance requirements; ability of the system to accept specified inputs; and performance of specified control actions. Verification is accomplished at the time and place of the demonstration (i.e., test).

d. **REVIEW OF TEST DATA** - Formal verification by examining data outputs to determine that they are generated in the proper format and at the appropriate point in the data processing. This method may apply to requirements which depend on a series of tests over more than one test occasion or under varied conditions of operation. Appropriate techniques for this method include the following:

(1) **Internal Interface Verification** - Tests that demonstrate proper data flow between routines, units, components, and the data base.

(2) **Output Verification** - Detailed auditing of output data to determine that outputs are in proper format and have correct values.

(3) **Logical Path Analysis** - Tests to indicate that the correct path is taken through the code in response to the input values specified.

(4) **Timing verification** - Test conducted to show that the computer program executes within the allowable time span. Verification is typically accomplished by review of detailed test procedures, including input data, and hardcopy printouts of test outputs. This is a common method for verification of requirements.

3.1.2.4 COMPUTER SECURITY REQUIREMENTS

Computer programs that handle DoD information (unclassified, classified, sensitive compartmented) must correctly perform their specified functions and do nothing else. Functional requirements for each computer program are identified in Section 4.2 of the Software Requirements Specification.

In addition, computer programs may have to meet special computer security requirements. For information on computer security requirements, consult the following documents:

a. CSC-STD-001-83, Department of Defense Trusted Computer System Evaluation Criteria; dated 15 August 1983

b. CSC-STD-002-85, Department of Defense Password Management Guideline; dated 12 April 1985

c. CSC-STD-003-85, Guidance for Applying the Department of Defense Trusted System Evaluation Criteria in Specific Environments; dated 25 June 1985

d. CSC-STD-004-85, Technical Rationale Behind CSC-STD-003-85; Computer Security Requirements; dated 25 June 1985

e. NSA/CSS Directive 10-27, Security Requirements for Automated Data Processing (ADP) Systems; dated 29 March 1984

f. Security Policy on Intelligence Information in Automated Systems and Networks; dated 4 January 1983 (formerly DCID 1/16)

These documents should be used for guidance in specifying security requirements and in planning, designing, implementing, and testing computer security features and assurances. Copies of these documents may be obtained from the Information Services Branch (C422). The Requirements, Analysis, and Planning for application Systems Evaluations Division (C23) is available to provide advice and guidance on Computer Security matters.

Computer Security requirements should be documented in Section 4.7 of the Software Requirements Specification.

The method of certifying that each requirement, including computer security requirements, has been satisfied must be identified in Section 5 of the Software Requirements Specification.

3.1.3 FORMAT FOR SOFTWARE REQUIREMENTS SPECIFICATION

SOFTWARE REQUIREMENTS SPECIFICATION

TABLE OF CONTENTS

Section 1.	GENERAL
1.1	Purpose of the Software Requirements Specification
1.2	Project References
1.3	Terms and Abbreviations
Section 2.	SYSTEM SUMMARY
2.1	Background
2.2	Objectives
2.3	System Definition
2.4	System Diagrams
2.5	Computer Program Identification
2.6	Assumptions and Constraints
Section 3.	ENVIRONMENT
3.1	Equipment Environment
3.2	Support Software Environment
3.3	Interfaces
3.3.1	Interface Block Diagram
3.3.2	Interface Definition
3.4	Security and Privacy
Section 4.	DETAILED CHARACTERISTICS AND REQUIREMENTS
4.1	Specific Performance Requirements
4.1.1	Accuracy and Validity
4.1.2	Timing
4.2	Computer Program Functions
4.2.1	Identification of Computer Program No. 1
4.2.1.1	Program Description
4.2.1.2	Detailed Functional and Performance Requirements
4.2.1.2.1	Title of Function
4.2.1.2.X	Title of Function X
4.2.1.3	Special Requirements
4.2.N	Identification of Computer Program No. N
4.3	Inputs-Outputs
4.4	Data Characteristics
4.5	Failure Contingencies
4.6	Design Requirements
4.7	Computer Security Requirements
4.8	Human Performance Requirements

Section 5.	TEST AND QUALIFICATION REQUIREMENTS
5.1	Introduction
5.1.1	Unit Tests
5.1.2	Integration Tests
5.1.3	Software System DT&E
5.1.4	System DT&E
5.2	Test Requirements

Section 6.	NOTES
------------	-------

Section 1. GENERAL

1.1 Purpose of Software Requirements Specification. This paragraph shall describe the purpose of the Software Requirements Specification in the following words, modified when appropriate:

This Software Requirements Specification for (Project Name) (Project Number) is written to provide:

- a. The software requirements to be satisfied which will serve as a basis for mutual understanding between the developer and the customer.
- b. A basis for the development of software system tests.

1.2 Project References. This paragraph shall provide a brief summary of the references applicable to the history and development of the project.

A list of applicable documents* shall be included. At least the following shall be identified by author or source, reference number, title, and security classification:

- a. System Acquisition Plan, Project Request or Contract.
- b. Previously developed technical documentation relating to this project.
- c. Significant correspondence relating to the project to include formal agreements to the requirements contained in the Software Requirements Specification.
- d. Documentation concerning related projects.
- e. Other manuals or documents that constrain or explain technical factors affecting project development.
- f. Standards or reference documentation, such as:
 - (1) Documentation standards and specifications.

*When applicable, specific reference should be made to the provisions of these documents in subsequent sections of the Requirements Specification.

- (2) Programming conventions.
- (3) DoD or Federal standards (data elements, programming languages, etc.).
- (4) Hardware manuals, support system documentation, etc., if necessary for an understanding of the Software Requirements Specification.

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

Section 2. SYSTEM SUMMARY

2.1 Background. Included within this paragraph shall be any information concerning the background of the uses and purposes of the system. Reference should be made to higher-order and parallel systems if needed to enhance the general description. The relationship between the project and other capabilities being developed concurrently shall be described.

2.2 Objectives. This paragraph shall contain a brief summary of the purpose of the proposed system. It shall also include a list of the functional areas of the system. It shall further identify and summarize the content and composition of the Software Requirements Specification. The paragraph shall also describe any anticipated operational changes that will affect the software system and identify any provisions within the system for accommodating them.

2.3 System Definition. This section shall describe (directly or by reference) system engineering documentation which graphically portrays the relationship of the items (hardware and computer programs) to be developed. It shall identify the functional areas of the system, including subsystems, and the individual items that must be developed.

2.4 System Diagrams. This section shall contain (directly or by reference) the system-level functional schematic diagrams. The diagrams shall be produced to the level of detail required to identify all system and subsystem functions, including each computer program and its major functions.

2.5 Computer Program Identification. This section shall contain a list of computer programs which are a part of the system/subsystem configuration described in this Specification. Computer programs shall be assigned unique identifiers and be listed in numerical sequence.

2.6 Assumptions and Constraints. Any assumptions and constraints that will affect development and operation of the system/subsystem shall be discussed in this paragraph. Any limitations affecting the desired capability (including the prediction of expected types of errors) and explicit identification of any desired capabilities that will not be provided by the proposed system/subsystem shall be discussed in this paragraph. Examples of assumptions include organizational actions, budget decisions, operational environment or deployed requirements. Examples of constraints include operational environment, operator skill level, human factors (man/machine interface), budget limitations or development schedules.

Section 3. ENVIRONMENT

3.1 Equipment Environment. This paragraph shall provide a description of the equipment capabilities required for the operation of the system/subsystem. This paragraph will present broad descriptions of the equipment presently available and the characteristics of any required new equipment based on the discussion in Section 3. A guideline for equipment to be described follows:

- a. Processor(s), including number of each on/off-line processor and sizes of internal storage;
- b. Storage media, including number of disk units, tape units, etc.;
- c. Output devices, including number of each on/off-line device;
- d. Input devices, including number of each on/off-line device;
- e. Communications net, including line speeds.

3.2 Support Software Environment. This paragraph shall provide a description of the support software with which the computer programs to be developed must interact. Included will be support software, input and equipment simulators, and test software, if needed. The correct nomenclature, level (version), and documentation references of each such software system, subsystem, and program shall be provided. In addition, the programming language and associated compilers, the operating system, and any Data Management System to be used shall be identified.

3.3 Interfaces. This paragraph shall provide a description of the interfaces with other systems and subsystems.

3.3.1 Interface Block Diagram. This section shall graphically portray all external (other systems) and internal (other subsystems) interfaces with the software system/subsystem. Each interface shall be given an identification (title or label) which will allow it to be identified in related documents and specifications. The diagram shall show the flow of data and of control through the interfaces.

3.3.2 Interface Definition. This section shall describe each interface identified in Section 3.3.1. This description shall include the type of interface, data transfer requirements, and design constraints imposed upon other equipment and computer programs as a result of the design of this software system.

3.4 Security and Privacy. This paragraph shall identify the classified components of the system/subsystem including computer programs, inputs, outputs and data bases. Consideration must be given to the fact that the combination of items of one classification may produce a component of a higher classification. This paragraph shall specify the level of classification of each component. It shall also prescribe any privacy restrictions associated with the data being handled.

Section 4. DETAILED CHARACTERISTICS AND REQUIREMENTS

This section shall define and specify all functional, operational and performance requirements; and all design constraints and standards necessary to ensure proper development and maintenance of the software system/subsystem.

4.1 Specific Performance Requirements. This section shall describe the specific performance requirements to be satisfied by the software system/subsystem. This presentation shall be a statement of requirements, evolved from the system analysis, on which the system design is to be based.* The requirements shall be stated in such a manner that software system/subsystem functions (paragraph 4.2) and tests can be related to them. A quantitative presentation of requirements shall be included, such as maximum allowed time from query to display of data, flexibility for adapting to changing requirements, etc.

4.1.1 Accuracy and Validity. This paragraph shall provide a description of the accuracy requirements placed upon the software system/subsystem. The following items must be considered:

*Anticipated deviations from any of the standards specified by the documents listed in paragraph 1.2 must be specifically indicated.

- a. Accuracy requirements of mathematical calculations.
- b. Accuracy requirements of data.
- c. Accuracy of transmitted data.

4.1.2 Timing. This paragraph shall provide a description of the timing requirements placed on the software system/subsystem. If the customer has not specified timing requirements, this paragraph shall describe the timing assumptions on which software development will be based. The following timing requirements shall be considered:

- a. Throughput time.
- b. Response time to queries and to updates of data files.
- c. Response time of major functions.
- d. Sequential relationship of functions.
- e. Priorities imposed by types of inputs and changes in modes of operation.
- f. Timing requirements for the range of traffic load under varying operating conditions.
- g. Sequencing and interleaving programs and systems (including the requirements for interrupting the operation of a program without loss of data).

4.2 Computer Program Functions. This section shall describe the individual functional requirements of each computer program identified in Section 2.5. Requirements shall be stated in quantitative terms with tolerances where applicable. This description should relate functional requirements to performance requirements and to computer programs that will provide the functions. It shall also describe how the aggregate of these functions satisfies the performance requirements in Section 4.1.

A section shall be included for each operational function, plus special functions such as sequencing control displays, error detection and recovery, input and output control, real-time diagnostics, operational data recording, etc. The description of these functions shall include relative sequencing, options, and other important relationships as appropriate. A subparagraph shall be prepared for each functional requirement.

4.2.1 Identification of Computer Program No. 1 This section shall contain the approved identification, nomenclature and authorized abbreviation for the computer program.

4.2.1.1 Program Description. This paragraph shall include a general description of the computer program and its function within the system/subsystem.

4.2.1.2 Detailed Functional and Performance Requirements.

4.2.1.2.1 Title of Function. The basic paragraph for each function shall describe the functional requirement and show its relationship with other functions. Each unique requirement shall be assigned a unique identifier which may be traced in subsequent project phases to (1) portions of the software design and (2) test cases.

4.2.1.2.X Title of Function X. Provide information as shown in 4.2.1.2.1 for each functional requirement to be satisfied by the computer program.

4.2.1.3 Special Requirements. This paragraph shall contain detailed descriptions of special data processing requirements which are distinguishable from other parts of Section 4.2.1.2. Examples are: instructions for special formats to accommodate testing, recoding or simulation; necessary procedures peculiar to the computer program; recovery requirements; and special security requirements.

4.2.N Identification of Computer Program No. N. Provide information for each computer program identified in Section 2.5, including a subparagraph for each section as identified in Section 4.2.1 and subsequent subparagraphs.

4.3 Inputs-Outputs. This section shall specify either directly or by reference to another part of the Software Requirements Specification the sources and types of input and output information to be used in the software system/subsystem. For inputs, this shall include a description of the information, its source(s) and, in quantitative terms, units of measure, limits and/or ranges of units of measure, and frequency of input information arrival. For outputs, this shall include a description of the information, its destination(s), and, in quantitative terms, units of measure, frequency of output information, etc. Descriptions and layouts or examples of hard-copy reports (routine, situational and exception) as well as graphic or display reports may be included. Inputs and outputs shall be related to the interfaces identified in Section 3.3. When an interactive system is being described, outputs must also be related to the system functions described in paragraph 4.2. When possible, these outputs should be related to computer programs that will produce them.

4.4 Data Characteristics. This paragraph shall specify, in descriptive and quantitative terms, the data requirements which affect the design of the software system/subsystem. It should include information on specific data elements by name and characters, if known. Also discussed shall be dictionaries, tables, and reference files, if applicable. An estimate of total storage requirements for the data and computer programs based on a summation of the requirements should be included. A description of the expected growth of the data and related components should be provided.

4.5 Failure Contingencies. This paragraph shall provide a discussion of the requirements for recovering from failures of the hardware or software system/subsystem, the consequences (in terms of system/subsystem performance) of such failures, and the alternative courses of action that may be taken to satisfy the information requirements. The following failure contingencies shall be included as appropriate:

a. Back-up. A discussion shall be provided of the back-up techniques for ensuring the continued achievement of system functions given in paragraph 4.2. "Back-up" means the redundancy available in the event the primary system element goes down. For example, a back-up technique for a disk output would be a tape output.

b. Fallback. An explanation of the fallback techniques for ensuring the continued satisfaction of the specific requirements of the system shall be provided. "Fallback" means the use of another system or other means to accomplish the system requirements. For example, the fallback technique for an automated system might be manual manipulation and recording of data.

c. Restart. A discussion shall be included of the restart capabilities for ensuring effective and efficient recovery from a temporary problem within the hardware or software systems. "Restart" capability means a program capability to resume execution from a point in the program before the problem occurred, along with appropriate restoration of data.

4.6 Design Requirements. This section shall contain detailed descriptions of design requirements for the software system/subsystem which are clearly distinguishable from other parts of Section 4. These may include, but are not limited to, requirements for:

a. The use of programming standards to assure: compatibility among computer programming components, the production of maintainable and understandable computer software, etc.

b. Program design methodologies such as top-down design, modular design, design considerations which ease modification, etc.

c. Expandability (growth potential) to facilitate modifications and additions to computer programs and features such as program hooks for future extensions.

d. Self-monitoring properties for both active and passive monitoring.

4.7 Computer Security Requirements. This section shall contain special computer security requirements, including requirements for protecting classified information in computer programs.

This section must clearly distinguish between mandatory requirements and design goals or options.

4.8 Human Performance Requirements. This section shall specify human performance/human engineering requirements for the software system/ subsystem (e.g., maximum time for human decision making, terminal characteristics, interface provisions intended to improve human interface to system, maximum display densities of information, clarity requirements for display, etc.). When the software system/subsystem directly supports a larger system, this section shall cite appropriate paragraph(s) of higher-level specifications which establish human performance/human engineering requirements for all system equipment. Requirements peculiar to the software system shall then be referenced on an add and/or delete basis.

Section 5. TEST AND QUALIFICATION REQUIREMENTS

This section shall specify test/qualification requirements, methods of qualification, and the necessary test tools and facilities to conduct the required qualification tests. This section shall establish the requirements for the test plans and procedures that must be formulated for qualification of the software. The intent of the test effort is to verify that the requirements stated in Section 4 of the Software Requirements Specification have been met.

NOTE: This section shall not incorporate detailed test planning documentation and operating instructions. Requirements specified in this section shall be the basis for preparation and validation of other test planning documentation.

5.1 Introduction. This section shall establish the requirements for development of a test plan and test procedures for the entire software system/subsystem and segments of the software system/subsystem. It shall specify which of the following levels of testing must be performed and which must be formally documented:

- a. Unit tests;
- b. Integration tests;
- c. Software system DT&E tests.

This section shall also describe methods of verifying the satisfaction of functional and performance requirements described in Section 4. Methods of qualification may include inspection of the computer program, review of analytic data, demonstration tests, and review of test data. Each method shall be defined. A Qualification Cross-Reference Index (Figure 3-1) identifying test category, qualification methods, requirement reference number and quality assurance reference number shall be included. Narrative information relating to test category shall be identified in subsequent subparagraphs.

5.1.1 Unit Tests. This section shall identify unit tests which are the only source of data to qualify specific requirements in Section 4. Unit tests accomplished in support of design and development which do not satisfy this criterion need not be specified in this section.

5.1.2 Integration Tests. This section shall identify tests which shall be conducted during software integration which require formal recognition by the customer and are oriented to verifying proper performance of the computer program prior to Software System DT&E. Integration tests which are conducted to support design and development need not be identified in this section.

5.1.3 Software System DT&E. This section shall identify requirements for formal qualification tests of the software system to demonstrate and verify that the requirements established in Section 4 have been satisfied, with only two exceptions:

- a. The requirement in Section 4 has been verified and satisfied by one of the tests included in paragraphs 5.1.1 and 5.1.2;
- b. The requirement in Section 4 cannot be demonstrated until System DT&E and will be identified in Section 5.1.4.

5.1.4 System DT&E. This section shall identify requirements specified in Section 4 which cannot be verified until System DT&E testing.

5.2 Test Requirements. This paragraph shall include the particular needs for conducting each type of testing. These shall include test formulas, algorithms, techniques and acceptable tolerance limits as applicable. Any other information needed to define the qualification methods described in Section 5.1 shall also be included.

TEST AND QUALIFICATION CROSS-REFERENCE INDEX

SRS PARAGRAPH REFERENCE	REQUIREMENT NAME (brief description)	REQUIREMENT ID NUMBER	QUALIFICATION METHOD (1)	QUALIFICATION TEST (2) LEVEL	SRS SECTION 5 PARAGRAPH REFERENCE

(1) QUALIFICATION METHOD

- I - INSPECTION
- A - ANALYSIS
- D - DETECTION
- T - REVIEW OF TEST DATA
- O - OTHER (REQUIRES EXPLANATION)

(2) QUALIFICATION TEST LEVEL

- 1. UNIT TEST
- 2. INTEGRATION
- 3. SOFTWARE DT & E
- 4. SYSTEM DT & E
- 5. OTHER (REQUIRES EXPLANATION)

Example of a Test and Qualification Cross-Reference Index

Section 6. NOTES

This section may be used to provide background material that would be of assistance to reviewers in understanding software requirements. It may include such things as Data Flow Diagrams, rationale for the allocation and groupings of requirements into computer programs, and other information which would help in review of the document.

3.2 SOFTWARE SYSTEM/SUBSYSTEM SPECIFICATION

3.2.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 3.3)

Software projects shall have a System/Subsystem Specification for the defined software end products to establish a documented preliminary design baseline from which the detailed design will be developed. The System/Subsystem Specification shall be developed from the Software Requirements Specification and shall contain the design information needed to support the detailed definition of the individual software system components.

The System/Subsystem Specification shall assign each unique requirement in the Software Requirements Specification to specific components and units of the software design and shall explicitly identify the mapping from requirements to software design.

The System/Subsystem Specification shall identify and name all software units in the software hierarchy.

3.2.2 GUIDANCE

3.2.2.1 INTERFACE CONTROL DOCUMENTATION

During the requirements definition phase, software developers should identify external software interface requirements and document them in the Software Requirements Specification. They should identify more detailed interface control information during the preliminary (or architectural) phase of development. This detailed information can be documented in the System/Subsystem Specification or in a separate Interface Control Document.

Although the separate Interface Control Document will not be completed until the preliminary design phase, software developers should decide during the requirements definition phase how the interface control information will be documented. The decision should be based upon the following factors:

- a. Number of external interfaces;
- b. Complexity of the interfaces;
- c. Number of organizations involved in defining and implementing the interfaces;
- d. Volatility of the interfaces (tendency to change).

If any (or all) of these factors is high, interfaces should be documented in a separate Interface Control Document. The plan for documenting detailed interface control information should be described in Section 8 of the Software Development Plan.

3.2.3 FORMAT FOR THE SOFTWARE SYSTEM/SUBSYSTEM SPECIFICATION

SOFTWARE SYSTEM/SUBSYSTEM SPECIFICATION

TABLE OF CONTENTS

Section 1.	GENERAL
1.1	Purpose of the System/Subsystem Specification
1.2	Project References
1.3	Terms and Abbreviations
Section 2.	SUMMARY OF REQUIREMENTS
2.1	System/Subsystem Description
2.2	System/Subsystem Functions
2.2.1	Functional Allocation Description
2.2.2	Functional Requirements Matrix
2.2.3	Accuracy and Validity
2.2.4	Timing
2.3	Flexibility
Section 3.	ENVIRONMENT
3.1	Equipment Environment
3.2	Support Software Environment
3.3	Interfaces
3.3.1	Interface Block Diagram
3.3.2	Software Interfaces
3.3.3	Hardware-to-Software Interfaces
3.4	Security and Privacy
3.5	Storage and Processing Allocation
Section 4.	DESIGN DETAILS
4.1	General Operating Procedures
4.2	System Logical Flow
4.2.1	Program Interrupts
4.2.2	Control of Computer Program Components
4.2.3	Special Control Features
4.3	System Data
4.3.1	Inputs
4.3.2	Outputs
4.3.3	Displays
4.3.3.1	Description of Displays
4.3.3.2	Display Identification
4.4	Program Descriptions
4.4.1	Computer Program Identification
4.4.1.1	Computer Program Component No. 1

4.4.1.1.1	Computer Program Component No. 1 Graphical Representation
4.4.1.1.2	Computer Program Component No. 1 Description
4.4.1.1.3	Computer Program Component No. 1 Interfaces
4.4.1.X	Computer Program No. X
4.4.1.X.1	Computer Program No. X Graphical Representation
4.4.1.X.2	Computer Program No. X Description
4.4.1.X.3	Computer Program No. X Interfaces
4.4.N	Computer Program No. N.
4.5	Data Base

Section 5. TEST AND QUALIFICATION

Section 6. NOTES

Section 1. GENERAL

1.1 Purpose of the System/Subsystem Specification. This paragraph shall describe the purpose of the SSS (System/Subsystem Specification) in the following words, modified when appropriate:

The System/Subsystem Specification for (Project Name) (Project Number) is written to fulfill the following objectives:

- a. To provide detailed definition of the software system/subsystem functions.
- b. To communicate details of the on-going analysis between users, customers, and developers.
- c. To define in detail the interfaces with other systems and subsystems and the facilities to be used for providing the interfaces.

1.2 Project References. This paragraph shall provide a brief summary of the references applicable to the history and development of the project. At least the following shall be specified by source or author, reference number, title, and security classification:

- a. Software Requirements Specification;
- b. Related System/Subsystem Specifications;
- c. Any other pertinent documentation or significant correspondence not specified in the Software Requirements Specification.

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

Section 2. SUMMARY OF REQUIREMENTS

Section 2 of the System/Subsystem Specification shall provide a summary of the characteristics and requirements of the software system/subsystem. This section shall be an expansion of the information published in the Software Requirements Specification to reflect the determination of additional details. The description shall reflect the translation of the computer system functions to the design of the computer programs and their components. Any changes to the characteristics and requirements set forth in Sections 2 and 4 of the Software Requirements Specification must be specifically identified.

2.1 System/Subsystem Description. This paragraph shall provide a general description of the software system/subsystem to establish a frame of reference for the remainder of the document. Higher order and parallel systems/subsystems and their documentation shall be referenced as required to enhance this general description. Included shall be a chart showing the relationship of user organizations to the major elements of the system and a chart showing the interrelationships of the system elements with the software system/subsystem. These charts shall be based on or be updated versions of the charts included in paragraph 2.4 of the Software Requirements Specification.

2.2 System/Subsystem Functions. This section shall describe the software system/subsystem functions and identify where in the software system/subsystem specific functional and performance requirements will be satisfied. There shall be both qualitative and quantitative descriptions of how the functions will satisfy the requirements.

2.2.1 Functional Allocation Description. This section shall describe the overall structure and functions of each computer program in the software system/subsystem down to component level. This identification, which may be in the form of hierarchical diagrams, shall identify and name all computer programs and their components and show their relationship with each other. Components of each computer program identified in Section 2.5 of the Software Requirements Specification shall also be identified in a table. Figure 3-2 is an example of such a table. The more detailed charts to be included in Section 4 shall be based on the charts included in this section.

2.2.2 Functional Requirements Matrix. This section shall trace the allocation of requirements (functional and other) identified in the Software Requirements Specification to computer program components. Each unique requirement shall be assigned to specific components. The name and mnemonic for each computer program and component shall be referenced in the tables. Figure 3-3 is an example of a Functional Requirements Matrix.

2.2.3 Accuracy and Validity. This paragraph shall provide a description of accuracy requirements imposed on the software system/subsystem. The requirements will be related to paragraph 4.1.1 of the Software Requirements Specification. The following accuracy requirements must be considered:

- a. Accuracy requirements of mathematical calculations.
- b. Accuracy requirements of data.
- c. Accuracy of transmitted data.

COMPUTER PROGRAM COMPONENT IDENTIFICATION**COMPUTER PROGRAM IDENTIFICATION**

	COMPONENT NAME	MNEMONIC	SYSTEM / SUBSYSTEM SPECIFICATION PARAGRAPH REFERENCE
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			

Example of a Table That Identifies Computer Program Components

Figure 3-2

2.2.4 Timing. This paragraph shall provide a description of the timing requirements placed on the software system/subsystem. If the customer has not specified timing requirements, this paragraph shall contain the timing assumptions on which software design is based. The requirements will be related to paragraph 4.1.2 of the Software Requirements Specification. The following timing requirements may be considered:

- a. Throughput time.
- b. Response time to queries and to updates of data files.
- c. Response time of major system/subsystem functions.
- d. Sequential relationship of system/subsystem functions.
- e. Priorities imposed by types of inputs and changes in modes of operation.
- f. Timing requirements for the range of traffic load under varying operating conditions.
- g. Sequencing and interleaving programs and systems (including the requirements for interrupting the operation of a program without loss of data).

2.3 Flexibility. This paragraph shall provide a description of the capability to be incorporated for adapting the software system/subsystem to changing requirements, such as anticipated operational changes, interaction with new or improved systems, and planned periodic changes. Components and procedures designed to be subject to change shall be identified.

Section 3. ENVIRONMENT

This section shall expand the description of the environment given in the Software Requirements Specification to reflect additional analysis. Although description of the environment may be more detailed as the result of on-going analysis and design, it must be consistent with the environment described in the Software Requirements Specification. Changes shall be discussed in terms of the impacts on the currently available environmental components (equipment, software, etc.) as well as the impacts on estimates and functions which were based on the original planned environment.

3.1 Equipment Environment. This paragraph shall identify and describe the equipment required for operation of the software system/subsystem. Included shall be descriptions of the equipment presently available as well as a more detailed discussion of the characteristics of any necessary new equipment.

FUNCTIONAL REQUIREMENTS MATRIX**COMPUTER PROGRAM IDENTIFICATION**

SPECIFICATION TYPE							
SOFTWARE REQUIREMENTS			SYSTEM / SUBSYSTEM				
PARAGRAPH REFERENCE	REQUIREMENT NAME (BRIEF DESCRIPTION)	REQUIREMENT ID NUMBER	COMPONENT(S) IN WHICH REQUIREMENT IS SATISFIED				

Example of a Functional Requirements Matrix

Equipment requirements shall be related to the requirements stated in paragraph 3.1 of the Software Requirements Specification. A guideline for equipment to be described follows:

- a. Processor(s), including number of each on/off-line processors and sizes of internal storage;
- b. Storage media, including number of disk units, tape units, etc.;
- c. Input/output devices, including number of each on/off-line I/O device;
- d. Communications net, including line speeds.

3.2 Support Software Environment. This paragraph shall provide a description of the support software with which the computer programs to be developed must interact. Included will be both support software and test software, if needed. The correct nomenclature and documentation references of each such software system, computer program, and program component shall be provided. Included must be a reference to the languages (compiler, assembler, program, query, etc.), the operating system, and any Data Management System (DMS) to be used. This description must relate to and expand on the information provided in paragraph 3.2 of the Software Requirements Specification. If operation of the computer programs to be developed is dependent upon forthcoming changes to support software, the nature, status, and anticipated availability date of such changes must be identified and discussed.

3.3 Interfaces. This section shall specify directly or by reference to separate Interface Control Documents, all external (other systems) and internal (between subsystems) interfaces necessary to ensure proper development of software for the system. Each interface shall identify the flow of data and control in one direction only. If data or control flows in both directions, each direction shall be specified as a separate interface. Optionally, interfaces may be documented in separate Interface Control Documents.

3.3.1 Interface Block Diagram. This section shall graphically portray the interfaces of the computer programs with other equipment and computer programs. Each interface shall be given an identification (title or label) which will allow it to be identified in related documents and specifications. The diagram shall show the data and control flow between the computer(s) and other system equipment. This paragraph shall also incorporate in subparagraphs as appropriate, a functional block diagram or equivalent representation of the external interfaces with each computer program.

3.3.2 Software Interfaces. This section shall specify the following for each software interface identified in Section 3.3.1:

- a. Interface identification (title or label).
- b. Purpose and functional description, including computer programs that interface with each other and software units that implement the interface.
- c. Direction of the flow of data and of the transfer of control.
- d. Formats and volumes of data to be passed and characteristics of the communications medium used for the transfer.
- e. Type of interface, such as manual or automatic.
- f. Interface procedures, including telecommunications considerations.
- g. Priority level of the interface data interrupt (if applicable).
- h. Maximum time allowed for the receiving software element to respond to the interface data interrupt and the effects of not responding within the allocated time.
- i. Design requirements imposed upon other computer programs as a result of the design of the interface.

3.3.3 Hardware-to-Software Interfaces. This section shall specify the following for each hardware-to-software interface identified in Section 3.3.1:

- a. Interface identification (title or label).
- b. Functional description, including the computer program and its software units that implement the interface.
- c. Direction of the signal.
- d. Format of the signal.
- e. Frequency of the signal interface.
- f. Priority of the signal.
- g. Maximum time allowed for responding to the signal, and the effect of not responding within the allocated time.
- h. Design requirements imposed upon the computer programs as a result of the design of the interface.

3.4 Security and Privacy. This paragraph shall describe the classified components of the system/subsystem, including computer programs, inputs, outputs, and data bases. These components will be related to paragraph 3.4 of the Software Requirements Specification. It shall also prescribe any privacy restrictions associated with the data being handled.

3.5 Storage and Processing Allocation. This paragraph shall describe the allocation of memory storage and processing time to computer program components, the executive routine, and the data base. In addition, the timing, sequencing requirements, and equipment constraints used in determining the allocation shall be described. The allocations must be consistent with the accuracy and validity and timing requirements for the system/subsystem described in Sections 2.2.3 and 2.2.4. Allocations in this specification are approximations and are not binding, but they should be made as close as possible to the storage space that will be needed. The total memory allocation and processing time for each computer program component and common data base shall be summarized in a table.

Section 4. DESIGN DETAILS

4.1 General Operating Procedures. This paragraph shall provide a general description of the system operating procedures. There shall be a description of the load, start, stop, recovery, and restart procedures.

4.2 System Logical Flow. This paragraph shall describe the logical flow of both data and control in the software system/subsystem. Logical flow will be presented primarily in the form of functional flow diagrams. A top-level flow diagram shall depict in a single figure the overall information flow of the software system/subsystem. This diagram shall refer to lower-level flow diagrams which shall provide more detailed information. Lower-level diagrams shall identify the computer program components identified in Section 2.2.

The diagrams shall provide an integrated presentation of the system/subsystem dynamics of entrances and exits and interfaces with all computer programs in the system. They will effectively represent all modes of operations, priorities, cycles, and special handling. They will show general flow from input, through the system/subsystem, to the generation of output.

4.2.1 Program Interrupts. This paragraph shall list all program interrupts and describe their effect on designing the control logic. Each interrupt shall be fully described as to source, purpose, type, and the required response of the executive control. The probable rate of occurrence of interrupts shall also be given.

4.2.2 Control of Computer Program Components. This paragraph shall describe how each computer program component obtains control. In narrative form it shall then describe its sequence of processing from initiation until termination. It shall also show details concerning assignment of priorities and cycle times to computer programs and computer program components.

4.2.3 Special Control Features. This paragraph shall describe all special control features which affect the design of the control logic but are not part of the normal operational functions.

4.3 System Data. This section shall describe and state the purpose of the inputs, outputs, data, and displays used in the software system/subsystem. Each description shall include the following information as applicable. Optionally, inputs, outputs, and displays may be described with the individual computer programs and components to which they relate.

4.3.1 Inputs. Each input to the software system shall be described as follows:

- a. Title and tag.
- b. Source.
- c. Format.
- d. Purpose.
- e. Units of measure.
- f. Limits or ranges of input values.
- g. Accuracy and processing requirements.
- h. Volume and frequency of arrival, including special handling (e.g., queuing, priority handling, etc.) for high density periods.
- i. Legality checks for erroneous information.

4.3.2 Outputs. Each output to the software system shall be described as follows:

- a. Title and tag.
- b. Destination.
- c. Format.

- d. Purpose.
- e. Units of measure.
- f. Limits or ranges of output values.
- g. Accuracy and precision requirements.
- h. Volume and frequency of output, including special handling for high-density periods (e.g., queuing, priority handling, etc.).
- i. Legality checks for erroneous information.

4.3.3 Displays

4.3.3.1 Description of Displays. This section shall describe the types of displays to be developed for the user interface, e.g., input screens, help screens, menu screens.

4.3.3.2 Display Identification. This section shall describe each display generated by the software system. For each display, the following information shall be given:

- a. Title and tag.
- b. Full name.
- c. Description of how the display appears to the user.
- d. Purpose of the display and its relationship to the processes in the system.
- e. Format, including data elements and sizes.
- f. Users/privileges, including type of user access-read/write/edit/delete.
- g. Protected areas.
- h. Means of presentation, e.g., CRT, printer, etc.
- i. Picture of the display (actual or mock-up).
- j. Identification of software units interfacing with the display, showing types of access (input/output).

NOTE: Section 4.3.3.2 may be published as an annex to the System/Subsystem Specification.

4.4 Program Descriptions. Paragraphs 4.4.1 through 4.4.N shall provide descriptions of the functions (related to paragraph 2.2 of the System/Subsystem Specification) of the computer programs and computer program components in the system/subsystem. If all computer programs are written in the same language, the introductory paragraph shall include the name of the language used. If all computer programs are not written in the same language, the statement, "(mnemonic name) is written in the (identify language)," shall be included in the introductory statements in 4.4.1 through 4.4.N.

4.4.1 Computer Program Identification. This section shall contain the lead phrase, "This section contains the detailed technical descriptions of the components of the (. NAME) computer program". The following subparagraphs shall be repeated for each component:

4.4.1.1 Computer Program Component No. 1. This paragraph shall identify the component by functional name followed by its mnemonic within parenthesis. Thereafter, the computer program component should be referred to by mnemonic only. It shall also include a brief abstract of the tasks of the component and its major functional interfaces.

4.4.1.1.1 Computer Program Component No. 1: Graphical Representation. This section shall present a graphical representation of the software units in the component. This may be done by structured design charts or equivalent. It shall show the relationships among units and depict the processing described in Section 4.4.1.1.2, including the sequence of operations and decision points in the component. The chart shall depict the overall information and control flow of the computer program component and shall reference the charts in paragraph 4.2 that identify the computer program.

4.4.1.1.2 Computer Program Component No. 1 Description. This section shall describe in words, figures, equations, and references to paragraph 4.4.1.1.1, the operation and design of the computer program component. Software units of the component shall be identified in a table or chart which depicts the allocation of requirements to individual software units. Figure 3-4 is an example of such a table. This section shall also describe the sequential functions of the component (what it does first, second, etc.) and relate them to the component's software units. For long or complex descriptions, the description should refer to graphical representations contained in Section 4.4.1.1.1. It shall also describe program logic and data flow depicted in Section 4.4.1.1.1.

UNITS OF COMPUTER PROGRAM COMPONENTS IDENTIFICATION

COMPUTER PROGRAM			COMPONENT	
	UNIT NAME	MNEMONIC IDENTIFIER	REQUIREMENTS ID / SRS PARAGRAPH REFERENCE	SSS / SPS PARAGRAPH REFERENCE
1.				
2.				
3.				
4.				
5.				
6.				
7.				
8.				
9.				
10.				
11.				
12.				
13.				
14.				

Example of a Table Allocating Requirements to Software Units

Figure 3-4

4.4.1.1.3 Computer Program Component No. 1. Interfaces. This section shall describe the relation of the computer program component to other computer program components, to that part of the data base external to the computer program, and where applicable to other computer programs. It shall also identify the specific units of the component that interface with other components and data bases and describe interface information such as input/output formats, and table, item, file, and buffer descriptions.

4.4.1.X. Computer Program Component No. X.

4.4.1.X.1 Computer Program Component No. X Graphical Representation.

4.4.1.X.2 Computer Program Component No. X Description.

4.4.1.X.3 Computer Program Component No. X Interfaces.

4.4.N Computer Program N. Section 4.4.1 and subparagraphs shall be repeated for each computer program and computer program component in the software system/subsystem.

4.5 Data Base. This paragraph shall include, directly or by reference to a Data Dictionary, a definition of the levels of data hierarchy (i.e., data base, file, record, field) and the content of each incorporated in the system/subsystem data base. Each item shall be described as follows:

- a. Title and tag.
- b. Description and purpose of content.
- c. Number of records or entries.
- d. Storage, to include type of storage, amount of storage and, if known, beginning and ending addresses.
- e. Source of input.
- f. Method of access.
- g. Classification.
- h. Data retention.

Section 5. TEST AND QUALIFICATION

This section shall consist of the following statement:

Detailed Test and Qualification Provisions are contained in the Test Plans and procedures for this software system/subsystem. The Software System DT&E Plan contains a detailed Test Case Matrix which relates each functional requirement in the Software Requirements Specification to the applicable test. The Test Plans and Test Procedures also specify any special test tools and capabilities required to qualify the software system/subsystem.

Section 6. NOTES

This section may be used to provide background material that would be of assistance to reviewers in understanding the design of the computer programs. It may include such things as alternate design solutions, rationale behind the design that is proposed, and other information which would help in review of the document.

3.2.4 FORMAT FOR INTERFACE CONTROL DOCUMENT

INTERFACE CONTROL DOCUMENT

TABLE OF CONTENTS

Section 1.	GENERAL
1.1	Purpose of the Interface Control Document
1.2	Project References
Section 2.	INTERFACES
2.1	Interface Block Diagram
2.2	Software Interfaces
2.3	Hardware-to-Software Interfaces

INTERFACE CONTROL DOCUMENT

SECTION 2 - GENERAL

1.1 Purpose of the Interface Control Document. This paragraph shall describe the purpose of this Interface Control Document in the following words, modified when appropriate:

This Interface Control Document establishes the software interface requirements for (system name).

1.2 Project References. This paragraph shall provide a brief summary of the references applicable to the identification of interfaces to the (system name) software. It shall identify the following by author or source, reference number, title, date, and security classification:

- a. System Acquisition Plan, Project Request, or Contract.
- b. Previously developed technical documents relating to the system's interfaces.
- c. Software Requirements Specification.
- d. DoD or Federal Interface Standards.
- e. Significant correspondence relating to the System's Interfaces.

SECTION 2 - INTERFACES

This section shall specify all of the external (other systems) and internal (between subsystems) interfaces necessary to ensure proper development of software for the system. Each interface shall identify the flow of data and control in one direction only. If data or control flows in both directions, each direction shall be specified as a separate interface.

2.1 Interface Block Diagram. This paragraph shall graphically portray the relationships of the computer programs to other equipments and computer programs with which they must interface. Each interface shall be given an identification (title or label) which will allow it to be identified in related documents and specifications. The diagram shall show the data and control flow between the computer(s) and other system equipment. This paragraph shall also incorporate in subparagraphs as appropriate, a functional block diagram or equivalent representation of the interface requirements of each computer program.

2.2 Software Interfaces. This section shall specify the following for each software-to-software interface identified in Section 2.1:

- a. Interface Identification.
- b. Functional Description.
- c. Direction of the flow of data and of the transfer of control.
- d. Formats and volumes of data to be passed.
- e. Type of interface, such as manual or automatic.
- f. Interface procedures, including telecommunications considerations.
- g. Priority level of the interface data interrupt (if applicable).
- h. Maximum time allowed for the receiving software element to respond to the interface data interrupt and the effects of not responding within the allocated time.
- i. Design requirements imposed upon other computer programs as a result of the design of the interface.

2.3 Hardware-to-Software Interfaces. This section shall specify the following for each hardware-to-software interface identified in Section 2.1:

- a. Interface Identification.
- b. Functional Description.
- c. Direction of the Signal.
- d. Format of the Signal.
- e. Transfer Protocol used for the Signal Interface.
- f. Frequency of the Signal.
- g. Priority of the Signal.
- h. Maximum time allowed for responding to the signals, and the effect of not responding within the allocated time.

- i. Design requirements imposed upon the computer programs as a result of the design of the interface.

3.3 SOFTWARE PROGRAM SPECIFICATION

3.3.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 3.5)

Software developers shall update and expand the System/Subsystem Specification to produce a detailed Program Specification for the defined software end products. Upon conclusion of the Critical Design Review(s), the Software Program Specification becomes the baseline from which code will be produced.

The detailed design shall be described through the routine level of software organization and the lowest logical level of data base organization. It shall emphasize timing, storage, and accuracy and shall refer to and update the System/Subsystem Specification.

3.3.2 GUIDANCE

The Program Specification document is designed to present more detailed data than the System/Subsystem Specification. Although it is a separate document, it is in fact an expansion of the System/Subsystem Specification and could be viewed as an updated version of it. The purpose of the Program Specification is to add enough additional detail to the System/Subsystem Specification to allow the developer to begin producing code.

When a software system is a complex aggregate of computer programs, Section 4.4 of the Program Specification may be developed in increments and reviewed at different times. These reviews should correspond to the time that the detailed design of a computer program is completed. For less complex software systems, all computer programs may be reviewed at a single meeting.

The Software Acquisition Manager must approve the schedule for preparation and review of the Program Specification.

3.3.3 FORMAT FOR SOFTWARE PROGRAM SPECIFICATION

SOFTWARE PROGRAM SPECIFICATION

TABLE OF CONTENTS

Section 1.	GENERAL
1.1	Purpose of the Program Specification
1.2	Project References
1.3	Terms and Abbreviations
Section 2.	SUMMARY OF REQUIREMENTS
2.1	Program Description
2.2	Program Functions
2.2.1	Accuracy and Validity
2.2.2	Timing
2.3	Flexibility
Section 3.	ENVIRONMENT
3.1	Support Software Environment
3.2	Interfaces
3.3	Storage and Processing Allocation
3.4	Security and Privacy
Section 4.	DESIGN DETAILS
4.1	Operating Procedures
4.2	System Logical Flow
4.3	System Data
4.3.1	Inputs
4.3.2	Outputs
4.3.3	Displays
4.4	Program Description and Logic
4.4.1	Computer Program Identification
4.4.1.1	Computer Program Component No. 1
4.4.1.1.1	Computer Program Component No. 1
4.4.1.1.1.1	Graphical Representation
4.4.1.1.2	Computer Program Component No. 1
4.4.1.1.2.1	Description
4.4.1.1.3	Computer Program Component No. 1
4.4.1.1.3.1	Interfaces
4.4.1.1.4	Computer Program Component No. 1
4.4.1.1.4.1	Data Organization
4.4.1.1.5	Computer Program Component No. 1
4.4.1.1.5.1	Limitations

4.4.1.X	Computer Program Component No. X
4.4.1.X.1	Computer Program Component No. X Graphical Represnetation
4.4.1.X.2	Computer Program Component No. X Description
4.4.1.X.3	Computer Program Component No. X Interfaces
4.4.1.X.4	Computer Program Component No. X Data Organizations
4.4.1.X.5	Computer Program Component No. X Limitations
4.4.N	Computer Program No. N
4.5	Data Base Characteristics and Data Environment
4.5.1	Data Base Characteristics
4.5.1.1	Data Organization
4.5.2	Data Retention
4.5.3	Program Relationships

Section 5. TEST AND QUALIFICATION

Section 6. NOTES

Section 1. GENERAL

1.1 Purpose of the Program Specification. This paragraph shall describe the purpose of the Program Specification in the following words or appropriate modifications thereto:

The objective of this Program Specification for (Project Name) (Project Number) is to describe the design of the software system/ subsystem in sufficient detail to permit program production by the programmer/coder.

1.2 Project References. This paragraph shall provide a brief summary of the references applicable to the history and development of the project. A list of applicable documents shall be included. At least the following shall be specified, when applicable, by source or author, reference number, title, and security classification:

- a. Software Requirements Specification.
- b. System/Subsystem Specification.
- c. Related Program Specifications.
- d. Any other pertinent documentation or significant correspondence not specified in the Software Requirements Specification.

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

Section 2. SUMMARY OF REQUIREMENTS

2.1 Program Description. This paragraph shall provide a general description of each computer program in the system/subsystem to establish a frame of reference for the remainder of the document. It shall include a summary of requirements established in the Software Requirements Specification. The documentation of higher-order systems/subsystems and related computer programs shall be referenced, as required, to enhance the general description of the computer programs contained in the Program Specification.

2.2 Program Functions. This paragraph shall identify and describe the function of each computer program in the software system/subsystem. Although descriptions of the functions may be more refined as a result of detailed design activities, they must be consistent with the functions described in Section 2.2 of the System/Subsystem Specification.

2.2.1 Accuracy and Validity. This paragraph shall provide a description of accuracy requirements imposed on each computer program identified in the System/Subsystem Specification. The requirements will be related to paragraph 4.1.1 of the Software Requirements Specification and to paragraph 2.2.3 of the System/Subsystem Specification. Changes in the accuracy requirements from the next higher-order document must be explicitly identified. The following accuracy requirements must be considered:

- a. Accuracy requirements of mathematical calculations.
- b. Accuracy requirements of data.
- c. Accuracy of transmitted data.

2.2.2 Timing. This paragraph shall provide a description of the timing requirements placed on each computer program. The requirements will be related to paragraph 4.1.2 of the Software Requirements Specification and to paragraph 2.2.4 of the System/Subsystem Specification. Changes in the timing requirements from the next higher-order document must be explicitly identified. For each computer program, the following timing requirements may be considered:

- a. Throughput time.
- b. Response time to queries and to updates of data files.
- c. Response time of major program functions.
- d. Sequential relationship of program functions and data flows.
- e. Priorities imposed by types of inputs and changes in modes of operation.
- f. Timing requirements for the range of traffic load under varying operating conditions.
- g. Sequencing and interleaving programs and systems (including the requirements for interrupting the operation of a program without loss of data).
- h. I/O transfer time required for disk, drum, tape, etc.
- i. Internal processing time.

2.3 Flexibility. This paragraph shall provide a description of the capability to be incorporated for adapting the software system/subsystem to changing requirements, such as anticipated operational changes, interaction with new or improved programs, and planned periodic changes. Components and procedures designed to be subject to change will be identified. This paragraph will be related to paragraph 2.3 of the System/Subsystem Specification.

Section 3. ENVIRONMENT

This section shall provide a short description of the environment in which the software system will operate. It shall be based upon Section 3 of the Software System/Subsystem Specification and shall identify any changes that have occurred since then. If there have been no changes, a brief summary and reference to the appropriate sections of the System/Subsystem Section will be acceptable for the following sections.

3.1 Support Software Environment. Refer to Section 3.2 of the System/Subsystem Specification.

3.2 Interfaces. If applicable, refer to relevant Interface Control Documents.

3.3 Storage and Processing Allocation. Although still not binding, this section shall update the allocation of memory storage and processing time for each software unit, computer program component, computer program, common data base, and executive routines. The "as-built" Program Specification shall use actual results.

3.4 Security and Privacy. Refer to Section 3.4 of the System/Subsystem Specification.

Section 4. DESIGN DETAILS

4.1 Operating Procedures. This section shall describe any specific procedures necessary for operation of each computer program in the software system. It shall also describe the interaction of the applications software with the executive software.

4.2 System Logical Flow. Refer to Section 4.2 of the System/Subsystem Specification. This section shall include any additional information gained since preparation of sections 4.2, 4.2.1, 4.2.2, and 4.2.3 of the System/Subsystem Specification.

4.3 System Data.

4.3.1 Inputs. Refer to Section 4.3.1 of the System/Subsystem Specification.

4.3.2 Outputs. Refer to Section 4.3.2 of the System/Subsystem Specification.

4.3.3 Displays. Refer to Section 4.3.3 of the System/Subsystem Specification.

4.4 Program Description and Logic. Paragraph 4.4.1 through 4.4.N shall provide detailed descriptions of the computer programs and computer program components listed in paragraph 4.4.1 through 4.4.N of the System/Subsystem Specification.

4.4.1 Computer Program Identification. This section shall contain the lead phrase, "This section contains the detailed technical descriptions of the computer program components of the (NAME) computer program". The following subparagraphs shall be repeated for each component:

4.4.1.1 Computer Program Component No. 1. Refer to paragraph 4.4.1.1 of the System/Subsystem Specification.

4.4.1.1.1 Computer Program Component No. 1 Graphical Representation. Refer to 4.4.1.1.1 of the System/Subsystem Specification. This section shall include any additional information gained since preparation of the System/Subsystem Specification. If software units have been subdivided into routines, lower-level structured design charts (or equivalent) that identify all decision points in the component shall be included. The lower-level charts, including routines, should refer to higher-level representations as appropriate.

4.4.1.1.2 Computer Program Component No.1 Description. Refer to paragraph 4.4.1.1.2 of the System/Subsystem Specification. This section shall identify equations to be solved, algorithms used to solve the equations, timing and accuracy characteristics, and any special conditions for operation of routines in the component. Within the description of each unit, this section shall contain the following information for each routine:

- a. Routine name.
- b. Purpose (brief narrative summary).
- c. Assumptions.
- d. Sizing (code and data).
- e. Calling sequence, arguments and definitions, and error exits.
- f. Inputs, output, and use.
- g. Routines called by this routine, and calling routines.
- h. Engineering description (including equations and a processing flow description expressed in a Program Design Language or other suitable descriptive format).

i. Restrictions and limitations.

4.4.1.1.3 Computer Program Component No. Interfaces. Refer to paragraph 4.4.1.1.3 of the System/Subsystem Specification. This paragraph should include any additional information gained since preparation of the System/Subsystem Specification.

4.4.1.1.4 Computer Program Component No. 1 Data Organization. This paragraph shall contain a list and description of all data items and tables which are unique to and included within the computer program component. It shall describe the areas of memory available for temporary storage. This paragraph shall include all internally-defined symbols and constants and their equivalents and meaning.

4.4.1.1.5 Computer Program Component No. 1-Limitations. This paragraph shall summarize any known or anticipated limitations of the Computer Program Component. It shall include a listing of all restrictions and constraints which apply to the Computer Program Component, including timing requirements, limitations of algorithms and formulas used, limits of input and output data, associated error correction sensing, and the error checks programmed into the routines.

4.4.1.X Computer Program Component No. X.

4.4.1.X.1 Computer Program Component No. X Graphical Representation.

4.4.1.X.2 Computer Program Component No. X Description.

4.4.1.X.3 Computer Program Component No. X Interfaces.

4.4.1.X.4 Computer Program Component No. X Data Organization.

4.4.1.X.5 Computer Program Component No. X Limitations.

4.4.N Computer Program N. Section 4.4.1 and subparagraphs shall be repeated for each computer program in the software system/subsystem.

4.5 Data Base Characteristics and Data Environment. This paragraph shall provide a description of the files, tables, dictionaries, program interrelationships with the tables, storage allocation, intermediate data structures, and data retention requirements of the system/subsystem data base(s).

4.5.1 Data Base Characteristics. This paragraph shall describe each program-related file, table, dictionary or directory to include the following:

- a. Title and tag.
- b. Description of content.
- c. Parameters - start of file, end of file.
- d. Number of records or entries.
- e. Record parameters - start of record, end of record.
- f. Relationship of each record to the common data base, if applicable.
- g. Storage, to include type of storage, amount of storage and, if known, beginning and ending addresses.
- h. Normal order of the file and other orders required for special purposes.
- i. Classification.

4.5.1.1 Data Organization. This paragraph shall describe directly or by reference to a Data Dictionary the relationship of the items, tables, and files contained within the data base. It shall incorporate such information as the following:

- a. A list of files, specifying for each file the address in storage and the number of tables contained in the file, etc.
- b. A list of tables, specifying for each table the location within the file and number of words contained in the table, etc.
- c. A list of items, identifying item location within the table, number of lists, item type, scaling, etc.

4.5.2 Data Retention. Data retention requirements shall be described as follows:

- a. Historic retention to include collection of data to be retained, format, storage medium, and time parameters.
- b. Periodic report data, e.g., time retained after report generation and time retained to provide summary reports.
- c. Summary report data, such as time retained after summary report.

4.5.3 Program Relationships. This paragraph shall describe the interrelationships of the various computer program components to the various files and tables used in the data base(s).

Section 5. TEST AND QUALIFICATION

This section shall consist of the following statement:

Detailed Test and Qualification provisions are contained in the Test Plans and Procedures for this software system/subsystem. The Software System DT&E Plan contains a detailed Test Case Matrix which relates each functional requirement in the Software Requirements Specification to the applicable test. The Test Plans and Test Procedures also specify any special test tools and capabilities required to qualify the software system/subsystem.

Section 6. NOTES

This section may be used to provide background material which would be of assistance to reviewers in understanding the design of the computer programs. It may also include alternate design solutions, design standards, alternate algorithms, and alternate mathematical derivations.

3.4 SEPARATE DATA DOCUMENTATION

3.4.1 Introduction

A data base is a collection of interrelated data that is stored independent of the computer programs that use the data. Methods of retrieving and modifying the data are often centrally controlled by a Data Base Management System (DBMS), but the term "data base" is loosely used throughout this section to include such diverse collections as sequential files, direct-access files, inverted files, relational data bases, and network data bases.

If more than one software system or subsystem shares data or if data presents unique management and control problems, Software Acquisition Managers may wish to separate data documentation from the software specifications provided in the first three sections of Part III. If separate documentation is preferred, Software Acquisition Managers may use the Data Dictionary Document described in this section.

Under NSAM 81-2, there are three acceptable methods for documenting the data of a software system. They are:

- a. to describe the system data in the Software Requirements Specification, the System/Subsystem Specification, and the Software Program Specification;
- b. to describe the system data in a separate, printed Data Dictionary Document; or
- c. to describe the data in an automated Data Element Dictionary/Directory which is capable of displaying or printing the information for human use. It may also be able to interact with other system software to control the use of data in the system.

The purpose of documenting the design of data is the same as documenting the design of computer programs: to state the requirement in functional terms; to refine the functions in technical terms sufficient for preliminary design; to further refine the technical terms until they are adequate for detailed design; and to eventually complete the detailed design until it becomes the "as-built" documentation. No matter which method or combination is chosen for documentation of data, the same principal components must be documented: all files; all record types; all items within records (irrespective of their eventual logical or physical implementations, such as "set," pointer or key field value); and all pertinent information about the use of the data. This includes a physical description of the data; a logical description; and a functional description of how the data is used, how it is changed, how it is removed and how its integrity is ensured.

Descriptions of data in any of the three forms must be traceable back through the computer program functions in the Software Requirements Specification to the original statement of system requirements. As data documentation becomes progressively more codified during development, software developers must take particular care to assure that this traceability is maintained. The terminology used to describe the data must also remain consistent in all documentation produced during the acquisition process.

The format of the Data Dictionary that follows in Section 3.4.3 describes what must be included in the completed document. It can be used as a checklist for evaluating either of the other two acceptable methods of documenting data.

3.4.2 GUIDANCE

3.4.2.1 SELECTION CRITERIA FOR DATA DOCUMENTATION METHODS

Software managers should examine several factors before selecting the method of documenting data:

- a. Size and complexity of the system;
- b. Extent of sharing of data between systems;
- c. Use of a generalized data base management system;
- d. Availability of an automated data dictionary;
- e. Number of data descriptions;
- f. Complexity of data structures;
- g. Volatility of data descriptions;
- h. Number of analysts and programmers involved;
- i. Number of organizations involved;
- j. Shortage of clerical assistance;
- k. Users of the data documentation.

Systems that share little data with other systems might appropriately have their data descriptions included in the Software Requirements Specification, System/Subsystem Specification, and Software Program Specification.

Systems with more complicated data requirements, such as conversion of data from an old system to a new one or sharing of a data base by more than one software system, should have their data documented separately. This will emphasize the importance of documenting the data and require less duplication of descriptions.

The greater the number and strength of the factors listed above, the greater the value of documenting data separately, possibly by automated methods. The Data Dictionary Document provides a standard, consistent framework for documenting information about the data used by a software system. If an automated Data Dictionary System is available, the same information that would go into the printed Data Dictionary Document should be input to and available from the automated system.

All systems that use a commercial Data Base Management System (DBMS) with an automated data dictionary facility should use the automated means of documenting data. If a DBMS does not have the automated data dictionary, a stand-alone data dictionary should be used if available. If an automated data dictionary cannot be easily enhanced to support the data documentation requirements, then missing information should be documented in the printed form. Only in unusual cases should a new automated data dictionary be developed for a system.

If a system will use many data descriptions or if the descriptions will change frequently, an automated data dictionary will be useful even if the system does not use a data base management system. This will reduce typing and automate the extensive and tedious checking required to ensure completeness and consistency.

3.4.2.2 PREPARATION OF THE DATA DICTIONARY DOCUMENT

The decision to separately document data should be made as early as possible, preferably during the software requirements specification phase. The Data Dictionary Document as outlined in Section 3.4.3 is a companion to the Software Requirements Specification, the Software System/Subsystem Specification and the Software Program Specification. It supplements Section 4 of the Software Requirements Specification. It replaces Sections 4.5 of the System/Subsystem Specification and 4.5 of the Program Specification. By the Software Requirements Review, the Data Dictionary Document should be prepared and the following sections reviewed with the Software Requirements Specification at the Software Requirements Review meeting:

SECTION 1 - GENERAL should be complete.

SECTION 2 - SYSTEM SUMMARY should be complete.

SECTION 3 - DATA IDENTIFICATION AND PROCEDURES FOR ADMINISTRATION should be complete.

SECTION 4 - FUNCTIONAL DESCRIPTION OF THE DATA should be complete.

SECTION 5 - LOGICAL ORGANIZATION OF THE DATA may only exist in outline form, but should be complete to the extent that the logical representation of the data is known.

SECTION 6 - PHYSICAL ORGANIZATION OF THE DATA may only exist in outline form, but should be complete to the extent that the physical representation of the data is known.

Section 4.5 of the System/Subsystem Specification identifies and names the levels of data hierarchy. For each level, it identifies the name, contents and size of each data element. By Preliminary Design Review, the Data Dictionary Document should be prepared to this level of detail and reviewed with the System/Subsystem Specification at the Preliminary Design Review meeting. Sections one through four of the Data Dictionary Document, which were completed for the Software Requirements Review meeting, should be refined as a result of the preliminary design activities. In addition, Section 5 of the Data Dictionary Document should be complete. Section 6 may only exist in outline form, but should be complete to the extent that physical representation of the data is known.

Section 4.5 of the Software Program Specification contains a complete definition of the data base down through the lowest logical level of data base organization. By Critical Design Review, the Data Dictionary Document should be complete. It should then document the complete design of all data that is a part of the software system or subsystem. It should be reviewed with the Software Program Specification at the Critical Design Review meeting.

The Data Dictionary Document is a technical document prepared for programmers and data base administrators. When completed, it must be sufficiently detailed to permit program coding and data base generation by the development organization. The documentation in this section is intended to cover all types of systems; so it does not make specific data or presentation formats mandatory. In fact, the same information is collected whether or not the Data Dictionary is automated. Software developers should devise the physical formats most useful and comprehensible to them. To achieve consistency in documentation, however, the following practices should be followed for all Data Dictionary Documents:

a. Each graphic representation should be followed by a narrative explanation.

b. Each item of information shown in a graphic representation must be consistent with standard data element names.

These graphic representations should be in the form most appropriate to the data being described (e.g., record layout forms, hierarchical charts, or CODASYL schema diagrams). They may be hand-drawn or automatically produced using computer-supported design tools.

Just as for the other specifications described in Part III, the Final Data Dictionary Document, incorporating all approved changes, shall be delivered after completion of Software System DT&E.

3.4.2.3 STANDARD DATA ELEMENT NAMES

Many data element names and associated codes have been standardized to facilitate data exchange and commonality of data structures. Whenever applicable, these standard data elements must be used in all data base files. For data bases which are developed internally, the NSA Data Standards Center is the source for information and guidance on standardized data element names and codes. For contracted development, the Software Acquisition Manager is responsible for making the appropriate standards available to the contractor.

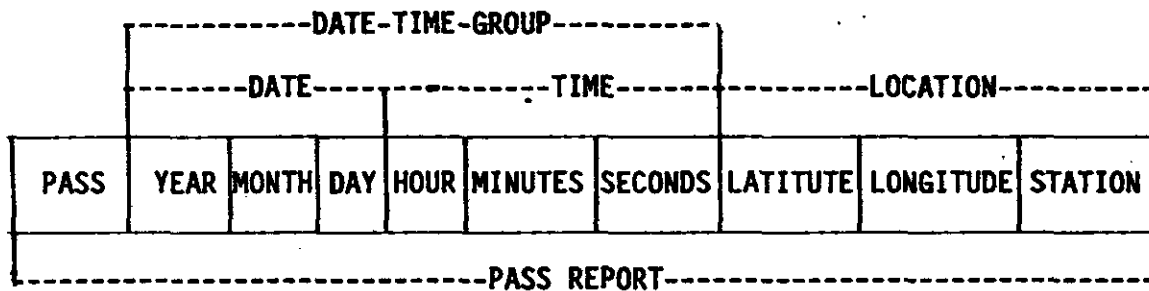
3.4.2.4 SAMPLE DATA DICTIONARY DOCUMENT ENTRIES

The purpose of the data dictionary is to catalog the data entities with their attributes. The data is divided into three types:

- a. **DATA ELEMENT** - the smallest definable, discrete data entity or unit of information.
- b. **DATA STRUCTURE** - an aggregation (collection) of related data elements, defined logically, but not in a physical format (i.e., it is a grouping of data elements with no specific format).
- c. **DATA STORE** - grouping of data structures and/or data elements into some meaningful collection.

The following example illustrates these concepts and the relationship between them. Assume that the system to be developed must create PASS REPORTS. This hypothetical report is made up of a unique PASS number, a Date-Time-Group, and a location. Refinement of the requirements reveals that both the date-time group and location information are divided into additional parts as follows:

EXAMPLE



In this example it should be clear that the PASS REPORT stores both data structures and data elements. This example also illustrates a data structure (Date-Time-Group) which is made up of smaller data structures (Date and Time) which are in turn made up of data elements. To summarize, the following lists recap the data which falls into each of the three categories:

DATA STORE: PASS REPORTS

DATA STRUCTURE: DATE, TIME, DATE-TIME-GROUP, LOCATION, all of
information of one PASS REPORT

DATA ELEMENT: PASS, YEAR, MONTH, DAY, HOUR, MINUTES, SECONDS,
LATITUDE, LONGITUDE, STATION

Figure 3-5 illustrates the relationships between data stores, data structures and data elements in the same example.

In the figure, the Pass Report data store is composed of several instances of Pass Report data structures. For convenience, only one Pass Report data structure is illustrated to consist of a Date-Time-Group data structure, a Location data structure and a Pass data element. This Pass Report data structure could represent all of the fields in a record. The Pass data store would then be analogous to a file.

The Date-Time-Group data structure is composed of the Date data structure and the Time data structure. The Year, Month, and Date data elements comprise the Date data structure. Likewise, the Hour, Minutes, and Seconds data elements comprise the Time data structure.

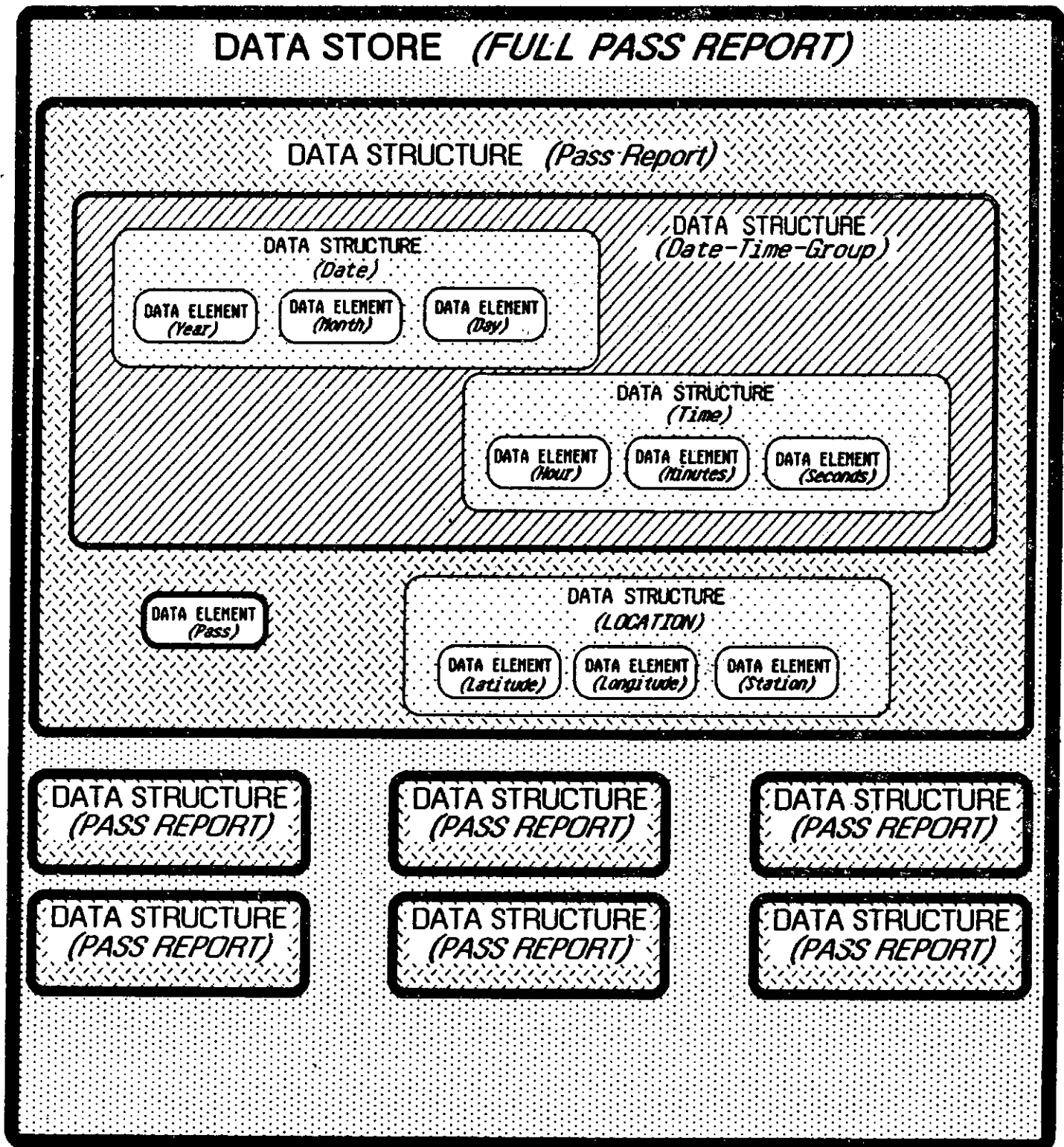


Figure 3-5
RELATIONSHIPS BETWEEN DATA STORES, DATA STRUCTURES, AND DATA ELEMENTS

With these concepts in mind, the following are sample Data Dictionary Document entries to illustrate the format described in Section 3.4.3. Notice that the attributes associated with a given entry will vary depending on the type of software system being developed, and the types of data processed by that system.

LOGICAL ORGANIZATION

DATA ELEMENT EXAMPLE

NAME: REFERENCE FREQUENCY - IFREQ

DESCRIPTION: This scalar contains a desired frequency used by the SWEEP process as a reference point. SWEEP will scan the frequency interval, + 100Hz about this point. The value of this variable is chosen by an operator.

PARENTAGE: DISPLAY-FREQ (data structure)
 MESSAGE-PACKET (data structure)

VALUES: 50.0 to 1.0 E9

UNITS OF MEASURE: KHz

NOMINAL VALUES: default value = 155.0 KHz

ALIAS: NEW-FREQ (data element)

SYNONYM: NOMINAL FREQUENCY

KEY: this is not used as an index to other data

DATA STRUCTURE EXAMPLENAME: DISPLAY-FREQ packet - DSPFRQDESCRIPTION: DISPLAY-FREQ packet contains all of the information that will be passed from the MAN-MACHINE-INTERFACE process to the DISPLAY-MASK process for output on the VDT display. It contains all of the information computed in the SHEEP process and informs the DISPLAY-MASK process which output display mask to use.PARENTAGE: ACTION-REQUEST (data store)LARGEST SOFTWARE ENTITY USING IT: process level (used by various routines)ALIAS: NEW-FRQUSAGE: MAN-MACHINE-INTERFACE process
DISPLAY-MASK processFORMAT: ID (data element)

DETECTOR-TYPE (data element)

GAIN-SETTING (data element)

TEST-TYPE (data element)

FREQUENCY (data element)

BANDWIDTH (data element)

DATA STORE EXAMPLENAME: MASK-1 - LOG MSKDESCRIPTION: MASK-1 is a VDT screen format mask that is displayed at login time on all operator terminals. This mask welcomes the user to the session, prompts for password, and positions the cursor for user entry.LARGEST SOFTWARE ENTITY USING IT: process level (used by two routines)USAGE: LOGIN processCONTENT: LOGIN-MASK (data structure)
USER-PRIVILEGES (data structure)RETENTION: permanentSCOPE: available to LOGIN process only

PHYSICAL ORGANIZATIONDATA STRUCTURE EXAMPLEDISPLAY-FREQ - DSPFRQ

ID (I*2)	DETECTOR-TYPE (I*2)	GAIN-SETTING (I*2)	TEST-TYPE (I*2)
FREQUENCY (R*8)			
BANDWIDTH (I*4)			

LENGTH: 20 bytesDATA STORE EXAMPLEMASK-1

LOGIN-MASK (488 bytes)
USER-PRIVILEGES (64 bytes)

PARTITIONS: The MASK-1 data store is partitioned into the LOGIN-MASK data structure and the USER-PRIVILEGES data structure.

ACCESS METHOD: The two partitions of the file are accessed directly using the ID field as the access key.

RELATIONSHIP OF COMPUTER PROGRAM COMPONENTS TO

PARTITIONS: The LOGIN process uses information from the USER-PRIVILEGES partition to create a table in memory containing access privileges for each user group. This table determines which prompts in the LOGIN-MASK to display to the user.

3.4.3 FORMAT FOR THE DATA DICTIONARY DOCUMENT

DATA DICTIONARY DOCUMENT

TABLE OF CONTENTS

SECTION 1 GENERAL

- 1.1 Purpose of the Data Dictionary Document
- 1.2 Project References
- 1.3 Terms and Abbreviations

SECTION 2 SYSTEM SUMMARY

- 2.1 System/Subsystem Description
- 2.2 Environment

SECTION 3 DATA IDENTIFICATION AND PROCEDURES FOR ADMINISTRATION

- 3.1 Data Identification
- 3.2 Organizational Responsibilities
- 3.3 Procedures
- 3.4 Security and Access Controls

SECTION 4 FUNCTIONAL DESCRIPTION OF THE DATA

SECTION 5 LOGICAL ORGANIZATION OF THE DATA

- 5.1 Design Description of the data
- 5.2 Graphical Representation of the Data
- 5.3 Data Elements
- 5.4 Data Structures
- 5.5 Data Stores
- 5.6 Relationship of Computer Program Components to the Data

SECTION 6 PHYSICAL ORGANIZATION OF THE DATA

- 6.1 Physical Description of the Data
- 6.2 Physical Description of Each Partition
- 6.3 Size and Storage Requirements
- 6.4 Relationship of Computer Program Components to the Partitions

SECTION 1 GENERAL

1.1 Purpose of the Data Dictionary Document. This paragraph shall describe the purpose of the Data Dictionary document in the following words, modified when appropriate:

The purposes of this Data Dictionary Document for (project name) (Project Number) are to document the definitions of data elements that are used in the system and to provide the basic design information necessary for the construction of the system files, dictionaries, and directories.

1.2 Project References. This paragraph shall identify the acquisition organization, development organization, and the user of the software system. A list of applicable documents shall be included. At least the following shall be specified, when applicable, by source or author, reference number, title, and security classification.

- a. Software Requirements Specification
- b. Software System/Subsystem Specification
- c. Program Specification
- d. User's Manual
- e. Program Maintenance Manual

1.3 Terms and Abbreviations. This paragraph shall provide an alphabetic listing of terms, definitions, and acronyms unique to this document and subject to interpretation by the user of the document. This listing will not include item names or data codes. This glossary may be provided as an appendix if it is too large to fit in this section of the document.

SECTION 2 SYSTEM SUMMARY

2.1 System/Subsystem Description. This section shall contain a general description of the software system/subsystem to establish a frame of reference for the remainder of the document. Higher order and parallel systems/subsystems and their documentation shall be referenced as required to enhance this general description. It shall also include a list of the functional areas of the system and show the relationship between the functional areas and the system/subsystem data which will be described in this document.

2.2 Environment. This section shall identify the hardware and software environment, including Data Base Management System, in which the system data will be used. If the system will share an integrated data base with other systems, that integrated data base shall be identified in this section. Included shall be a discussion of any impacts on the integrated data base of the data that are to be used in this system as described in Sections 4, 5, and 6. Such impacts may include recommended changes in the organization of the data in the integrated data base that would enhance system efficiency, the addition to the integrated data base of new data elements needed for this system, etc. Recommendations concerning changes in existing software which supports the integrated data base, such as data base management systems, should also be included. If the system data base is designed based on the assumption that planned or recommended enhancements to either the integrated data base or its support software will be available at the time of system implementation, these assumptions shall be stated.

SECTION 3 DATA IDENTIFICATION AND PROCEDURES FOR ADMINISTRATION

3.1 Data Identification. This paragraph shall describe the system labeling/tagging conventions to the extent necessary for the programmer to use the conventions as a practical working tool. For example, it should specify the naming conventions used to identify new versions of the data and any conventions adopted to relate the data components to software requirements and design specifications.

The data dictionary document is first written during the software requirements phase. At this time, the physical structure (e.g., file, record) or the implementation (e.g., linked list, index) of the data is unknown. Therefore it is defined only in functional terms. As the design evolves and becomes more detailed, the implementation becomes better defined. Additions are made to the existing data element definitions or new data elements are inserted as they become known to reflect the refined system and data definitions.

Data elements at one phase of development relate to elements at the preceding phase, but may be given different names because of language constraints. A naming convention that shows that the elements are directly related (or derived) enables one to easily discern the relationship.

3.2 Organizational Responsibilities. This section shall identify the organization responsible for managing the data. Although this organization may change as the system proceeds from development through acceptance (and for life cycle support), the planned organizations should be identified. If the developer does not know the organization, or if it has not been identified, it should be so stated.

3.3 Procedures. This section shall contain the procedures and instructions to be followed by all personnel who will contribute to the generation of the data and who will use it for testing or operational purposes. It shall also identify and describe support programs which will be used to assist in the implementation of the procedures. Manual procedures created specifically to handle the data shall also be included. The following shall be described:

- a. Procedures for the authorization of new data;
- b. Criteria for entering data into the test data base;
- c. Criteria for entering data into the operational data base;
- d. Rules for the submission of data, including formats for data description sheets and cards;
- e. Review procedures to assure data integrity;
- f. Review procedures to assure that performance goals are met.

3.4 Security and Access Controls. This section shall describe the procedures for protecting the data from unauthorized access. It shall describe the classified components of the system and relate the access controls to them. It shall also prescribe any privacy restrictions associated with the data being handled.

SECTION 4 FUNCTIONAL DESCRIPTION OF THE DATA

This section shall describe the functional relationship between the data and the computer programs that will make use of the data. The description should be related to Sections 4.2 and 4.4 of the Software Requirements Specification.

The means and timing (who, when, and how frequently) of the following shall be described in narrative form:

a. Input Source. This includes the sources from which the data will enter the system, such as: other systems, operators at terminals, internally-generated computer processes, etc. Any displays (masks, forms, screens, menus) used for data entry shall be included. Also, any validity checking of the input data should be stated.

b. Uses. This means the manner in which the data will be used, such as: reference material for analysts, raw source data for computer processing, control for other processes, etc. The uses shall be related to the computer program functions identified in Section 4.2 of the Software Requirements Specification.

c. Change. This means the processes by which the data is changed, such as: by computer programs acting on the data, through processes controlled by operators at terminals, by periodic batch replacement, or by an independent auxiliary computer.

d. **Removal.** This means the data retention requirements of the data, such as: permanent retention, periodic removal, aperiodic removal, etc. It also includes the aging sequence or algorithm that will be used to identify the data that will be retired and the method of moving inactive data to backup storage.

e. **Integrity.** This means the processes by which the data will be reviewed to assure that it remains valid, such as: computer processes (periodic validation programs, performance analysis programs, etc.) and manual review. Also included shall be descriptions of backup and recovery procedures in case the active versions are lost or damaged.

SECTION 5 LOGICAL ORGANIZATION OF THE DATA

This section shall describe the conceptual scheme of the organization of the data and define the data which is identified in the System/Subsystem Specification.

5.1 **Design Description of the Data.** This section shall describe the logical relationships among the data. It should be in conceptual terms with no reference to physical organization on storage devices or eventual method of implementation. It shall include the following information:

- a. Constraints or influences affecting all or part of the design, such as existing files, software or systems
- b. Rationale for the design, including the reason for adopting the selected design
- c. Indication of any design strategies abandoned during the design (if relevant)
- d. Grouping of the data (or design of the data base) including logical partitioning of the data (or data base) and the identification of the levels of data hierarchy.

5.2 **Graphical Representation of the Data.** This section shall contain a graphical representation of the data (chart or diagram). It shall depict the relationship among all the data stores, data structures, and data elements.

5.3 Data Elements. This section shall provide in alphabetical order a listing of data elements used in the software system. A data element is the lowest level of definable, discrete data useful in the system. Any further division of the data would cause it to lose meaning. The following list of attributes, both logical and physical, shall be used to define each data element:

- a. Name of data element;
- b. Description - what the data element does and how it is used in the system;
- c. Parentage - next highest related class of data (i.e., data structure) that directly uses the data element;
- d. Values or range of values;
- e. Units of measure - given for any data element that has dimensional meaning;
- f. Nominal values - initial value, default value, typical value (if they are appropriate);
- g. Alias - when a data element is called by two or more different identifier names and all have the same attributes;
- h. Synonyms - when two or more data elements have different names and different attributes, but have the same functional meaning;
- i. Key - identify whether data element acts as an index to other related data.

Note that a data element could become a field in a record or cell in a table when the focus of attention shifts from the logical to the physical view of the data.

5.4 Data Structures. A data structure is a grouping of data elements with no specific format that are treated by the system as a single logical unit. This section shall contain a description of every data structure defined. The following information shall be provided:

- a. Name of the data structure;
- b. Description of the data structure (what differentiates it from other data structures);

- c. Parentage - data structures and stores containing the data structure;
- d. Largest software entity using the data structure;
- e. Alias - when a data structure is called by two or more different names and both have the same attributes;
- f. Usage - an attribute that indicates the number of times and which processes use a data structure;
- g. Format - list of the elements and the order in which they appear.

Note that a data structure could be a node in a linked list or a record when the focus of attention shifts from the abstract to the physical view of the data.

5.5 Data Stores. This section shall contain a description of every data store defined. The following information shall be provided:

- a. Name of the data store;
- b. Description of the data store (what it contains in general);
- c. Largest software entity using the data store;
- d. Usage - an attribute that indicates the number of times and which processes use a data store;
- e. Content - list the structures and/or elements;
- f. Retention - timespan for keeping the data store in the system;
- g. Scope - accessibility of the data store by processes in the system.

Note that a data store could be a file or a table when the focus shifts from the logical to the physical view of the data.

5.6 Relationship of Computer Program Components to the Data. This section shall portray graphically the relationship of the various computer program components in the software system/subsystem to the stores, structures, and elements. It shall provide set-use matrices that reflect the cross-indexing of all of the above. Figure 3-6 is an example of a Data Store Set-Use Matrix and Figure 3-7 is an example of a Data Structure Set-Use Matrix.

DATA STORE SET-USE MATRIX

COMPONENT DATA STORE	INITIALIZATION	USER-INPUT	VALIDATION	PASS-CAPTURE	REPORT
Pass Report				X	X
Mask-1	X	X	X		
Mask-3		X	X		X
Screen Table	X				

FIGURE 3-6 EXAMPLE OF A DATA STORE SET-USE MATRIX

DATA STRUCTURE SET-USE MATRIX

COMPONENT DATA STORE	INITIALIZATION	USER-INPUT	VALIDATION	PASS-CAPTURE	REPORT
Date-Time-Group				X	X
Date				X	X
Time				X	X
Location				X	X
Login-Mask	X	X	X		X
User-Privileges		X	X		
Report-Mask		X	X		X

FIGURE 3-7 EXAMPLE OF A DATA STRUCTURE SET-USE MATRIX

SECTION 6 PHYSICAL ORGANIZATION OF THE DATA

This section shall describe the conceptual scheme of the organization of the data and define the data which is identified in the Software Program Specification.

6.1 Physical Description of the Data. This section shall describe the physical layout of the data on the storage devices. It shall describe how indices, pointers, chains or other methods are used to locate the data. When appropriate, a graphic representation of the data should be included. It should also describe compaction and coding techniques that are used, including a graphic representation of the data before and after.

The Physical representation of all data defined in Section 5 of the Data Dictionary Document must be described in this section. Aggregates of data which were only described as data stores or data structures must now be described in physical terms. In general the following transformations occur:

Data Stores may become:

- Files
- Tables
- Linked Lists
- Relations
- Reports
- Masks

while data structures may become:

- Records in Files
- Rows in Tables
- Nodes in Linked Lists
- Tuples in Relations
- Subparts in Reports
- Fields in Masks

While this list is not complete, it should illustrate the way logical representations and physical representations of the same data evolve through the development cycle.

6.2 Physical Description of Each Partition. If the active version of any files are separated into a number of physical partitions, this section shall describe the content and function of each partition. This section shall provide the following information for each physical partition of each file, including the "aged" or "historical" files and the backup copies:

- a. Partition name
- b. Password Information: the type of passwords used and the access privileges which these give to the user (The actual passwords are not included.);
- c. Security classification of the content;
- d. Recording format, density, parity, record length, block size;
- e. Primary and secondary storage media;
- f. Storage device and location;
- g. Criteria for moving partitions from one medium to another;
- h. Restrictions and limitations on usage;
- i. Access method;
- j. A description of the relationships between record types in the files, such as networks or hierarchies (not including the mechanism used to link records together).

6.3 Size and Storage Requirements. This section shall provide the following information for each partition of each file. Information in this section must be consistent with Section 4.5 of the System/Subsystem Specification and Section 4.5 of the Program Specification.

- a. File size in bytes, cylinders or tracks.
- b. Any size parameters necessary for individual Data Base Management Systems (DBMS).

6.4 Relationship of Computer Program Components to the Partitions.

This section shall relate the various software units and routines (as identified in the Software Program Specification) to the physical data. If all the data of a given file are accessible (read, modified, or written) by one routine, it is unnecessary to list every item in the file. The primary purpose of this section is as a reference for future modification.

PART IV - SOFTWARE DEVELOPMENT PRACTICES

4.1 Software Analysis and Design

4.1.1 Policy and Requirements Summary (from NSA/CSS Software Acquisition Manual 81-2, Policy 4.1)

Software developers shall use a top-down approach to design software. Top-down requires that the design be performed by starting with the top-level system function and proceeding through a downward allocation, evaluation, and iteration to successively lower levels of design. This design approach enhances design traceability, completeness, and comprehensiveness. Procedures for applying the selected design methodologies shall be documented in a Software Standards and Practices Manual.

Design shall be initiated by establishing a functional design hierarchy, where the top represents the overall mission to be performed by the total software.

- (a) Successive levels shall be obtained by breaking down and partitioning the software into blocks with progressively greater functional detail;
- (b) The software requirements shall be allocated and mapped onto this design hierarchy;
- (c) The lowest level of the design hierarchy shall be defined so that its software elements perform the routine algorithm and data processing functions necessary to implement all of the input-to-output paths of the requirements.

4.1.2 Introduction

The techniques presented in the following sections include discussions of design methods and design representation schemes. A design method is a set of rules and guidelines used for abstracting, decomposing, and solving problems. Software analysis and design are mental activities applied to develop a conceptual model of the software system's functional and informational requirements. Analysis and design methods provide guidance to the software developer in translating the conceptual model into a software design that will satisfy the requirements of the software system. Graphical Representation schemes are the means by which developers express the conceptual model in real terms. They communicate design information to others and assist developers in dealing with problem complexity and technical detail.

A methodology is a collection of methods chosen to complement one another, along with the rules for their application. It includes documentation, procedures, and representation schemes. The purpose of this section is to describe a recommended methodology and related graphical representation techniques to support top-down analysis and design. The methodology is consistent with the documentation formats described in

Section 3 of this manual. The recommended methods are Structured Analysis and Structured Design. The recommended design representation schemes are Data Flow Diagrams, Structure Charts, N² charts, and Program Design Language (PDL). Together, the analysis and design methods make a methodology that satisfy the requirements of top-down analysis and design.

TABLE OF CONTENTS

Section 1. Structured Analysis

- 1.1 Introduction
- 1.2 Data Flow Diagrams
- 1.3 Process Logic
- 1.4 Data Dictionary
- 1.5 Conclusion

Section 2. Structured Design

- 2.1 Introduction
- 2.2 Method
- 2.3 Design Evaluation
 - 2.3.1 Coupling
 - 2.3.2 Cohesion
 - 2.3.3 Ordering of Modules and Decisions
 - 2.3.4 Additional Heuristics
- 2.4 Summary

Section 3. N2 Charts

Section 4. Real-Time Design Issues

Section 5. Program Design Language

- 5.1 Introduction
- 5.2 Description
 - 5.2.1 Sequence
 - 5.2.2 IF-THEN-ELSE Structure
 - 5.2.3 DO-WHILE Structure
 - 5.2.4 DO-UNTIL Structure
 - 5.2.5 CASE Structure
 - 5.2.6 DO Group Exit Structure
 - 5.2.7 Library Segmentation
- 5.3 Complexity Measurement of PDL
- 5.4 Program Design Language Example
- 5.5 Program Design Language Standards

Section 6. Decision Table Standards

- 6.1 Introduction
- 6.2 Description
 - 6.2.1 Parts of a Decision Table
 - 6.2.2 Decision Table Form
 - 6.2.3 Types of Decision Tables

TABLE OF CONTENTS (CONTINUED)

Section 7. Summary and Recommendations

Section 8. References

Section 1. Structured Analysis

1.1 Introduction

Structured Analysis provides a method for decomposing and documenting the functional requirements of a system. The method requires the integrated and disciplined use of graphic models of the logical system (Data Flow Diagrams), a description of the sequence of steps necessary to transform inputs to outputs (Process Logic), and a precise definition of data interfaces (Data Dictionaries). Data Flow Diagrams model the flow of data through a system. Process Logic describes the sequence of steps necessary to transform inputs to outputs, and the Data Dictionary contains information about all data mentioned on Data Flow Diagrams or in Process Logic.

1.2 Data Flow Diagrams

Data Flow Diagrams have four basic elements: (1) Data Flows, represented by arrows; (2) Processes, represented by circles; (3) Stores of Data, denoted by parallel lines; and (4) External Entities, shown as squares. Figure 4-1 is an example of a Data Flow Diagram.

DATA FLOWS

A Data Flow is a pipeline through which data flows. The flow may consist of a single data element or sequences of data elements. A sequence of data elements may be ordered according to rules of "syntax" that define the legal structure of the Data Flow. Data Flows should only be used to show the flow of data, not the flow of control.

PROCESSES

A Process is a point where data transformation occurs. The transformation may be simple, such as placing data in a store, or quite complex, such as converting analog data to a series of digital messages for distribution to other external systems.

DATA STORES

A Data Store is a "frozen" flow of information. It represents the place where data is stored when there is a delay between the time it leaves a Process and the time that another Process is ready to receive it.

EXTERNAL ENTITIES

External Entities represent external sources or users of data which interface with the system. External Entities may be other systems or even people who must interact with the system.

Product Ordering System

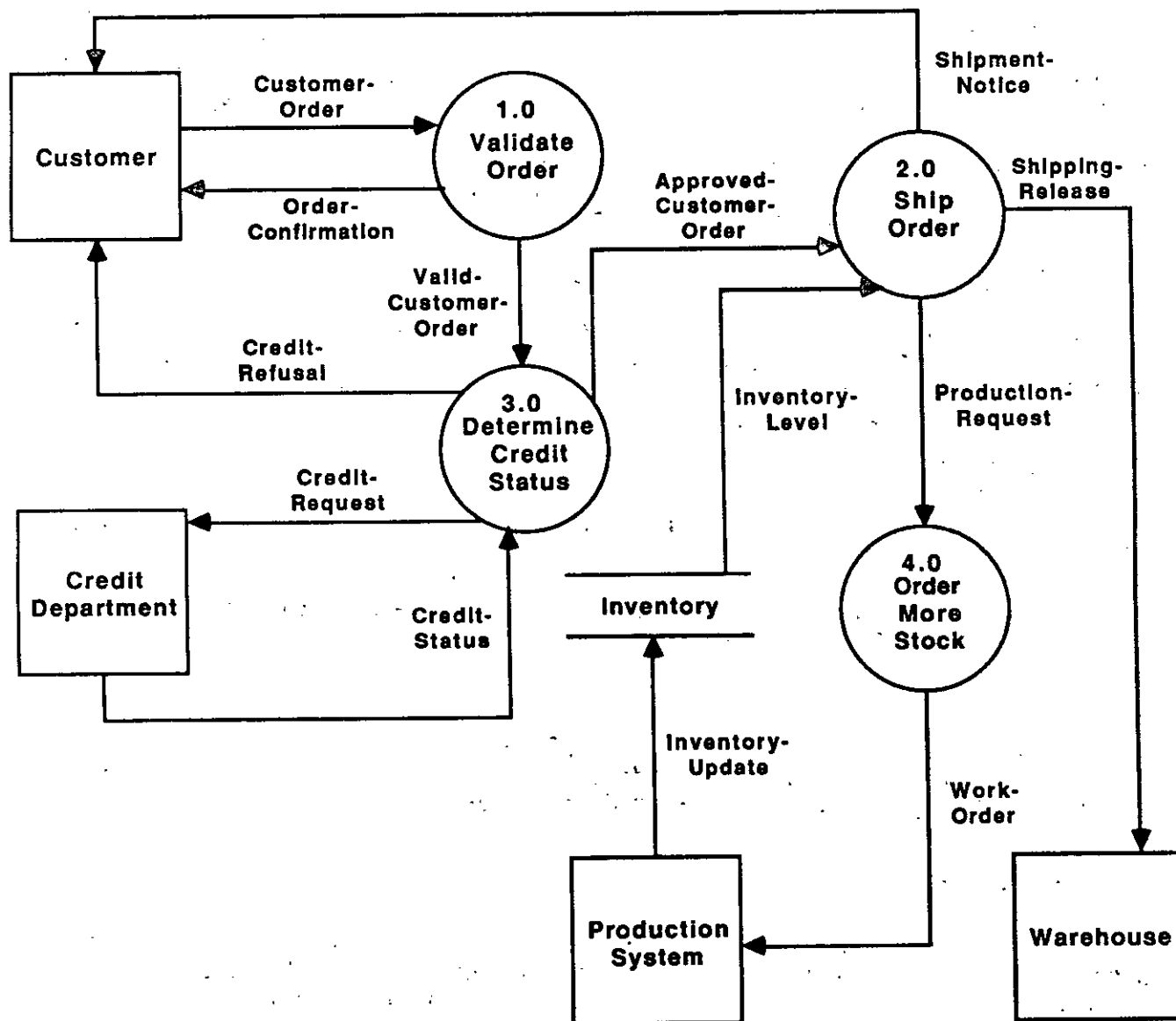


Figure 4-1 Level 0 Data Flow Diagram

1.3 Process Logic

In Data Flow Diagrams for large systems, each Process will represent a major transformation of data that may involve much buried detail. To attempt to describe the logic of such a Process would require so much detail that the description would not be understandable. Hence the practice is to break each high-level Process into more detailed, lower-level Data Flow Diagrams. This expansion should continue until all software functions are shown. The result is that the system may have a hierarchy of diagrams as in Figure 4-2. The number of levels should be chosen so that processes at the lowest level can be described in simple, well-defined logical descriptions called Process Logic.

Process Logic should describe the sequence of actions that must occur for each low-level process to transform its input data into its proper output data. Structured English, Decision Tables, and even prose English are acceptable methods for describing Process Logic. Structured English resembles a high-level Program Design Language. It combines the syntax of structured programming constructs (sequence of operations, repetition of operations, selection of operations) with the semantics of English.

Decision Tables are well suited for describing small, complex actions. If neither Structured English nor Decision Tables is appropriate, any method which effectively communicates the sequence of steps required to accomplish the function can be used.

1.4 Data Dictionary

A Data Dictionary also accompanies the set of Data Flow Diagrams for a system. It documents all data in the system, including individual data elements, data structures, and data stores. For more information on the Data Dictionary, see Section 3.4 of this manual.

1.5 Conclusion

The combination of Data Flow Diagrams, the Data Dictionary, and Process Logic describe the functions that the system is to perform. A complete Software Requirements Specification, however, must also contain other essential information, such as performance requirements and constraints, resource constraints, and man-machine interface requirements.

Although Structured Analysis is not particularly helpful for defining the other types of requirements, it is an effective technique for analyzing and defining the functional and interface requirements for a software system. Not only does it help the requirements analyst understand the role of data and data transformation in satisfying software requirements, but it provides a good technique for breaking high-level requirements into more detailed requirements and graphically representing the manner in which they will be satisfied.

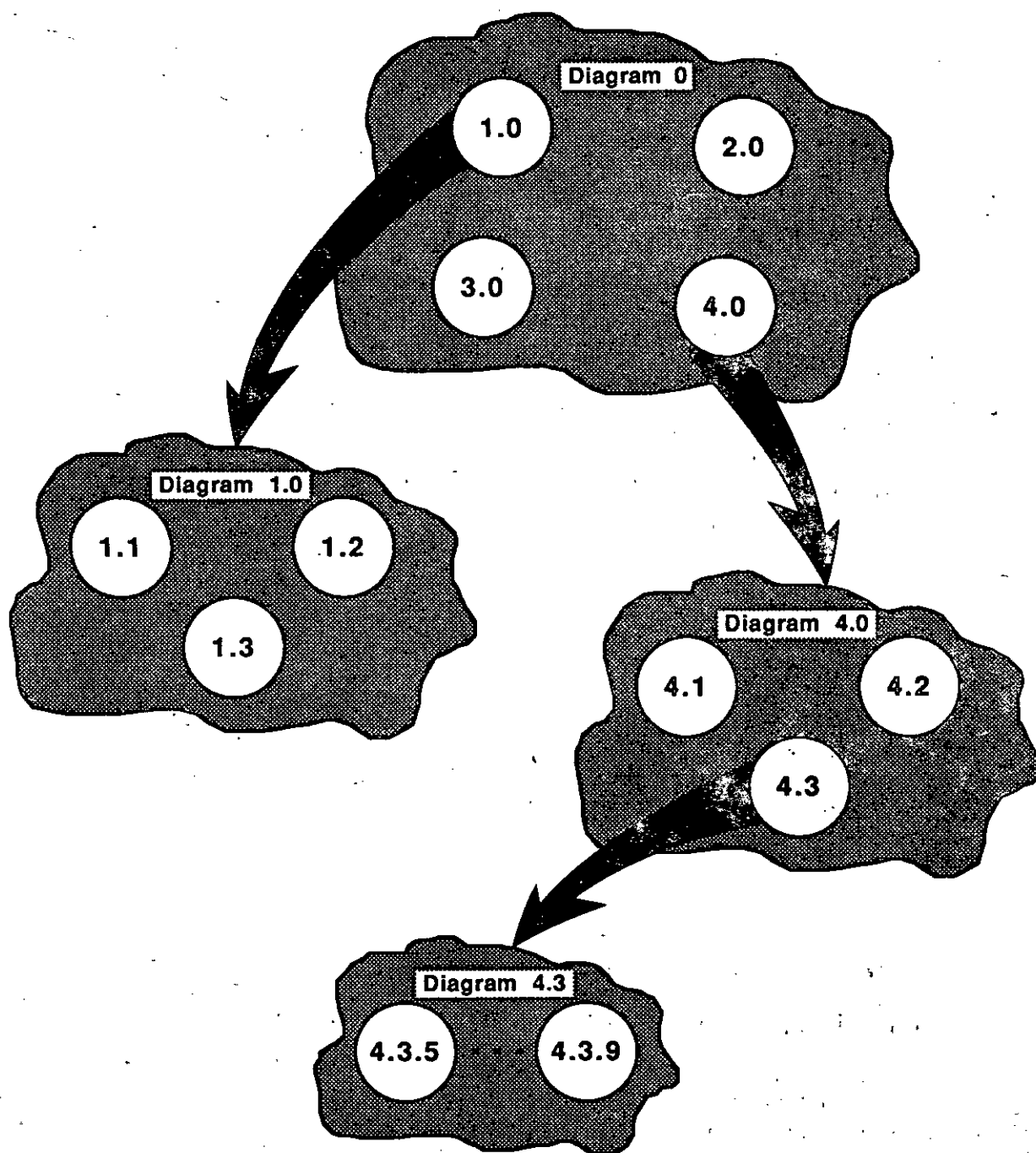


Figure 4-2 Process Decomposition

Data Flow Diagrams may be used to supplement Section 4 of the Software Requirements Specification and included as appendices to the Specification.

It must also be stressed that the products of Structured Analysis do not describe the architecture or design of the system. Some design process, such as Structured Design, must be used to transform the requirements into a design that meets the specified constraints.

For a complete description of Structured Analysis, please see references (1) and (2) identified in Section 8.

Section 2. Structured Design

The following sections discuss Structured Design which is based on concepts originated by L.L. Constantine (and later published by him while working with E. Yourdon) and by G. J. Myers and W. P. Stevens.

2.1 Introduction

Structured Design methods include techniques, strategies, and heuristics for producing "good" designs and graphical representation schemes for documenting the designs. The graphical representations are Data Flow Diagrams and Structure Charts. Data Flow Diagrams are discussed in Section 1. The Structure Chart, which will be discussed in this section, is a modular hierarchy diagram used to record design decisions. It shows how functions are allocated among software units and depicts the resulting unit interfaces.

2.2 Method

As the basis for software design, Structured Design requires the designer to consider the flow of data through a system. The flow of data is then documented in a Data Flow Diagram. As described in Section 1, the bubbles on the Diagram represent the points of transformation where the data must be changed. There may be many intermediate transformations or only one, but ultimately, the data must reach its proper output form. Once the points of transformation are identified, the designer can produce Structure Charts to represent the functions necessary to change the data. To complete the Structure Chart, the functions must be decomposed into subfunctions that when combined will satisfy the requirements of the major function. The goal of this decomposition process is to divide the software into separate pieces that can be built with minimal effect on other parts of the software; or in other words, to produce simple, independent modules.

In addition to identifying software units to perform functions, the software architecture must include units to control the processing flow between functional units. Thus, units near the top of the structure are primarily controlling units. They pass data to subordinate units, control their execution, and receive their outputs. The lower-level units do the actual processing or computation and do not direct other units.

Figure 4-3 depicts the Structure Chart for a program called "Locate Target." The flow of control is shown by the lines connecting the modules. It proceeds downward from left to right. Labeled arrows are used to show the name and direction of data passing between the modules. Arrows with open circles at the end indicate data items that are not control parameters. An arrow with a closed (darkened) circle would indicate the passing of a control flag or switch.

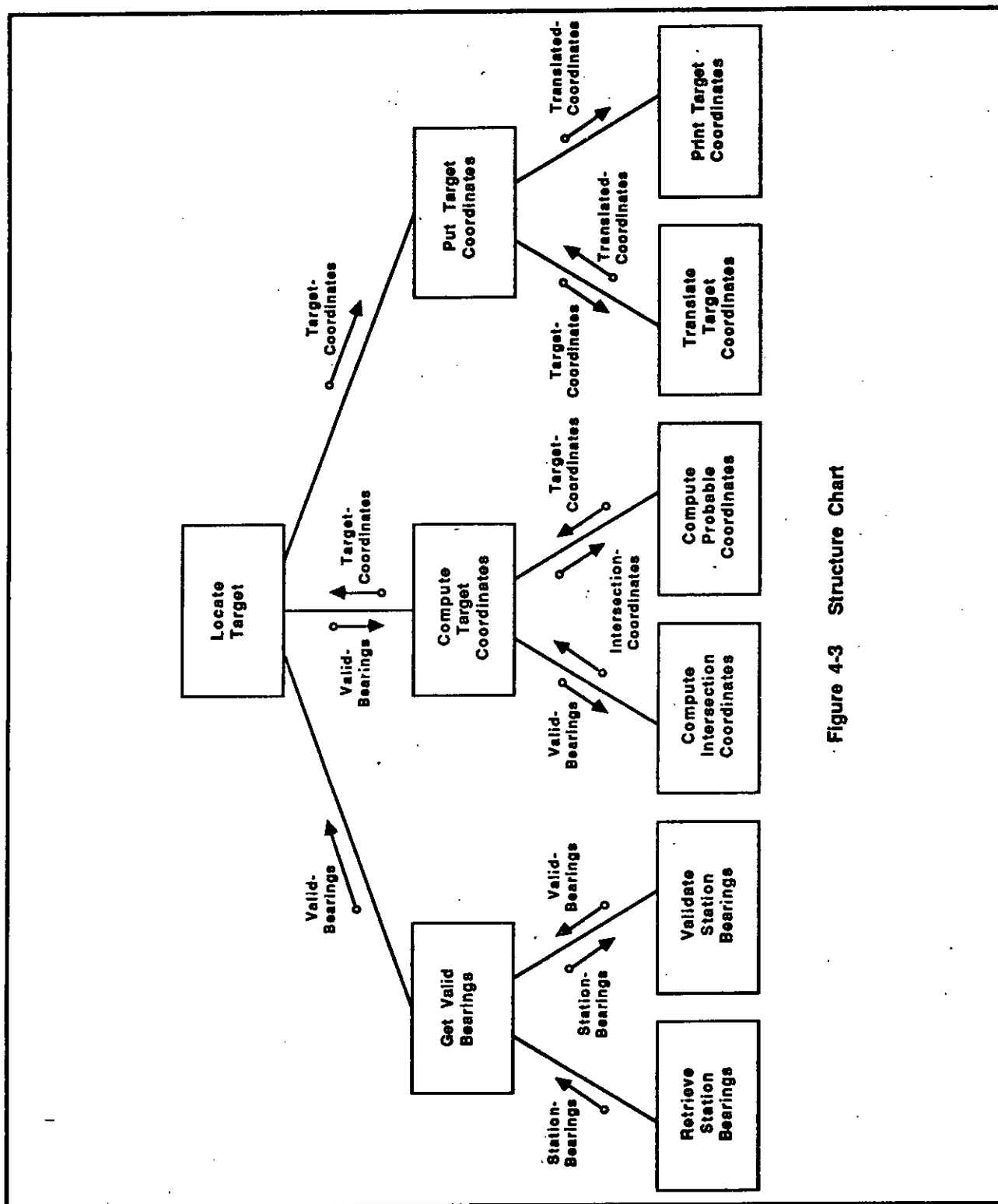


Figure 4-3 Structure Chart

This example shows a calling structure three levels deep, with the second level containing a module for each major function of the program. Generally, a structure chart may be of any width or depth so long as each module contains a single entry and exit point.

2.3 Design Evaluation

Good software design involves much more than the construction of Structure Charts. The first attempts to produce Structure Charts from Data Flow Diagrams probably will not represent the best way to organize software units to accomplish their intended tasks. They may need to be changed or rearranged. Thus, in addition to rules for the construction of Structure Charts, Structured Design provides guidelines for evaluating and improving software designs. Three of the most important are as follows:

Coupling - measures the relationship between modules

Cohesion - measures the substance of a single module

Ordering of modules and decisions.

Using heuristics described in the following sections, each of these design qualities (or attributes) may be evaluated.

2.3.1 Coupling

Coupling is a measure of the relationship between modules. The premises of the coupling guidelines are that the more independent a module, the easier it will be to understand; and the fewer the number of paths along which changes and errors can propagate, the better the design.

Structured Design identifies five levels of coupling. They are listed in order from most to least desirable:

a. Data - The interface level where only the data elements needed as input are passed to a module as explicit parameters. The calling module has no knowledge of how the data will be used.

b. Common Data - The interface level where the required data is part of a larger data collection. The use of common data structures are examples of this level of interface. The flow of data may be impossible to trace through a system designed with common coupling.

c. Control - The interface level where a control field is established; e.g., the use of a variable to indicate the type of processing required. This requires the sending module to set a value in a control field and the receiving module to test the value.

d. External - The interface level where a module receives its input by inspecting and using data variables which have been defined and reside in another module. Files that are shared between modules provide external coupling.

e. Content - The interface level where one module must understand the mechanics of other modules; e.g., to know internal switch settings. Content coupled modules may actually share lines of code.

In reality, the five levels of coupling are not precisely defined or discrete. The implication, however, is that simple module interfaces improve reliability and reduce a software system's sensitivity to change. The goal is to reduce, or loosen, connections between modules.

2.3.2 Cohesion

Cohesiveness is a measure of the strength of association of elements within a module. Good modular designs require that software be decomposed into small, independent functional modules. Measuring cohesion enables designers to recognize functionally cohesive and nonfunctionally cohesive modules. A scale of cohesiveness proceeding from best to worst is as follows:

a. Functional - The case where a module performs a single, discrete, logical transformation. A functionally-bound module is one in which all of the elements contribute to the execution of one and only one task; e.g., sine and cosine functions.

b. Sequential - The case where a program is modularized on the basis of the control structure organization. A sequentially-bound module is one in which the output of one processing element serves as input to the next element; e.g., editing transactions and updating a master file.

c. Communicational - The case where a program is modularized by grouping input and output activities in the same module. A communicationally-bound module is one in which all of the elements operate upon the same input data set and/or produce the same output data; e.g., a module that performs all I/O functions on a specific file.

d. Temporal - The case of creating separate modules to handle time-oriented activities. A temporally-bound module is one whose elements are only related in time; e.g., initialization and termination routines.

e. Logical - The case of modularizing a program by logically grouping activities; e.g., printing a number of different error messages originating in various segments. A "logical" module usually requires a control data element to be passed to it to determine the type of processing to be performed.

f. **Coincidental** - Coincidental cohesion occurs when there is little or no relationship among the elements of a module. It sometimes results from attempts to modularize by arbitrarily segmenting programs based upon the number of statements.

These six levels of cohesion are neither discrete nor exclusive. The objective is to segment programs into functionally cohesive modules whose elements are all strongly interrelated. Structured Design asserts that coupling and cohesion are related. Of the two, however, cohesion is the more important concept. Lower levels of cohesiveness (e.g., coincidental) can be expected to increase coupling (e.g., content) as flags and switches are introduced into the code.

2.3.3 Ordering of Modules and Decisions

In addition to coupling and cohesion, Structured Design includes a heuristic to evaluate the organization of modules and decisions. The heuristic uses two terms, scope of control and scope of effect, which are defined as:

a. **Scope of Control** - The scope of control of a module is the set of all modules that are subordinate to the module in the program organization and the module itself.

b. **Scope of Effect** - The scope of effect of a decision is the set of all modules that contain some code whose execution is based upon the outcome of the decision.

For any given decision, Structured Design asserts that the scope of effect should be a subset of the scope of control of the module in which the decision is located. If the scope of effect is within the scope of control, then all modules affected by the decision are organized together. This minimizes the complexity of program paths and reduces the strength of coupling.

2.3.4 Additional Heuristics

Two additional heuristics for examining program organization are also associated with Structured Design. They are:

a. **Module size** - Modules should be small enough to be understandable and have a cyclomatic complexity of ten or less. (See Section 1.2 of Programming Standards; Software Segmentation, Module Size, and Complexity.)

b. **Reasonable span of control** - The number of modules immediately subordinate to a given module should be no more than 9; preferably no more than 7. Spans of control outside this range usually indicate that the module is too complex.

2.4 Summary

Structured Design does not provide precise rules for designing software. It does provide valuable guidelines for developing and evaluating software designs. It also provides excellent graphical representation techniques for identifying and documenting software units that must be built. Structure Charts are particularly valuable for depicting the software architecture of a system as developed during the preliminary design phase. They also help to define module interfaces. Structured Design is particularly suited for design problems where Structured Analysis has been used to define requirements.

For a complete description of Structured Design, please see references (4), (5), (6), and (7) identified in Section 8.

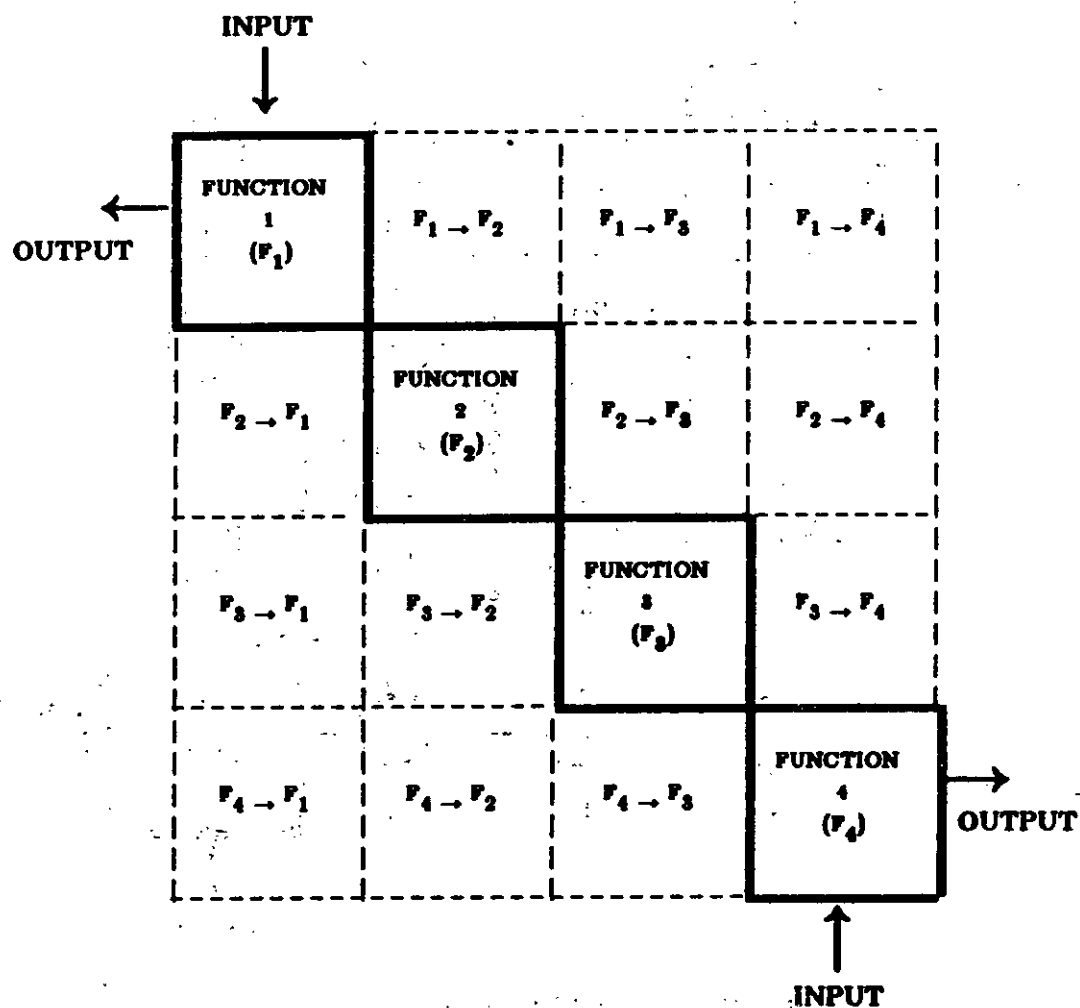
Section 3. N² Charts

N² Charts provide an excellent method for developing and graphically depicting interfaces. They highlight interfaces and interrelationships among system elements and their functions. They can be used to describe interfaces at any level, whether between systems, subsystems, computer programs, components, units or routines. By emphasizing the presence or absence of interfaces, they assist in identifying overly-complex, awkward, and missing interfaces.

The basic N² Chart and four simple rules for its construction are shown in Figure 4-4. System elements are placed on the diagonal. The remaining squares in the NxN matrix represent interface inputs and outputs. Outputs are horizontal (left or right) and inputs are vertical (up or down). A blank intersection indicates that there is no interface. The description of the interface can be given in the intersecting square or the interfaces on the chart can be represented by numbers (Figure 4-5) accompanied by a list describing each interface.

N² Charts are a recommended graphical technique for developing and analyzing interface relationships and dependencies during requirements definition, architectural design, and detailed design.

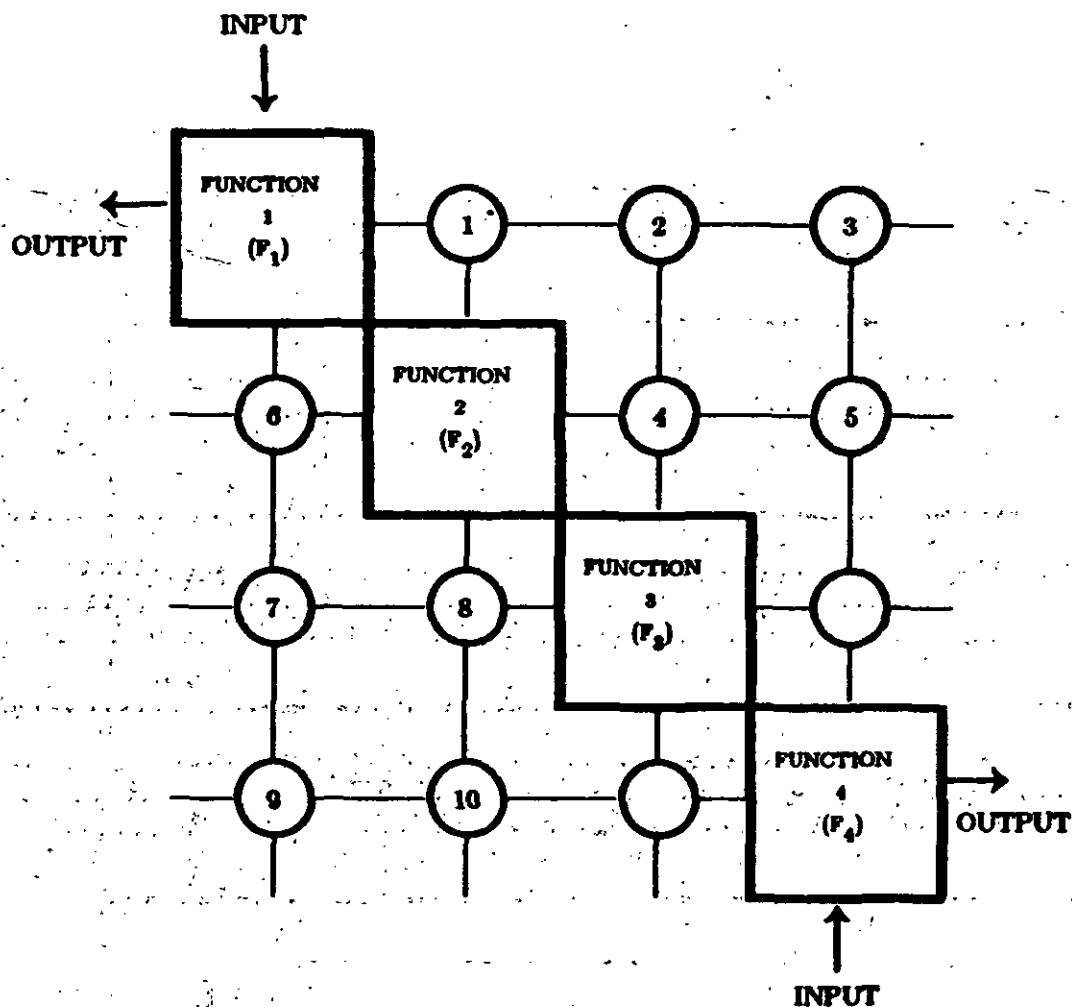
For a complete description of N² Charts, please see references (8) and (9) identified in Section 8.



BASIC N^2 CHART RULES

- ALL FUNCTIONS ARE ON THE DIAGONAL
- ALL OUTPUTS ARE HORIZONTAL (LEFT OR RIGHT)
- ALL INPUTS ARE VERTICAL (UP OR DOWN)
- ALL NON-FUNCTION SQUARES DEFINE ONE-WAY INTERFACES BETWEEN THE ASSOCIATED FUNCTIONS

Figure 4-4 Basic N^2 Chart



DESCRIPTION OF INTERFACE:

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____
9. _____
10. _____

Figure 4-5
N² CHART-CIRCLE FORMAT

Section 4. Real-Time Design Issues

Structured Analysis and Structured Design may not adequately address the design of concurrent process in real-time or near real-time systems. Because software developers often find it difficult to design and implement systems with concurrent processes, interrupts, and severe timing constraints, many techniques for dealing with those problems are currently being investigated. To date, there is little consensus on the best solution. Perhaps the best description of current research on software design methods for real-time systems can be found in the February 1986 issue of the IEEE Transactions on Software Engineering (Reference 3). This is a special issue on software design methods. Many of the articles deal with real-time design issues.

Several techniques described in that issue and in other references may help one to design real-time systems. Three of the most promising involve the use of Abstract Data Types, Petri Nets, or Transformational Schemas. Abstract Data Types (Reference 12) identify design objects as formally specified entities taken directly from the user's environment. The relationships between entities, including real-time issues, may be analyzed apart from the detailed description of the entity. Petri Nets (Reference 13) are graphs which may be used to model and analyze concurrent processes and timing requirements of real-time systems. Transformational Schemas (Reference 14) represent extensions to Data Flow Diagram Notations to include the aspects of timing and system-level control.

Rather than provide detailed discussions here of these or other techniques, designers of systems with real-time requirements are encouraged to recognize the limitations of most current design methods and to be aware of new techniques that help to deal with those limitations. Somehow, real-time systems do get built today, but much work is currently being done to help developers build them better tomorrow.

Section 5. Program Design Language

5.1 Introduction

Program Design Language (PDL) is a structured, natural language design notation for describing the control structure and general organization of computer software. Its primary purpose is to ease the translation of design specifications into computer instructions. It provides a means by which designers can communicate their ideas with programmers, other designers, and users. It serves the same function as a flow chart, but it translates more readily to source code and can be inserted into the source code as comments.

PDL has been characterized as a "pidgin" language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language). It is important to note that PDL's most important feature is the natural language representation of the software design. The "keywords" (structured programming constructs) impose a structure on the design, but it is essentially an English language document. Since the PDL representation is textual in nature and derives its structure from the keywords, PDL designs are more readable and understandable than graphical representations such as flowcharts.

5.2 Description

This section describes the syntax for a set of structures recommended for use as PDL. In the examples, a lower case letter enclosed in parentheses (e.g., (p), (q)) represents a conditional expression written in English (not a programming language). The term "English Language" used in the examples indicates the English language representation of the function performed by the module.

5.2.1 Sequence

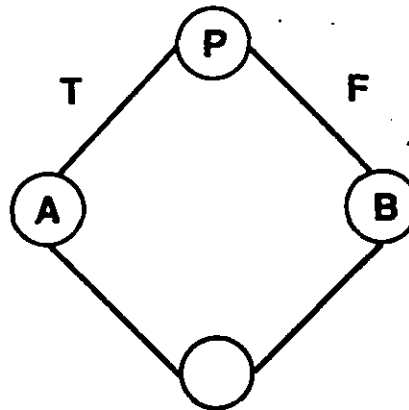
Sequential statements, the basic form of control flow, would have the following language text representation:

```
Get next Master-Record
Determine Transaction-Type
```

```
·
·
·
```

5.2.2 IF-THEN-ELSE Structure

The IF-THEN-ELSE structure causes control to be transferred to one of two functional blocks of code based on the evaluation of the truth of a conditional statement (p) acceptable to the source language. The flowgraph* for the IF-THEN-ELSE structure is:

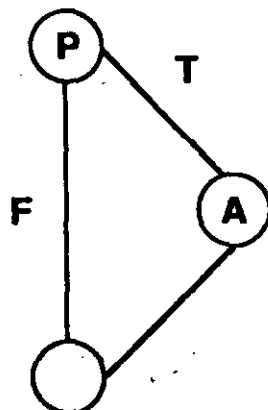


and the language text to be used to represent the IF-THEN-ELSE is:

```
IF (p) THEN
    English Language
ELSE
    English Language
ENDIF
```

***NOTE:** Flowgraphs are used to show flow of control. The flow is always assumed to be downward unless otherwise indicated.

The ELSE and its associated block of code in the IF-THEN-ELSE structure may be regarded as optional; if the ELSE is omitted, the flowgraph becomes:

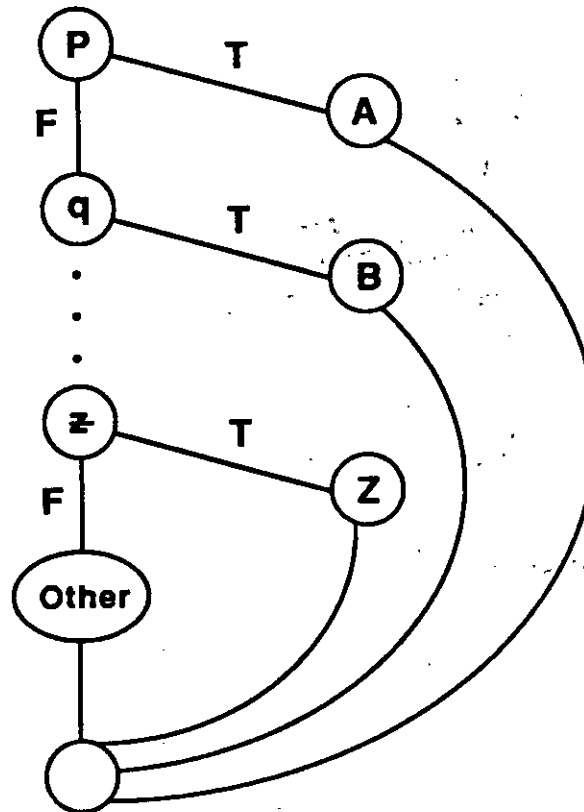


and the language text is:

```
IF (p) THEN
  English Language
ENDIF
```

The keyword THEN is optional in both cases and if present is treated as a comment.

The last keyword in the IF-THEN-ELSE structure is ELSEIF. The ELSEIF is used when parallel conditions, not subordinate conditions, are being tested. The flowgraph for this structure is:



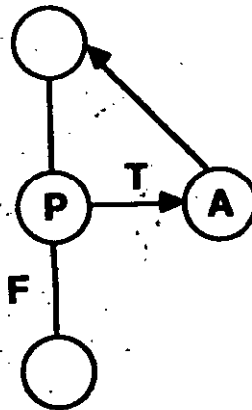
and the language text is:

```

IF (p) THEN
    English Language
ELSEIF (q) THEN
    English Language
.
.
.
ELSEIF (z) THEN
    English Language
ELSE
    English Language
ENDIF
  
```


5.2.3 DO-WHILE Structure

The DO-WHILE structure allows execution of a functional block of code A while condition (p) is true. The flowgraph for the DO-WHILE structure is:

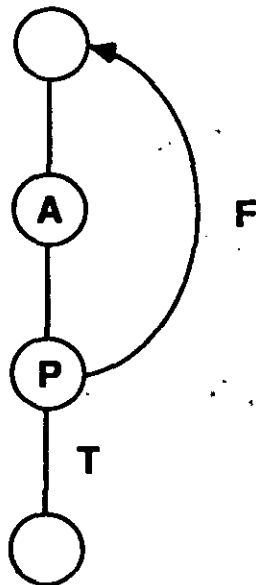


and the language text to be used is:

```
DO WHILE (p)  
  English Language  
ENDDO WHILE
```

5.2.4 DO-UNTIL Structure

The DO-UNTIL structure allows execution of a functional block of code A until a condition (p) becomes true. The functional block of code A is executed at least once. The flowgraph for the DO-UNTIL structure is:



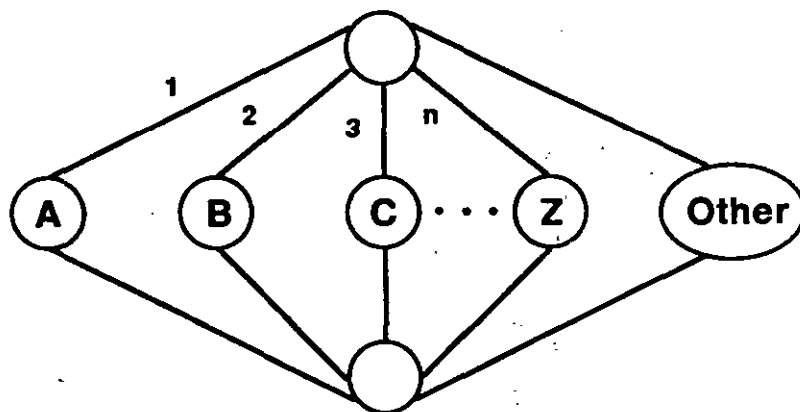
and the language text to use is:

```

DO UNTIL (p)
  English Language
ENDDO UNTIL
  
```

5.2.5 CASE Structure

The CASE structure causes control to be passed to one of the set of functional blocks of code (A, B, ..., Z) based on the value of an integer variable *i*. The flowgraph for the CASE structure is:



and the text to be used to represent the CASE is:

```

DO CASE (i)
  CASE_A:
    English Language
  .
  .
  .
  CASE_Z:
    English Language
  CASE_OTHER:
    English Language
ENDDO CASE
  
```

If the CASE_OTHER (default) is not used, the text which represents the CASE is:

```
DO CASE (i)
  English Language
CASE_A:
  English Language
.
.
.
CASE_Z:
  English Language
ENDDO CASE
```

These two representations of a CASE figure can be expressed by the testing of parallel conditions. For example:

```
DO CASE (i)
CASE1:
  S1
CASE2:
  S2
CASE_OTHER:
  S3
ENDDO CASE
S4
```

is

```
IF CASE1 THEN
  S1
ELSEIF CASE2 THEN
  S2
ELSE
  S3
ENDIF
S4
```

While,

```
DO CASE (i)
CASE1:
  S1
CASE2:
  S2
CASE3:
  S3
ENDDO CASE
S4
```

is

```

IF CASE1 THEN
  S1
ELSEIF CASE2 THEN
  S2
ELSEIF CASE3 THEN
  S3
ENDIF
S4

```

The CASE structure is a multi-branch, multi-join control structure used to express the processing of one of many possible uses. Although it is conceptually represented by testing parallel conditions, its actual implementation in a particular programming language may be quite different. The forms may range from the IF . . . ELSEIF . . . ELSEIF . . . ELSE . . . ENDIF construct to a computed GOTO.

5.2.6 DO Group Exit Structure

When writing structured code, it is sometimes necessary to exit prematurely from a DO structure. Although such a premature exit can be achieved by appropriately setting and testing logical switch variables, more readable programs are usually obtained using the following keywords:

UNDO

The UNDO keyword is used to indicate a premature exit from a DO structure. It indicates a transfer to the first statement immediately following the closing statement of the DO structure to which it is applied.

CYCLE

The CYCLE keyword is used to indicate a premature exit from an iteration of a DO structure. It indicates that control proceeds to the closing statement (ENDDO) of the DO structure to which it is applied. Any subsequent actions take place as if the closing statement (ENDDO) had been normally encountered as the next statement in the DO structure.

Use of UNDO and CYCLE

These features must be used sparingly if high quality programs are to be obtained. UNDO and CYCLE must be looked upon as convenient ways out of a difficult situation and not as a standard way to design programs.

Consider the following example:

```
DO WHILE (p)
  IF (q) THEN
    A
  CYCLE
ENDIF
B
ENDDO WHILE
```

Such CYCLE usage is not recommended because the same algorithm can be expressed in a simpler and strictly structured form as:

```
DO WHILE (p)
  IF (q) THEN
    A
  ELSE
    B
  ENDIF
ENDDO WHILE
```

Similarly:

```
DO WHILE (TRUE)
  S1
  UNDO IF (p)
ENDDO WHILE
```

should instead be written as:

```
DO UNTIL (p)
  S1
ENDDO UNTIL
```

In the preceeding example, the DO WHILE (TRUE) construct represents an infinite loop.

5.2.7 Library Segmentation

The CALL construct facilitates expressing major subfunctions to be performed where another segment is required for specification to a lower level of detail. The CALL represents the use of a subroutine. The language text used to represent this capability is:

```
CALL (English Language)
```

5.3 Complexity Measurement of PDL

The Cyclomatic Complexity Measure discussed in Section 4.4, Programming Standards, may also be applied to PDL. In fact, measuring complexity of PDL is a valuable technique for controlling the testability, simplicity, and maintainability of a design. Using the complexity measure as a guide can help the designer in making several design decisions. It can help partition requirements on a data flow diagram; it can help evaluate the design; and it can help limit the complexity of code.

5.4 Program Design Language Example

The following is an example of PDL as used to describe a simple preprocessor design for an unspecified target language.* The first level specification might appear as:

PREPROCESSOR

```

INPUT CONTROL CARDS
INITIALIZE WORKSPACE
INPUT SOURCE IMAGE INTO BUFFER
DO WHILE SOURCE DECK NOT EMPTY
    DETERMINE STATEMENT TYPE AND PARAMETERS
    GENERATE TARGET CODE
    INPUT SOURCE IMAGE INTO BUFFER
ENDDO
RETURN
```

"PREPROCESSOR" makes references to several subspecifications. Each has been given a unique, descriptive name (e.g., "INITIALIZE WORKSPACE") by which a subsequent refinement may be located at the next design level. The subspecifications can then be expanded into any needed detail at succeeding levels. For example, the next level of design for the "DETERMINE STATEMENT TYPE AND PARAMETERS" subspecification might appear as:

*Material for this example was taken from Standardized Development of Computer Software by Robert C. Tausworthe of the Jet Propulsion Laboratory.

DETERMINE STATEMENT TYPE AND PARAMETERS

```

INITIALIZE POINTER TO FIRST CHARACTER IN BUFFER AND
  ROOT OF TEMPLATE GRAPH
DO WHILE INPUT POINTER IS NOT AT THE END OF THE INPUT
  BUFFER
  IF THE INPUT CHARACTER MATCHES THE TEMPLATE NODE
    CHARACTER THEN ADVANCE INPUT POINTER AND GRAPH
    POINTER
  ELSE
    IF THERE IS AN ALTERNATE TEMPLATE NODE THEN
      EXECUTE GRAPH NODE ACTION CODE FOR
      CURRENT NODE
      SET GRAPH POINTER TO ALTERNATE NODE
    ELSE
      SET STATEMENT TYPE TO "UNRECOGNIZED"
      UNDO
    ENDIF
  ENDIF
  IF THE INPUT BUFFER IS EXHAUSTED AND GRAPH NODE
    IS A LEAF THEN SET STATEMENT TYPE OF LEAF
    NUMBER
  ENDIF
ENDDO
RETURN

```

Note that a PDL description is procedural. Since nonprocedural information such as data structure definitions are absent, all of the information needed to understand the software may not be present. In the preceding example, the "template graph" data structure has not been described and the PDL description is less understandable without that information. The design representation does much to promote understandability, but it is not the whole answer.

The technique has, however, made it possible to describe the algorithms being developed in a structured, procedural, and readable manner before any code has been written. It allows alternative designs to be reviewed for completeness, correctness, etc., before they are committed into formalized documentation and code.

5.5 Program Design Language Standards

When used to document the detailed design of a software unit, PDL becomes part of the Detailed Design specification. The following suggested standards will improve the readability, reliability, and maintainability of the design:

- a. PDL statements should be written independent of the programming language, hardware, and operating system to be used.
- b. The logic of a software element described in PDL flows downward; control should never pass upward.
- c. Statements referencing other elements such as subroutine calls should contain the reason for the call such as:

CALL LOGRTN TO LOG REASON FOR TERMINATION

- d. PDL statements should describe functions and not just reflect the source code statement(s) to be used. For example, the statement:

SET RETURN CODE TO REFLECT A READ ERROR

is preferred to:

SET RETURN CODE TO -1.

- e. When defining an IF-THEN-ELSE construct, the most frequently occurring event will be tested for first. For example, when performing an I/O operation, success rather than errors is more probable. For this reason code for processing a successful I/O operation would precede error handling code within the body of the IF-THEN-ELSE construct.
- f. Conditions that will cause termination of a DO loop should be identified in the DO statement. This requirement will negate the need for the DO FOREVER statement.
- g. IF-THEN-ELSE constructs should be used in place of CASE statements. They more closely reflect English narrative.

Section 6. Decision Table Standards

6.1 Introduction

A decision table is a special form of table that codifies a set of decision rules based on a clearly identified set of conditions and the resulting actions. It represents in tabular form, the following three items which are contained within the body of the decision table.

- a. Conditions - Factors to consider in making a decision.
- b. Actions - Steps to be taken when a certain combination of conditions exist.
- c. Rules - Specific combinations of conditions and the actions to be taken under those conditions.

A decision table is a method of representing requirements or program design and is best applied where there are numerous, detailed, interacting decisions involved in a problem. Its use requires a shift in emphasis from process logic or program design language (both of which emphasize sequence) to a technique that considers the resulting actions as primary.

6.2 Description

6.2.1 Parts of a Decision Table

The various parts of a decision table are:

- a. Condition stub - All or part of a condition statement; i.e., a logical question, or relational or state condition that is answerable by a yes or no.
- b. Condition entry - completion of the condition statement.
- c. Action stub - All or part of an action statement. An explicit statement of the action to be taken.
- d. Action entry - Completion of the action statement.
- e. Rules - Unique combinations of conditions and the actions to be taken under those conditions.
- f. Header - A title and/or code that identifies the decision table.
- g. Rule identifiers - Codes that uniquely identify each rule within a table.

- h. Condition identifiers - Codes that uniquely identify each condition statement/entry.
- i. Action identifiers - Codes that uniquely identify each action statement/entry.
- j. Notes - Comments concerning the contents of the table. Notes are not required, but might be used to clarify table items.

The parts of a decision table are diagrammed in Figure 4-6.

The major steps in developing a decision table are:

- a. Define the specific boundaries of the problem.
- b. Identify the conditions.
- c. Identify the necessary actions.
- d. Identify the rules that define the "if . . . then" relationship between conditions and actions.

6.2.2 Decision Table Form

As seen in Figure 4-7, rules (vertical columns R1, R2, and R3) are the guides for a decision table user. Conditions are matched against those stated in the table, allowing a specific rule to be located which then identifies the actions to be taken.

6.2.3 Types of Decision Tables

There are three types of decision tables: limited entry, extended entry, and mixed entry. These three types are defined as follows:

- a. Limited Entry Table (Figure 4-7) - the entire condition or action must be written in the stub. The entry is used to show whether a particular condition is true, false, or not pertinent (Y, N, or - (or blank)), and whether a particular action should (or should not) be performed (X or - (or blank)).
- b. Extended Entry Table (Figure 4-8) - Part of the condition and action statements are extended into the condition and action entries.
- c. Mixed Entry Form (Figure 4-9) - Combines both limited and extended entry forms. The two forms may be freely mixed within a table, but a single condition or action row must be in just one format.

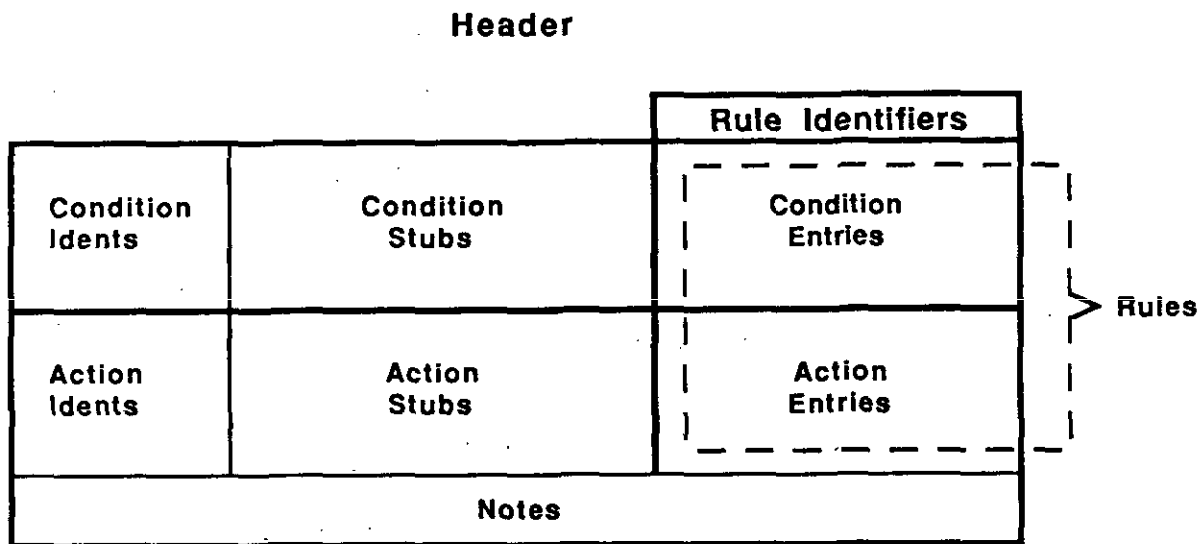


Figure 4-6 Decision Table Parts

Credit Order Approval Procedure

		R1	R2	R3
C1	Credit Limit OK	Y	N	N
C2	Pay Experience Favorable	—	Y	N
A1	Approve Credit	X	X	—
A2	Return Order to Sales	—	—	X
Note: Be certain that the latest credit information is used.				

Figure 4-7 Limited Entry Decision Table

Credit Order Approval Procedure

		R1	R2	R3
C1	Credit Limit	OK	Not OK	Not OK
C2	Pay Experience	—	Favorable	Unfavorable
A1	Credit Action	Approve	Approve	Don't Approve
A2	Order Action	—	—	Return to Sales

Figure 4-8 Extended Entry Decision Table

Credit Order Approval Procedure

		R1	R2	R3
C1	Credit Limit OK	Y	N	N
C2	Pay Experience	—	Favorable	Unfavorable
A1	Credit Action	Approve	Approve	Don't Approve
A2	Return to Sales	—	—	X

Figure 4-9 Mixed Entry Decision Table

Section 7. Summary and Recommendations

The methodologies described in this section may be used during both the analysis phase and the design phase. The chart below identifies each method and the software development activities where they are most appropriate.

METHOD	REQUIREMENTS ANALYSIS	PRELIMINARY DESIGN	DETAIL DESIGN
Structured Analysis	X	X	
Structured Design		X	
N2 Charts	X	X	X
Program Design Language			X
Decision Tables	X	X	X

Structured Analysis can be used for any application. However, it tends not to be sufficient for very large, complex real-time software efforts. Also, data flow diagrams have no mechanism for accurately specifying parallel aspects of processing. For these applications, or applications using real-time interrupts, the additional real-time design techniques mentioned in Section 5 should be used to supplement Structured Analysis to show processing requirements related to concurrency.

Structured Design is recommended for developing and representing the architectural design of a software system. Structured Design, is, however, primarily concerned with organizing program modules; other techniques must be used to specify the actual content of each module. Program Design Language (PDL) is an excellent technique for the detailed design of individual modules established by using the Structured Design technique.

N2 charts can be useful additions to the structured methods when the data flows and interfaces are large and complex. They highlight the interfaces between units, programs, or systems and can effectively provide traceability for large numbers of interfaces that may otherwise become overwhelming.

Decision tables may be used as process logic in the analysis phase or as a supplement to Program Design Language during detailed design. They are well suited for showing complex combinations of inputs that would result in long, deeply nested descriptions in PDL. Decision tables are also valuable for developing test cases since each column is, in effect, a test case.

Section 8. References

Structured Analysis

- (1) DeMarco, Tom, Structured Analysis and System Specification, New York, Prentice Hall, 1979.
- (2) Gane, C., Sarson, T., Structured Systems Analysis: Tools and Techniques, McDonnell Douglas Corp., 1982.

Software Design

- (3) IEEE Transactions on Software Engineering, Vol SE-12, Number 2, February 1986, Special Issue on Software Design Methods.
- (4) Stevens, W.P., Myers, G.J., Constantine, L.L., "Structured Design," IBM Systems Journal, Vol 13, No. 2, 1974, pp. 115-139.
- (5) Stevens, W.P., Myers, G.J., Constantine, L.L., Structured Design, New York, Yourdon, Inc., 1975.
- (6) Yourdon, E., Constantine, L.L., Structured Design, New York, Yourdon, Inc., 1975.
- (7) Myers, G.J., Reliable Software Through Composite Design, New York, Petrocelli/Charter, 1974.

N2 Charts

- (8) Lano, Robert J., A Technique for Software Systems Design, Elsevier North-Holland Inc., 1979.
- (9) Class Notes, from a seminar conducted by the National Cryptologic School, titled "Software Requirements Analysis and Specification," taught by Robert J. Lano.

Real-Time Software Design

- (10) Booch, G., Software Engineering with Ada, Menlo Park, Benjamin/Cummings, 1983.

- (11) Agerwala, T., "Putting Petri Nets to Work," Computer, December 1979, pp. 85-94.
- (12) B.H. Liskov and S. Zilles, "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, Volume SE-1, Number 1, March 1975.
- (13) Coolahan, J.E. and Roussopoulos N., "A Timed Petri Net Methodology For Specifying Real-Time System Timing Requirements," Proceedings of the International Workshop on Timed Petri Nets, Torino, Italy, July 1985, IEEE Catalog Number 85CH2187-3.
- (14) Ward, P.T., "The Transformational Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," IEEE Transactions on Software Engineering, Volume SE-12, Number 2, February 1986.

Program Design Language

- (15) PDL/81 Design Language Reference Guide, Caine, Farber and Gorden, Inc., 1981.
- (16) PDL/81 Document Language Reference Guide, Caine, Farber and Gorden, Inc., 1981.

Decision Tables

- (17) Hurley, R., Decision Tables in Software Engineering, New York, Van Nostrand Reinhold, 1983.

4.2 UNIT DEVELOPMENT FOLDERS

4.2.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 4.2)

Software developers shall prepare and maintain a Unit Development Folder (UDF) for each software unit produced by the developer in order to provide: (1) an organized, accessible collection of all requirements, design data, code, and test data pertaining to that unit, as these data are produced; and (2) unit-level schedules and status information. A UDF must be established for each unit as soon as the unit is identified and maintained in an accessible location throughout the software acquisition activity. They may be organized in notebooks, folders, or on-line files.

A UDF shall contain, as a minimum: the unit requirements; preliminary unit design (including storage and timing budgets); detailed unit design; current unit code; unit test plans, procedures, data, and test results; and reviewers' comments applicable to that unit.

4.2.2 GUIDANCE

The Unit Development Folder contains the material applicable to one software unit. Section 6 of the Software Standards and Practices Manual establishes criteria for breaking down the project's software into units that are appropriately sized and configured. Section 3.2 of the Software Configuration Management Plan should establish rules for naming and identifying software units. A unit may be at any level of the software hierarchy. Typical criteria for selecting units are:

- a. The unit performs a well-defined function;
- b. The unit is amenable to development by one person within the assigned schedule;
- c. The unit is an aggregate of software to which the satisfaction of requirements can be traced;
- d. The unit is amenable to thorough testing;
- e. The unit is appropriately sized. Typical units are between 100 and 1000 higher-order language statements. Units may, however, be smaller or larger depending on specific project needs.
- f. During detailed design, the unit has an estimated cyclomatic complexity of 7 or less; during coding, the unit has a cyclomatic complexity of 10 or less.* NOTE: If the detailed design results in a unit with a complexity greater than 7, the unit should be decomposed into two or more routines, each with complexity of 7 or less. When the routines that make up the unit are coded, then each routine should have a complexity of 10 or less.

*SOURCE: NBS Special Publication 500-99 "Structured Testing: A Testing Methodology Using the McCabe Complexity Metric," by Thomas J. McCabe, 1982, p.12

The Software Development Manager may adapt the organization and content of Unit Development Folders to reflect needs of the project. Sections of the UDF may be assigned to different people or they may all be assigned to one person. Sections may be expanded, contracted, or resequenced to suit specific situations. Some considerations for selecting the structure of a UDF are:

- a. Each section should contribute to the visibility and management of the development process;
- b. The content and format of each section should be clearly defined;
- c. The structure should be flexible enough to apply to a variety of types of software units;
- d. As nearly as possible, the sections should be chronologically ordered.

Usefulness of the Unit Development Folders depends upon the realism of the schedule and the participation and commitment of the unit developers. It also depends upon the interest and concern that software managers demonstrate in monitoring and achieving the milestones identified on the cover sheet.

4.2.3 FORMAT FOR UNIT DEVELOPMENT FOLDERS

1. Cover Sheet. Figure 4-16 is a sample cover sheet for a Unit Development Folder. Each UDF cover sheet shall identify the name of the computer program, the name of the computer program component, and the names of the routines included in the unit. The cover sheet shall identify the sections selected for the UDF and describe the section. Each section shall include space for the following:

- a. Schedule due date;
- b. Date actually completed;
- c. Name of person responsible for completing the section (originator);
- d. Name of reviewer and date of review.

For multiple routine units, a one-page composite schedule illustrating section schedules of each section may be included.

2. UDF Change Log. Figure 4-11 is a sample UDF Change Log. The UDF Change Log shall be included to document all UDF changes after the initial development is completed and the unit is put into a controlled test or maintenance environment.

3. References. There will be times when code listings, test outputs or other computer output are too large to be included in the UDF binder. When this happens, the material may be placed in a separate, clearly-identified location. Relevant sections of the UDF shall then identify the location. Figure 4-12 illustrates a typical reference log with spaces for identification of updates made after the initial document is completed.

4. Sections of the Unit Development Folder

Section 1. Requirements. This section identifies the specific requirements that are to be satisfied by the unit. It should also refer to the higher-level requirement (document and section number) from which it was derived.

If unit or higher-level requirements change during development of the unit, the new requirements statement shall be added and recorded. Any assumptions, ambiguities, interpretations or conflicts concerning the requirements shall be stated, reviewed, and approved as part of the requirements section.

Section 2. Design Description. This section contains the design description of the unit including the preliminary design, the detailed design, and the "as built" design. During the development process, this section contains the

UNIT DEVELOPMENT FOLDER COVER SHEET						
COMPUTER PROGRAM		COMPONENT		UNIT NAME		ROUTINES INCLUDED
CUSTODIAN						
SECTION NUMBER	DESCRIPTION		DATE		ORIGINATOR	REVIEWER/DATE
			DUE	COMPLETED		
1	REQUIREMENTS					
2	DESIGN	PRELIMINARY				
		DETAILED				
		INSPECTION				
		AS-BUILT				
3	FUNCTIONAL CAPABILITIES LIST					
4	UNIT CODE	CLEAN COMPILE				
		INSPECTION				
		COMPLETE				
5	UNIT TEST PLAN, PROCEDURES AND TEST DATA					
6	TEST CASE RESULTS					
7	PROBLEM REPORTS	DESIGN				
		CODE				
8	NOTES					
9	REVIEWER'S COMMENTS					

EXAMPLE OF A UDF COVER SHEET

Figure 4-10

EXAMPLE OF REFERENCE LOG FOR MATERIAL IN A SEPARATE LOCATION

Figure 4-12

current working version of the design. It must be maintained and annotated as changes are made to the initial or preliminary design until the detailed design documentation is completed and approved as being a "code-to" specification for the unit. The format and content of this section should be suitable for inclusion in the Program Specification. When integration testing is completed, the design should be updated to reflect the "as-built" design. This section shall also contain allocated storage and timing budgets and be updated to reflect actual budgets.

When the initial detailed design is completed and ready to be reviewed, a design inspection for the unit shall be conducted (Policy 4.3). Section 2 cannot be completed until the design inspection has been successfully completed.

Section 3. Functional Capabilities List. This section is a list of testable functions performed by the software unit. It describes what things a particular unit does, preferably in sequential order. Normally, an adequate level of breakdown has been achieved if each testable capability can be described in one or two sentences of ordinary length.

Because the Functional Capabilities List is generated from the requirements of Section 1 and the detailed design of Section 2, it provides the basis for planned and controlled unit-level testing. In some cases, however, some of the data processing functions will only be indirectly related to unit requirements. When this happens a convenient breakdown may relate to major segments of code and/or decision points.

The Functional Capabilities List provides a consistent approach to testing which can be reviewed, audited, and understood by an outsider. Items in the list should be mapped to test cases to provide the rationale for each test.

Section 4. Unit Code. This section contains the current source code listings for each of the routines in the unit. For units with multiple routines, sub-section separators should be used. The completion date for this section is the scheduled date for the first error-free compilation or assembly. After the first error-free compilation or assembly and a code inspection are completed, the code is ready for unit-level testing. If unit code is contained in a separate location, this section must contain a cross reference to the location, including where it is located.

When the developer completes a clean compile of the unit code, a code inspection shall be conducted (Policy 4.3). Section 4 cannot be completed until the code inspection has been successfully completed.

Number 5. Unit Test Plan, Procedures and Test Data. This section is a description of the testing approach for the unit. The Unit Test Plan must conform to the standards established in the General Test Plan (Policy 5.1). An introductory paragraph shall describe the testing approach for the unit. Included for each test case of the unit shall be the following:

- a. Identification of any test tools or drivers used;
- b. Description of all test input data and/or drivers for the case;
- c. Description of expected outputs, including numerical or other demonstrable results;
- d. Acceptance criteria for the case;
- e. Identification of the requirements or functional capabilities demonstrated by the case;
- f. Test scenario description explaining how the tests will be executed.

This section shall also include a summary to demonstrate that all testable requirements and functional capabilities have been tested. Figure 4-13 gives a sample format for demonstrating completeness. It shall also identify the test data to be used in the unit test.

Number 6. Test Case Results. This section contains documentation that testing has occurred as described in Section 5. This documentation is a compilation of all currently valid test case results and analyses necessary to demonstrate that testing is complete. Test outputs should be identified by test case number and listings should be clearly annotated to facilitate review of the results. If test drivers, test tools, data bases and unit code are revised during unit testing, revision status should be recorded to facilitate retesting. It may be necessary to record test outputs in separate binders.

Number 7. Problem Reports. This section contains status logs and copies of Design Problem Reports, Design Analysis Reports and Discrepancy Reports (as required) to document all design and code problems and changes subsequent to baselining. This insures traceability for all problems and changes. Individual status logs should summarize actions and disposition of all problems.

[illegible]

4-51

Number 8. Notes. This section contains any memos, notes, reports, etc., which relate to the contents of the unit or to problems and issues involved.

Number 9. Reviewer's Comments. This section is a record of reviewers' comments from the section-by-section review and sign-off and from scheduled independent audits. Comments are usually also provided to project and line managers responsible for development of the unit.

4.3 SOFTWARE DESIGN AND CODE INSPECTIONS

4.3.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 4.3)

Software developers shall conduct unit design and code inspections to facilitate the early detection of errors. These inspections shall be accomplished by having the software unit design and code reviewed by a team of individuals (normally between 3 and 7 people) who have technical abilities to review the material.

4.3.2 INTRODUCTION

Software inspections are formal peer reviews of design and code. Although similar to walk-throughs, they are more structured and make more use of inspection results than walk-throughs. An excellent comparison of the two peer-review techniques was made by Michael E. Fagan in his article, "Design and Code Inspections to Reduce Errors in Program Development," published in the IBM Systems Journal in 1976 (Vol. 13, Number 3):

Walk-throughs ... are practiced in many different ways, in different places, with varying regularity and thoroughness. This inconsistency causes the results of walk-throughs to vary widely and to be nonrepeatable. Inspections, however, having an established process and a formal procedure, tend to vary less and produce more repeatable results.

The basic objectives of inspections are to detect errors and ambiguities in software products close to the time that the products are first produced. Designers and programmers find out almost immediately what types of errors they are making. By being a part of the inspection process, they learn how to find their own mistakes and how to improve the quality of their work. In addition to improving quality, inspections can actually reduce development time by reducing the time and effort required for error correction during the various levels of software testing. They allow participants to learn much about the software products in a short time and help them to handle subsequent development and testing with more confidence and fewer false starts.

4.3.3 THE INSPECTION PROCESS

The following sections describe how inspections should be conducted and identifies who should participate in them. The types of inspections and the six formal steps for conducting an inspection will also be described. Much of the information was taken from the following publications:

a. "Inspections in Application Development - Introduction and Implementation Guidelines," IBM Technical Newsletter, Number GN20-3184, 1981.

b. "Design and Code Inspections to Reduce Errors in Program Development," M.E. Fagan, IBM Systems Journal, Volume 15, Nr. 3, 1976, pp. 182-211.

4.3.3.1 PARTICIPANTS IN THE INSPECTION PROCESS

The inspection team is normally composed of the moderator, the author (developer), and one or more inspectors. Inspectors typically include the technical people responsible for the prior and succeeding development phases. For example, detailed design inspections should include the developer of the preliminary design, the person who will code the unit, and the person responsible for testing the unit. An inspection team should be made of three to seven people.

The moderator controls the inspection activities and is the leader of the inspection meeting. The moderator schedules each step of the inspection process, selects the inspectors, prepares inspection reports, evaluates error rates, determines whether a reinspection is required, and assigns each person a role. A "reader" must be appointed to read the material aloud. A recorder should keep a problem list of all errors. Another person may be assigned to find all standard violations.

4.3.3.2 TYPES OF INSPECTIONS

Inspections are most commonly used immediately following detailed unit design and unit coding. Preliminary design, test plans, test cases, and manuals may also be inspected. Documentation inspections can also be used to review requirements.

4.3.3.2.1 DETAILED DESIGN INSPECTIONS

The purpose of the Detailed Design Inspection is to find errors and ambiguities in the detailed design materials. No attempt should be made to correct the errors during the inspection. Major emphasis should be placed on examining the design for errors in logic, external linkages, and data areas.

In order to conduct a detailed design inspection, certain documents must be complete and ready for the inspection. A completed copy of the detailed design for the unit and any supporting prose discussing external linkages or control blocks are necessary. It also helps to have the preliminary design documentation on hand to compare with the detailed design. A detailed design inspection checklist which contains a set of prompters or clues to help uncover errors should be used at this inspection. Figure 4-14 is an example of a detailed design inspection checklist.

Logic (LO)

- .Are all constants defined?
- .Are all unique values explicitly tested on input parameters?
- .Are values stored after they are calculated?
- .Are all defaults checked explicitly: for example, blanks in an input stream?
- .If character strings are created, are they complete? Are all delimiters shown?
- .If a keyword has many values, are they all checked?
- .Are all keywords tested in a macro?
- .Are all keyword-related parameters tested in a service routine?
- .Are all increment counts properly initialized (0 or 1)?
- .After processing a table entry, should any value be decremented or incremented?
- .Is provision made for possible processing at logical checkpoints in the program (end-of-file, end-of-volume, etc.)?
- .Is all I/O performed on opened files?
- .Are routine error conditions adequately provided for (INVALID KEY, ON SIZE ERROR, etc.)?
- .Are literals shown where there should be constant data names?
- .On comparison of group items, should all fields be compared?
- .Is the value of a data item used before the item is initialized?
- .Are all data areas shown in design necessary or are some extraneous?

Data Area Usage (DA)

- .If design is dependent on building/creating/deleting various data areas, are they all designated?
- .Should a called macro provide any INCLUDEs for any data areas that the macro expanded code may depend on?
- .Does design show explicitly which area to use in a data area; that is, if there are multiple save areas?
- .If the program stores into a data area, does it store into the correct field?
- .If a value is fetched from a data area, is the correct field fetched?
- .Should the data area be boundary-aligned?
- .Does a save area have multiple uses? Can conflicts arise?

SAMPLE DETAILED DESIGN INSPECTION CHECKLIST**FIGURE 4-14**

Test and Branch (TB)

- .Are all three conditions tested; that is, greater than, equal to, and less than zero?
- .After a linkage, should a return code be tested?
- .Is a SORT or a MERGE operation tested for successful completion?
- .Are branch legs correct; that is, should YES be NO and NO be YES?

Return Codes/Messages (RM)

- .Are messages issued for all error conditions?
- .On exits, should a return code be set or a message issued?
- .Does the message say what it means?
- .Could more information be supplied in the message?
- .Do return codes in the design for particular situations match the global definition of the return code as documented?

Register Usage (RU)

- .If a specific register is required, is it specified?
- .Does any macro expansion use a register already in use without saving the data?
- .Is the integrity of all input registers maintained?

More Detail (MD)

- .Does the design specify a process ambiguously, or does the process require more than ten instructions?

External Linkages (EL)

- .Should a standard linkage be used rather than coding a subroutine inline?
- .Is the designated linkage the right one for the function to be performed?
- .Is the data area mapped as the receiving module expects it to be?

Standards (ST)

- .Are any programming standards for the project in jeopardy of compromise because of the design?

Initial Design Documentation (HL)

- .Does the detailed design match the initial design?
- If not, the initial design documentation could be in error.

Performance (PE)

- .Does the design impair the performance of this module to any significant degree?

Figure 4-14 (continued)

The Software Development Manager defines the exit (or completion) criteria for the inspection. The exit criteria define the things that must be complete before the software product can proceed to the next development phase. Without explicit exit criteria, it is not possible to certify that any given phase (for example, detailed design) is complete. Two examples of detailed design inspection exit criteria are: PDL will be written to the level where one line of PDL will equate to 3 to 10 lines of actual code; and all rework from the detailed design inspections will be complete and verified.

4.3.3.2.2 CODE INSPECTIONS

The Code Inspection ensures that the code is correct and that the code matches the design. The participants in this inspection must include the moderator and the developer of the unit being inspected. Other inspectors may include the designer of the unit, the team leader, and the person who will be responsible for maintenance or testing of the unit.

Program listings (the first clean compile/assemble) and prologues must be complete before a code inspection. Other materials are the detailed design documentation and any design change requests. There should also be a code inspection checklist specific to the language of the code being inspected. This checklist will prompt the inspectors to find common error types in the code listing. Figure 4-15 presents a guide to the types of questions that should be in a code inspection checklist.

Some examples of the code inspection exit criteria are: all code listings are sufficiently commented and all coding standards have been followed; the code accurately implements the detailed design; and all rework from the code inspections is complete and verified.

4.3.3.2.3 OTHER INSPECTIONS

The inspection types discussed above are the most commonly used. Inspections may also be used to ensure the correctness of other deliverables. It may be valuable to inspect the Software Requirements Specification with the user present to ensure that all of the user's requirements are being satisfied and that the requirements are understood by everyone. Test plans and test cases may be inspected to ensure a complete, accurate, and comprehensive function verification. Even publications like the User's Manual can be inspected for clarity and completeness.

4.3.3.3 FORMAL INSPECTION STEPS

An inspection consists of six formal steps: Planning, Overview, Preparation, Inspection Meeting, Rework, and Follow-up.

FORMAT:

- Are nested IF's indented properly?
- Are comments accurate and meaningful?
- Are meaningful labels used?
- Does the code essentially correspond to the outline of the module in the prologue?
- Are installation programming standards followed?

ENTRY & EXIT LINKAGES:

- Are initial entry and final exit correct?
- For each external call to another module:
 - Are all required parameters passed to each called module?
 - Are the parameter values which are passed set correctly?
- Are subroutines entered and exited properly?

PROGRAM LANGUAGE USAGE:

- Is the optimal verb or set of verbs used?
- Is the installation-defined restricted subset of the language used throughout the module?

STORAGE USAGE:

- Is each field initialized properly before its first use?
- Is the correct field specified?
- Is each field declared as the correct variable type?

TEST AND BRANCH:

- Is the correct condition tested?
- Is the correct variable used for the test?
- Is each branch target correct and exercised at least once?

PERFORMANCE:

- Is logic coded optimally?
- Are normal error/exception routines provided?

MAINTAINABILITY:

- Are listing controls utilized to enhance readability?
- Are labels and routine names consistent with the logical significance of the code?

LOGIC:

- Has all design been implemented?
- Does the code do what the design specified?
- Is each loop executed the correct number of times?

GENERAL CODE INSPECTION CHECKLIST

THE PLANNING STEP

During this step, the moderator chooses the participants for the inspection and ensures that the inspection materials are distributed to the inspection team. The moderator also schedules the overview and inspection meetings, allowing sufficient study time for the inspectors.

THE OVERVIEW MEETING

The purpose of the overview meeting is to educate the inspection team. The designer or coder of the unit should describe the unit's major functions and functional relationships and give a detailed description of the materials. This meeting is attended by the inspection team and any other project personnel who may need a more detailed understanding of the unit's function. This meeting should last no longer than one hour.

THE PREPARATION STEP

All participants prepare for the inspection meeting individually. Their goals are to become thoroughly familiar with the inspection materials and to identify potential defects that should be discussed at the inspection meeting. While examining the inspection materials, the inspectors make sure that the work product being inspected matches the materials from the previous phase. For example, the unit code should not deviate from the detailed design specification.

The time required to prepare for the inspection meeting will vary depending upon the type of materials being inspected. Preparation for a detailed design inspection should take about one hour for every 100 lines of PDL. For a code inspection, preparation should take about one hour for every 125 lines of code.

THE INSPECTION MEETING

Only the inspection participants should attend this meeting. Other project personnel may interfere with the objective of the meeting - to find errors. At the beginning of the meeting, the moderator describes the sequence in which the materials are to be inspected. The moderator conducts the meeting, not the author. The author's role is usually limited to answering technical questions.

During the Planning step, the moderator appoints one of the inspectors to read aloud the inspection materials. It is usually more effective for the reader to paraphrase the materials instead of reading them verbatim. Paraphrasing tends to keep the other participants more alert and helps the author determine whether the materials can be understood. As the reading proceeds, each participant looks for errors or ambiguities in the materials and for adherence to the exit criteria.

As errors are found, the recorder (if a recorder was not appointed, the moderator performs this function) records them in a problem list and classifies them by error type, error category, and error severity. The error type is used to tell whether there was an error in the logic, prologue, program language usage, data area usage, etc. Each code or design error is placed into one of three error categories; missing, wrong, or extra. The error severity is either major or minor. An error which causes the unit to malfunction or which would cause incorrect results to be attained is considered a major error. Examples of minor errors would be standards violations or defects in prologues or code comments. Figure 4-16 is an example of a Detailed Design problem list and Figure 4-17 is an example of a Code Inspection Problem List. Problem lists are hand-written during the inspection meeting. As errors are found, they are recorded in the problem list. Each entry should contain the error type, error category, error severity, and a short description of the error.

After the inspection is complete, the moderator seeks the team's agreement on the problem list and decides whether a reinspection is required. This decision is based upon a project's standards. For example, an organization may decide that the detailed design material may not exceed an error rate of 5% (5 defective lines of PDL for every 100 lines of PDL inspected) and that code may not contain more than one major problem per 25 lines of source code. Within one day of the inspection meeting, the moderator distributes to the participants the module detail report which summarizes the number and types of errors found and states whether a reinspection is required. The problem list is attached to this report.

At the detailed design inspection, materials are examined for consistency with the initial design, correctness of every logic path through the unit, and ambiguities in the design statements which could lead to coding errors. The emphasis is on detecting omissions.

Code inspections emphasize the detection of wrong rather than missing or extra code. The correctness of structured code may be verified by either of two methods. The best method is to trace main line logic through every subroutine until the main line logic has been completely traced. Then all remaining secondary paths are traced. Another method is to trace the code in a sequential page order, starting with the main line segment, followed by the lower-level segments.

An inspection meeting should never last longer than two hours. The participants become less efficient at finding errors after two hours. At a detailed design inspections, about 130 lines of PDL can be reviewed per hour. For a code inspection, about 150 lines of code can be reviewed every hour.

THE REWORK STEP

In the rework step, the author corrects the problems specified in the problem list. The moderator estimates the time it should take to correct the problems in the inspected materials and schedules the follow-up meeting accordingly.

Example error types: RM - Return Codes/Messages-
 DA - Data Area Usage
 MD - More Detail
 EL - External Linkages
 LO - Logic
 MN - Maintainability
 PE - Prologue/Prose
 ST - Standards
 RU - Register Usage
 PD - Preliminary Design Documentation
 US - User Specifications
 TB - Test and Branch
 OT - Other

Error categories: M - Missing
 W - Wrong
 E - Extra

Error severity: MAJ - Major
 MIN - Minor

The following are some sample entries from a detailed design inspection problem list:

1. LO/W/MAJ PDL line 30: Initialize NAME field to all blanks.
2. TB/W/MAJ PDL line 105: Should test for NAME = spaces, not NAME not = spaces.
3. DA/W/MIN PDL line 4: NAME field should be alphabetic, not alphanumeric.
4. ST/W/MIN Throughout the document, the PDL keywords should be in capital letters.

SAMPLE DETAILED DESIGN INSPECTION PROBLEM LIST

FIGURE 4-16

Example error types: CC - Code Comments
 DA - Data Area Usage
 DE - Design Error
 EL - External Linkages
 LO - Logic
 MN - Maintainability
 PE - Performance
 PR - Prologue
 PU - Programming Language Usage
 RU - Register Usage
 SU - Storage Usage
 TB - Test and Branch
 OT - Other

Error categories: M - Missing
 W - Wrong
 E - Extra

Error Severity: MAJ - Major
 MIN - Minor

The following are some sample entries from a code inspection problem list:

1. LO/W/MAJ Line 169: While counting the number of leading spaces in NAME, the wrong variable (I) is used to calculate "J".
2. PU/E/MIN Line 175: In NAME-CHECK, the check for SPACE is redundant.
3. DA/W/MAJ Line 299: The underscore symbol should be used instead of the vertical bar.
4. LO/W/MAJ Line 352: The code does not match the specification. Any combination of alpha, blanks, and numbers should be allowed.

SAMPLE CODE INSPECTION PROBLEM LIST

FIGURE 4-17

THE FOLLOW-UP STEP

During the follow-up step, the moderator verifies the completeness and accuracy of the reworked materials and gives formal approval to the work, thereby allowing the development effort to move forward. If the amount of rework warrants another inspection meeting, one may be scheduled at this time.

4.3.4 PERSONNEL CONSIDERATIONS WHEN USING INSPECTIONS

Data from inspections should never be used to evaluate software developers. The goal of inspections is to detect errors. The number of errors detected depends upon the thoroughness of the inspection. If many errors are found during an inspection, this means that the inspection process is working, not that the developer is inadequate. If people fear that data from inspections will be used to evaluate them, inspections will be of little benefit to a project.

11-2

4.4 PROGRAMMING STANDARDS

4.4.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 4.4)

Software projects shall use programming standards to promote uniformity, readability, understandability, maintainability, and other quality characteristics of the software products. Where applicable, such standards should also contribute to portability of software between computers and to compatibility with existing and future support software. Either the programming standards specified in the contract or identified in the NSA/CSS Software Product Standards Manual or developed for a specific project shall be used. The developer shall demonstrate plans to periodically audit design and code to assess adherence to the project programming standards. Programming standards shall be documented in a Software Standards and Practices Manual.

4.4.2 GUIDANCE

The Programming Standards define the standards to be followed during software design, code, and subsequent life cycle support. The objectives of the standards are twofold: (1) to produce a high quality software product (code and documentation), and (2) to reduce the life cycle costs of that product. A project's programming standards shall contain, as a minimum:

1. The standards to be used for preface test and in-line comments;
2. Higher order and assembly language coding standards for every language that is used by the project to generate code;
3. Structured programming standards for every higher order language that is used by the project to generate code;
4. Project-unique design standards (e.g., duty cycle, memory utilization, maximum routine size, input/output, interfaces).

Assembly language shall be allowed only in areas where code efficiency, storage or machine dependency requires it. The developer must identify and justify all areas where assembly language is required and obtain approval from the Software Acquisition Manager before proceeding.

Programming standards are necessary if the software development process is to become a disciplined activity. However, the standards cannot be applied in a "cookbook" manner. They require judgment in their application to ensure compliance and to provide waivers for specific instances when there is an appropriate rationale.

4.4.3 PROGRAMMING STANDARDS AND GUIDELINES

TABLE OF CONTENTS

SECTION 1.	INTRODUCTION
1.1	Programming Style
1.2	Software Segmentation, Module Size, and Complexity
1.3	Ada
SECTION 2.	STRUCTURED PROGRAMMING
2.1	Theory
2.2	Constructs
SECTION 3.	IMPLEMENTATION
3.1	FORTRAN 77
3.2	PL/I
3.3	C
3.4	Structured Programming in S/370 Assembly Language
3.5	Ada
SECTION 4.	MODULE CONSTRUCTION
4.1	Completeness
4.2	Function
4.3	Entry/Exit
4.4	Declarations
4.5	Imbedded Constants
4.6	Arguments
4.7	Exponents
4.8	Mixed Mode Arithmetic
4.9	Explicit Branching
4.10	Error Handling
4.11	Range Checks
4.12	Indices and Subscripts
4.13	Loop Termination
4.14	Use of Labels and Names
4.15	Global and Shared Variables
4.16	Standard Linkage Conventions
4.17	Input and Output
4.18	Naming Standards
4.19	Preface Commentary Standards
4.19.1	NAME Section
4.19.2	PURPOSE Section
4.19.3	INPUT/OUTPUT Section
4.19.4	PARAMETER Section
4.19.5	CALLS Section

- 4.19.6 GLOBAL DATA Section
- 4.19.7 RESTRICTIONS Section
- 4.19.8 ABNORMAL END Section
- 4.19.9 METHOD Section
- 4.20 In-Line Commentary Standards
- 4.21 Indentation and Paragraphing Standards

- SECTION 5. PRODUCT DEVELOPMENT LIBRARY
 - 5.1 Introduction
 - 5.2 Manual Product Development Library
 - 5.3 Basic Product Development Library
 - 5.4 Full Product Development Library with Management Data Collection and Reporting

Section 1. INTRODUCTION

1.1 Programming Style

When the design of a new software system is complete and well documented in the Program Specification, the process of translating the detailed design into executable software begins. If the architectural and detailed design describe each software capability as a standalone unit, the translation process will be relatively straightforward. The interfaces between software units will become argument lists for subroutine calls; or they will turn into data structures that are read into or written out of independently executing programs. If the data referenced in the Program Specification is already defined in the Data Dictionary Document, the declaration of the argument list or data structures will be easily accomplished.

In a similar fashion, if the Program Specification portrays the design as a hierarchy of independent units with clearly defined interfaces, the software can be developed in a top-down, structured manner. Units at the top of the hierarchy can be programmed and tested thoroughly without considering the processing that occurs at lower levels of the software architecture. The technique of defining the interfaces between units so that one unit does not need to know how the other translates inputs into outputs is called "information hiding." If information hiding is considered during the design phases, it will be easy to achieve in the coding phase by developing the interfaces exactly as they were designed and documented in the Program Specification. After the top level units in the architecture are implemented and tested, lower levels of units can be added and tested in the same way. Succeeding levels of software can then be built from the top of the architecture to the bottom until the entire software system is coded.

A top down, structured approach, combined with the information hiding, has several advantages:

- a. The system is easy to understand because each layer in the software hierarchy is made of units and routines small enough to be easily understood. This is especially important when changes need to be made to the completed system.
- b. Isolating problems in the software is easy because each layer in the software architecture can be completely tested before moving to the next lower level.
- c. When units are completely defined by a Program Design Language and a description of their interfaces, they can be coded by someone who has little knowledge of the system or of the surrounding modules. Each unit can be implemented and tested independently. This gives the Software Development Manager tremendous flexibility in assigning software units to programmers for coding and unit testing.

It should be apparent that a detailed design that embodies structured programming techniques and structured design techniques makes the programming phase simpler. Program Design Language will have the same complexity value as the code that is developed. A unit with a span of control of three can be coded to have the same span of control. The key to success in creating modular, structured software is to design it that way.

Given that the modularity, control structure, and communications within a software system are largely defined during the design phase, software projects can gain additional benefits by adopting standards for programming style. These are to some degree dependent upon the programming language which will be used. Later sections in this chapter address language-specific standards. However, some style guidelines are applicable regardless of the implementation language. The goal of project-specific programming standards is to make the software easy to read and understand. Well-structured programs meet this goal. They can be read from beginning to end without having to jump back and forth through the source code. They are also organized hierarchically. The highest level of the hierarchy may contain subroutine calls, procedure calls, or macro calls to lower-level routines. When completed, the lower-level routine will return to the higher-level and resume execution following the point-of-call.

Well-structured, readable code is attained by using the basic constructs of structured programming described later in this chapter. Each of the constructs has a common characteristic: a single entry point and a single exit point. This means that the only way to enter a software element is at the beginning, and the only way to leave it is at the end. All that can be seen from the outside of an element of code and the single entry and exit points are the variables that enter and leave at those points.

Stylistic conventions adopted in the analysis and design phases should continue during the programming activities. Names used in the Data Dictionary Document to reference data should be emulated in data declarations. Data names should convey what the data is and how it is used. In-line comments explain functions which may not be discernible from the program statements. Minimizing the use of global data reduces the risk of unpredictable results. These issues and other important considerations are discussed more fully in the remaining sections of this chapter.

1.2 Software Segmentation, Module Size, and Complexity

The intent of modularization is to segment software into discrete functional elements that are understandable, maintainable, and testable. Two principles govern the modularization process: functional decomposition should result in independent, cohesive units that are a natural decomposition of the problem; and modularization should be governed by "size" or "testability" of units. In other words, units must not be so complex that they are untestable. Size limitations are useful indicators for software units that may need additional decomposition, but they should not force artificial segmentation of functional blocks of code. In addition, a block of code which is less than maximum length does not guarantee a module with reasonable complexity.

Complexity Measures provide an excellent quantitative basis for modularization and allow identification of software units which may be difficult to test or maintain. They should be used in conjunction with size limitations to aid in controlling software segmentation. The complexity measure briefly presented in this section is derived from T. J. McCabe's work on complexity measures.*

The approach taken is basically a mathematical technique, drawn from graph theory, which is used to measure the number of paths through a program. Complexity is controlled by setting the limits on the number of paths that a software element may contain. The approach is complicated by the fact that any program with a backward branch contains an arbitrary number of paths. This complexity measure focuses on the set of basic paths that, when taken in combination, will generate every possible path.

The paper describes three methods of evaluating the complexity of a program. Two of the methods require that the control paths of the program be expressed as a directed graph. The program control graph has a unique entry and exit node, each node in the graph corresponds to a block of code whose flow is sequential, and arcs correspond to branches in the program.

*McCabe, T. J., "A Complexity Measure," Proceedings of the Second International Conference on Software Engineering, IEEE, October 1976, San Francisco, California.

See also McClure, C. L., "A Model for Program Complexity Analysis," Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978, Atlanta, Georgia.

The third method avoids drawing program control graphs and determines complexity by counting the number of conditions (IF tests) in a structured program and adding one. Compound IF statements such as:

```
IF C1 AND C2 THEN
```

are treated as

```
IF C1 THEN IF C2 THEN
```

which contains two conditions. Case structures can be simulated by parallel IF statements. For example:

```
DO CASE (i)
CASE1: S1
CASE2: S2
ENDDO CASE
```

is "equivalent" to:

```
IF CASE1 THEN S1
ELSE IF CASE2 THEN S2
```

If there is a default case, a DO CASE with N cases will have N-1 conditions. Case statements lacking a default case can be evaluated by counting the number of cases; i.e., N cases then N conditions.

Consider the following code fragments:

```
IF p THEN
  code A
ELSE IF q THEN
  code B
ELSE IF r THEN
  code C
ELSE
  code D
ENDIF
ENDIF
ENDIF
```

Fragment A

```
IF p THEN
  code A
ENDIF
IF q THEN
  code B
ENDIF
IF r THEN
  code C
ELSE
  code D
ENDIF
```

Fragment B

Using the predicate counting method (number of IF tests plus 1), both fragments have the same complexity value (4).

While this method is perhaps the easiest to use, a more useful complexity measure is to compute the number of possible paths through a structured program. If we compute the number of possible paths through each fragment, fragment A has a complexity value of 4, but fragment B has a complexity value of 8.

The point of this discussion is that the traditional approach to combatting complexity (i.e., dividing the program into small, well-structured, independent modules that interact in a restricted manner) is not a complete methodology. Quantitative techniques should be used to analyze program complexity. They should include, at a minimum, an examination of the number of possible execution paths and the control structures and variables used to direct path selection.

Establishing reasonable limits on module size and complexity for each project is essential. Software Development Managers must set and enforce standards for module size and complexity. The standards should be rationally enforced to avoid artificial segmentation. The intent of size and complexity standards is to produce software that is understandable, maintainable, and testable.

For more detailed discussions on software complexity and its relation to software testing, consult McCabe's article in the "Proceedings of the Second International Conference on Software Engineering," and NBS Special Publication 500-99, "Structured Testing: A Testing Methodology Using the McCabe Complexity Metric," 1982.

1.3 Ada*®

This section describes the principal features of Ada, a new programming language developed under sponsorship of the Department of Defense. Ada is a high-level, general purpose programming language designed initially for large, real-time embedded computer systems applications. It can also be used for many other types of applications.

Ada was designed to improve three areas relating to computer programming: program reliability and maintenance, programming as a human activity, and language efficiency. Emphasis was placed on program readability over ease of writing. The syntax of the language uses English-like constructs rather than encoded forms. Ada also provides the ability to assemble a program from independently-produced software components.

An Ada program is made of one or more program units. There are four forms of Ada program units. Subprograms define executable algorithms. Package units define collections of entities. Task units define parallel computations, and generic units define parameterized forms of packages and subprograms. These program units can be compiled separately. Each program unit normally consists of two parts: a specification containing information that must be visible to other program units, and a body containing implementation details that need not be visible to other program units.

Within the context of these program units, Ada has many of the best features found in PASCAL, ALGOL, PL1, FORTRAN, and COBOL. It also contains additional features not found in other languages. The combination of desirable features of current programming languages and new features embodying improved software engineering methods make Ada a good language for many applications. Following is a description of the most distinctive Ada features.

SEPARATION OF PROGRAM SPECIFICATION AND PROGRAM BODY

In Ada, the specification of a program unit is its interface with the outside world. It tells software developers what they need to know to use the program unit. It identifies the data objects (parameters) that must be supplied when the unit is called and the data objects that the unit will return to the caller. The body contains the code statements that are executed when the program unit is called. Once the program unit's specification has been compiled, it can be put into an Ada library. Developers can then design and code new program units that call the first unit. They do not need to know anything about the body of the first unit.

®Ada is a registered trademark of the U.S. Department of Defense.

*The information in this section was taken from two sources: (1) Reference Manual for the Ada Programming Language, ANSI/MIL-STD 1815A-1983; dated 17 February 1983, and published by the United States Department of Defense; and (2) Software Engineering with Ada, by Grady Booch, the Benjamin/Cummings Publishing Company, Inc., 1983.

Separation of specification and body supports "top-down" design and programming. If a high-level program unit calls a lower-level unit, then the high-level unit can be designed, coded, and compiled when the specification of the lower-level unit is completed. Using a dummy version of the lower-level unit's body allows the higher-level unit to be tested before the body of the lower-level unit has ever been designed.

Separation also loosens connections ("coupling") between program units. It makes the coupling that does exist explicit and understandable. In Ada, software elements are coupled through their specifications, not their bodies. This reduces the "ripple" effect when changes are made to Ada source code.

DEFINITION OF NEW DATA TYPES

Most programming languages have a small set of built-in data types, such as integers, real numbers and character strings. Typically, they allow programmers to define and name multiple instances of a given data type, such as integer or string variables; but programmers often need to use data objects that are not part of the built-in sets.

Ada allows programmers to define new types of data objects built from primitive data types. Just as the familiar arithmetic operations (+, -, *, and /) are legal operations on objects of the built-in data type "integer," Ada allows programmers to define "legal" operations of the new data type.

The legal operations of user-defined data types can be grouped with the names and structures of the data types themselves into a package and placed in an Ada library. Then programmers can treat the new user-defined data types as if they were built-in types. They can manipulate objects of the new types with sets of operations defined for those types. Ada prevents manipulation of those objects unless they have been explicitly defined.

User-defined data types support several software engineering principles. They support the idea of programming in terms of objects natural to an application and operations natural to the manipulation of those objects. This idea is called "object-oriented" programming. They also support the concept of "information hiding" by keeping from users all information that they do not need to know. In addition, user-defined data types loosen coupling between program units. The less a unit knows (and uses) about implementation of the objects it manipulates, the lower its coupling with other elements that use the same objects.

STRONG TYPING

Ada requires that every declaration of a data object specify its data type. The data type determines the operations that may legally be applied to objects of that type and the values that such objects may have. This is called strong typing.

Strong typing supports the principle that errors should be detected as early as possible in the development process. Because the types of all objects are known at compile time, the compiler can detect many errors involving type conflicts. It can detect attempts to assign a value to an object that is not in the set of legal values for objects of that type. At compilation time, Ada can also check errors between separately-compiled programs.

GENERIC PROGRAM UNITS

A generic program unit is a template or skeleton of package program units or subprogram program units that can be written once and tailored to meet specific needs at translation time. Thus, generic program units define a unit template, along with generic parameters, that allow programmers to use the generic unit in many instances. They are not executable; so they may not be used directly. To create a generic program unit, programmers take the specification part of a package or subprogram and add a prefix, called the generic part, that defines the generic parameters. By specifying values for the generic parameters, programmers may use the generic unit in other program units without explicitly creating separate units to process objects of different types if the algorithms are identical.

Generic program units support the principle of procedural abstraction. Programmers may work with a high-level procedure, such as a sort, and ignore low-level details such as data types. They may use the high-level algorithm without having to write the low-level coding details about specific data types.

TASKING

In his book, Software Engineering with Ada, Grady Booch says the following:

In the real-world problem space, there are often a number of activities logically occurring at the same time. For example, an aircraft autopilot may be continuously

monitoring several sensors, such as airspeed and angle of attack, waiting for user requests, and controlling several independent devices, such as control surfaces and throttles. In addition, some devices, such as navigation aids, may use asynchronous hardware interrupts to request service.

Most existing high-order languages provide little support for dealing with such concurrent activities. Software developers would have to use the facilities of the host operating system or write unique multitasking assembly language routines. Ada, however, is designed to handle this type of real-time application.

Ada supports real-time applications by allowing a developer to specify that certain program units run concurrently. Tasking program units are entities that proceed in parallel except at points where they synchronize. Ada provides commands that allow tasks to synchronize or "rendezvous," under a variety of conditions.

PORTABILITY

Portability means the ability to transfer computer programs from one computer to another. Any language with machine-independent constructs provides portability, but Ada offers a higher degree of portability than most other languages. There are several reasons for this:

a. Input/Output. In most languages I/O is limited and inflexible. The result is that individual computer systems must supplement the standard I/O of the language with nonstandard features. These nonstandard features must be recoded when a program is "ported" from one machine to another. In Ada, the problem is circumvented by the package feature. A group of I/O routines can be grouped in one package in much the same way that a package implements a new user-defined data type. The user can define a package of I/O routines to provide a specific type of I/O capability or to access a specific I/O device.

b. Low-Level Features. However much developers may hide low-level details of physical implementation, they must always implement low-level programs and data structures. Because most high-level languages do not provide a convenient way of expressing those low-level details, they must be implemented in assembly language. Ada, however, has features for providing such low-level details so that no assembly language code is required. It provides the capability for generating the correct code to manipulate objects of the given type on a new machine.

c. Standardization. Most languages have no strict standard. Even those that do have nonstandard dialects that are similar but not identical. The Department of Defense is placing much emphasis on ensuring that there are no dialects of Ada, either subsets or supersets. They have developed elaborate procedures for testing, validating, and certifying Ada compilers. These procedures contain tests to verify that every approved Ada compiler implements and correctly interprets the requirements of the language as stated in the Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A current version dated 17 February 1983). In fact, the Ada Joint Program office has registered the item Ada as a trademark in an effort to ensure that only compilers that have passed the validation tests are even called "Ada".

CONCLUSION

The Department of Defense sponsored the development of Ada to deal with many of the well-documented problems associated with the development and maintenance of software systems. By having many modern software engineering principles embedded in the language, advocates believe that the use of Ada will improve productivity, allow code to be reused, reduce time required for module integration, and reduce costs of life-cycle support. As more compilers are certified and their performance improved, the use of Ada in Government and in industry will grow. This is not because the use of Ada has been mandated by the Department of Defense, but because many who know the language believe that it has great promise.

Section 2. Structured Programming

Structured programming is designed to minimize software complexity. Its major objective is to simplify the program development process by standardizing the number of logic structures used to construct software. Much of a program's complexity is caused by the presence of arbitrary branching; forward and backward branches that are not part of well-defined structures, for example. Arbitrary branching makes it difficult to determine the state of a program at any given point (e.g., value of a variable, program paths traversed). Furthermore, as a program undergoes change during development and maintenance, its complexity often increases.

2.1 Theory

Structured programming copes with complexity by using a few simple structures that aid in minimizing the number of software interactions and interconnections. It is based on new mathematical foundations for programming. Four propositions are central to structured programming (see Mills, H.D., "Mathematical Foundations for Structured Programming," FSC 72-6012, IBM Corp., February 1982). They are, in order of logical precedence:

a. Top-Down Corollary: Every proper program logic can be represented by one of the following three structures:

- (1) DO f THEN g
- (2) IF c THEN f ELSE g
- (3) WHILE c DO f

where "f" and "g" are proper programs each with one entry and one exit, "c" is a determinable condition (i.e., a test) and "IF", "THEN", "ELSE", "WHILE", and "DO" are logical connectives. The "Top-Down Corollary" guarantees that structured programs can be written or read "top-down", i.e., in such a way that the correctness of each segment of a program depends only on segments already written or read and on the functional specifications of any additional segments referred to by name.

b. Structure Theorem: Every proper program logic is equivalent to a program obtained by iterating and nesting the three structures (1), (2), and (3) above. The "Structure Theorem" proves that arbitrary flowcharts with unrestricted control branching can be represented using a finite site of structures with a reduction in the complexity.

c. Correctness Theorem: If a program is structured as in (b) and if the domain of the data space on which "f" operates in (3) is not redefined dynamically in the looping process, the correctness of the entire program can be proved by successively proving that the data spaces for each structure at each level of iteration or nesting are transformed in a specified way. The "Correctness Theorem" shows how standard mathematical practices can be applied to proving the correctness of structured programs.

d. Expansion Theorem: The freedom by which a proper program logic "f" may be refined into one of the forms (1), (2), or (3) above is limited as follows:

(1) DO g THEN h can replace "f" whenever there exists a functional decomposition of "f" into "g" and "h" in which $f=h(g)$, i.e., "f" is the result of program logic "h" operating on the computer state at the completion of "g".

(2) IF c THEN g ELSE h can replace "f" whenever a logic condition "c" can be found whose domain is the same as that of "f". Then "g" and "h" are fully determined.

(3) WHILE c DO g can replace "f" whenever a function "g" can be chosen which, when iterated, ultimately reaches "f". The condition "c" is determined as that condition which recognizes that "g" has reached "f".

2.2 Constructs

As shown in Section 2.1, any proper program (i.e., a program with one entry and one exit) is equivalent to a program that contains only the following logic structures:

- Sequences of two or more operations
- Conditional branch to one of two operations and return
(IF a THEN b ELSE c)
- Repetition of an operation while a condition is true
(DO-WHILE p)

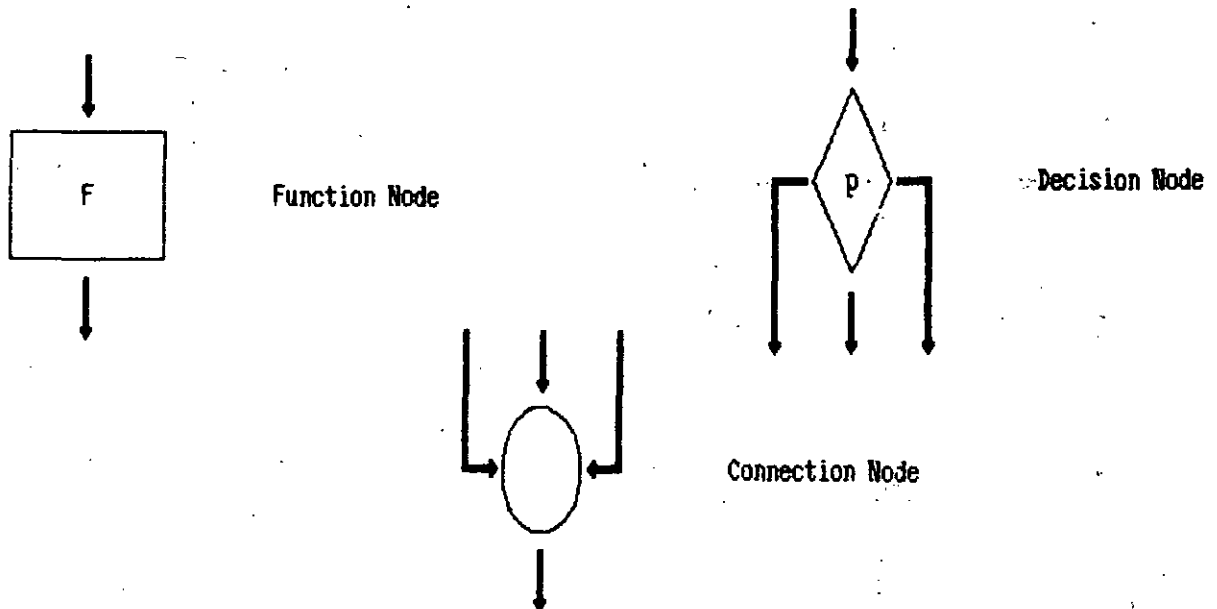
Each of the above structures represents a proper program. A large and complex program may then be developed by appropriate sequencing and nesting of these three basic structures. The logic flow always proceeds from the beginning to the end without arbitrary branching.

Two extensions to these three basic logic structures have been defined to improve readability of source code. They are DO-UNTIL and CASE. These do not affect the spirit of structured programming, and in some cases may result in more efficient use of computer time and storage. DO-UNTIL is a variant of the DO-WHILE and provides an alternate form of looping structure. It differs from DO-WHILE in that the condition is tested after the operation rather than before. CASE is a multibranch, multijoin control structure used to select one of the many possible processing alternatives.

Thus, structured language code need contain only the following five logic control structures:

- Sequence of two or more operations
- Conditional branch to one of two operations and return (IF p THEN a ELSE b)
- Repetition of an operation while condition is true (DO WHILE p)
- Repetition of an operation until a condition is true (DO UNTIL p)
- Selection of one of several possible operations (CASE)

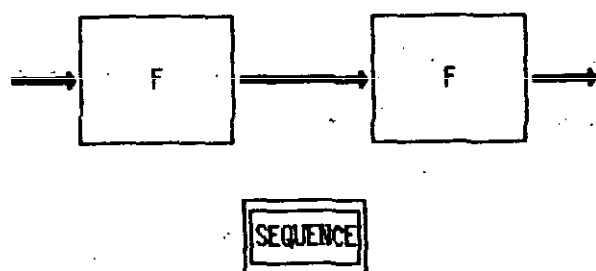
These control structures can be graphically represented by three kinds of nodes; function, decision, and connection. The symbols for these nodes are:



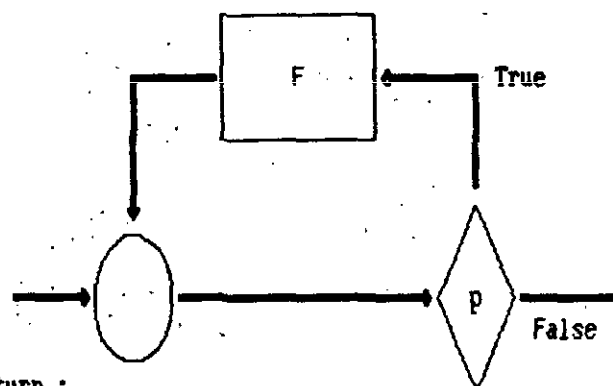
The flow of execution through these nodes follows the arrows. The function node represents a defined set of processing (which might be a single statement or as much as an entire program). A decision node represents a multi-way branch whose outcome is defined by the state of the tested condition. Finally, the collector node represents the joining of two or more logic paths.

Having described the basic constructs and a nomenclature for representing them, graphic representations of the five programming constructs are as follows:

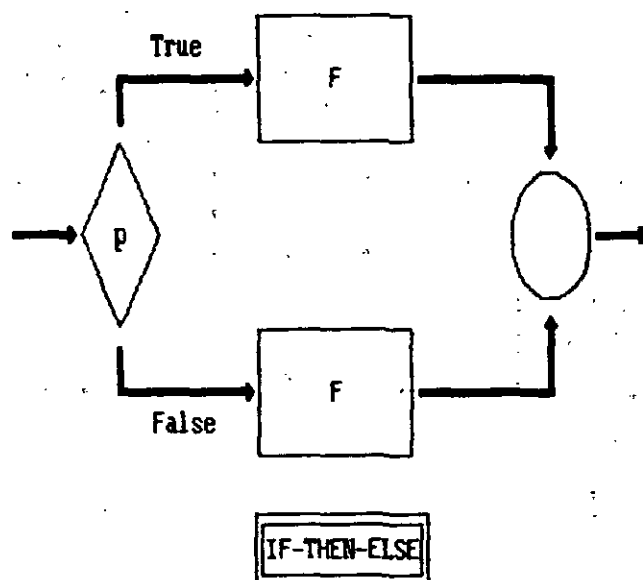
Sequence of two operations :



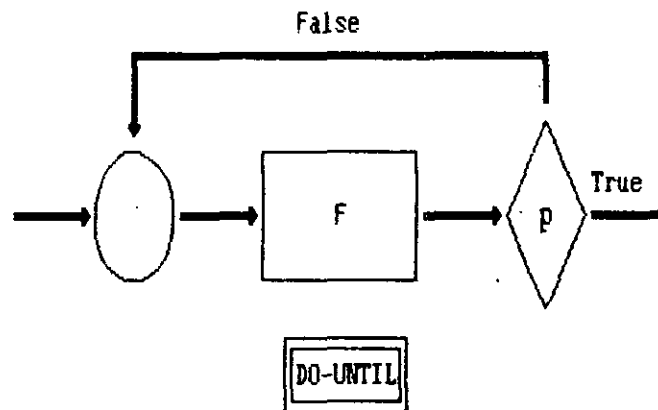
Operation repeated while a condition is true :



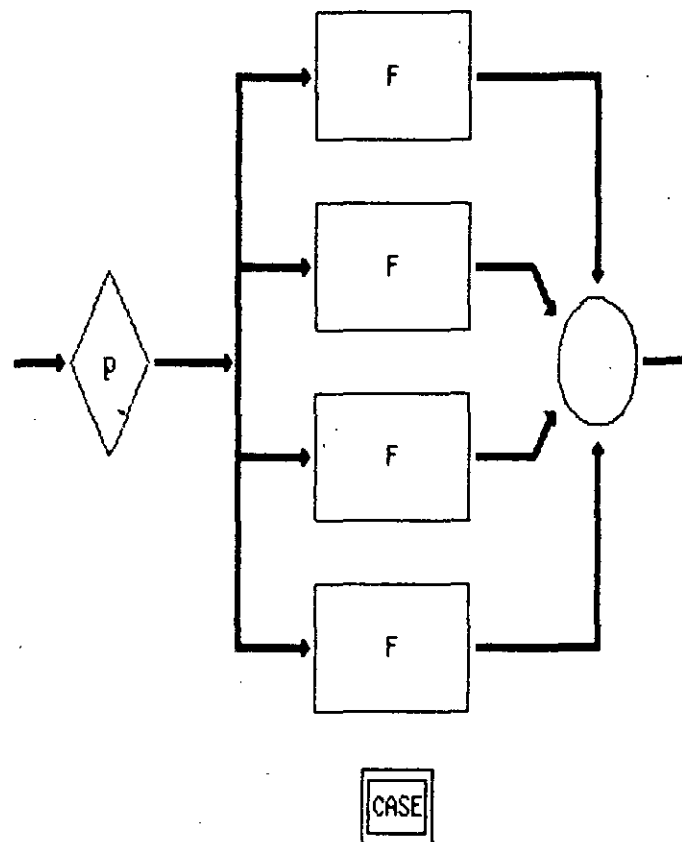
Conditional branch to one of two operations and return :



Operation repeated until a condition is True :



Conditional branch to one of four operations and return :



Section 3. Implementation

This section discusses the actual implementation of the structured programming constructs for each of several higher-order languages.

3.1 FORTRAN 77

While FORTRAN 77 does feature an IF-THEN-ELSE structure, the remaining constructs must be simulated. Though this "simulation" adds to complexity (due to the needed branching around blocks of code), it can enhance the readability of code if used properly.

IF-THEN-ELSE Structure

The IF-THEN-ELSE structure causes control to be transferred to one of two functional blocks of code (A or B) based on the evaluation of a logical expression (p). As mentioned above, FORTRAN 77 directly supports this construct as follows:

```

      IF (p1) THEN
        code A
      ELSEIF (p2) THEN
        code B
        .
        .
      ELSEIF (pn) THEN
        code Y
      ELSE
        code Z
      ENDIF

```

In the above example, a series of tests are performed until the expression is "true" or the "n" cases are exhausted. In the latter case, code block Z (the default code) is executed.

DO Structures

The DO structures allow iterative execution of a functional block of code (A) based on a logical expression (p). If the test is made prior to the execution of code (A), it is a DO-WHILE structure. If the test is made after code (A), it is a DO-UNTIL structure.

The FORTRAN DO is essentially a specialized DO-UNTIL and its use to simulate the DO structure is rather clumsy. A body of code is iterated until a loop index exceeds a test value. In addition, the statements within its range are executed at least once. The DO structures can easily be simulated using a logical IF in conjunction with GOTO statements.

DO-WHILE

The code structure used to represent the DO-WHILE is:

```
0010      CONTINUE
          IF ( p ) THEN
              code A
              GOTO 0010
          END IF
```

DO-UNTIL

DO-UNTIL is simulated thusly:

```
0010      CONTINUE
          code A
          IF ( .NOT. (p) ) GOTO 0010
```

FORTRAN DO

The FORTRAN DO is a command to execute the statements that physically follow it, up to and including the numbered statement indicating the end of the DO's range. Since the FORTRAN DO is essentially a specialized DO-UNTIL structure, it is desirable to have specific guidelines for using this FORTRAN capability. It is acceptable to use a FORTRAN DO where the loop must progress incrementally. For example, only repetitive structures that require varying indexes and that also use the indexes in the body of the loop should be implemented using the FORTRAN DO.

In general, thinking in terms of FORTRAN DO's, rather than DO-WHILE or DO-UNTIL constructs, encourages the idea that loops should progress arithmetically with a control variable running from 1 to n in steps of m. Although there are methods for mapping an arbitrary sequence into an arithmetic progression, they tend to obscure the actual logic and make the program more susceptible to error. The control structure should not conceal the reason for the loop or terminating condition.

CASE Structure

The CASE structure causes control to be passed to one of a set of functional blocks of code (A, B, ..., Z) based on the value of an integer variable *i*. The CASE structure can be simulated using either a computed GOTO statement or a sequence of IF-THEN-ELSE structures. The choice should be based on the density of the set as expressed by the following ratio:

$$\frac{\text{number of CASE values}}{(\text{maximum CASE value}) - (\text{minimum CASE value}) + 1}$$

The density of a set increases as the ratio approaches one. Except for high density sets, the CASE structure should be simulated using IF-THEN-ELSE structures.

Computed GOTO Simulation of the CASE Structure.

The value of the integer variable, *i*, may need to be normalized since control is passed to one of the set of functional blocks of code (A, B, ..., Z) based on the value of *i*, equal to (1, 2, ..., *n*). Some FORTRAN compilers provide that if the value of *i* is outside the range 1 *i* *n*, the next statement is executed. Others will treat it as being an undefined statement. It is therefore recommended that the value of *i* be tested before entering the CASE control logic structure. Out-of-range values should result in control being transferred to the default code (if present) or to the end of the case structure.

The default code and the "GOTO 00n0" immediately following the computed GOTO statement are provided for compilers that provide for execution of the next sequential statement when *i* is not within the range of the computed GOTO.

Statements with the structure are indented from the DO CASE and ENDDO comment lines (which define the beginning and end of the figure). Statements within each case are indented from each CASE comment line.

If the functional blocks of code are identical for more than one case, the appropriate entries in the computed GOTO statement should contain the same label. Further, if no action is to be performed for specific values of *i*, the appropriate entries should point to the end of the structure.

NOTE: The statement lines prefixed with "/* comment */" are for illustrative purposes only and do not represent executable FORTRAN code.

Consider the following example:

```

/* comment */ DO CASE (I)
                GO TO (0010, 0040, 0010, 0020, 0001, 0030) , I

/* comment */ CASE OTHER :

    0001          CONTINUE
                  default code
                  GOTO 0040

/* comment */ CASE 1: CASE 3:

    0010          CONTINUE
                  code for cases 1 and 3
                  GOTO 0040

/* comment */ CASE 4:

    0020          CONTINUE
                  code for case 4
                  GOTO 0040

/* comment */ CASE 6:

    0030          CONTINUE
                  code for case 6

    0040          CONTINUE

/* comment */ ENDDO CASE

```

For $i=1$ and $i=3$, the same functional block of code will be executed; for $i=2$, no processing will be performed; and for $i=5$ and i out-of-range, the default code will be executed.

IF Simulation of the CASE Structure

Implementing the CASE structure using a computed GOTO can waste memory when the density of the set is low. Consider the following example:

```

CASE values = 1, 2, 3, 4, 5, 100, 200
Density      = .035

```

If the computed GOTO statement is used, 200 branch labels (for CASE values 1 through 200 inclusive) must be specified in the statement. Memory can be reduced if the CASE is simulated using the IF-THEN-ELSE structure.

CASE structure simulation using the IF-THEN-ELSE structure is relatively straightforward. Each CASE value is tested by a series of parallel tests, i.e., an IF...ELSE IF...ELSE...ENDIF sequence. If the CASE values are not equally probable, it may be better to use the IF...ELSE IF...sequence, testing the most likely values first.

Restricted FORTRAN Statement Usage

In order to maintain structured programming concepts, certain allowable statements shall not be used except as required in the definition of standard program structures. Unless required by standard program structure definition, unconditional branching should not be used.

The GOTO statement is used in the definition of the following structures: DO-WHILE, DO-UNTIL and CASE. The computed GOTO is used in the CASE standard structure. These shall not be used except in the indicated structures.

The arithmetic IF should not be used. The IF-THEN-ELSE structure (with nesting sometimes required) will provide the same capability.

Statement numbers shall be used only on CONTINUE and FORMAT structures.

Other uses of the preceding statements shall be avoided.

3.2 PL/I

PL/I supports all of the basic structured programming structures. Once the principles of structured programming are understood, writing structured programs in PL/I can be accomplished with relative ease.

IF-THEN-ELSE Structure

The IF THEN-ELSE structure causes control to be transferred to one of two functional blocks of code (A or B) based on the evaluation of a logical expression (p).

```

      If (p) THEN
        DO;
          code A
        END;
      ELSE
        DO;
          code B
        END;

```

The ELSE is vertically aligned with the IF, and statements controlled by the THEN and ELSE shall be indented to show the span of control. A DO group (i.e., DO; ...END;) should be used in the THEN and ELSE portions of the IF statement to ensure that the span of control is explicit and understood. The ELSE statement is optional and if not used would result in the following code:

```

      IF (p)
        DO;
          code A
        END;

```

DO Structures

The DO structures allow interactive execution of a functional block of code (A) based on a logical expression (p). If the test is made prior to the execution of code (A), it is a DO-WHILE structure. If the test is made after code (A), it is a DO-UNTIL structure. PL/I contains the control structures necessary to implement the structured programming structures.

DO-WHILE

The basic PL/I format for the DO-WHILE is:

```
DO WHILE (p):
    code A
END;
```

One form of the DO-WHILE with indexing is:

```
DO id1 = exp1 to exp2 by exp3 WHILE (p);
    code A
END;
```

The repeat option provides an alternative method of specifying successive values of the control variables as in:

```
DO id1 = exp1 REPEAT (exp2) WHILE (p);
    code A
END;
```

Another variation leaves the predicate implicit in the indexing parameters:

```
DO id1 = exp1 TO exp2 BY exp3;
    code A
END;
```

DO-UNTIL

The DO-UNTIL control structure executes a function and then tests a logical expression to determine whether to repeat it. The code within the loop is always executed at least once and the loop terminates on a "true" condition. The basic PL/I DO-UNTIL is:

```
DO UNTIL (p);
    code A
END;
```

Consult a PL/I manual for variations of this basic format.

CASE Structure

The CASE structure selects one of a set of functional blocks of code based on the value of an expression (e):

```

SELECT (e);
    WHEN (e1)
        code A;
    WHEN (e2)
        code B;
    .
    .
    WHEN (en)
        code Z;

    OTHERWISE
        default code;

END

```

In this example, e and e1...en are expressions. When control reaches the SELECT statement, (e) is evaluated. The expression in the first WHEN clause (e1) is evaluated and compared with (e). If the two are equal, code (A) is executed; otherwise, successive expressions (e2...en) are evaluated until either a match is found (and corresponding code is executed) or the choices are exhausted. In the latter case, the default code is executed unconditionally.

INCLUDE Capability

The capability of nesting blocks of code with other code blocks is useful to top-down programming. A program, in PL/I terms, can consist of one or more external and/or internal procedures, or code incorporated from a library with an %INCLUDE statement.

3.3 C

Programs written in C offer some of the best opportunities for good structured code. The language supports all of the structured constructs discussed in section 2.2. In addition, C promotes the concept of building a program with modular units, many of which can be reused in other programs/functions.

IF-THEN-ELSE Structure

The IF-THEN-ELSE structure causes control to be transferred to one of two functional blocks of code based on the evaluation of a logical expression (p). The construct can easily be extrapolated to an n-way branch. The more general form follows:

```

IF (p1)
{
    code A;
    .
    .
}
ELSE IF (pn)
{
    code Y;
}
ELSE
{
    code Z;
}
ENDIF

```

In the above examples, a series of tests are performed until either the expression evaluates to "true" or the "n" cases are exhausted. In the latter case, code block Z (the default code) is executed.

DO Structures

The DO structures allow interactive execution of a functional block of code (A) based on a logical expression (p). If the test is made prior to the execution of code (A), it is a DO-UNTIL structure.

While C supports the spirit of both the DO-WHILE and DO-UNTIL constructs, their syntactical appearance can easily confuse the issue; the DO-UNTIL construct looks like a DO-WHILE loop.

DO-WHILE

C's DO-WHILE construct is coded as follows:

```

WHILE (p)
{
    /* while */
    code A;
} /* while */

```

As long as condition p evaluates to "true", the code contained within the braces is executed.

DO-UNTIL

As noted above, this construct can be visually deceiving. Its format suggests a DO-WHILE loop:

```

DO
{
    /* do */
    code A;
} /* do */
WHILE (p);

```

Close examination of the above logic should convince one that the spirit of the DO-UNTIL construct is preserved. Code will be performed (at least once) until the condition p evaluates to false.

CASE Structure

The CASE structure selects one of a set of functional blocks of code based on the value of an expression (e). The C equivalent of the CASE statement is the SWITCH statement. Its format is:

```

SWITCH ( e )
{
    /* case */
    CASE v1 :
        code a;
        BREAK;
    CASE v2 :
        code b;
        BREAK;
    .
    .
    .
    DEFAULT :
        code z;
        BREAK;
} /* case */

```

The expression (e) is successively compared to values v1...vn. If a case is found whose value is equal to the value of e, the program statement/block following the respective CASE statement is executed. Note that contrary to typical C form, multiple statements within a CASE code block are separated solely by semicolons, not enclosed in braces!

The BREAK statement signals the end of a particular case and effectively transfers program flow to the statement following the SWITCH's closing brace. Failure to include a BREAK statement results in the execution not only of the statements in the case which matched, but all subsequent cases until either a BREAK statement of the SWITCH's closing brace is encountered.

The DEFAULT case is optional. It resembles the ELSE statement in the IF-THEN-ELSE construct. If the expression (e) does not match any of the preceding case values, the DEFAULT case code is executed. The BREAK statement in the DEFAULT case shown above is somewhat superfluous since there are no executable statements prior to the SWITCH's terminating brace. Its inclusion is simply a style preference to aid in future maintenance.

3.4 Structured Programming in S/370 Assembly Language

Since assembly language is so closely tied to a specific processor, it is impossible to present a generic approach to implementing the structured constructs. Since it is widely used, S/370 Assembly Language is described here as a representative example. The concepts are easily transportable to other assembly language environments.

IF-THEN-ELSE Structure

The IF-THEN-ELSE structure causes control to be transferred to one of two functional blocks of code (A or B) based on the evaluation of a predicate (p). Macro implementation of this structure is:

```
IF (p)
  code A
ELSE
  code B
ENDIF
```

The format of the predicate (p) may take one of three forms. They are:

- a. condition mask
- b. operation code, operand 1, operand 2, condition mask
- c. compare instruction, operand 1, condition mask, operand 2 and are discussed in the paragraphs below.

a. Condition Mask

The user can supply a condition mask having a value from 1 to 14, or a one or two character alphabetic mnemonic indicator. The valid alphabetic expressions are identical to the extended branch mnemonic instruction codes except that the leading "B" is dropped. In addition, relational expressions for comparison purposes such as LT and LE are included as valid operands. The list of the mnemonics and equivalent masks is shown in the following table.

Mnemonics and Equivalent Conditional Masks

Mnemonic	Meaning	Conditional Mask
O	Ones or Overflow	1
P	Plus	2
H	High	2
GT	Greater Than	2
L	Low	4
M	Minus or Mixed	4
LT	Less Than	4
NE	Not Equal	7
NZ	Not Zero	7
E	Equal	8
EQ	Equal	8
Z	Zero	8
NL	Not Low	11
NM	Not Minus	11
GE	Greater Than or Equal	11
NP	Not Plus	13
NH	Not High	13
LE	Less Than Equal	13
NO	Not Ones	14

Thus the instructions:

IF(EQ) and IF(8)

will result in the same mask value to the branch or conditional instruction.

The alphabetic indicators listed above are usually associated with the instructions which set the condition code and are more meaningful when used in conjunction with them. Therefore, the user is allowed to write both the instruction and the condition code mnemonic (or actual value) to be used in the branch instruction. Any two-operand instructions are permitted. Two formats exist for this option, one specifically for compare instructions and one for all other types.

b. Operation Code, Operand 1, Operand 2, Condition Mask

In non-compare instructions the user supplies an operation code, two operands, and a condition mask. The instruction is written as it would appear in an assembly language program with the exception that the operation code and the first operand are separated by a comma, and the entire instruction is enclosed in parentheses. The instruction is followed by the conditional mask desired. Thus;

```
IF (AR,R2,R5,NM)
```

is interpreted as "if the content of register 2, after the content of register 5 has been added to it, is not minus, then ..."

c. Compare Instruction, Operand 1, Condition Mask, Operand 2

The format for the compare instruction differs from the non-comparison type in that the relational operand appears between the two operands being compared rather than after the second operand.

```
IF (CLC, O(8,R2), GT, 8R2))
```

is interpreted as "if the logical comparison of the character string referenced by O(8,R2) is greater than the character string referenced by 8(R2), then ..."

The ELSE in the IF-THEN-ELSE structure is optional and if not used, the macro reduces to:

```
IF (p)
  code A
ENDIF
```

Code blocks shall be indented to indicate the scope of control of the logic structure.

DO Structures

The DO structures allow iterative execution of a functional block of code (p) based on the evaluation of a predicate (p). If the test is made prior to the execution of code A, it is a DO-WHILE structure. If the test is made after code A, it is a DO-UNTIL structure.

Since the assembly language has branch instructions which can decrement (or increment), compare, and branch (or not) depending upon the result of the comparison, an indexed DO macro is presented. The indexed DO is a specialized DO-UNTIL structure.

DO WHILE/UNTIL

The DO macro is used to control looping operations. The generalized form of the code to represent the DO-WHILE is:

```
DO WHILE = (condition)
  code A
ENDDO
```

and the format used to represent the DO-UNTIL is:

```
DO UNTIL = (condition)
  code A
ENDDO
```

Statements within each structure (code A) should be indented to indicate the scope of control of the logic structure.

The keyword operands (UNTIL=, WHILE=) are used with a predicate whose format is identical to those described in the preceding IF-THEN-ELSE section. Thus:

```
DO WHILE = (NP)
  code A
ENDDO
```

provides for looping while the condition code is not positive, whereas:

```
DO UNTIL = (CLI, BYTE, EQ, C'A')
  code A
ENDDO
```

provides for looping until the logical comparison of BYTE and C'A' is equal.

Indexing DO

The execution of the indexing DO is governed by the keyword operands FROM=, TO=, BY=. Each of the three keywords can indicate a register and an optional value. The code structure is:

```
DO FROM = (r1,i), TO = (r2,j), BY = (r3,k)
  code A
ENDDO
```

with the recommended indentation as indicated.

CASE Structure

The CASE structure causes control to be passed to one of a set of functional blocks of code (A, B, . . . , Z) based on the value of an integer variable. The macro for the CASE structure is:

```
CASEENTRY (r)
  CASE Number, Number, . . .
  code A
  CASE Number, Number, . . .
  code B
  .
  .
  .
ENDCASE
```

The CASE is used to implement a decision process where there are a number of conditions and subsequent actions to be taken under those conditions. Control is transferred to a block of code having a given case number, when that number is found in a specified general register (r). The determination of the correct block of code to execute involves the use of a vector of addresses.

The starting macro (CASEENTRY) has one operand. It specifies a general register which contains the case number of the block of code to be executed. The other macro in the CASE macro set which has operands is the CASE macro. The case numbers, which are to be assigned to the block of code following the CASE macro, are listed as operands. Zero and negative numbers are not valid case numbers. More than one case number may be assigned to each CASE macro and they need not be in any specified sequence. If more than one case number is to be assigned, they must be separated by commas.

Note that the CASE structure is not defined if the value of the integer variable (general register) is out of range. Therefore, the general register must be checked before entering the CASE structure. In addition, every case number must be specified within the range of case numbers to ensure that the proper code is executed. Where the density of the set of case numbers is sparse (i.e., few case numbers having a wide range), alternate control logic should be considered such as an IF...ELSE IF...ELSE IF...ELSE...ENDIF...ENDIF...ENDIF construct.

Restricted Use of Assembly Language Statements

In order to preserve the structured programming concept, certain instructions should be used on an exception basis only. These are: branch instructions--branch on condition, branch on count, and branch on index. They involve the language capability which affects change of control.

3.5 Ada

Ada supports all of the structured programming structures. Further, Ada permits these structures to be employed in concurrent processing (via the Tasking model inherent with every validated Ada compiler) as well as in sequential operations.

Booch¹ gives the following high-level view of Ada:

"An Ada system is composed of one or more program units, each of which may be separately compiled. Program units consist of "subprograms", "packages" and/or "tasks". A "subprogram" is either a procedure or function, and it expresses a single action. A "task", on the other hand, defines an action that is logically executed in parallel with other tasks. A task may be implemented on a single processor, a multiprocessor, or a network of computers. A package is a collection of computational resources, which may encapsulate data types, data objects, subprograms, tasks or even other packages. A package's primary purpose is to express and enforce a user's logical abstractions within the language."

For a more detailed explanation, a copy of the Language Reference Manual should be consulted.²

IF-THEN-ELSE Structure

The IF-THEN-ELSE structure causes control to be transferred to one of several functional blocks of code based on the evaluation of the logical expression (P).

```

if P then
    code-block A;
else
    code-block B;
end if;

```

Ada allows a variant on the IF-THEN-ELSE, denoted IF-THEN-ELSEIF. The IF-THEN-ELSEIF structure allows control to pass to one of multiple blocks based on further evaluation of additional logicals.

¹Booch, Grady: "Software Engineering with Ada," The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CCA, 1983

²"Ada Programming Language," ANSI/ML-STD-1815A, Department of Defense, 22 January 1983

```

if P then
  code-block A;
elseif P2 then
  code-block B;
elseif P3 then
  code-block C;
...
else
  code-block Z;
end if;

```

In any case, the subsequent sub-blocks are optional. That is, an if-then may exist without either an else or elseif part.

```

if P then
  code-block A;
end if;

```

(In any case, execution continues at this point.)

In the above example, if P evaluates as TRUE, then code-block A is executed, then control falls to the next instruction after the "end if". If P is FALSE, then control passes to the next instruction after the "end if".

Ada also permits qualified execution of multiple conditionals. That is, if P were actually made up of two evaluations (e.g., A=B and B=C), then normally A=B would be evaluated and B=C would be evaluated. Those results would be combined in a Boolean AND operation to produce a logical result. Ada permits the user to define conditional evaluation of other parts of the overall condition with the "and then" and the "or else".

If P1 and then P2 then ... indicates that if-and-only-if the P1 conditional evaluates to TRUE is the evaluation of the P2 conditional to be performed. Similarly, if P1 or else P2 then ... indicates that if-and-only-if the P1 conditional evaluates to FALSE is P2 to be evaluated. This saves time on evaluations and can prevent the evaluation of an erroneous condition. Further, it explicitly defines what the evaluation is and leaves the choice up to the user. Many languages employ the "lazy evaluation" as implied by the "and then" and the "or else," but it is left up to the user to find out what each implementation does. With Ada, the user will always have both options.

CASE Structure

The CASE statement is another example of a conditional. This structure is most useful when many alternatives are available or when an expression may take on many values and an "if" would be unwieldy. In the following example, P1 is an object/expression which will evaluate to a discrete value. O1 and O2 are individual/sets/ranges of values which P1 may take on and the code-block is that code to be executed when P1 evaluates to any of the values in the associated "when" statement. (The "=>" is read as "then".)

```

case P1 is
  when O1 =>
    code-block;
  when O2 =>
    code-block;
  ...
  when other => (this is optional if-and-only-if the above
conditions exhaust the entire range of values to which the expression P1 can
evaluate.)
    code-block;
end case;

```

DO Structures

DO Structures are constructed based on the basic loop ... end loop; construct.

```

loop
  code-block(s);
end loop;

```

The above construct will cause the execution of the contained code-block(s) "forever". To better fit the accepted corollary, Ada has a "while P loop ... end loop;" construct. The following example reflects the loop execution while some condition(P) evaluates to TRUE:

```

while P loop
  code-block(s)
end loop;

```

There is no specific "DO-UNTIL", but it may be simulated with the basic loop structure and the "EXIT WHEN" statement immediately before the "end loop" statement. The "EXIT WHEN" indicates that when condition P evaluates to TRUE, control transfers to the instruction immediately following the nearest "end loop" statement.

```

loop
  code-block(s);
  exit when P;
end loop;

```

To improve the maintainability of code, Ada permits "user-defined" iteration. The form is "FOR identifier in some-range-of-value loop ... end loop;". In the following example, P1 and P2 represent begin and end points of some discrete type (e.g., integer, enumeration, character, etc.) where P1 is less than P2; over which V is incremented one member at a time. To reverse (decrement) the direction, the word "reverse" follows the keyword "in".

```

for V in P1 .. P2 loop
  code-block(s);
end loop;

```

Include Capability

The capability to reuse blocks of code in other code blocks is a useful programming technique. A program may import external source code, in-line, with the pragma INCLUDE ("file name where source code resides") statement.

Similarly, operations, types and objects may be "imported" from libraries via the "context clause" by means of the "with" statement. "With" makes the visible elements of a library package available to the Ada program. This differs from the "include" Pragma in that WITHing sets up external references to object code and the source file is not changed during compilation.

Section 4. Module Construction

Each software module shall be constructed to have the following characteristics.

4.1 Completeness

The beginning and end of any program or control segment shall be contained in a single structured module.

4.2 Function

Each module shall perform a single well-defined function, where all the elements of the module contribute to performing only one function. In addition, implementation details of the module should be hidden from all other modules. Should the implementation details change, only the module performing the function would be directly affected and only that module would require modification.

4.3 Entry/Exit

Each module shall contain a single entry and a single exit. Upon completion of processing, modules shall return to the calling routine, not exit to another module. Interfaces (entry and exit conditions and parameters) shall be clearly indicated and defined.

4.4 Declarations

Declarative and data statements in a module shall be placed after the preface commentary block and before the first executable statement. Although more than one data declaration per line is permissible, care should be taken to ensure that complex data structures are defined clearly.

4.5 Imbedded Constants

Literal constants shall not be imbedded in executable statements and shall be avoided in declarations or record and array sizes. Constants shall be defined in the declaration section of the program so that needed changes can be made in one line of code. Examples of possible imbedded constants are:

- . Array sizes
- . Value of pi
- . Indexing information in DO FOR loops
- . Array subscription
- . Record sizes or line lengths.

4.6 Arguments

In general, arguments in call statements shall not contain arithmetic or logical expressions. Each argument shall be represented by a single variable. The number and types of arguments in the call to a module shall be the same as the module's formal parameter list; i.e., no short calls. In addition, any dummy elements used in the call must be identified.

4.7 Exponents

Whole numbers used as exponents shall be expressed as integers; i.e., A**3 not A**3.0.

4.8 Mixed Mode Arithmetic

Mixed Mode arithmetic expressions shall be critically examined and avoided if possible. When no other recourse is available, evaluate the expression carefully to ensure that the type conversion yields the expected results. To help future code maintenance, a comment should bring attention to the conversion. The intent is to make any type conversions explicit.

4.9 Explicit Branching

In order to implement structured programming, it is necessary to prohibit the use of explicit branching (GOTO type instructions). Only branching instructions which are necessary to simulate structured programming constructs are permitted.

4.10 Error Handling

All errors shall be logged to record their occurrence and to facilitate later analysis of error conditions. There must be a record of all errors that occur in the system.

All error conditions shall have known responses. The potential effects of errors shall be known in advance and responses determined before they occur.

Errors shall not be permitted to propagate. They should be discovered at the place where they occur and shall be handled in such a manner that one error does not cause others, producing a cascading effect.

Upon encountering an error which does not permit further execution of the module, the module or operating system shall free any resources the module has acquired before it terminates.

4.11 Range Checks

Limit checks shall be performed to ensure that variable contents are within the expected range of values. If a variable value does not fall within the specified limits, appropriate error action shall be taken. The domain of all variables shall be specified during design, declared during implementation, and checked during operation. Examples of variables which shall be range checked are:

- . Parameters
- . Array subscripts
- . Variable parameters in DO CASE structures
- . Variables used as initial, incremental or terminal values in looping structures.

4.12 Indices and Subscripts

Loop index parameters and array subscripts shall be expressed as integer constants or variables. When the language supports it, they may be expressed as enumeration data types.

4.13 Loop Termination

Loop termination shall be guaranteed. Termination conditions shall not depend on the correctness of assumptions made concerning the loop parameters unless these assumptions have been verified immediately prior to loop entry.

4.14 Use of Labels and Names

Labels and names shall be meaningful and obviously unique. Labels such as LABOT1 and LABØT1 are confusing, and typographical errors are easy to make and difficult to discover.

4.15 Global and Shared Variables

Global or shared variables should be avoided. When a variable is shared between two or more modules, each module must trust the other modules to use the variable properly. Reliance on correct use involves a risk that may not be acceptable.

Variables which are required in more than one module should be communicated as arguments. Limiting the scope of variables helps to ensure that the exact meaning of a variable is known.

In those cases where there is clear technical justification for the use of common data, the following restrictions will apply:

- . Only common areas which can be accessed by name will be used (e.g., FORTRAN's named COMMON).
- . The length and description of common data areas will be the same for each module using the block.
- . When possible, only one copy of the common block will be maintained and, when possible, must be "INCLUDE"d in each module rather than coded in-line.

4.16 Standard Linkage Conventions

If the module is being implemented in assembly language, standard conventions for communications between modules shall be defined and used. The goal is to construct modules which are independent, self-contained, and which communicate with other modules in well-defined ways. Some items which should be considered are:

- . Registers shall be saved before relinquishing control to another module and restored when control is returned.
- . Communication shall be via an argument list.
- . There shall be a standard method for communicating the calling module's return address.

4.17 Input and Output

Input routines and output routines shall be collected into a limited number of modules. Assembling these routines into a small number of modules facilitates the monitoring and controlling of input and output data. In addition, modification is made easier by having these routines restricted to a limited area.

4.18 Naming Standards

The names for units of code and data shall be descriptive of their functions or content. They shall be related to other software and supporting items, while following language conventions and constraints. Special prefixes and suffixes may be appended to indicate additional information such as data types of variables or task entry points.

4.19 Preface Commentary Standards

The following contains standards for a set of comments to be used as in-source documentation for every program module. Each module of code shall contain a standardized block of comment statements, immediately following the module declaration, which describes the module's function, usage and operation requirements. The preface shall be identified by a unique comment line indicator to facilitate automated searches and extractions of information. The preface commentary shall contain, at a minimum, the following information.

4.19.1 NAME Section - The NAME section of the preface block shall contain the name used to identify the module, its version identifier, and its entry point.

4.19.2 PURPOSE Section - The PURPOSE section of the preface block shall contain a brief description of the purpose/function of the module.

4.19.3 INPUT/OUTPUT Section - The INPUT/OUTPUT section of the preface block shall contain the name of each I/O file used by the module with an indication of whether the file is input to the module, output from the module or both. In addition, modules utilizing interactive screen I/O should provide a brief synopsis of the screen's function.

4.19.4 PARAMETER Section - The PARAMETER section of the preface block shall provide a definition of all arguments required by the module and all values (output parameters) returned by the module. The definition of parameters shall include: name, data type, size, and functional use.

4.19.5 CALLS Section - The CALLS section of the preface block shall provide a list of all routines called by this program module and a list of all routines that call this program module.

4.19.6 GLOBAL DATA Section - The GLOBAL DATA section of the preface block shall provide a list of all common or global data used in the program module.

4.19.7 RESTRICTIONS Section - The RESTRICTIONS section of the preface block shall contain a list of any special or unusual features which restrict or limit the module's performance characteristics.

4.19.8 ABNORMAL END Section - The ABNORMAL section of the preface block shall contain a list of abnormal return conditions and actions.

4.19.9 METHOD Section - The METHOD section of the preface block shall contain a detailed description of the methods used by the module to perform the function described under PURPOSE. This functional description shall reference any documents which contain pertinent illustrations, graphs and tables, or which provide more detail concerning equations solved and algorithms employed.

4.20 In-Line Commentary Standards

Commentary within the body of source code shall define the processing being performed. These comments shall identify the purpose of each control statement. A control statement is defined as one which conditionally alters a data value or alters the sequential execution of statements. For ease of reading, comments may be grouped at the beginning of a set of logically-contiguous statements. They should be positioned so as not to obscure the visual cues about program structure provided by indentation and paragraphing. As a minimum, in-line comments should precede blocks of one or more of each of the following:

- IF statements - Describe the reason the action is performed if the test is satisfied
- Input/Output statements - Identify the nature of the record or file being processed
- DO statements - Describe the reason the action is being performed
- CALL statements - Describe the reason the procedure is being called

4.21 Indentation and Paragraphing Standards

Source code shall be clearly indented to indicate the span of control of a structured figure so that logical relationships in coding correspond to physical positions on program listings. Executable statements are restricted to one per line. The continued portions of statements requiring more than one line should be indented in a block beneath the first line of that statement. More than one data declaration per line is permissible; some logical order of data declaration should be used, however (e.g., alphabetical). Lengthy statements, such as complicated conditional branches, should be broken down into sequences of shorter code. Blank lines or their equivalent shall be used to provide proper separation of routines and to enhance readability.

Section 5. Product Development Library

5.1 Introduction

A product development library is a practical necessity in the implementation of modern programming technology. It serves as a repository for the data necessary for the orderly development of computer programs. A product development library has four components:

1. Internal libraries are used to store data (e.g., source code, object code, input data) in machine readable form.
2. External libraries are used to store corresponding data in hard copy (human readable) form.
3. Computer procedures provide easy access to computer processing of data stored in internal libraries. These include procedures for compiling and assembling source code, link editing, and system testing.
4. Office procedures specify such things as the maintenance procedures for both internal and external libraries, and standard formats for external libraries.

The components of the product development library should establish and ensure that there is an exact correspondence between the internal (computer readable) and external (programmer readable) versions of the developing system. This basic correspondence is established by the computer and office procedures.

A product development library may be implemented in any of the following forms: a manual system, a basic system with some automated support, and a third level (the most automated level) with full management data collection.

5.2 Manual Product Development Library

The most rudimentary system is a manual, non-automated one. The manual system is the easiest and least expensive way of collecting data. It should be implemented in accordance with the following:

- a. General Requirement. A computer program and data repository shall be established and maintained under control of a librarian in a central location. Facilities and procedures for the generation, storage, and maintenance of all software developed shall be implemented. The procedures established shall provide the following:

(1) The identification and delegation of responsibilities for clerical and record keeping functions associated with the programming process and the maintenance of the library.

(2) The delegation of responsibilities for all machine operations with regard to such items as project initiation/termination, program test philosophy, output media/frequency, etc.

(3) The procedures for recording, cataloging, and filing of all code generated on the project, both intermediate and final; and for the retention of superseded corrected code for stated retention periods.

(4) A method for controlling the version(s) of the code contained in the library and the method for providing visibility into this process by configuration management personnel.

(5) A method for collecting and disseminating basic management data on the use of the library facilities and status of the programming activities.

5.3 Basic Product Development Library

A basic system should be implemented in accordance with the following:

a. General Requirement. A data repository shall be maintained in two forms. Data is stored in an internal library which is computer-resident in machine readable form. The identical data is also stored in an external library which is in hard copy form, in organized project notebooks. Procedures shall be established for this repository to define:

(1) The delegation of clerical and record keeping operations associated with the programming process and the maintenance of both the internal and external libraries.

(2) The delegation of all machine operations with regard to such areas as project initiation/termination, program test philosophy, output media/frequency, etc.

(3) The requirements for the recording, cataloging, and filing of all code generated in the project, both intermediate and final; and for the retention of superseded corrected code for stated retention periods.

b. Source Data Maintenance. The product development library shall be structurally organized to provide the following capabilities:

(1) Data File Storage - The capability to store within the internal library files of source and object code, control language, code enabling the loading and execution of several files stored, etc.

(2) Data Access - Direct access of a single unit of stored data in a timely fashion. Further, the capability to perform a multi-library search during retrieval of input for purposes of compilation, linkage, execution, and output processing.

(3) Library Backup - A capability to recover from inadvertent loss or destruction of data. This involves back-up storage on a storage unit independent of the internal library (e.g., magnetic tape, disk pack, punched cards) and also the regeneration of the library data files from this back-up storage. This back-up capability shall selectively regenerate and restore data so that it is possible to recover portions of the total product development library data base without the need to perform full storage dump and restore operations.

(4) Data Maintenance. The capability to add, delete, or replace one or more units of data in an internal library file; to copy one or more segments of data from one library to another. Further, the automatic generation of program stubs for segments of source code which remain to be developed (coded) and are not currently stored within the library. Lastly, the capability to merge two data files from two different product development libraries into a single data file.

c. Output Processing. This involves the output of data stored in the file(s) and the output of related control data for use by both programming and management personnel. As a minimum, data output shall include:

(1) Project Status Listings - Status information with regard to a section within a project to include such items as module name, date/time information retrieved, storage space allocated and used, module language, creation/last update, number of lines per module.

(2) Source Listings - Printed output of any segment which is in card-image format. The output contains such items as print date/time, segment type, line numbering and nesting-indentation, segment update information.

(3) Program Structure Report - This output (for a specified module) illustrates a hierarchically nested and indented list of all modules which are referenced by an "include" statement, either in the originally requested module or in modules which are themselves "included." Provisions are also included with regard to the "call" statement. An alphabetically arranged list of all referenced modules follows the hierarchical listing.

(4) **Magnetic Tape** - The capability to copy one or more specific segments of data or files of data onto magnetic tape for permanent storage or for distribution to other computer facilities.

d. **Programming Language Support.** This capability includes the validation and compilation of program source code stored in the product development library. The product development library shall interface with or invoke pre-compilers and compilers for the purposes of syntax checking of both structured and unstructured source code, compiling the source code, and storing the resultant object module within the library. Further, it shall provide load module generation to execute the necessary computer programs (i.e., to convert object modules into executable programs, to store the resulting machine code in the library, to load the program for subsequent execution).

e. **Library System Maintenance.** To provide procedures to generate and maintain the product development library system. This includes product development library installation; initialization of product development library projects; creation of any product development library data files (sections); deletion of a section, library, or project; and backup/restore either on a project, library, or section level.

f. **Privacy/Access Constraints.** The product development library shall provide for the integrity of the data stored. This includes maintaining the separation of different versions of projects/programs under development, control over inadvertent destruction of data, and protection of data from unauthorized access.

5.4 Full Product Development Library with Management Data Collection and Reporting

A full product development library with management data collection and reporting shall contain the functions defined for the Basic Product Development Library and the additional capability of providing full privacy/access constraints, documentation support, and management data/collection/reporting as specified herein:

a. **Privacy/Access Constraints.** The product development library shall provide control over the integrity and security of the data stored within the library; namely, control over different versions of projects/programs under development, control over the inadvertent destruction of data, and control with regard to the protection of data from unauthorized access. The design requirements listed herein are not intended to support the protection of classified data.

(1) **Data Integrity** - Procedures shall be established for maintaining control over multiple versions of a program and also control over various change levels of program source segments with a specific

version. A version is defined as a complete program or program system as it exists at a specific point in time, usually related to a specific milestone in the development cycle. The objective of version and level control is to allow a development and/or maintenance group to manipulate multiple versions of the same program system (e.g., operational version, development version, maintenance version) stored in separate libraries without unnecessary duplication of source or object code. With version control, it is possible to merge data from several libraries for the purpose of performing tests.

(2) Data Destruction - Procedures shall be established to limit data destruction (and likewise data recovery) to authorized personnel only. At a minimum, these procedures shall establish controls at the project, library, and data file levels as defined for the designated product development library installation. Further, the product development library design shall prevent the addition or copying of data segments into a product development library file if segments with the same names already exist in the file.

(3) Data Protection - The product development library shall contain facilities for the protection of data stored in project files and for the prevention of unauthorized access and update.

b. Management Data Collection and Reporting (MDCR). This capability involves the collection and storage of data related to program development and maintenance and subsequently, the generation of management reports containing the data and/or summaries of data. A separate management data file is generated and maintained for each project for which management data is collected. The minimum requirements for the MDCR are as follows:

(1) Functional Requirements. The major processing functions required to satisfy the MDCR functional requirements are:

(a) Collecting - A facility to automatically gather and store necessary data in individual computerized data bases such as test data, source code, object code, etc.

(b) Updating - A capability to add, delete or replace management data.

(c) Archiving - The capability to record historical management data.

(d) Reporting - A facility to retrieve and output management data in a convenient and meaningful manner.

(2) Collection and Reporting Levels. The MDCR shall collect, update, accumulate, and report data at different levels of the software hierarchy.

(3) Data Classes and Types. Collected data stored in the product development library to support planning and managing the software development shall be as follows:

(a) Project Environment Definition - This data, entered at the system level during the early stages of a project (i.e., Software Requirements Specification of System/Subsystem Specification), shall define such items as project title, narrative description of project, milestone schedule including start/end dates, project personnel needed (job type, familiarity with hardware and program language, experience level, number, etc.), estimated travel/workspace requirements, project complexity and hierarchical structure, information regarding estimated amounts and types of documentation to be produced (i.e., standards, users guides, test plans and reports, installation manuals, etc.), and project summary reporting format/frequency.

NOTE: Storage of the above data in the product development library is optional since it may be acceptable to collect this data on a one-time manual form.

(b) System Description Definition - This data describes the modules and programs constituting the software system being developed. This data, most of which should be automatically collected, contains such items as module or program name, development state/end dates, programmer(s) responsible, number of lines of source code, program language, revision number/date, etc.

(c) Computer Utilization Definition - This data describes the usage of the computer facility. It includes such times as estimated computer turnaround time, job name, job submitted/received times, programmer name, and number of runs per job.

(d) Resource Cost Definition - This data describes the dollar cost and/or man-months needed for the project in terms of several categories (e.g., personnel, travel, computer time) stored on a reporting cycle/period basis. The data consists of such items as project name, man-month effort for each reporting cycle by personnel type (i.e., managers, programmers, analysts, clerical), material costs (supplies, equipment rental, etc.) for each reporting cycle, personnel/travel costs, computer time/costs, and miscellaneous costs (special hardware, office space, etc.).

(e) Program Product Definition - This data includes information pertinent to the quality assurance and production of source code such as number of compilations/assemblies, lines of source code affected (input, added, deleted, changed), information pertinent to efficiency/documentation improvements realized, errors experienced (keying clerical, standard omissions/misinterpretations, etc.), and extensions/changes to initial software requirements specification for project/program.

(4) Report Classes. A report generation capability shall be included in the MDCR to output at least the following information:

(a) Program Module Statistics - Statistics output relative to each of the hierarchical levels. Detailed reports produced at the module and unit level contain such information as period of reporting cycle, module name/programmer, module version, last date modified, number of update executions, number of lines of source code involved both cumulative and for the reporting period. Summary type reports are produced at the project level and contain such information as name, start date, number of programs/modules, number of lines of higher order language source, and number of lines of assembly code.

(b) Computer Utilization Statistics - These reports include both detailed reports and summary reports for the project manager. Examples of data reported are job name, number of runs both to date and for the reporting period, average computer turnaround time both to date and for the reporting period, and maximum computer turnaround time both to date and for the reporting period.

(c) Program Product Statistics - These reports track the program development and update activity. Data shall include reasons for making the updates.

(d) Program Structure Reports - These reports list the total tree structure of the program starting at the module specified by the user and including all "included" or "called" source modules. The reports will also include a cross reference listing of all "included" or "called" source modules.

(e) Historical Reports - The archival data to be retrieved would be identified by using search criteria supplied by the user such as project name, date range, and range of project size (expressed in terms of project source modules). These reports will contain cost data and key historical data items from the other MDCR Report Classes cited in this paragraph for both active and inactive software projects.

c. Documentation Support. The basic requirements include the storage, update and output of product development library data as specified for the MDCR. Other desirable, but not required, capabilities are as follows:

(1) Print in a user specified sequence one or more modules of program stored in the product development library. There also should be a capability to merge segments of program design language, program source code, and textual data into one output listing.

(2) Format an output listing in accordance with the following user supplied information: header to be printed at top of page; spacing between segments of data; spacing between lines of output; and number of lines to print on a page.

(3) Automatic page numbering beginning with a user-supplied page number.

(4) Print a title page containing: document title, document date, author's name, organization and address.

(5) Generate a magnetic tape in print image form. This allows the distribution of documentation in machine-readable form for easy storage and reproduction.

PART V - SOFTWARE TEST AND OPERATIONS

5.1 SOFTWARE GENERAL UNIT TEST PLAN

5.1.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 5.1).

Software developers shall prepare a General Unit Test Plan which defines the scope and standards for the testing that must be successfully completed for each software unit.

The General Unit Test Plan shall contain, as a minimum:

- a. General project standards for unit test thoroughness;
- b. A description of how the developer will conduct and monitor unit testing to assure compliance with the Plan;
- c. An identification of unit test input data that must be supplied by external sources and the plan for obtaining these data.

5.1.2 GUIDANCE

This document should not describe any test cases. It is a project-wide scope and standards document for unit-level testing. Unit-specific test information (e.g., unit test case descriptions) appears in each Unit Development Folder.

5.1.3 FORMAT FOR THE GENERAL UNIT TEST PLAN

GENERAL UNIT TEST PLAN

TABLE OF CONTENTS

Section 1.	Purpose and Scope
Section 2.	Applicable Documents
2.1	Standards
Section 3.	General Project Standards for Unit Testing
Section 4.	Unit Test Implementation
Section 5.	Unit Test Input Data from External Sources
Section 6.	Quality Assurance

Section 1. Purpose and Scope. This section shall describe the purpose and scope of unit testing and shall describe its relationship to other test phases.

Section 2. Applicable Documents. This section shall identify all documentation that is used as a basis for this test plan, including documentation that is required to support the unit testing that is being specified. The document title, number, date of issue, etc., shall be presented for each document listed.

2.1 Standards. This paragraph shall reference any specifications or other relevant standards that will be used during the test design, during the preparation of the test plan, or during unit testing.

Section 3. General Project Standards for Unit Testing. This section shall describe general project standards for thoroughness of unit testing. Examples of standards for unit tests are as follows:

- a. Verification of all computations using nominal, singular, and extreme data values;
- b. Verification of all data input options;
- c. Verification of all data output options and formats, including error and information messages;
- d. Exercise of all executable statements in each unit at least once;
- e. Test of options at branch points in each unit.

Section 4. Unit Test Implementation. This section shall describe how the developer will conduct and monitor unit testing to assure compliance with this plan. Some examples are:

- a. A description of the criteria and techniques to verify compliance of the code with unit-level design budgets (e.g., sizing, accuracy, timing);
- b. A description of how the project will verify, at the unit level, that applicable interface requirements have been satisfied;
- c. A description of the support software that will be needed for unit testing (e.g., test data generators, test drivers, dynamic path analyzers).

Section 5. Unit Test Input Data from External Sources. This section shall identify unit test input data that must be supplied by external sources (e.g., users) and the plan for obtaining these data.

Section 6. Quality Assurance. This section shall define the developer's management procedures for reviewing and evaluating unit test results.

5.2 SOFTWARE SYSTEM INTEGRATION AND TEST PLAN

5.2.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 5.2)

Software projects shall plan for and conduct software integration and test activities which will be accomplished prior to Software System DT&E. These test activities shall, as a minimum:

- a. Integrate the software units into a cohesive, nominally executing software product;
- b. Demonstrate the satisfaction of the complete requirements set in the Software Requirements Specification;
- c. Demonstrate software operability over a range of operating conditions (normal and abnormal).

These activities shall be described in a Software System Integration and Test Plan. Essential test resources, including test support tools and equipment, shall be available and ready by the start of associated test activities.

A preliminary version of the Software System Integration and Test Plan shall be prepared prior to PDR. The complete plan shall be prepared prior to CDR and, as appropriate, updated prior to the initiation of the integration and test activity.

Before integration testing begins, the Software Acquisition Manager shall meet with the Developer to evaluate the adequacy of the test planning and preparation activities. After integration testing is completed, the Software Acquisition Manager shall evaluate test results and verify that the software is ready for Software System DT&E.

5.2.2 GUIDANCE

The Software Integration and Test activity includes the integration of software units into builds and the integration of single builds into larger builds. When all of the builds have been assembled, integrated, and tested, the final build is a fully integrated software system that executes in a controlled test environment. The Software Integration and Test Plan describes the approach for all levels of software integration and the plan for testing the software elements as they are joined to form an operational configuration.

For each build, the first integration tests should demonstrate that software units execute together. Once this is proved, a second group of tests should be planned to verify the control paths through the software. This includes testing to ensure that controls exercised through the man-machine interface and other external sources are correct and that internal control paths function as they were designed to function. The next group of

integration tests should test the internal data integrity of the software. These tests may require instrumentation to monitor data values and system changes affected by changes in data values. After the software passes these integration tests, a fourth group of tests should be conducted to verify support to external software interfaces, performance, accuracy, error recognition, and recovery. As new builds are added to existing builds, additional tests should be conducted to demonstrate that features and capabilities of earlier builds still function correctly with the new features and capabilities.

5.2.3 FORMAT FOR THE SOFTWARE SYSTEM INTEGRATION AND TEST PLAN

SOFTWARE SYSTEM INTEGRATION AND TEST PLAN

TABLE OF CONTENTS

Section 1.	Purpose and Scope
1.1	Relationship to Other Test Activities
Section 2.	Applicable Documents
2.1	Development Specifications
2.2	Standards
2.3	Other Publications
Section 3.	Integration and Test Identification
Section 4.	Resources Required
4.1	Personnel Requirements
4.2	Facilities/Hardware
4.3	Interfacing/Support Software
Section 5.	Test Management
5.1	Integration Test Team Organization & Responsibilities
5.2	Responsibilities of Other Organizations
5.3	Product Control
5.4	Test Control
5.5	Evaluation and Retest Criteria
5.6	Test Reporting
5.7	Test Review
5.8	Test Data Environment
Section 6.	Test Structure and Design
6.1	Test Levels
6.2	Test Approach
6.3	Test Inputs
6.4	Test Cases/Classes of Tests
6.5	Test Identification
Section 7.	Software Requirements to be Satisfied Through Integration Testing
7.1	Software Requirements
7.2	Requirements Verification Traceability
Section 8.	Schedules

Section 9. Turnover Criteria
9.1 Beginning of the Integration Test Activity
9.2 Completion of the Integration Test Activity

Section 10. Notes

Section 1. Purpose and Scope (PDR). This section shall describe the purpose and scope of software system integration testing and describe its relationship to other test phases. It shall describe the top-level concepts and goals for integration and the strategy for fulfilling these goals. It shall also describe how the integration strategy will provide for verification of essential system services, such as an augmented operating system or a new file manager, before beginning integration of software using these services.

1.1 Relationship to Other Test Activities (PDR). This section shall describe an activity network which shows the interdependencies among the various integration test events and schedules.

Section 2. Applicable Documents (PDR). This section shall identify all documentation that is used as a basis for this test plan, including documentation required to support the integration testing that is being specified. For each document listed, the document title, number, date of issue, etc., shall be given.

2.1 Development Specifications (PDR). This paragraph shall list all development specifications (e.g., Software Requirements Specification, System/Subsystem Specification, Program Specification, Interface Specifications) that identify the items being tested.

2.2 Standards (PDR). This paragraph shall list any specifications or other relevant standards that are used during the test design, during preparation of the test plan, or during integration testing.

2.3 Other Publications (PDR). This paragraph shall identify reference documents, manuals, diagrams, exhibits, etc., that are used with this test plan or activities described in this test plan.

Section 3. Integration and Test Identification (PDR). This section shall identify the computer programs that will be tested and turned over for formal Software System Development Test and Evaluation. It shall also identify the criteria used to select "Builds" (or capability increments) for integration and testing.

Section 4. Resources Required.

4.1 Personnel Requirements (PDR). This section shall describe the responsibilities, authority, and particular knowledge and skills required for personnel conducting integration testing activities. It shall also describe the number of people involved in this activity.

4.2 Facilities/Hardware (PDR). This section shall identify the facilities, computer hardware, and interfacing hardware to be used for integration. It shall also identify any planned transitions in facility location, computer hardware, or interfacing hardware during integration, and the plans for accommodating problems associated with the transitions (if any).

4.3 Interfacing/Support Software (PDR). This section shall describe interfacing and support software needed to conduct the integration test activity (e.g., operating system, pre-processors, test drivers, test data generators, post-processors, other computer programs).

Section 5. Test Management

5.1 Integration Test Team Organization and Responsibilities (PDR). This paragraph shall describe individual and organizational responsibilities for conducting and coordinating the integration testing activity. It shall also identify the names of people who are responsible for the activities.

5.2 Responsibilities of Other Organizations (PDR). This paragraph shall identify any responsibilities or requirements for participation of customer, user, or other organizations in the integration testing activity.

5.3 Product Control (PDR). This paragraph shall identify the software and hardware control procedures to be used during integration testing. These procedures can be included by reference to the Configuration Management Plan, if applicable.

5.4 Test Control (CDR). This paragraph shall describe procedures to be used by integration test personnel for controlling integration test activities and products produced during those activities (e.g., deck submittal, output storage, team meetings, etc.).

5.5 Evaluation and Retest Criteria (CDR). This section shall define the developer's management procedure for reviewing and evaluating test results. The evaluation criteria (successful test criteria or accept/reject limits) for each test shall be the means for determining success or failure of the tests. This section shall also identify the criteria for retesting integration tests.

5.6 Test Reporting (PDR). This paragraph shall identify requirements and procedures for preparing and reviewing reports of formal and informal testing at each level of integration testing. This includes reporting results of integration testing to both the developer's and the customer's project management.

5.7 Test Review (PDR). This paragraph shall identify the informal developer reviews of each level of integration testing. It shall also identify the formal developer review at the conclusion of integration and test that will determine the readiness of the software system for formal Software System DT&E.

5.8 Test Data Environment (PDR). This section shall identify the activities required to generate and validate the software data base to be used in integration testing. It shall also identify the activities required to produce: (a) nominal or calibration test data, and (b) stress test data (including noisy and otherwise imperfect data) for each level of integration.

Section 6. Test Structure and Design

6.1 Test Levels (PDR). This section shall identify the "Builds" (capability increments) which will undergo integration and testing. Within each Build, it shall identify the levels of integration and test to be conducted (e.g., task, sub-function, function).

6.2 Test Approach (PDR). This paragraph shall identify the test approach for each test level and Build (or capability increment). It shall also describe the concepts and methods for stressing the software system (e.g., with noisy and otherwise imperfect input data, or with peak loads) during integration testing.

6.3 Test Inputs (CDR). This section shall identify how each software entity to be tested (at each level of integration) will obtain necessary test data (e.g., from a test generator, from deliverable software previously verified during integration).

6.4 Test Cases/Classes of Tests (PDR). This section shall identify the integration tests and classes of test cases to be executed for each capability increment. The level of detail should be appropriate for the project. Some examples of classes of tests include:

- a. Computational tests to verify the accuracy of computer programs in achieving quantitative results:

- b. Data handling tests to demonstrate that specific features are effectively accomplished (e.g., data collection and merger, data conversion, bad data disposition, data accountability, linking table functions, etc.).

- c. Interface tests to demonstrate that groups of handler routines function properly and that the peripheral device reactions are acceptable. All interfaces must be tested. Tests should include each input/output

medium, exercise all transaction formats and code translation, and verify error rejection according to the performance requirement.

d. Processing tests to demonstrate the effectiveness of equipment performance, data flow, privacy checks (authentication, access restrictions and reaction to illegal penetration attempts), operator functions, and related features as listed in the Development Specifications.

e. Saturation tests to overload the computer by creating the conditions necessary to activate overload software safeguards (e.g., threshold alarm notice, overload spillover function, halt input, remove low speed devices, etc.).

f. Recovery tests to demonstrate the ability to recover all data lost at disruption time. A test for each (disk/drum/tape) recovery method must be conducted and provide evidence that all data was in fact recovered.

6.5 Test Identification (CDR). For each Build, this section shall identify and describe each integration test that will be conducted. Each test shall be described as follows:

- a. Test Identification.
- b. Purpose.
- c. Software elements to be integrated and tested.
- d. Input data.
- e. Evaluation method/acceptance criteria.

Section 7. Software Requirements to be Satisfied Through Integration Testing

7.1 Software Requirements (PDR). This section shall identify (possibly by reference to specification documents or specification identifiers) the requirements whose satisfaction will be demonstrated by the integration test activity.

7.2 Requirements Verification Traceability (PDR). This section shall present the relationship of the software requirements to the test structure and design described in Section 6.

Section 8. Schedules (PDR). This section shall refer to the schedule information in the integration and test activity network of Section 1.1. It shall identify the schedule for conducting particular tests or levels of tests identified in Section 6 and the order in which the tests will be conducted. The schedule may be presented in terms of general periods (weeks or months) for the tests.

Section 9. Turnover Criteria

9.1 Beginning of the Integration Test Activity (PDR). This section shall describe the criteria for beginning integration testing.

9.2 Completion of the Integration Test Activity (PDR). This section shall define the criteria for ending the integration test activity and beginning Software System DT&E testing.

Section 10. Notes. This section may be used to present examples, charts, exhibits, flow charts, traceability matrices, data value content, etc., that are too lengthy or bulky to be included in the test plan.

5.3 SOFTWARE SYSTEM DEVELOPMENT TEST AND EVALUATION (DT&E) PLAN

5.3.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 5.3)

Software developers shall have a Software System DT&E Test Plan and Test Procedures to provide a controlled definition of the project's DT&E test program.

A preliminary Software System DT&E Test Plan shall be reviewed at PDR. The complete Software System DT&E Test Plan shall be reviewed in a CDR session for the purpose of achieving written agreement with the developer that the execution of the defined test cases in a manner described in the Plan, using an approved data base and with test results which satisfy the defined acceptance criteria, will result in acceptance of the software end products.

The DT&E test program shall be directed by a developer test organization which is independent in the sense that it is not subordinate to those responsible for designing and coding the software system.

For interactive or operator-oriented software, where the test results are dependent on a specific scenario of conditions, software system DT&E test procedures shall be required. These procedures shall be derived from the Software System DT&E Test Plan and shall be placed under project configuration control prior to the start of DT&E testing.

5.3.2 GUIDANCE

The Software System DT&E Plan is prepared in two stages: (1) a preliminary version by PDR, and (2) a complete version by CDR. The format described in Section 3 notes the parts that should be completed by PDR. The remaining parts should be completed by CDR. Software Development Test and Evaluation (DT&E) is the final level of software testing. It is a planned, structured demonstration that the software satisfies functional and performance requirements. It requires custom participation and approval.

Software DT&E must include enough tests to demonstrate the following:

- a. The software can support the full range of operational capabilities required by the Software Requirements Specification.
- b. The software satisfies performance requirements and operational and development constraints.
- c. The software supports external interface requirements.
- d. The software can support man-machine and system control interfaces.

5.3.3 FORMAT FOR THE SOFTWARE SYSTEM DT&E TEST PLAN

SOFTWARE SYSTEM DT&E PLAN

TABLE OF CONTENTS

Section 1.	General
1.1	Introduction
1.2	Purpose
1.3	Criteria for Conducting Software System DT&E
1.4	Project References
Section 2.	Test Requirements and Acceptance Criteria
Section 3.	Test Descriptions
3.1	Classes of Tests
3.2	Test Case Structure
Section 4.	Software Requirements/Test Specification
Section 5.	Resources Required
5.1	Personnel Requirements
5.2	Facilities/Hardware
5.3	Support Software
Section 6.	Data Base for Software System DT&E
Section 7.	Test Management
7.1	Protocols
Section 8.	Customer Support Requirements
Section 9.	Software System DT&E Test Schedules
9.1	Master Test Activity Schedule
9.2	Activity Network for DT&E Testing
Section 10.	Software Modifications and Retest Criteria
Section 11.	Notes

Section 1. General

1.1 Introduction (PDR). This section shall contain a brief overview of the entire document. It shall emphasize the important concepts or considerations involving Development Testing and Evaluation of the software system and the plan describing the activities.

1.2 Purpose (PDR). This section shall describe the objectives and purpose of the DT&E plan. It shall explain the relationship of Software System DT&E to other test phases, including integration testing and system-level testing.

1.3 Criteria for Conducting Software System DT&E (PDR). This section shall define the criteria by which the software system will be judged ready for DT&E. (This section should be the same as Section 9.2 of the Software System Integration and Test Plan.)

1.4 Project References (PDR). This section shall list the reference documents applicable to the development of the software system.

Section 2. Test Requirements and Acceptance Criteria (PDR). This section shall contain the test requirements for the software end-product acceptance and criteria for acceptance of the software system by the customer.

Section 3. Test Descriptions

3.1 Classes of Tests (PDR). This section shall describe the classes of software DT&E tests and their dependency relationships with each other. For each class of tests, this section shall identify:

- a. Test purpose.
- b. Software requirements to be demonstrated by the test.
- c. Special software, hardware and facility configurations to be used.
- d. Test input environment and output conditions.
- e. Critical analysis techniques relating test output to acceptance criteria.

3.2 Test Case Structure (CDR). This section shall describe in subparagraphs 3.2.1 through 3.2.n each test to be conducted for software DT&E testing. For each test, it shall identify:

- a. Test identification.
- b. The software requirement to be demonstrated by the test.
- c. Test input requirements.
- d. Software and hardware configuration to be used.
- e. Support software to be used.
- f. Major software entities to be exercised by the test.
- g. Test output requirements.
- h. Test output analysis method.
- i. Uniquely-identified acceptance criteria.

Section 4. Software Requirements/Test Specification (CDR). This section shall present a table correlating software requirements (from the Software Requirements Specification) with tests (DT&E tests plus applicable lower-level tests) which demonstrate satisfaction of the requirements. Figure 5-1 is an example of a Software Requirements/Test Case Matrix.

Section 5. Resources Required (PDR).

5.1 Personnel Requirements (PDR). This section shall describe the number and qualities of people who will conduct Software System DT&E.

5.2 Facilities/Hardware (PDR). This section shall identify the facilities, computer hardware, and interfacing hardware to be used in DT&E testing. It shall also identify any planned transitions in facility location, computer hardware or interfacing hardware for DT&E testing.

5.3 Support Software (PDR). This section shall describe the support software to be used in DT&E testing. It shall also include plans for obtaining this software and for validating it prior to the start of DT&E testing.

Section 6. Data Base for Software System DT&E (PDR). This section shall describe the data base to be used in DT&E testing. It shall also include plans for obtaining approval of it prior to the start of DT&E testing. In addition, this section describes any plans for obtaining data from support software, support hardware or "live" sources.

Section 7. Test Management (PDR). This section identifies the participants in the Software DT&E phase and the job that each participant will perform. It must identify who is responsible for obtaining, providing, and validating the following:

- a. Software under test.
- b. Support software.
- c. Hardware.
- d. Facilities.
- e. Test drivers.
- f. Test data bases.

7.1 Protocols (CDR). This section shall describe the protocols to be followed by the developer in reviewing, reporting, and accepting test executions and results. This includes the means of controlling changes to test plans, test procedures, and other test materials.

Section 8. Customer Support Requirements (CDR). This section shall summarize requirements for customer-provided hardware, software, data, documentation, facilities, and support during DT&E testing.

Section 9. Software System DT&E Test Schedules.

9.1 Master Test Activity Schedule (PDR). This section indicates the time-phasing of the various classes of DT&E tests. This includes the expected duration of each category of DT&E tests. This schedule must be compatible with development schedules and delivery dates.

9.2 Activity Network for DT&E Testing (CDR). This section shall contain an activity network for DT&E testing. It must include each test case and show the order in which each will be conducted and the interdependencies among test events. The activity network should also show the time-duration for each activity.

Section 10. Software Modifications and Retest Criteria (CDR). This section shall define the criteria and procedures for incorporating software modifications and performing retest with the software modification.

Section 11. Notes. This section may be used to present examples, charts, exhibits, flow charts, traceability matrices, data value content, etc., that are too lengthy or bulky to be included elsewhere in the test plan.

5.4 OPTIONAL TEST AND BUILD DELIVERY DOCUMENTATION

5.4.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policies 5.2 and 5.3)

For interactive or operator-oriented software, where the test results are dependent upon a specific scenario of conditions, the Software Acquisition Manager or the Software Development Manager may require that Software Test Procedures be prepared. These procedures shall be derived from the Software System Integration and Test Plan and the Software System DT&E Plan and shall be placed under project configuration control prior to the start of the test activity.

The Software Acquisition Manager may require that the developer prepare a Software Test Report to document the status and results of integration tests and Software DT&E tests.

5.4.2 GUIDANCE

5.4.2.1 SOFTWARE TEST PROCEDURES

Software Test Procedures contain the detailed procedures for conducting integration tests and acceptance (DT&E) tests. They relate to specific tests identified in Section 6 of the Software System Integration and Test Plan and Section 3 of the Software System DT&E Plan. For tests identified in those plans, Software Test Procedures describe the steps necessary to conduct the tests. They also provide a place to document the results of each test.

Software tests are often conducted by test personnel who did not design or write the code being tested. If high-quality tests are to be conducted, software testers must pay careful attention to the task of writing test procedures. They should keep the following two points in mind when developing the procedures:

- a. The test procedures should be recipe-like and describe the step-by-step actions necessary to conduct the tests and evaluate the results. This allows them to be executed and understood by personnel who are not intimately familiar with the design.

- b. The test procedures must be procedure-oriented rather than essay-style generalities. This allows members of the acquisition team to certify that the procedures were followed to the letter.

At the discretion of the Software Acquisition Manager, separate Test Procedures may be prepared for each test or the procedures may be combined in a single document to cover a group of tests.

5.4.2.2 SOFTWARE TEST REPORT

Software Test Reports record the results of any level or type of software tests. They document the results of testing that has been conducted at various stages of development and provide the Software Acquisition Manager with a means of assessing whether the software can progress to the next stage of development.

Although not mandatory, Software Test Reports are recommended for reporting the results of testing at several points:

- a. completion of integration testing of each build;
- b. completion of integration testing of the entire software system;
- c. completion of Software DT&E.

5.4.2.3 BUILD DESCRIPTION DOCUMENT

The Build Description Document identifies the functional capabilities of each incremental build of software that is delivered for implementation. It also identifies all physical products to be delivered and provides instructions for implementing the build in an operational configuration. A Build Description Document should accompany the delivery and implementation of each incremental build.

5.4.3 FORMAT FOR SOFTWARE TEST PROCEDURES

SOFTWARE TEST PROCEDURES

TABLE OF CONTENTS

Section 1.	Test Identification
Section 2.	Equipment and Software
Section 3.	Input Data
Section 4.	Procedures
4.1	Initiation
4.2	Operation
4.3	Termination/Restart
Section 5.	Test Results
Section 6.	Evaluation

Section 1. Test Identification. This section shall identify the specific test from the Software System DT&E Plan (Section 3) or the Software System Integration and Test Plan (Section 6) to which these procedures apply.

Section 2. Equipment and Software. This section shall identify all equipment and test support software required for the test.

Section 3. Input Data. This section shall describe the input data required to conduct the test.

Section 4. Procedures. This section shall contain all of the test steps required to conduct the test. Steps shall be written and listed sequentially in the order in which they will be implemented. They shall identify all actions required by the person conducting the test. They shall also describe the expected result from each action that is taken. Figure 5-2 is an example of a Test Procedure Table that may be used to document test procedures. Procedures shall be specified for the following:

4.1 Initiation. This section shall list the step required to begin the test, including the following:

- a. Equipment Activation.
- b. Input Data Activation.
- c. Program Initiation.
- d. Setting of Parameters.

4.2 Operation. This section shall identify each step in the test procedure. For each step, the expected result shall also be identified.

4.3 Termination/Restart. This section shall identify the steps required to terminate and (as applicable) restart the test. This shall also include procedures for assuring the necessary output data is available for evaluation.

Section 5. Test Results. This section shall list the results of the test. It shall identify each step where a discrepancy occurred. It may include opinions as to the cause of the discrepancy and recommendations for correction. It shall also identify each discrepancy report prepared as a result of the test.

SYSTEM NAME:

TEST IDENTIFICATION:

PERSON RESPONSIBLE FOR TEST:

DATE TEST CONDUCTED:

Number	Test Procedure	Expected Results	Results Obtained? (yes/no)

NOTES: (*Explanation of discrepancies; Recommendations for corrective action.*)

Figure 5-2 Test Procedure Table

Section 6. Evaluation. This section shall contain an evaluation of the test. It shall identify successes, deficiencies, limitations or other constraints detected during the test. It shall also include recommendations for subsequent actions as a result of the test.

5.4.4 FORMAT FOR SOFTWARE TEST REPORT

SOFTWARE TEST REPORT

TABLE OF CONTENTS

Section 1.	General
1.1	Purpose of the Software Test Report
1.2	Project References
Section 2.	Test Results
Section 3.	Test Evaluation
Section 4.	Recommendations

Section 1. General.

1.1 Purpose of the Software Test Report. This paragraph shall describe the purpose of this Software Test Report. It shall summarize the results of tests that have already been conducted and identify testing that remains to be conducted.

1.2 Project References. This paragraph shall identify documents applicable to software testing of the system. It shall identify the following by author or source, reference number, title, date, and security classification:

- a. Software Requirements Specification.
- b. Software Test Plans.
- c. Software Test Procedures.
- d. Software Development Plan.
- e. Previously developed technical documents related to software testing.

Section 2. Test Results. This section shall identify each test that has been conducted during the applicable testing period. For each test, it shall summarize the objective of the test and the result of the test. It shall also identify all Software Problem Reports that have been prepared as a result of the test.

Section 3. Test Evaluation. This section shall present an evaluation of the software tests that have been conducted. It shall identify functional deficiencies, limitations, or constraints that have been detected during the testing process. For each deficiency, this section shall describe the impact on the software and on the development effort if the deficiency is not corrected. It shall also identify the effort required to correct the deficiency.

This section shall also evaluate whether the software that has been tested is ready for subsequent stages of testing or acceptance (if applicable).

Section 4. Recommendations. This section shall contain recommendations for correcting deficiencies (where and how).

5.4.5 FORMAT FOR BUILD DESCRIPTION DOCUMENT

BUILD DESCRIPTION DOCUMENT

TABLE OF CONTENTS

Section 1.	General
1.1	Purpose of the Build Description Document
1.2	Build-Related Documentation
Section 2.	Inventory of Materials Released
Section 3.	Functional Capabilities and Software Elements
Section 4.	Implementation Instructions
Section 5.	Possible Problems and Errors

Section 1. General.

1.1 Purpose of the Build Description Document. This paragraph shall describe the purpose of the Build Description Document in the following words, modified as appropriate:

This document describes the capabilities and products of Build (Build name, number) of the (name) Software System/Subsystem for the (name) System.

1.2 Build-Related Documentation. This paragraph shall provide a summary of the references related to the products and capabilities of the Build software. It shall identify the following by author or source, reference number, relevant section, title, date, and security classification:

- a. Software Requirements Specification.
- b. Software Development Plan.
- c. Software System/Subsystem Specification.
- d. Software Program Specification.
- e. Software System Integration and Test Plan.
- f. Software System DT&E Plan.
- g. Previously-developed technical documents relating to Incremental Development of the software.

Section 2. Inventory of Materials Released. This section shall identify all physical products which are part of the Build delivery. It shall also identify all utility and support software required to operate, install, or regenerate the incremental capabilities of the Build.

Section 3. Functional Capabilities and Software Elements. This section shall identify the functional capabilities and the software elements to be implemented in the Build Software. This identification may be done directly or by reference to related specifications.

Section 4. Implementation Instructions. This section shall describe (either directly or by reference) the procedures for installing and implementing the software in this incremental Build.

Section 5. Possible Problems and Errors. This section shall identify potential problems and known errors which need to be corrected in the Build software. It shall also state the actions that are being taken to resolve the problems and correct the errors.

5.5 SOFTWARE MANUALS

5.5.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 5.4)

Software developers shall produce software manuals which contain instructions necessary to operate, use, and maintain the deliverable software system.

During the system acquisition planning phase, Software Acquisition Managers shall identify the set of software manuals that software developers must produce to meet the requirements of this policy. These manuals may include a User's Manual, an Operations Manual, a Software Maintenance Manual, and a Firmware Support Manual.

The Operations Manual shall contain, as a minimum, a description of how to set up, execute, select options and interpret output for software operation.

The User's Manual shall include, as a minimum, the operational procedures needed to provide users with instructions necessary to execute the software. It shall also relate the operational procedures to the operational system functions.

A Software Maintenance Manual shall contain, as a minimum, sufficient information to enable an experienced programmer to modify the code and data in the computer programs and the procedures of the software system. It shall also contain instructions for installing programs and procedures in the operational environment.

By Preliminary Design Review, early versions of each software manual shall be available for review. The early versions shall describe the man/machine interface with the software, computer hardware and other system equipment.

5.5.2 GUIDANCE

Software Acquisition Managers may select several types of manuals to satisfy the requirements of this policy. A variety of acceptable formats are included here. They are as follows:

a. User's Manual

1. User's Manual
2. Software System Users Manual
3. Positional Handbooks

b. Computer Operation Manual

1. Computer Operation Manual

c. Software Maintenance Manuals

1. Program Maintenance Manual
2. Firmware Support Manual

Manual content and format should be specifically designed to meet the needs of the intended user. Software Acquisition Managers may require that manuals be developed from formats in this section or they may use the formats as guidelines to develop new formats that are more appropriate for their needs. In either case, manuals should be organized to explain the system in terms of application and operation and should be as self-contained as possible. Reference to other documents should be minimal. The text must be factual, concise, specific, clearly worded, and illustrated. Sentence forms should be simple and direct. Abbreviated tabular data such as charts, tables, checklists, and diagrams should be used whenever practicable.

Technical detail reflected in the manuals should be expressed in wording that is easily understood. Unless essential for practical understanding and application, discussions of theory should be omitted. Manuals should not use phraseology requiring a specialized knowledge. They should emphasize specific steps to be followed, the results which may be expected or desired, and the corrective measures required when such results are not obtained.

5.5.3 DESIGN REQUIREMENTS FOR SOFTWARE MANUALS

The primary requirement of a manual is that the intended audience be able to use it. A well-designed manual will help meet this objective. Following are specific requirements for designing software manuals:

- a. **FORMAT.** Each document shall have the following parts: a title page, a release page, a change log, a table of contents, the main body of text, a glossary, and illustrations. Optional parts are a list of figures or illustrations, a list of tables, appendices, and an index. Each page shall contain the page number and page content heading.
 1. The title page contains the title of the manual and the date of the manual.
 2. The release page contains a description of the version of the software system with which the manual is compatible.
 3. The change log indicates change pages and shows the publication date of each page.
 4. The table of contents contains a list of topic headings taken from the main body of the text. The page number of the topic headings shall be listed.
 5. The optional list of figures or illustrations contains a list of the titles of figures or illustrations. The page numbers of the figures or illustrations shall be listed.

6. The optional list of tables contains a list of table titles. The page numbers of the tables shall be listed.

7. The main body of the text is divided into chapters. Each main topic shall constitute a chapter.

8. The glossary shall contain all specialized terms used within the manual.

9. The optional appendices may contain any auxiliary material deemed necessary in the use of the manual or the software system. Examples are tables and worksheets.

10. The optional index contains reference page numbers of each topic listed.

b. PAGINATION. Pages may be numbered consecutively throughout the document, or through a chapter only.

c. TOPIC HEADINGS. Topic headings shall clearly indicate the order of subordination. Parts, chapters sections, paragraphs, figures, and tables shall have brief descriptive titles. Major headings may be centered. Subordinate topic headings shall be left-justified on the page. Indented headings may be used if further subordination is required.

d. ILLUSTRATIONS AND DIAGRAMS. Illustrations and diagrams shall be used whenever the result will be a more effective presentation of information.

e. NOMENCLATURE. Nomenclature shall be consistent throughout a particular set of manuals. Standard acronyms and abbreviations may be used if they are first defined in the text. They shall also be defined in the glossary.

f. SPACE CONSERVATION. Layout shall not constrain usability or clarity of material. If blank space improves the effectiveness of communication, blank portions of pages are permitted.

g. USERS AIDS. Summaries and printed tables shall be provided where appropriate to aid the user of the manual.

h. COLLATING, DRILLING, AND BINDING. Collating, drilling, and type of covers shall be as directed by the customer and/or user.

5.5.4 FORMAT FOR USER'S MANUAL

USER'S MANUAL

TABLE OF CONTENTS

Section 1.	General
1.1	Purpose of the User's Manual
1.2	Terms and Abbreviations
1.3	Security and Privacy
Section 2.	System Summary
2.1	System Functions
2.2	System Operation
2.3	System Configuration
2.4	Performance
2.5	Data Base
Section 3.	Operating Instructions
3.1	System Initialization
3.2	Execution Options
3.3	User Inputs
3.4	System Inputs
3.5	Execution
3.6	Restart and Recovery
3.7	Outputs
3.8	Termination
3.9	Error Messages
Section 4.	Data Update Procedures
4.1	Frequency
4.2	Restrictions
4.3	Sources
4.4	Access Procedures
4.5	Update Procedures
4.6	Recovery and Error Correction Procedures
4.7	Termination Procedures
Section 5.	Notes

Section 1. General.

1.1 Purpose of the User's Manual. This paragraph shall describe the purpose of the User's Manual. It shall also identify for whom it is intended.

1.2 Terms and Abbreviations. This section shall provide an alphabetic listing or include in an appendix a glossary of terms, definitions, and acronyms used in this document and subject to interpretation by users of the system. This list will not include item names or data codes.

1.3 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.

Section 2. System Summary.

2.1 System Functions. This section shall describe the functions of the system and show how they support the user. The description shall include:

- a. The purpose, reason, or rationale for the system.
- b. Capabilities and operating improvements provided by the system.
- c. Additional features, characteristics, and advantages considered appropriate in furnishing a clear, general description of the system and the benefits derived from it.
- d. Functions performed by the system, such as preprocessing or postprocessing data input or output from a primary processor; maintenance of data files; etc.

2.2 System Operation. This paragraph shall explain the relationships of the functions by the system with its sources of input and destinations of output.

2.3 System Configuration. This section shall describe the system to which this user's manual applies. It shall identify the hardware and software required to support the operational environment of the system. It shall also provide a brief description of the purpose of each part of the system.

2.4 Performance. This section shall describe the overall performance capabilities of the system. Some examples of performance measures and information of interest are as follows:

- a. Input - types, volume, rate of inputs accepted.
- b. Output - types, volume, accuracy, rate of outputs that the system can produce.
- c. Response time - include qualifications, where necessary, that affect response time in processing operational reports, such as listing a tape, compiling an object program, etc. Type and volume of input and equipment configuration are examples of items that may influence running time and, consequently, response time.
- d. Limitations - for example, maximum size per unit of input, format constraints, restrictions on what data files may be queried and by what location, language constraints.
- e. Error rate - capabilities for detecting various legal and logical errors and the means provided for error correction.
- f. Processing time - show typical processing times.
- g. Flexibility - note provisions allowing extension of the usage of the system.
- h. Reliability - note system provisions that support, for example, alternate processing or a switch-over capability.

2.5 Data Base. This section shall identify the data files that are referenced, supported, or kept current by the system. The description should be brief and should include the type of data in the file and how the data is used. If the system does not have a file query capability, this section shall also include a description of data elements in the data base. This description shall include information such as the following:

- a. Data element name.
- b. Synonymous name.
- c. Definition.
- d. Format.
- e. Range of values
- f. Unit of measurement.

g. Data item names, abbreviations, and codes.

If this information is contained in a data dictionary, this section may refer to an entry in the dictionary. Any variations from the format or data items in either inputs or outputs shall be identified.

Section 3. Operating Instructions. The following sections shall provide instructions for operating each function identified in Section 2.1. As appropriate, operating instructions may be given for each function sequentially or they may be given for interrelated functions in the order that procedures are implemented by the user.

3.1 System Initialization. This section shall describe the steps necessary for initializing the system prior to its operation. If a separate Operator's Manual has been prepared, this section may refer to appropriate sections that provide system initialization procedures.

3.2 Execution Options. This section shall describe the execution options available to the user when executing the system and its individual functions.

3.3 User Inputs. This section shall describe user inputs to the system.

3.4 System Inputs. This section shall identify inputs to the system (other than user inputs). As applicable, descriptions of inputs may include the following:

- a. Purpose.
- b. Content.
- c. Associated inputs.
- d. Origin of inputs.
- e. Data files associated with the inputs.
- f. Security considerations.

3.5 Execution. This section shall describe the step-by-step procedures for executing functions in the system. Included shall be the sequence of steps required to access the data base. Also included shall be the steps necessary to produce the various displays and retrievals available to the users. Graphical representations of the displays shall be included along with a description of their relationships to the procedures and functions.

3.6 Restart and Recovery. This paragraph shall describe the procedures for restarting execution of the functions when they have been interrupted.

3.7 Outputs. This section shall identify outputs from the system. As applicable, descriptions of outputs may include the following:

- (1) Output - list the outputs produced by the system showing their relationship to the inputs.
- (2) Purpose of output - explain the reason for the output and note conditions or events that require its generation by the system.
- (3) Content of output - describe in general terms the information provided by the output.
- (4) Associated outputs - identify other system outputs that complement the information in this output.
- (5) Distribution of outputs - note the recipients of the output.
- (6) Security considerations.
- (7) Other - describe additional items of general information.

3.8 Termination. This section shall describe procedures for terminating execution of the functions. It shall give procedures for both normal and abnormal termination.

3.9 Error Messages. This section shall list all error messages produced by the software system that can be displayed to the user. It shall also describe the error associated with each message and identify the proper user response to the message.

Section 4. Data Update Procedures (if applicable).

4.1 Frequency. This paragraph will describe the frequency of data updates. Information such as the events that cause the update may be included.

4.2 Restrictions. This paragraph shall describe any restrictions on updating the data base. Included may be such factors as:

- a. Users authorized to update.
- b. Time periods when such updating is allowed.
- c. Information for ensuring that only authorized updates are allowed.

4.3 Sources. Included in this paragraph will be a list of the sources used to obtain the data that will make up each update.

4.4 Access Procedures. This section shall describe the sequence of steps required to access the data base. Included will be such information as the name of the system or subsystem being called and other control information such as access restrictions.

4.5 Update Procedures. Paragraph 4.5.1 through 4.5.n shall provide information to enable an authorized user to update data in the system data base. For each type of update procedure, information such as the name of the operation, input formats, and sample responses may be included.

4.6 Recovery and Error Correction Procedures. This section shall identify error codes and messages. It shall also indicate their meanings and describe corrective actions that should be taken. Any user-initiated recovery procedures and validity checks shall also be included.

4.7 Termination Procedures. This paragraph shall present the step-by-step sequence of actions necessary to terminate the update.

Section 5. Notes. This section shall contain any general information that will help the user understand the operation of the system and how the User's Manual relates to its operation.

5.5.5 FORMAT FOR SOFTWARE SYSTEM USER'S MANUAL

TABLE OF CONTENTS

Section 1. INTRODUCTION

Section 2. GLOSSARY

Section 3. SOFTWARE SYSTEM CAPABILITIES

Section 4. FUNCTIONAL DESCRIPTION

Section 5. USAGE INSTRUCTIONS

Section 6. OPERATING INSTRUCTIONS

Section 7. APPENDIX

Content. The user's manual for a software system or subsystem shall contain the following:

Section 1. Introduction. A description of the manual's purpose, scope, organization and content.

Section 2. Glossary. Terms used in the manual.

Section 3. Software System Capabilities:

a. **Purpose.** A description of the purpose of the software system.

b. **General Description.** A description of the system, giving an overview of the system and its operation.

c. **Function Performed.** Identification of the specific functions performed by the software system. (This may be in terms of systems operations, uses, outputs, etc.)

Section 4. Functional Description. A description of each specific function with the software system. The following subparagraphs shall be repeated for each function:

a. **Title of Function.** A descriptive title of the specific function.

b. **Description of Function.** A summary description of the specific function, including:

- (1) Purpose and uses of function.
- (2) Description of system inputs.
- (3) Description of expected output and results.
- (4) Relationship to other functions.
- (5) Summary of function operation.

Section 5. Usage Instructions. How to use each specific function. This shall include the following:

a. Preparation of Inputs. A definition of the system inputs, other than those required to operate the computer program (see paragraph f2). These inputs constitute the basic data that are to be processed by the software system. The definition shall include:

- (1) Title of inputs.
- (2) Description of inputs.
- (3) Purpose and use.
- (4) Input media.
- (5) Limitation/restrictions.
- (6) Format and content.
- (7) Sequencing (e.g., formalized deck structure).
- (8) Special instructions.
- (9) Relationship of inputs to outputs.

b. Results of Operation: A definition of expected results after completion of software system operation.

- (1) Description of results.
- (2) Form in which results will appear.
- (3) Output format and content.
- (4) Instruction on use of outputs.
- (5) Limitations/restrictions.
- (6) Relationship of outputs to inputs.
- (7) Examples.

c. Error messages. A description of error messages associated with the function and corrective action to be taken.

Section 6. Operating Instructions. Procedures required to operate the software system. It shall include the following:

a. Operating Procedures. The step-by-step procedures required to:

(1) Initiate the Software System. Procedures shall include reading computer programs into the computer, establishing the required mode of operation (if more than one), initially setting required parameters, providing for inputs and outputs, and operating the computer programs.

(2) Maintain Software System Operation. Procedures shall be specified to maintain operation of the computer program where operator intervention is required.

(3) Terminate and Restart the Software System. Procedures shall be specified for normal and unscheduled termination of computer program operations, as well as restarting the computer programs.

(4) Software System Generation Procedure. This section shall describe procedures that are necessary to create a new version of the master code media (for example, master tape).

(5) Symbolic Updating Procedures. This section shall describe the procedures to be followed to effect symbolic changes to the software system. This discussion should present a step-by-step description of the updating processes. As necessary, appropriate data sets and inputs will be described.

b. Operator Inputs. A complete description of all the control formats of each computer program, including the function or purpose of each field, will be given. All inputs, other than those described in paragraph e(1), required to operate the computer program shall be defined in a similar manner as follows:

(1) Title of input.

(2) Purpose and use.

(3) Input media.

(4) Limitations/restrictions.

(5) Format and content.

c. Operator Outputs. A complete description of the output format other than those described in paragraph e(2), of each computer program. This shall include samples of each type of possible

output format. Furthermore, a cross-reference list between output fields and input control fields shall be given, so that the users of the software system may readily determine the effect of certain input fields on each output field. This subparagraph shall include, but not be limited to, the following:

- (a) Title of output.
- (b) Purpose and use.
- (c) Output media.
- (d) Output format.
- (e) Output content (symbols, codes, etc.).

g. Appendix. The appendix may include information bound separately for convenience, as in the case of classified appendix or a large body of data.

5.5.6 FORMAT FOR POSITIONAL HANDBOOKS

POSITIONAL HANDBOOK

TABLE OF CONTENTS

Section 1. INTRODUCTION

Section 2. OPERATOR POSITIONS

Section 3. NORMAL OPERATING PROCEDURES

Section 4. EMERGENCY OPERATING PROCEDURES

For each operator position, Positional Handbooks describe the procedures for the operating computer-associated consoles and related equipment. Operating procedures are based upon the functions to be performed by the software system. They describe the responsibilities, duties, and operating procedures of operational positions. Emphasis is on the actions required by operators of the position; discussion of sub-system functions and theories of operations is included only as necessary to clarify instructions in operational procedures.

The handbooks provide operators with a comprehensive understanding of position responsibilities and duties. They provide the operator with a ready reference of position duties and the necessary step-by-step procedures for the operation of position equipment. A handbook is normally prepared for each operating position. Individual handbooks may be combined into a single volume for the total software system.

Positional Handbooks will be prepared using the following guidelines:

Section 1. INTRODUCTION. The introduction to each handbook shall specify the operator position covered and briefly explain the extent of the instructions provided. When necessary, the introduction may also contain special information concerning any noteworthy or unusual features of the contents.

Section 2. OPERATOR POSITION. A comprehensive description of the specific operator position shall be provided. This will include a complete job description of the specified position, describing in detail positional responsibilities and duties, defining the knowledge and capabilities the operator must possess in order to perform his position duties, and listing operator positions in his charge and operator positions to whom he is directly responsible.

Section 3. NORMAL OPERATING PROCEDURES:

a. This section shall contain introductory material providing general information concerning operator action procedures. It shall sequentially number paragraph headings and designate the precise title of each action which can be taken at the operator position. Graphical representations of displays that are part of the operating procedures shall accompany the description. The text for each action shall include an explanation of why the action is taken and shall detail step-by-step procedures, restrictions, and results. The explanation of the purpose of the action should summarize those factors that need to be considered before the action is taken. Additionally, factors and procedures that need to be considered after the action has been decided upon but before the required action procedures are initiated should be discussed.

b. Step-by-step procedures shall be listed specifying the sequence of operations to accomplish the action.

c. A listing of restrictions shall specify the computer program restrictions unique to the action which, if applied to the procedure being taken, would be identified as an illegal action.

Section 4. EMERGENCY OPERATING PROCEDURES. This section shall discuss emergency conditions, such as computer failure or equipment malfunction, that require startover operations to initiate or reestablish cycling of the operational computer program. This section shall contain such information as the effects of computer downtime on data storage, descriptions of the various modes of startover that may be used to reinitiate operation of the system, and listings of the duties and responsibilities of the operator position during startover.

5.5.7 FORMAT FOR COMPUTER OPERATION MANUAL

COMPUTER OPERATION MANUAL

TABLE OF CONTENTS

Section 1.	General
1.1	Purpose of the Computer Operation Manual
1.2	Project References
1.3	Terms and Abbreviations
1.4	Security and Privacy
Section 2.	System Overview
2.1	System Application
2.2	Program Inventory
2.3	File Inventory
2.4	Processing Overview
Section 3.	System Operation
3.1	System Preparation and Setup
3.1.1	Power On/Off
3.1.2	Initiation
3.2	Operating Procedures
3.3	Input/Output
3.4	Monitoring Procedures
3.5	Auxiliary/Off-Line Routines
3.6	Recovery Procedures
3.7	Special Procedures
Section 4.	Diagnostic Features
4.1	Error Detection/Diagnostic Features
4.2	Computer System Diagnostic Features
Section 5.	Notes

Section 1. General.

1.1 Purpose of the Computer Operation Manual. This paragraph shall describe the purpose of the Computer Operation Manual. It shall also identify for whom it is intended.

1.2 Project References. At least the following documents, when applicable, shall be specified by author or source, reference number, title, date, and security classification:

- a. Users Manual.
- b. Program Maintenance Manual.
- c. Other pertinent documentation on the project.

1.3 Terms and Abbreviations. This section shall provide an alphabetic listing or include in an appendix a glossary of terms, definitions, and acronyms used in this document and subject to interpretation by operators of the system.

1.4 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.

Section 2. System Overview.

2.1 System Application. A brief description of the system shall include its purpose and uses.

2.2 Program Inventory. This paragraph shall provide an inventory in tabular form of the software used by the system. This listing shall include the program full name, program ID, and classification of the program.

2.3 File Inventory. This paragraph shall list all permanent files that are referenced, created, or updated by the system. Included should be information such as the file name, file ID, storage medium, and required storage (number of tapes or disks).

2.4 Processing Overview. This paragraph will provide information which is applicable to the processing of the system. Separate paragraphs may be used as needed to cover system restrictions, waivers of operational standards, information oriented toward specific support areas (e.g., tape library) or other processing requirements such as the following:

- a. Interfaces with other systems.
- b. Other pertinent system-related information.

Section 3. System Operation.

3.1 System Preparation and Setup. This section, in the subparagraphs below, shall describe the procedures for system preparation and setup prior to system operation.

3.1.1 Power On/Off. This paragraph shall explain the step-by-step procedures required to power-on and power-off the equipment for operational and stand-by mode.

3.1.2 Initiation. This section shall describe the following:

1. The equipment setup and the procedures required for pre-operation.
2. The procedures necessary to bootstrap the system and to load programs.
3. The common commands for system initiation (e.g., program interrupt/recovery and system priority organization).

3.2 Operating Procedures. This section shall describe the steps for system restart after system initiation. If more than one mode of operation is available, instructions for the selection of each mode shall be provided. This section shall contain sufficient detail so that all option points are well identified and so that recovery from any step in error may be accomplished without starting over if technically possible.

3.3 Input/Output. This section shall describe input/output media and explain detailed procedures required for each. This section shall briefly describe the operating system control language. This section shall also list operator procedures for interactive messages/replies (e.g., describe password use, log on/log off procedures, and file protection requirements).

3.4 Monitoring Procedures. This section shall describe the requirements for monitoring the computer program in operation. Trouble and malfunction indications shall be delineated, with corresponding corrective actions. Evaluation techniques for fault isolation shall be described to the maximum extent practical. Conditions which require computer shutdown or aborting, and specific abort procedures, shall be described. Procedures for on-line interventions, trap recovery, and operation communications shall also be included.

3.5 Auxiliary/Off-Line Routines. Procedures required to operate all auxiliary/off-line routines of the system shall be explained.

3.6 Recovery Procedures. This section shall explain procedures to follow for each trouble occurrence or program error (e.g., give detailed instructions to obtain system dumps). This section shall describe the steps to be taken by the operator to restart system operation after an abort or interruption of operation. Procedures for recording information concerning a malfunction shall also be included.

3.7 Special Procedures. This section shall include any additional instructions required by the operator (e.g., system alarms, program/system security considerations, preparation of the computer system for a diagnostic run, switch over to a redundant system).

Section 4. Diagnostic Features.

4.1 Error Detection/Diagnostic Features. This section shall briefly describe the error detection/diagnostic features of operational programs. The purpose for each error detection/diagnostic feature shall be described. For each description, its value/meaning shall be stated.

4.2 Computer System Diagnostic Features. This section shall inform the operator of available software for hardware diagnostics by reference to the applicable publication.

Section 5. Notes. This section shall contain any general information that will help the computer operator understand the operation of the system.

5.5.8 FORMAT FOR PROGRAM MAINTENANCE MANUAL**PROGRAM MAINTENANCE MANUAL****TABLE OF CONTENTS**

Section 1.	General
1.1	Purpose of the Program Maintenance Manual
1.2	Project References
1.3	Terms and Abbreviations
1.4	Security and Privacy
Section 2.	System Description
2.1	System Application
2.2	General Description
2.3	Program Descriptions
2.4	Unique Run Features
Section 3.	Environment
3.1	Equipment Environment
3.2	Support Software
3.3	Data Base
3.3.1	General Characteristics
3.3.2	Organization and Detailed Description
Section 4.	Program Maintenance Procedures
4.1	Conventions
4.2	Verification Procedures
4.3	Error Conditions
4.4	Special Maintenance Procedures
4.5	Special Maintenance Programs
4.6	Listings
Section 5.	Notes

Section 1. General.

1.1 Purpose of the Program Maintenance Manual. This paragraph shall describe the purpose of the Program Maintenance Manual. It shall also identify for whom it is intended.

1.2 Project References. This paragraph shall provide a brief summary of references applicable to the history and development of the software system. It shall also identify the acquisition organization, the development organization, and the user organization(s). At least the following documents shall be identified by author or source, reference number, title, date, and security classification:

- a. Software Requirements Specification.
- b. Software System/Subsystem Specification.
- c. Software Program Specification.
- d. Data Dictionary Document.
- e. User's Manual.
- f. Computer Operation Manual.

1.3 Terms and Abbreviations. This section shall provide an alphabetic listing or include in an appendix a glossary of terms, definitions, and acronyms used in this document and subject to interpretation by maintainers of the software system.

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

Section 2. System Description.

2.1 System Application. This section shall explain the purpose of the system and the functions that it performs.

2.2 General Description. This section shall provide a description of the system in terms of its overall functions. This shall include a description of the functions of each computer program and computer program component. It shall also include a graphical representation showing the relationship of each computer program with other major elements of the system.

2.3 Program Descriptions. The purpose of this section is to provide information that would be of value to software maintainers in understanding the software system. This section shall first identify all computer programs that make up the system (Special Maintenance programs shall be discussed in Section 4.4, Special Maintenance Procedures). Following the identification, all computer program components shall be identified. Following a narrative and graphic description of each component, each software unit shall be described as follows:

- a. Identification-Unit title and tag.
- b. Functions - a brief description of the unit functions.
- c. Detailed Documentation - Refer to the applicable portion of the Program Specification (Section 4.4.1 - 4.4.n) or provide equivalent level of information.

2.4 Unique Run Features. This section shall describe any unique features of the operation of the software system 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 support software used by the system and identify the appropriate versions or release numbers 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 including security considerations.

3.3.1 General Characteristics. This section shall provide a general description of the characteristics of the data base, including:

- a. Identification - name and mnemonic reference. List the programs and program components using the data base.
- b. Data Permanency - note whether the data base 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 data base by software elements in the system.

3.3.2 Organization and Detailed Description. This section shall refer to the specific location within the Program Specification or Data Dictionary Document where the internal structure of the data base is located. If internal structure is not documented in those documents, the equivalent level of detail shall be documented in this section.

Section 4. Program Maintenance Procedures. This section shall provide information on the specific procedures necessary for the programmer to maintain the software that makes up the system.

4.1 Conventions. This paragraph will explain all rules, schemes, and conventions that have been used with 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, units, routines, records, data fields, storage areas, etc.
- b. Procedures and standards for charts, listings, serialization, abbreviations used in statements and remarks, and symbols 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 shall include requirements and procedures necessary to check the performance of a software element following its modification. Procedures for periodic verification of the software may also be included.

4.3 Error Conditions. A description of error conditions shall 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 which have not been described elsewhere in this section. Specific information that may be appropriate for presentation includes:

- 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 shall be described in the same manner as those described in paragraphs 2.3 and 2.4.

- a. Input-Output Requirements. Included in this paragraph shall be the requirements concerning materials needed to support the necessary maintenance tasks. 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 shall 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.

Section 5. Notes. This section shall contain any general information that aids in understanding this manual or in understanding how to document, maintain, develop, or modify the software system.

5.5.9 FORMAT FOR FIRMWARE SUPPORT MANUAL

FIRMWARE SUPPORT MANUAL

TABLE OF CONTENTS

Section 1.	General
1.1	Purpose of the Firmware Support Manual
1.2	Project References
1.3	Terms and Abbreviations
1.4	Security and Privacy
Section 2.	Device Information
2.1	Device Description
2.2	Installation and Repair Procedures
2.3	Limitations
Section 3.	Programming Equipment and Procedures
3.1	Programming Hardware
3.2	Programming Software
3.3	Programming Procedures
Section 4.	Vendor Information
Section 5.	Notes

Section 1. General.

1.1 Purpose of the Firmware Support Manual. This paragraph shall describe the purpose of the Firmware Support Manual. It shall also identify for whom it is intended.

1.2 Project References. This paragraph shall provide a brief summary of references applicable to the history and development of the parts of the system that contain firmware. It shall also identify the acquisition organization, the development organization, the user organization, and the organization(s) responsible for providing life-cycle support for the system. At least the following documents shall be identified by author or source, reference number, title, date, and security classification:

- a. Software Requirements Specification.
- b. Software System/Subsystem Specification.
- c. Software Program Specification.
- d. Data Dictionary Document.
- e. User's Manual.
- f. Computer Operation Manual.
- g. Program Maintenance Manual.

1.3 Terms and Abbreviations. This section shall provide an alphabetic listing or include in an appendix a glossary of terms, definitions, and acronyms used in this document and subject to interpretation by maintainers of the firmware parts of the system.

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

Section 2. Device Information.

2.1 Device Description. This section shall contain a complete physical description of the firmware components of the system, e.g., Read-Only-Memory (ROM), Programmable Read-Only-Memory (PROM), Erasable Read-Only-Memory (EROM), switches, or strapping. As a minimum, the following shall be described for each firmware component:

- a. Memory size (e.g., words, bytes, or bits);
- b. Operating characteristics (e.g., access time, power requirements, and logic levels);
- c. Pin functional descriptors;
- d. Logical interfaces;
- e. Manufacturer's part number;
- f. Internal and external identification scheme used on any device.

2.2 Installation and Repair Procedures. This section shall contain procedures for installation, replacement, and repair for each firmware device or board. This shall include procedures for removal and replacement of boards; addressing scheme and implementation; socket number for each device; description of the host board layout; and the available (unused) portion of each firmware device or board.

2.3 Limitations. This section shall describe the operational and environmental limits to which the devices may be subjected and still maintain satisfactory operation.

Section 3. Programming Equipment and Procedures.

3.1 Programming Hardware. This section shall describe the equipment to be used for programming and reprogramming each firmware device. It shall identify computer peripherals, general-purpose equipment, and special equipment used for device loading, burn-in, and test (including verification that the proper content is stored). Each piece of equipment shall be identified by manufacturer, manufacturer's designation, and any special features. A description of the major function of each piece of equipment shall also be included.

3.2 Programming Software. This section shall describe the software to be used for programming and reprogramming each firmware device, including all software for device loading, burn-in, and test. Each software item shall be identified by vendor, vendor's designation, version, and any special features. A description of the major function of each piece of software shall also be included.

3.3 Programming Procedures. This section shall describe the procedures to be used for programming and reprogramming each firmware device including logic data generation and device loading, burn-in, and test. It shall identify all equipment and software necessary for each procedure.

Section 4. Vendor Information. This section shall contain data supplied by the original device vendor or refer to the relevant documentation. The description shall describe the capabilities of the device and the methods of reaching the capabilities.

Section 5. Notes. This section shall contain any general information that aids in understanding this manual or in understanding how to document, maintain, develop or modify the firmware devices that are used in the system.

PART VI - SOFTWARE PRODUCT ACCEPTANCE

6.1 SOFTWARE END-PRODUCT ACCEPTANCE PLAN

6.1.1 POLICY AND REQUIREMENTS SUMMARY (From NSA/CSS Software Acquisition Manual 81-2, Policy 6.1)

Software developers shall follow an orderly procedure, governed by a Software End-Product Acceptance Plan, to prepare for and achieve acceptance of all software deliverables and services called for in the System Acquisition Plan. This plan shall be maintained until acceptance, along with a file of acceptance related data which record receipt and acceptance of all end products and services.

As a minimum, the following end products shall be required in a form and format designated by the System Acquisition Organization:

- (a) Software Program Source Code and Listings;
- (b) Build Description Documents (when the software is developed and delivered in increments);
- (c) Software Requirements Specification;
- (d) Software System/Subsystem Specification;
- (e) Software Program Specification;
- (f) Software Manuals;
- (g) Software Configuration Management Data Base;
- (h) Software Test Plans and Test Cases;
- (i) Software Test Procedures;
- (j) Test Programs and test data files;
- (k) Software Test Reports;
- (l) Hardware diagnostic software for all hardware components in the operational hardware configuration;
- (m) Additional end products required by the Software Life Cycle Support organization.

6.1.2 FORMAT FOR SOFTWARE END-PRODUCT ACCEPTANCE PLAN

SOFTWARE END-PRODUCT ACCEPTANCE PLAN

TABLE OF CONTENTS

- Section 1. Purpose
- Section 2. Applicable Documents
- Section 3. Acceptance Identification
- Section 4. Items of Acceptance
- Section 5. Implementation

Section 1. Purpose. This section describes the purpose of this plan, which is to present an orderly procedure for preparing for and achieving acceptance of all software products required by the acquisition organization.

Section 2. Applicable Documents. This section shall list all references that relate to acceptance of end-products and services by the acquisition organization. This may include the System Acquisition Plan, contractual references, change proposals, Contract Data Requirements List, etc.

Section 3. Acceptance Identification. This section shall define the meaning of "customer acceptance" of items. It shall also define what constitutes "close-out" of an item. Two example statements are as follows:

(1) "Customer approval" of this Software End-Product Acceptance Plan indicates that the customer agrees to the criteria for acceptance of items listed in this plan and that items for acceptance are limited to those described in the plan.

(2) "Close-out" of an item means that no additional work by the developer is expected or required unless changes or maintenance are necessary.

This section shall also identify the instrument of acceptance for each end item.

Section 4. Items of Acceptance. This section shall contain a table or list summarizing the end-product and services which are required by the acquisition organization. The list should be as follows:

Item Number	Item Name	Reference
.	.	.
.	.	.
.	.	.
.	.	.

Following the list, a summary sheet for each listed item shall be included. The summary sheets shall include the following information:

- a. Descriptive title;
- b. Degree of customer concurrence required;

- c. Format of the deliverable item;
- d. Schedule for producing, delivering, and (if applicable) obtaining approval of the item;
- e. References to the item (contract or otherwise);
- f. Criteria for acceptance of the item;
- g. Name of person who will accept the item;
- h. Procedure and schedule for customer review and acceptance.

Section 5. Implementation. This section shall identify the developer's plan for:

- a. Tracking progress toward close-out of each end-product and service;
- b. Conducting the acceptance audit.

