JPL PUBLICATION 79-56

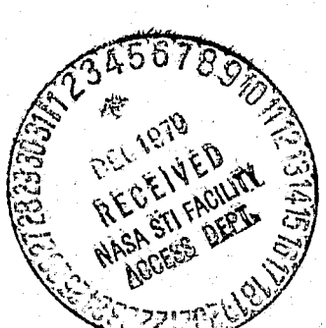
Preparation Guide for Class B Software Specification Documents

Robert C. Tausworthe

October 1, 1979

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California



JPL PUBLICATION 79-56

Preparation Guide for Class B Software Specification Documents

Robert C. Tausworthe

October 1, 1979

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under NASA Contract No. NAS7-100.

Contents

Abstract

This Guide provides general conceptual requirements and specific application rules and procedures for the production of Software Specification Documents in conformance with Deep Space Network software standards and "Class B" standards. Class B documentation is identified as the appropriate level applicable to implementation and sustaining engineering and operational uses by qualified personnel, engineer or equivalent. Specific characteristics of Class B documents are defined in this Guide.

Preparation Guidelines for Class B Software Specification Documents

1.0 Introduction

1.1 Purpose

This Guide provides general conceptual requirements and specific application rules and procedures for the production of Software Specification Documents (SSDs) in conformance with Deep Space Network (DSN) software standards (Refs. 1-3) and "Class B" standards, as introduced in Ref. 4. Class B documentation is herein identified as the appropriate level applicable to implementation and sustaining engineering and operational uses by qualified personnel, engineer or equivalent. Specific characteristics of Class B documents are defined in this Guide.

1.2 Scope of Application

This Guide applies to SSDs produced for DSN subsystem software formally transferred to operations. This Guide bases document contents on the presumption that SSD users in the DSN are qualified programmers with skills equivalent to at least three years experience, one year of which has been spent in DSN subsystem software implementation, sustaining, or as a Deep Space Station (DSS) software specialist.

1.3 Applicable Documents

.1 *Software Implementation Guidelines and Practices*, DSN Standard Practice 810-13, Aug. 1977 (JPL internal document); also available in *Standard Practices for the Implementation of Computer Software*, JPL Publication 78-53, Chapter 1, edited by A. P. Irvine, Jet Propulsion Laboratory, Pasadena, CA, Sept. 1, 1978.

.2 *Preparation of Software Specification Documents*, DSN Standard Practice 810-19, Mar. 1977 (JPL internal document); also available in *Standard Practices for the Implementation of Computer Software*, JPL Publication 78-53, Chapter 4, edited by A. P. Irvine, Jet Propulsion Laboratory, Pasadena, CA, Sept. 1, 1978.

.3 *DSN Engineering Documentation Management Plan*, DSN Standard Practice 810-26, Nov. 1977 (JPL internal document).

1.4 Description

The word "documentation," as used in this Guide, refers specifically to information recorded about a computer program to explain the pertinent aspects of that program to sustaining and operational personnel.

This Guide does not counter the top-down, concurrent documentation and implementation principles put forth in the existing DSN Standards; rather, the guidelines herein address the appearance of an SSD strictly as a delivered, "as-built" specification of the program. Minimum redundancy among SSD entries is sought, and the code listing is viewed as the bottom-level realization of the top-down program documentation hierarchy.

The rules in DSN Software Standard Practices 810-13 and 810-19 remain in effect. The "correctness assessment" level of documentation necessary for Class B sustaining and operations purposes is construed in this Guide to be that which enables the user to understand the contents of the SSD as required for its effective usage. Descriptions of SSD items to the level of detail required for coding without functional ambiguity are not required, because the code listings are part and parcel of the as-built SSD. Alternate forms of design documentation equivalent to flowcharts and narratives are defined, but not authorized, by this Guide. Specification of the specific form of program design documentation remains a Software Requirements Document (SRD) prerogative.

This Guide identifies the intended users and usages of the SSD, and defines the experience level that the preparer(s) of the SSD may be assumed to have. This Guide emphasizes a balanced exposure of function, interface, algorithm, data structure and flow, and code descriptions, with threadability from requirements, through design, into the code. The guidelines are based on an actual survey of user needs.

The Guide further addresses the role of quality assurance (QA) as it relates to SSD preparation. The QA criteria and procedures contained herein, however, are illustrative only, inserted as a typical, advisory baseline to aid the preparer(s) of an SSD. Standard procedures and practices for QA involvement in SSD preparation are not governed by this Guide.

1.5 Acknowledgments

The contents of this Guide were formed through the cooperative efforts of many individuals who have proposed, evaluated, and provided material. The team established to determine DSN users and their SSD needs included R. B. Hartley, R. R. Miner, M. J. Polsley, and C. L. Gee. Every DSN Development Section supervisor and Cognizant Development Engineer (CDE), as well as all DSS software personnel, several elements of the DSN Program Office, DSN Software Quality Assurance, and DSN Operations, have had the opportunity to review and comment. This Guide is believed to be a least-squares-fit response to all of the inputs received. The participation of all who contributed is gratefully acknowledged.

2.0 SSD Basic Usage Requirements

The principal users of the SSD during the production phase are the CDE and other programmers who use the SSD as a medium to record design information needed for developing the program. This Guide does not specifically address the needs of these users. Rather, its orientation is toward the use of the SSD in the post-production period. However, the SSD is meant to be something that is needed continually and naturally by the developers during implementation so as to avoid a lengthy documentation effort later. The rules in this Guide, therefore, recognize the need for concurrency in the production of code and documentation.

The several uses to which these as-built SSDs are put indicate what the basic requirements of the document are.

Sustaining engineering and DSN Standard Practices require functional threadability from the top down, stepwise from design into the code, and also sufficient description and rationale so that errors, located by a combination of analysis and test, can be repaired (both in code and documentation).

Test preparation, training, and acceptance testing activities require that adequate functional behavior and environmental descriptions of the subsystem be provided (or referenced) in the SSD. (Software Operator's Manuals (SOMs) for use in the Deep Space Stations should contain only operational procedures so as to keep the SOM small, handy, and most useful to DSS personnel.) Threadability from requirements into the operational features of the software is needed to demonstrate acceptability.

Effective hardware and system *diagnostics and trouble shooting* require that the SSD provide the means for an analyst to locate an appropriate section of code given a particular subsystem operational phenomenon (e.g., occurrence of an anomaly). The SSD thus contains function-to-code threadability indicators together with sufficient functional, procedural, and interface information so that the reader can comprehend the software/hardware/subsystem interactions.

Quality assurance, while not a user of the program, is nevertheless a transfer signatory and, therefore, a user of the SSD. QA thus requires that documentation be supplied in a form that is readily auditable. The conformance between design information and code must be evident, and descriptions must adhere to DSN and project standards. Only the executable code itself is a naturally true, as-built representation of the program. Comments within and specifications concerning this code are mere descriptions meant to increase understanding. To be effective, all descriptions must be accurate. It is, therefore, the task of QA to certify that such descriptions reflect the as-built character of the program to that degree sufficient for its intended operational and sustaining usages.

Developers of other programs also may be users of a given SSD, as a source of subroutines, handlers, interface information, etc.

Therefore, the objective of this Guide is to specify the level and kind of detail necessary to

(a) Understand the program from the top down, especially in control and data flow.

(b) Thread from an observed performance phenomenon through the function, interface, and design descriptions into the code.

(c) Locate, diagnose, and correct errors.

(d) Understand the program function sufficiently well to generate and evaluate acceptance tests.

(e) Audit the documentation (including comments) and code for conformity.

3.0 Content Requirements and Guidelines

Users of a program have a right to expect that the documentation they see will measure up to the same professional quality standards as the program itself. Documentation quality is characterized by four attributes: completeness, accuracy, clarity, and economy.

Good documentation is also characterized by order and form, which displays a clear plan or design to whatever the writer wishes to communicate. Clear documentation does not fall into order by mere chance. Order results from careful arrangement of suitable materials to fit a definite purpose.

3.1 Content Guidelines

The SSD content guidelines contained herein were determined through surveys, interviews, and reviews involving implementors, operations, Deep Space Station software personnel, QA, and the DSN program office. The list of topics in this section is a compendium of information deemed necessary by these functionaries in performance of their assigned tasks. The order of these elements in the list does not indicate expressed priority nor is it the SSD table of contents (the SSD table of contents is found in Ref. 2):

(a) Text and code table(s) of contents

(b) Hardware and software subsystem interfaces

(c) Subsystem capacities (e.g., timing and loading factors)

(d) Data base organization and characteristics

(e) End-to-end functional descriptions and modes of operation

(f) Program and data design overview and organization

(g) Module interface definitions and data flow descriptions

(h) Top-down, function-oriented procedural descriptions (flowcharts and text, or equivalent)

(i) Glossary of all acronyms, mnemonics, variables, data structures, etc.

(j) Annotated code listings

(k) Code concordance listing

(l) Memory map/overlay map and descriptive overview

(m) Complete information to repair or rebuild the program (e.g., SYSGEN, link-edit code, disk-catalog code, compiler options, special procedures, etc.)

The approach this Guide pursues toward fulfilling SSD documentation goals is one which encourages a *balance* among descriptions of the items above. The preparer of the SSD may assume that the readers being addressed have a certain level of experience: three years programming experience, with one of these being in familiar contact with DSN subsystem software. Information normally retained by such individuals or readily accessible to them may be abbreviated in the SSD (some material may need only a reference to applicable source documents). None of the topics above, however, should be omitted altogether.

The criteria governing the amount of information required in the SSD are contained in the next section.

3.2 Definition of Class B Detail Requirements

Class B specifications oriented for sustaining and operations shall define every factor of the software item being described to the extent that qualified personnel not part of the implementation team can satisfactorily perform their assigned functions using only information supplied. The level of detail required is that amount needed (1) to locate and correct errors or to implement changes, perhaps after some reasonable reflection, analysis, and testing to discover how a specific part of the program works; (2) to define program functions sufficient for test generation and verification purposes; and (3) to identify subsystem performance phenomena with portions of the software which apply to these phenomena.

Documentation must exist to a sufficient degree that the program is understandable by the intended readers on an individual module basis when read from the top downward through the documentation hierarchy. Control logic must be decidable without delving into lower documentation levels, and the role of each of the steps of a module algorithm must be clear.

Class B documentation criteria recognize the listings as an integral part of the SSD, essential to a complete and final understanding of the program design.

4.0 Class B SSD Guidelines and Standards

4.1 General SSD Preparation Guidelines

The SSD combines text, figures, charts, tables, examples, and listings together in such a way as to describe the program in an effective way. Textual descriptions alone usually cannot adequately convey understanding to readers, nor can flowcharts alone, tables alone, nor listings. Rather, the SSD must reach a balance among the narrative, graphics, tables, and code.

This Guide also stresses a balance among descriptions of the

(a) Functional specification

(b) Data structures

(c) Data flow

(d) Program logic

(e) Source code

The overall volume of the SSD may vary considerably, but completeness with economy and without redundancy should be the criterion for the inclusion of material. However, as a rule of thumb, the figures in Table 4-1 indicate the typical numbers of pages per 1000 executable statements usually deemed adequate. (An executable statement does not include data declarations, parameter definitions, equivalences, nor compiler directives.) Table 4-2 gives a typical volume example based on these figures.

The following general guidelines and later detailed standards promote an effective balance of material in the SSD. Examples of the application of these guidelines appear in Appendix A.

1. Keep the material produced to the minimum necessary to satisfy Class B level of detail, subject to DSN Standard Practices (Refs. 1 and 2) and as specialized by the other rules in this Guide.

2. Use references generously to reduce the size of the SSD.

   Reference, rather than repeat, information appearing elsewhere in the the SSD (including the code), system/subsystem documents, or other maintained, transferred documents.

   Do not refer to material in unmaintained documents, such as the SRD or Software Definition Document (SDD). References to the SOM, Software Test and Transfer (STT) document, text-

Table 4-1. Estimated documentation needed per 1000 lines of pure source code

SSD topic	Number of pages
Functional Descriptions	
Overview	1
External data flow	1
Database	1
Detailed functional specifications	6
	9
Program Design	
Overview	3
Internal data flow	2
Interface	1
Data structure description	5
Logic (flowchart, or equivalent, and narrative)	12
	23
Miscellaneous	
Table of contents	1
Standards and conventions	1
System, environment, external interfaces	1
Test specifications	1
I/O format, memory maps, tables, etc.	2
Glossary	8
	14
Code Listings	
Code table of contents (entry points)	1
Executable code (40 lines/page)	25
Nonexecutable code	1
Comments (320 comments)	8
Concordance listing	5
Job control, SYSGEN, link edit, disk catalog, etc.	1
	41

Table 4-2. Typical Class B SSD: 30,000 lines of code

SSD section	Number of pages
Title page	1
Table of contents	30
Introduction	3
Standards and conventions	30
System, environment, and interfaces	30
Functional specifications	270
Program specifications	690
Test specifications	30
Appendices	300
Listing	1230
Total SSD volume	2614

books, and other such material are encouraged when applicable.

3. Provide function-to-code threadability.

Organize the SSD to display the data flow and program structure in a way the reader can grasp and use as a road map toward finding the detail to be extracted.

Direct function-to-code correspondence is not required. Rather, one should be able to follow the functions, flow of data, and control through the existing documentation as threads or pointers from the topmost functional descriptions, into the design, down through the hierarchic layers, and, finally, into the particular code segments which apply. The module Dewey-decimal scheme of numbering and organizing the documentation should be carried all the way into the code as an aid in identifying the correspondences between design and code.

4. Use descriptive aids, such as diagrams, tables, examples, etc., throughout the SSD to promote understanding; pick the best description method suited to the ideas to be communicated.

It is possible for programs to be described entirely without graphical or other aids. However, the judicious use of figures and tables can drastically shorten the time it takes a reader to comprehend textual (or code) descriptions of program behavior.

5. Do not use graphics, tables, or other aids when textual descriptions alone suffice (are clear), and the aids do not promote a better understanding.

For example, flowcharts (or equivalents) are not necessary to describe a few-step algorithm that is adequately described by text alone.

6. Limit the narrative accompanying a flowchart, table, or other descriptive aid to explanations and information not contained in the aid but needed for understanding.

Supply explanations, such as rationale for steps, instructions on how to interpret a graphic (when nonstandard), and significance of operations, as needed for understanding the SSD when read from the top down.

Narrative should not ordinarily be required for a 2- or 3-box flowchart (or equivalent) when sufficiently annotated.

7. Avoid redundancy between program design information (Section 5 of the SSD) and comments in the code.

Limit annotations in the code to code-level issues and special supporting information needed to understand how program specifications have been implemented into code. If design information is necessary for understanding the coding, give a reference in the code to the proper point in the design. This practice makes the code an extension of the program specification and avoids duplication of documentation.

8. Number all algorithm steps and parts of graphic displays for use in narrative references.

It is not necessary to provide narrative for every number. Narrative descriptions may cover functional groupings, or may be absent altogether if the algorithm or graphic is sufficiently annotated.

9. Provide a glossary defining each program mnemonic constant, parameter, variable, and data structure. Also, define all acronyms used in the SSD.

For each give the mnemonic derivation and defining characteristics, such as units, type, substructure, usage, etc. Correspond names appearing in the design and code portions of the SSD, if different. This glossary may be in the code or in a separate SSD appendix.

10. Write descriptions as "as-built" specifications.

Do not describe functions or actions using the words "shall" or "will" (these imply intent or requirement of something yet to be built). Rather, describe the program function, design, interfaces, etc., as accomplished facts.

4.2 Environment Specification Standards

Much of the system environment specifications (Section 3 of the SSD) will be standard among DSN applications. In such cases, references to standard documents, other SSDs, subsystem/system documents, etc., describing the particular system details are encouraged. The remaining descriptions in the SSD are for top-level descriptions of

(a) Subsystem interfaces

(b) Block diagrams

(c) Special hardware device interface characteristics

(d) Special configurations of the operating system

(e) Special interfaces with other system software databases

(f) Special program job control and link-edit code particulars

If "boilerplate" material (common to many SSDs) is deemed necessary for inclusion, the Software Production Management and Control (SPMC) facility can provide such duplicate sections of the SSD as needed automatically.

Section 5.4 of the SSD addresses external subroutine (library and common software) interfaces. If these details are inserted in Section 3 of the SSD, then omit them (except by reference) in Section 5.4.

1. Provide a one-page block diagram and accompanying overview narrative which shows the program properly embedded in its hardware and software environment and identifies the major system and external data interfaces.

> This overview occupies the area between Sections 3 and 3.1 of the SSD (see Figure 2-1 of Ref. 2). See Appendix A for a typical "siting diagram." Discuss hardware and software constraints or restrictions, use of privileged instructions, interrupts, and other environmental topics.

2. Give a high-level overview description of the operating system, specialized for the program being described, as applicable.

> Describe only the modifications or additions made to a standard reference version of the operating system.

3. Document or reference system/subsystem and all other pertinent interfaces.

4.3 Functional Specification Standards

The software functional specification principally occupies Section 4 of the standard SSD (see Figure 2-1 of Ref. 2). It is purely technical in nature; it pertains wholly to the *external* characteristics of the program. It responds to, extends, refines, and documents the technical concepts laid down in the SRD and SDD. It is not a mere restatement of the SRD or the SDD or the SOM – it *defines* the way the program responds, as built. This definition is maintained throughout the entire life of the software.

Program functional descriptions should describe *what* the external characteristics of a program are and should leave descriptions concerning *how* these functions are implemented to the program specification section (SSD Section 5). Functional descriptions may be structured into hierarchic layers of increasing detail to foster readability (this follows the normal DSN standard breakdown of information). Appendix A illustrates several suggested hierarchy methods and a typical optional function description graphic format.

1. State and display processing specifications primarily as functional "black box" transformations of input data and input conditions into output data and output conditions. Include timing and response considerations as appropriate.

2. Identify and describe the distinct program modes of operation in the program functional overview (Section 4.1 of the SSD). State the logical conditions or events which invoke and terminate each such mode.

> A mode is defined as a way of operating a program to perform a certain subset of the processing requirements that normally are associated together in the program function.
>
> A graphic illustration is often useful and, perhaps, sufficient here. Typical mode diagrams and descriptions are illustrated in Appendix A. Intricate mode selection and transition logic may, perhaps, be better described using decision tables (also in Section 4.2 of the SSD). See Appendix C for decision table standards.

3. Use references to functional descriptions in the SOM in lieu of duplicating that information in the SSD whenever the SOM contains extensive definitive functional information.

> Normally, functional material properly belongs in the SSD, with only operational procedures contained in the SOM. However, if it is deemed necessary to have this information in the SOM, then avoid duplicating it in the SSD.

4. If internal program descriptions are required to clarify a program function, then describe only the pertinent assumed high-level characteristics of the internal design. Refer the reader to the full programming description (in SSD Section 5 or an appendix) for other details.

> For example, describe intermediate data sets in terms of their information content, the substructure imposed by functional relationships between items, correlations among data items, and data

flow among modules and tasks. Leave the other details of data structuring and format (e.g., to accommodate accessibility and storage) to SSD Section 5.

4.4 Programming Specification Standards

The design process culminates in a body of information contained in the Program Specification of the SSD (Section 5, as shown in Figure 4-2 of Ref. 2, annotated as in Figure 4-1 to identify the Dewey-decimal numbering of subsections). In the guidelines which follow, the term "module" refers to a flowchart (or equivalent) and its accompanying narrative. The flowchart displays the control logic and operations constituting the algorithm, and the narrative extends, explains, and expands upon the procedural material. Sections 5.2, 5.3, and 5.4 utilize Dewey-decimal module identifiers to distinguish the SSD entries. For example, module 1.2.3 would be found in Section 5.2(1.2.3). Descriptions of material to be entered in each section are rather explicitly defined in 810-19. The strict use of the outline in Ref. 2 is not mandatory. However, it should be used to the extent feasible to keep commonality among SSDs. Required topics must be addressed.

Other forms of documentation, such as SDDL (Ref. 5), Caine-Farber-Gordon PDL (Ref. 6), CRISP-PDL (Ref. 7), and decision tables (Ref. 8) of program descriptions, are deemed "equivalent" to DSN standard flowcharts when they adhere to the standards contained in Appendices B and C of this Guide. Authorization of alternate methods for use is not the province of this Guide; such decisions are properly addressed in the SRD or SDD.

It is usually just as important to describe the structure, significance, flow, and use of internal data as it is to describe the algorithms processing these. Program data structures, external subroutine interfaces, and input/output (I/O) and resource design specifications thus also appear in SSD Sections 5.4, 5.5, and 5.6 (external subroutine interfaces may appear only as references, if detailed previously in Section 3 of the SSD).

5. PROGRAM SPECIFICATIONS

- 5.1 Overview
- 5.2 (*Chart number*) Main Program Detailed Design
- 5.3 (*Alpha Chart Number*) Subroutine Detailed Designs
- 5.4 External Interfaces (common software, operating system, data, and hardware interfaces, as appropriate)
- 5.5 Data Structure Definitions (internal data)
- 5.6 I/O and Resource Design

Fig. 4-1. The Program Specification Section of the SSD

The following rules guide the content of the programming specification:

1. Begin the Programming Specification with a comprehensive overview (SSD Section 5.1).

> A well-written overview will make it possible to shorten the descriptions of individual modules. Briefly describe the design philosophy, approach, and rationale. Identify key factors which had a major influence on the design if important to understanding the program, such as (1) schemes for resource allocation and protection; (2) methods for process synchronization and communication; (3) task breakdown, timing, priorities, and task interfaces; and (4) overlay structures and resource management, etc. Discuss the major algorithms and data structures, and describe the breakdown of the program functions into individual program components. Levels of access and data flow between program processing modules should be addressed for top-level function-to-design threadability.
>
> Program structural information that describes the breakdown of functions into subfunctions and shows interfaces and data flow (especially with the immediate environment) is particularly useful. Such information need not be extremely detailed – for example, data flow information concerning type of data, source, and destination is sufficient, generally speaking, in this overview.

2. For each module, provide the following information (in Sections 5.2 and 5.3 of the SSD):

> (a) The module "chart" number, name, and effective date. The effective date is advisory only, a means for the programmer to keep track of module revisions.
>
> Identification information should be placed at the upper right of each page for ease in module location. The formats in Ref. 2 apply to flowcharts and narrative. See appendices for standards on equivalent forms.
>
> (b) A short description of the module function, as needed for understanding the algorithm which follows.

This description should, only as appropriate, discuss data structures, data flow, interfaces, inputs, actions, and outputs, and should identify constraints which affect the design or operation, such as (1) conditions under which module is executed, (2) critical maximum time of execution, (3) data ordering, (4) machine timing characteristics, (5) task communications and interactions, (6) special hardware interactions, (7) accuracy requirements as related to computer word size and the need for single- or multiple-precision computation, or (8) unique or unusual coding required.

(c) A description of the algorithm steps in terms of a control flowchart (or equivalent) and narrative.

(d) Name (or initials) of "Designer," "Checker," and "Proj. Engr."

In cases where the programmer performs all of these functions, a single name or set of initials is sufficient. (It should appear on the "Proj. Engr." line on flowcharts; see appendices for equivalent forms.)

3. Strive to limit module designs to 10 or fewer algorithmic steps. Describe each step in a clear, functionally definite way.

4. Number each algorithmic step (flowchart box) and use this number for making references in the narrative to the step and for cross-referencing this step into the code.

In cases where later alterations add a step, minimize the rework by labeling this step by an intermediate identifier, such as "5A" (between steps 5 and 6). Steps may also be numbered in multiples of 10 to further aid in later entry (or deletion) of numbers. Standards for narrative step references are shown in Ref. 2. Appendix A illustrates code-step reference annotation.

5. Do not refine an algorithmic step (e.g., do not "stripe" its flowchart box) if that step is clearly perceived as a single entity, its inputs and outputs are well understood at this point in the SSD, and the corresponding code is a straightforward realization of that entity (or follows a well-known or referencible algorithm).

As a general guide, strive to make each "unstriped box," or equivalent, in the SSD average about 10-15 lines of code. This number is expected to vary widely, fewer lines in logical instances and, perhaps, more in computational cases. Appendix A gives an example of Class B detail for assembly language programs, corresponding to the Class A example shown in Figures B-4 and B-5 of Ref. 2.

6. Include statements which provide rationale, assumptions, or other clarifying explanations of the algorithm as needed to lend meaning and understanding.

Explaining the intended significance of an action (such as, for example, setting or testing a flag) can save a reader much time in understanding how and why an algorithm works. It is particularly important to provide information for most loops, stating what assumptions are valid during each iteration (i.e., stating the loop invariant). Every abnormal exit of a module should be fully explained in the design documentation.

7. Keep design documentation of data entities at the data structure level; leave storage structure descriptions to the code.

For example, for operations that operate on record data structures, identify fields by name rather than by substructure position. Describe the substructure positional parameters in the code listings. In this way, the SSD tends to be insensitive to substructuring of the data. Enter all field names into the glossary.

8. If overlays and memory swapping are used, provide usage maps and explanatory material that together correspond memory areas with program execution information, as necessary for understanding which segmentation or configuration is in effect with the several program parts.

9. Provide other auxiliary documentation as may be required to bridge the design to the code, such as (1) timing diagrams, (2) features of the code which link performance to design, or (3) examples of inputs and outputs.

4.5 Code Documentation Standards

Just as the programming specification (SSD Section 5) contains information to bridge functions to data structures and algorithms, so the code descriptions may need information to bridge data structures and algorithms into programmed instructions. Such descriptions are supplied chiefly in the form of comments, although some may also appear in the form of timing diagrams, coding standards, etc., elsewhere in the SSD (as per rule 4.4.9 above).

The orientation of the code commentary is toward describing the code design, rather than the program design. Such an

orientation tends to insulate the design portion of the SSD from code-design anomalies and remove redundancy within the SSD. The line between program design and code design is not a clear one, however, in all cases. There is little adverse effect if the program design leaks into the commentary, or if code design leaks into the program specification, so long as top-down readability is maintained. Redundancy is prone to error or extra expense when changes are made and, therefore, should be limited as much as is consistent with understandability and costs.

The following guidelines promote the use of code listings as an integral part of the SSD, not as a separate stand-alone item. The criteria for correspondence between the programming specifications and the code listings are the following:

(a) The executable code must be a faithful representation of the design as documented in the program specifications. Specifically, this means that the two must be logically equivalent, with steps in one readily identifiable with steps in the other. Code must remain one-entry, one-exit at the module level, and equivalent steps in separate flow paths on a flowchart (or equivalent) must translate into separate blocks of code. The programmer is otherwise free to translate the specifications into the most efficient logically equivalent form.

(b) The storage structures in the code must be a faithful representation of the design as documented in the program specifications. Specifically, this means that the data declarations and usages must conform under a suitable and documented translation. Conformance in structure name, numbers of fields, field names, field types, and ranges of values is required. Commentary should be present which defines pertinent aspects of this correspondence.

Specific rules to effect these criteria are the following:

1. Enter into a special, easily located part of the program documentation (such as Section 2 of the SSD) all general, program-wide coding conventions and standards which relate how flowchart (or equivalent) specifications are coded.

> Make the set of conventions complete enough so that, except for special cases, they are sufficient for a reader to understand how the code corresponds to the program specifications when not commented.

2. Document interfaces between major program segments (overlays and tasks) and between the program and the operating system (as applicable to the code design). Define priorities, task save areas, working areas, global common, executive services, I/O interfaces, error recovery, etc.

3. Provide a cross-index table or table of contents of the code module entry points.

> This index should ideally list each module by name in Dewey-decimal or alphabetic order, name the file the source is in, and give a locator into the listing. However, a table of contents for each of the separate compilations is acceptable.

4. Use the same symbolic names for code entry points, variable names, etc., as appear in the program procedural design, insofar as permitted in the implementation language.

> Clearly annotate listings when alternate names or labels have had to be used, and insert such correspondences in the glossary.

5. Provide a comment banner at the entry of each flowcharted segment of code, subroutine, etc.

> This block should contain (1) module name, (2) module type (subprogram, subroutine, macro, main program, task, etc.), (3) Dewey-decimal identifier and revision number (if applicable), (4) programmer name, and (5) effective date. The effective date is advisory only, a mechanism for distinguishing module versions.

6. State pertinent code-level assumptions upon entry or exit from a module before the beginning of its compiled (or assembled) statements.

7. Annotate the algorithmic steps within coded modules using the same identifying numbers as appear in the flowchart (or equivalent).

> If a cross-reference number also appears on a flowchart box, then supply this number as well as the box-reference code. Appendix A illustrates such annotation of an assembly language program coded from a flowchart.

8. Insofar as feasible, put all configuration-dependent parameters or compile-parameter options together in one clearly identified place in the code.

> Give a prescription for changing them should the parameters require alteration. Include device addresses, dedicated interrupt locations, etc.

9. Name constants and compiler parameters to indicate purpose, rather than value. Define the mnemonic derivation in comments.

> Use separate constants (parameter names) for different purposes, even if some of these have the same values. Avoid the use of "magic numbers," or numbers whose meanings are not implied by their value.

10. Avoid using literal constants in the code except when these are true recognizable constants of the problem.

> Literals should not be used to address data structures by making use of assumed structural formats, especially if this practice will limit the extendability, modifiability, or flexibility of the code being written.

11. If the data structures as defined in the design have required further detailing at the code level, then insert such information as annotations located with the storage structure declaration, properly referenced. Record further detailings of other resource access requirements in a similar manner.

12. Provide all information necessary to rebuild or repair the program (such as disk catalog code, job control code, compiler options, build procedures, etc.) properly described in text or annotated on listings. Locate all SYSGEN, link-edit, and program source and object files, if not governed by standards.

> Include listings of special SYSGEN code or other code modifying the standard operating systems, if applicable. Annotate the modifications so as to be readily discernable and understandable.
>
> Include listings of the link-edit job control code, annotated as necessary for understanding.

13. Provide a complete, definitive glossary of variables appearing in the code.

14. Provide a concordance listing of program variables, labels, and constants.

5.0 Documentation Procedures and Protocols

The secretariat function referred to in Refs. 1 and 2 is performed for DSN Data Systems by the Software Production Management and Control (SPMC) facility. This facility supports the production of software in a number of ways, such as by providing data preparation or date entry services; by generating, updating, and editing textual and graphics materials; and so forth. Some of the specific document production activities are:

(a) Computer processing of text and structured flowcharts.

(b) Modcomp processing of Caine-Farber-Gordon Program Design Language (CFG-PDL).

(c) Automatic production of certain document matter such as the title page, table of contents, etc. ("canned" output of these items is available as a guide).

(d) Manual production of unstructured flowcharts, figures, illustrations, and other graphics.

(e) Reproduction of document material.

(f) Page numbering, application of running head information, etc., for source listings.

(g) Updating and maintaining the current SSD and source code master copies in the SPMC library.

(h) Proofreading documentation produced against CDE input.

(i) Overall checking of documentation for preliminary or final issue.

A specific SPMC operator is assigned to each CDE. All materials, including flowcharts, text, and other program descriptions utilized in SSD production, are input to, and received from, this operator.

An element of documentation cannot be said to exist until it has been given to SPMC for processing; it is not complete until subsequently signed by the CDE and others; and it is not certified until all QA discrepancies have been removed.

Before starting an SSD, the CDE should familiarize himself with DSN standards, Refs. 1 and 2, as well as DSN Data Systems guidelines and SPMC procedures and services listed above.

The aim of the SPMC is to raise software implementation productivity by relieving the CDE of many clerical and routine documentation tasks, using specially trained personnel. The following rules promote further productivity by increasing the efficiency of SPMC resources:

1. Submit legible input in reasonable quantities. Avoid extremes, such as one page at a time, or a full binder of handwritten material all at once.

2. Affirm that the red-lined printout submitted to the SPMC for updating is the current copy in the SPMC database.

> Updating from an obsolete copy of an SSD will increase the turnaround time, and there is a risk in processing some information in error. Check with the assigned operator for the proper current copy for red-lining.

3. Flowcharts may be submitted to SPMC (a) hand-drawn, with or without template, on a 21.6 X 27.9-cm (8-1/2 X 11-inch) or 27.9 X 32.2-cm (11 X 17-inch) paper, (b) as CRISPFLOW written on keypunch coding sheets, (c) as CRISPFLOW written on regular paper, printed or in longhand, or (d) as red-lined corrections of previous submissions.

> CRISPFLOW manuals are available in the SPMC library. Lowercase characters will be converted to capitals before the plots are made. It is the programmer's responsibility to ensure that the charts are readable.*

4. When approved in the SRD/SDD, equivalent forms of documentation may be submitted to SPMC (a) printed on keypunch coding sheets, (b) hand-written (printed or longhand) on regular 21.6 X 27.9-cm (8-1/2 X 11-inch) paper, or (c) red-lined corrections to previous submissions. Alternatively, the programmer may build and edit his own program design files.

> SPMC will provide entry and editing of user CFG-PDL files, and will maintain CFG-PDL source for each CDE. It is the CDE's responsibility to see that the SPMC master is kept current.

*Structured flowcharts processed by the SPMC currently cost about $2 to $4 each; structured charts submitted in CRISPFLOW form by the CDE cost even less. Unstructured flowcharts run about $35 to $40.

5. Each CDE should confer with the assigned SPMC operator to clarify the following items:

(a) The schedule required for SSD production.

(b) The general structure of the final document. In particular, the expected use of flowchart-equivalent documentation, in which sections of the SSD, and how this should be integrated with word-processor text.

(c) The transfer of existing material from the SPMC database into the SSD, such as standard "boilerplate," excerpts from other SSDs, SOMs, etc.

(d) The way the SSD will be submitted to QA and published in final form.

(e) The segmentation of the documentation database necessitated by word-processor limitations.

> It is very important that the CDE be made to understand how the word-processor limitations affect the documentation to be produced, so that he may take part in organizing the segmentation to fit with the way the SSD is to be split into several volumes, when applicable. It is also important that the SPMC operator be apprised of the content of each volume, of multiple volumes are necessary.

(f) The way published SSDs can most easily be upgraded by the use of document amendments in the form of change pages (rather than complete reissue).

(g) The techniques used to thread the program specifications (Section 5 of the SSD) into code.

> The SPMC can, perhaps, aid in preparation or generation of indexes or a code table of contents, if instructed properly.

Appendix A
Documentation Examples

This appendix contains sample pages of documentation prepared according to the standards of this Guide. These SSD excerpts are meant primarily to exhibit documentation style and level of detail rather than to describe correctly the DSN subsystems from whose documentation these samples were taken. In some instances, where precise data was unavailable or inadequate, fictitious information was inserted to show the type and style of information that is required at those points.

In these samples, text in italics is not part of the SSD, but contains comments about the material at that point.

Examples A.1 through A.7 are representative of material from each of the various sections of the outline suggested in the applicable DSN Standard. The next, A.8, presents a narrative-flowchart-code example of documentation and code in the detail desired, and illustrates the conventions for documenting threadability from design into code. Example A.9 gives typical Class B single-item design descriptions that need not be further refined in the programming specifications (SSD Section 5).

Example A.10 shows CFG-PDL, HIPO (hierarchic input-processing-output), and decision-table equivalents to documentation items. The examples shown in A.11 are miscellaneous items, such as state diagrams, data structure diagrams, glossary items, program structure chart style samples, and the like.

A.1 Example of INTRODUCTION Material

1.0 INTRODUCTION

1.1 Purpose and Scope of SSD

This document describes the Deep Space Station Communication Terminal Program, also known as the Communications Monitor and Formatter (CMF) program, since it resides in and operates the CMF assembly.

This specification applies to that software produced in exact and complete fulfillment of requirements set forth in Reference 1.3.1, DSN Ground Communications Requirements. Previous versions of the CMF program were only partially compliant with these requirements. The software requirements for this program are covered by Reference 1.3.2, as amended by the ECOs listed with the reference.

1.2 General Program Description

The Deep Space Station Communication Terminal Program, hereinafter merely referred to as the Comm Terminal program, or CMF program, resides in the CMF assembly, which is the Deep Space Station (DSS) terminus of the High-Speed Data Lines (HSDLs). The CMF program, in conjunction with the CMF hardware, multiplexes and processes all high-speed data transmitted to and from a particular DSS.

The CMF assembly is composed of two Modcomp II-25 computers (one prime and one backup), plus special peripheral hardware devices for interfacing with the HSDL and with DSS equipment. The CMF assembly, in turn, is part of the Station High-Speed Data Subsystem (SHS), with assembly identification number 91.6. The SHS is then part of the Ground Communications Facility (GCF) High-Speed Data Subsystem, or GHS. The GHS, depicted in Figure 1.2-1, also includes the Central Communications Terminal (CCT), which is the JPL terminus of the HSDL. The CCT includes the Error Detection and Correction (EDC) assembly, the High-speed SWitch (HSW) assembly, the Central Communications Monitor (CCM) assembly, and their interconnections. The GHS is one component of the Mark-III Data System (MDS). The CMF program is a real-time operational program, interactive with other GHS computers and with other MDS computers at a deep space station.

The primary function of the CMF program is to provide the station with the ability to communicate with the outside world. The data from the MDS computers is made ready for transmission by calculating the error polynomial (encoding), so that when the data and polynomial are received, a test of the polynomial (decoding) can determine whether any errors were introduced in the transmission process. Similarly, data received at the station is decoded to detect any received errors. If errors are detected, the data block error status bits are set to indicate that condition. All data blocks, good and bad, are delivered with the error status bits set to indicate the data quality.

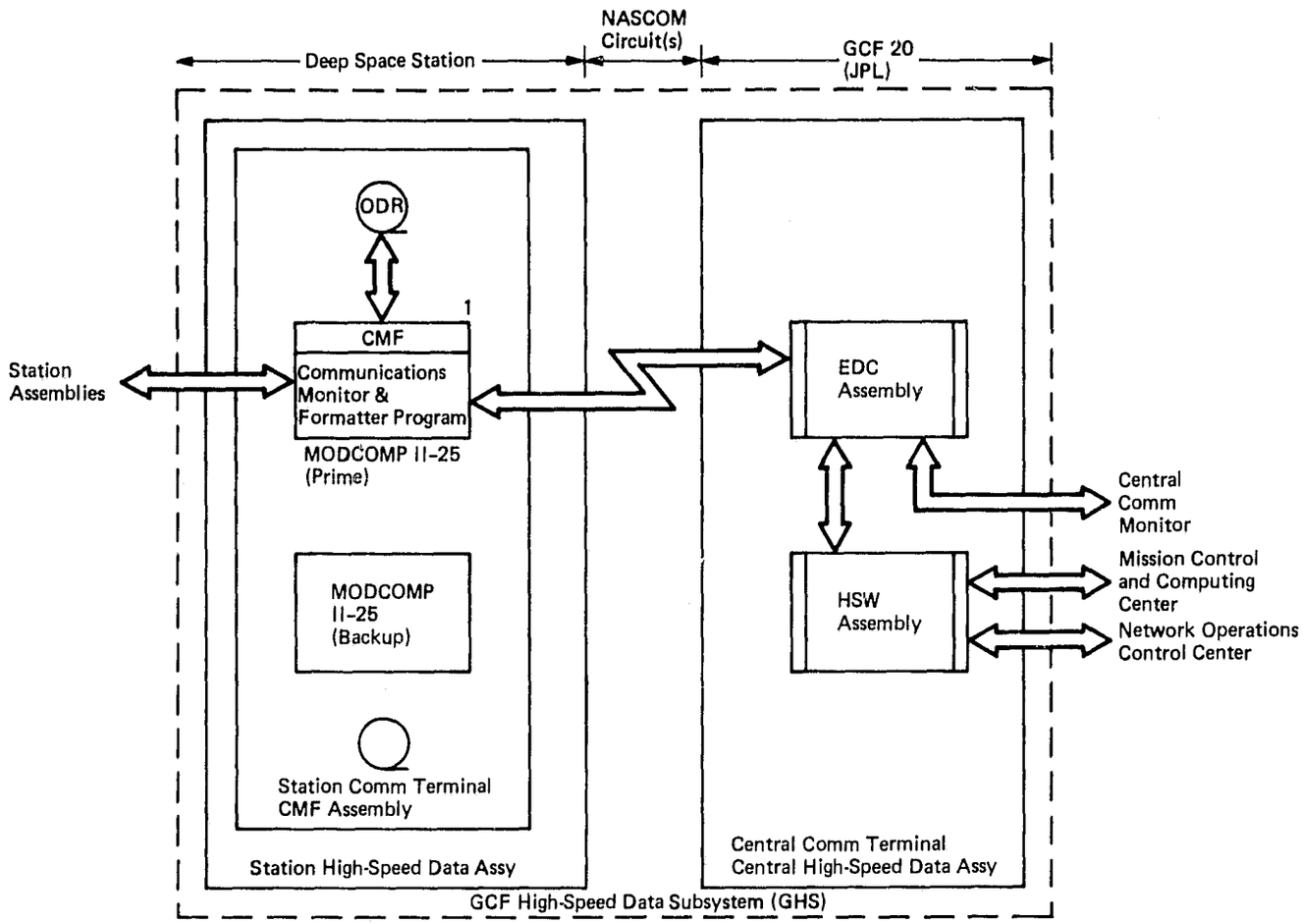


Fig. 1.2-1. GCF High-Speed Data Subsystem overview

A second function of the CMF program is to build an Original Data Record (ODR) by logging all outbound data received from the station assemblies. The station assemblies may indicate whether the data is to be logged, transmitted or both. Normally, all data are both logged and transmitted; log-only and transmit-only are special cases. The CMF recovers temporary ODRs held by the other MDS assemblies, if necessary to perform this function.

The third function of the CMF program is the monitoring of the health of the CMF assembly and the entire GHS. Monitoring of the GHS assembly itself is done within the program; a periodic station summary is reported to the Digital Instrumentation Subsystem (DIS) Monitor and the Data System Terminal (DST). Additional backup displays at the DSS are available on operator request. Monitor information is also periodically formatted into a data block and sent over the HSDL to the CCM computer at JPL. The CCM provides global monitoring of the GHS in a single location.

The fourth function of the CMF program is to provide line printer service to the DIS Monitor Subsystem. A request for service is queued if the line printer (printer/plotter) is busy (such as may often be the case at a conjoint station). When the printer is no longer busy, the request in the queue is honored.

1.3 Applicable Documents

.1 DSN Ground Communications Requirements: GCF Functional Requirements (1975 through 1978), DSN Document 823-1, SS91.0-95.0, Dec. 1, 1975 (JPL internal document).

.2 Software Requirements Document: High-Speed Data Subsystem Station Communications Terminal Program, DSN Document SRD-DMH-5115-OP-D, Sept. 20, 1978 (JPL internal document).

Requirements are augmented by the following ECR/ECOs:

(ECR/ECO list appears here.)

.3 Software Specification Document: High-Speed Data Subsystem Error Detection and Correction Program, DSN Document SSD-DMH-5110-OP, Oct. 20, 1978 (JPL internal document).

.4 Software Specification Document: High-Speed Data Subsystem High-Speed Switch Program, DSN Document SSD-DMH-5111-OP, Oct. 15, 1978 (JPL internal document).

.5 Ground Communications Facility High-Speed Data Subsystem Test Plan and Test Procedures, DSN Document SD512008, Rev. A, Aug. 8, 1978 (JPL internal document).

.6 Software Test and Transfer Document: High-Speed Data Subsystem Communications Terminal Program, DSN Document STT-DMH-5115-OP-D, Sept. 1, 1978 (JPL internal document).

.7 Software Implementation Guidelines and Practices, DSN Standard Practice 810-13, Aug. 1977 (JPL internal document).

.8 Preparation of Software Requirements Documents, DSN Standard Practice 810-16, Dec. 1975 (JPL internal document).

.9 Preparation of Software Definition Documents, DSN Standard Practice 810-17, July 1976 (JPL internal document).

.10 Preparation of Software Specification Documents, DSN Standard Practice 810-19, Mar. 1977 (JPL internal document).

.11 Preparation of Software Operator's Manuals, DSN Standard Practice 810-20, Feb. 1977 (JPL internal document).

.12 Preparation of Software Test and Transfer Documents, DSN Standard Practice 810-21, Nov. 1976 (JPL internal document).

.13 "MDS Test Software Plan," IOM 3380-76-195, Data Systems Section, Apr. 16, 1976 (JPL internal document).

. . . *and so on.*

A.2 Example of STANDARDS AND CONVENTIONS Material

2.0 STANDARDS AND CONVENTIONS

2.1 Specification Standards and Conventions

2.1.1 Applicable Documentation Standards

This document was composed in accordance with the guidelines set forth in DSN Standard Practice 810-19, Preparation of Software Specification Documents, dated March 1, 1977 (see 1.3.10).

2.1.2 Exceptions to Specified Documentation Standards

The creation of these detailed specifications involved more than ten people over a three-year span, during which time the DSN guidelines for programming and documentation were in a state of flux. Although the information is now complete, documentation for single tasks may reflect both the individual style of programmers as well as individual interpretations of evolving standards.

Specifically, these differences are the following:

(A list of exceptions appears here.)

2.1.3 Special Documentation Standards

2.1.3.1 Register/bit nomenclature

Register usage is often specified as either R or REG (e.g., R1 or REG1). Bits are specified as B or BIT (e.g., B15 or BIT 15). When a bit is indicated as "SET," this means it possesses, or is changed, to a value of one (=1). The bit is changed to zero (=0), or has the value zero, when the word "RESET" is used.

2.1.3.2 Decimal/hex notation

A number is hexadecimal if it is preceded by a pound (#) sign; otherwise it is decimal.

2.1.3.3 REX service notation

The Request for EXecutive (REX) services are referenced in two ways. Either REXZZ or REX,#ZZ is used as the striped box identifier, where ZZ is the REX number. Both forms indicate the same REX subroutine.

2.1.3.4 Register save areas

Most subroutines save the registers they utilize in dedicated save areas. The labels of these save areas are not listed on the documentation. Consult the code listings for further information.

2.1.3.5 Message codes

When specific message codes are referenced by the program, the documentation presents within quotation marks the ASCII message corresponding to the code instead of the code itself.

. . . *and so on.*

2.2 Programming Conventions

2.2.1 Applicable Policy and Procedure Documents

For the most part, the CMF program was coded in accordance with DSN Standard Practices 810-13, Software Implementation Guidelines and Practices, dated August 1, 1977, and 810-19, Preparation of Software Specification Documents, dated March 1, 1977, which require the program to be coded in accordance with the principles of structured programming (see 1.3.7 and 1.3.10).

2.2.2 Exceptions to Established Policies and Procedures

There are several exceptions to the established policies scattered throughout most of the tasks. One task in particular, TRS, is wholly unstructured.

Waivers for these exceptions are listed below; the actual waivers appear as Appendix 7.10.3.

(A list of waivers appears here.)

2.2.3 Special Policies and Procedures

(A list of special policies and procedures applied to this program appears here, only as needed to record the as-built character of the program.)

2.2.4 Applicable Programming Standards

2.2.4.1 Operating system

The CMF program utilizes the MDS Standard MAX-III Operating System (SOS) generated by the standard DSN Data Systems Job Control Code (see Appendix 7.11).

2.2.4.2 Existing programming standards

The MDS project did not levy project-wide programming standards other than the following:

a. Header cards are required on all subroutines, subprograms, and tasks.

. . . *and so on.*

2.2.5 Exceptions to Specified Programming Standards

None.

2.2.6 Special Programming Standards

2.2.6.1 Argument transfer

Arguments are transferred to and from subroutines by way of registers.

2.2.6.2 Register storage

All registers used by a subroutine are saved immediately upon entry and restored immediately before exit.

2.2.6.3 Subprogram and subroutine naming

The name of a coded module is also the name under which it is documented.

. . . *and so on.*

2.2.7 Programming Language

All modules are coded in Modcomp Assembly language to expedite the real-time functions of the program.

2.3 Test and Verification Standards

2.3.1 Applicable Test and Verification Standards

No standard tests or verifications were available for this program. However, DSN Standard Practices 810-13 and 810-21 (see 1.3.7 and 1.3.12) state the standard test and verification policies. MDS Test Software was covered by the Test Software Plan (see 1.3.12).

2.3.2 Exceptions to Specified Test and Verification Standards

2.3.2.1 No peer review

DSN Standard Practices require peer review of all design items. However, resources for peer review were not allocated for this program. A waiver was issued (see Appendix 7.10.2.3).

2.3.3 Special Test and Verification Standards

Test standards for the CMF program are implicit in the GCF High-Speed Data Subsystem Test Plan and Test Procedures (see 1.3.13).

2.4 Quality Assurance Standards

2.4.1 Applicable QA Standards

QA involvement is covered by DSN Standard Practices 810-13, DSN Software Implementation Guidelines and Practices; 810-19, Preparation of Software Specification Documents; and 810-21, Preparation of Software Test and Transfer Documents (see 1.3.7, 1.3.10, and 1.3.12).

2.4.2 Exceptions to Specified QA Standards

None.

2.4.3 Special QA Standards

None.

A.3 Example of ENVIRONMENT AND INTERFACES

3.0 ENVIRONMENT AND INTERFACES

The DSS Command Subsystem is part of the DSN Mark III Data System (MDS) implementation. The MDS is fully described in References 7.2.3 and 7.2.21. The Command Subsystem at 26-meter and 64-meter stations consists of two equipment strings, designated Alpha and Beta. At conjoint stations a third string, termed Gamma, is shared by the 26-meter and 64-meter wings. Figure 3.0-1 illustrates the MDS Command Subsystem interfaces. Reference 7.2.21 contains illustrations of the detailed MDS configurations, including the DSS Command Subsystem, for each DSS.

A command string comprises the CPA, the Command Modulator Assembly (CMA), and the Command Switch Assembly (CSA) (not shown in Figure 3.0-1). The CSA is only located at 64-meter and conjoint stations and is used for switching between the dual exciter assemblies at those stations. The CPA maintains interfaces with:

(a) The Command Modulator Assembly.

(b) The Frequency and Timing Subsystem (FTS) via the Time Format Assembly (TFA).

(c) The Digital Instrumentation Subsystem (DIS), Communications Monitor and Format Assembly (CMF), and possibly the host computer via the Star Switch Controller (SSC).

(d) The Data System Terminal (DST) of the Station Monitor and Control (SMC) Subsystem, via the host.

3.1 Hardware Configuration and Interface

The CPA is a Modcomp II-25 minicomputer, to which is connected a group of DSN standard peripheral devices. These devices include a moving head disk, a local subsystem keyboard/character printer (local subsystem terminal), and an operator control console consisting of two CRT terminals and a keyboard/character printer (DST).

3.1.1 Computer

The CPA minicomputer, Modcomp II-25, is a general-purpose 16-bit digital computer with an 800-nanosecond cycle time. The computer possesses the timing, interrupt, and I/O interface features necessary to support DSN standard and JPL-supplied peripheral devices.

The specifications for the CPA computer are presented in the technical manual for the Modcomp II Computer, References 7.2.11 and 7.2.12.

3.1.2 Standard Peripherals

The CPA has associated with it a moving-head disk with a capacity of 2.5 megabytes. The disk interfaces with the Central Processing Unit

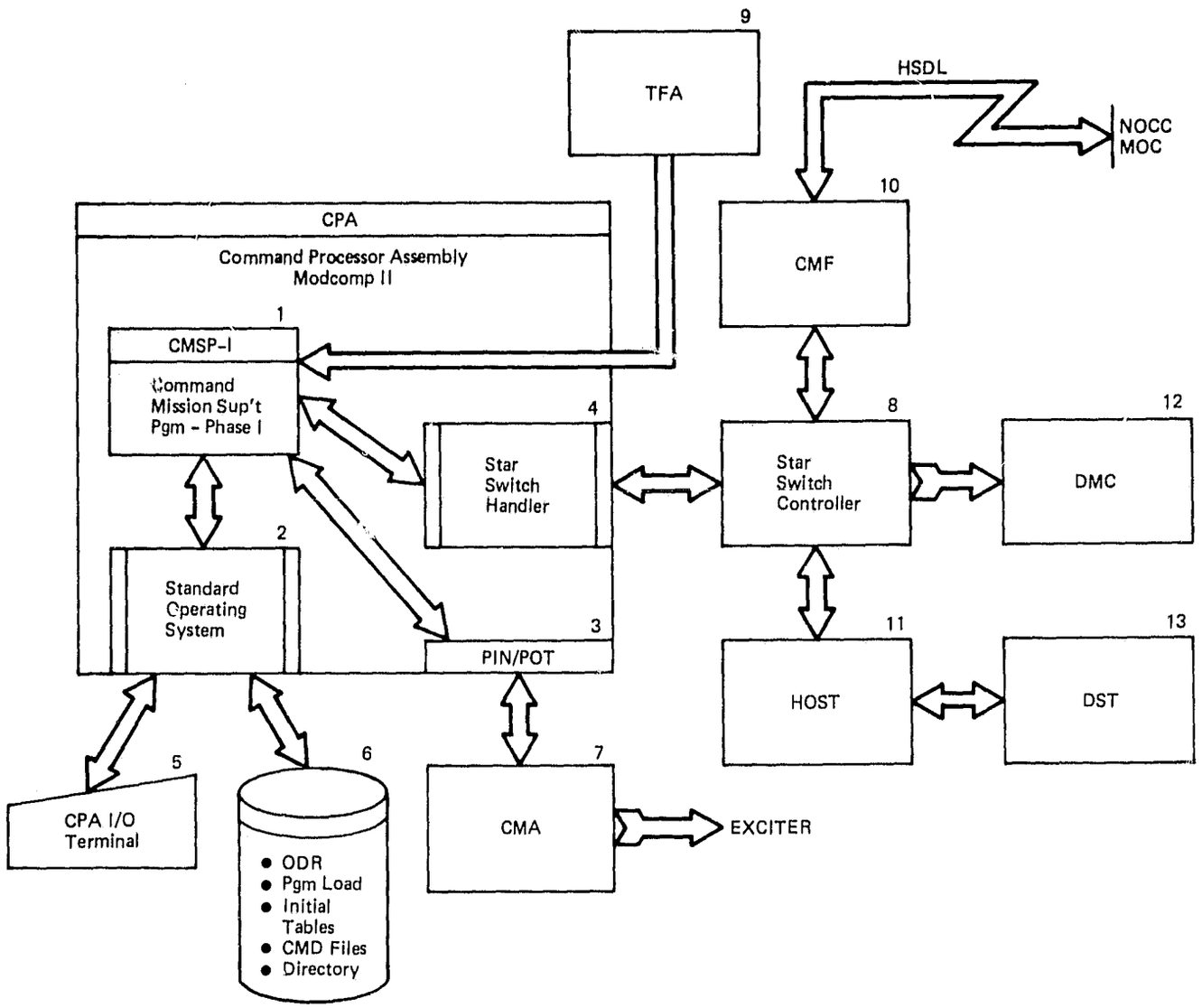


Fig. 3.0-1. Command Subsystem internal and external interfaces

(CPU) through a party-line I/O bus and performs direct-memory I/O operations under the control of an internal Direct Memory Processor (DMP). The moving-head disk provides storage for the CPA Operating System, the command application programs, disk-resident command files, and a temporary ODR. The specifications for the moving-head disk interfaces are presented in the reference manual for the Modcomp II computer, Reference 7.2.18.

The CPA is connected to a local subsystem character printer/keyboard device. The character printer is a serially asynchronous, impact printer with a printing speed of up to 120 characters per second. The keyboard is capable of generating the full set of 128 ASCII characters. The local subsystem terminal uses the standard RS-232C interface. The terminal provides operator communication facilities locally at the CPA. The specifications for the local subsystem terminal interfaces are presented in the Functional Specifications for Terminet 1200 Data Communication Printer, Reference 7.2.14.

(Other standard peripherals are described here.)

3.1.3 Special Peripheral Devices

The CPA also contains or interfaces with JPL-designed equipment. The special equipment interfacing with the CPA includes the CMA, the XDS-920 PIN/POT Emulator, the external DMP, the SIA/SSC, and the TFA.

3.1.3.1 The Command Modulator Assembly

The CMA interfaces are described fully in the Command Modulator Assembly Technical Manual, Reference 7.2.9.

3.1.3.2 XDS-920 PIN/POT Emulator

The CMA was originally designed to operate with an XDS-920 computer through a 24-line, parallel input/output interface. With the replacement of the XDS-920 computer by the CPA (a 16-bit machine), a special XDS-920 PIN/POT Emulator operates with the CPA so as to emulate the I/O signal characteristics of the EOM, PIN, POT, and SKS computer instructions of the XDS-920, respectively.

The specifications for the XDS-920 PIN/POT Emulator are contained in Reference 7.2.19.

(All other interfaces are similarly specified using references or detailed descriptive material.)

3.2 Software Environment and Interfaces

The Command Subsystem software interfaces defined in this document are differentiated as either external or internal with respect to the CPA. Interfaces that are with facilities existing outside the CPA will be referred to as external, and interfaces with modules within the CPA will be considered internal.

3.2.1 Operating System Interfaces

The CPA Operating System is a resident portion of the CPA software. The Operating System (OS) provides input/output control, task management, data management, and executive services in support of the CPA applications software. The CPA OS is generated from a standardized operating system package, based on the Modcomp MAX III Operating System, Extended Version Revision F (see Reference 7.2.17), and includes:

(a) Features to load from disk dynamically and to execute nonresident, transient tasks.

(b) The Basic I/O System (BIOS) that supports standard peripherals, such as a moving head disk, character printer/keyboard device, and CRT/keyboard terminal.

(c) Supervisor calls (REXs), which allow application tasks to request resource use, I/O services, task scheduling, task execution, delay, and utility services.

(d) Features for multiplexing non-interrupt CPA time to support a true priority structure for application tasks.

The standard OS package also provides a special SSC handler for intersubsystem communications and a host symbiont task which will enable the CPA to communicate with the DST when the CPA and DST are not directly connected to each other.

The CPA SYStem GENeration (SYSGEN) process performs the following:

(a) Defines the external interrupt structure.

(b) Selects the real-time clock frequency.

(c) Sizes global common for CPA applications.

(d) Supplies resident operator communications task.

(e) Supplies task-embedded interrupt processors.

(f) Includes specialized instructions and executive services.

Specific details on system generation are contained in Appendix 7.11; usage of the MAX-III features is described in Reference 7.2.17; SSC handler and host symbiont interfaces are detailed in References 7.2.20 and 7.2.21.

3.2.2 Interfaces With Other Subsystems

The data interfaces of the Command Subsystem are with the Exciter/Transmitter Subsystems, the Frequency and Timing Subsystem, the Ground Communication Subsystem, the Station Monitor and Control Subsystem, the Mission Operations Center (MOC), and the Network Operations Control Center (NOCC).

3.2.2.1 Exciter Transmitter Subsystem interface

The CPA interfaces with the Exciter/Transmitter Subsystems through the CMA. The CPA initially configures the CMA for the proper command modulation method, and then proceeds to transfer to the CMA the idle sequences and command data for radiation. (The CMA modulates the selected subcarrier with the command data, forming a composite signal, which is then sent to the Exciter/Transmitter Subsystems for transmission to the spacecraft.) The CPA, utilizing the CMA as a communications link, monitors the Exciter/Transmitter status and confirms that the proper command signal has been transmitted to the spacecraft. Specific bit-level interface values for all parameters are contained in the CMA Technical Manual (Reference 7.2.9).

(Interfaces with other DSS, NOCC, and MOC Subsystems are described or referenced here.)

3.3 Operator Interfaces

(Operator interfaces are described here, or the SOM is referenced.)

A.4 Example of FUNCTIONAL SPECIFICATIONS

4.0 FUNCTIONAL SPECIFICATIONS

4.1 Functional Overview

The CMD program controls the Command Processor Assembly (CPA) to perform the following functions:

(a) Receive and store flight project control and command data.

(b) Receive and store configuration and standards and limits.

(c) Generate command composite signal and initiate its transmission to the spacecraft

(d) Perform system validation

(e) Generate a temporary Original Data Record (ODR).

(f) Format and transmit data to MOC and NOCC.

(g) Display command data and subsystem performance to DSS operations personnel.

(h) Accept manual control from SMC for configuration and commands.

4.2 Software Configuration and Modes of Operation

The Command Mission Support Program (Phase I), or CMSP-I Program, is governed by control mode, program state, and operational mode.

The two control modes are

automatic (or REMOTE)
MANUAL (or local to DSS)

The CPA subsystem states are

BEGIN
INITIALIZATION
REAL-TIME COMMAND
END

The six operational modes are

CALIBRATE-I
CALIBRATE-II
IDLE-I
IDLE-II
ACTIVE
ABORT

4.2.1 Control Mode

The REMOTE mode is characterized by operation in which the Command Subsystem is directed by the remote control center. Command control and data input messages are transmitted over HSD lines to the CPA, and the CPA acknowledges this input and reports the operational status of the subsystem to the remote control center. The DSS remains passive, simply monitoring subsystem operation. The REMOTE mode is the primary method of operation of the Command Subsystem.

The MANUAL mode is an emergency method of operation. In the MANUAL mode, command, control, and data input functions are assumed by the DSS operator (but responsibility is still retained by the remote control center). Control and data input directives are entered via the DSS centralized or local subsystem terminals. Program responses are directed to both the DSS display consoles and the remote control center. During manual commanding, voice communication between the DSS and control center is maintained, but otherwise the control center is passive.

4.2.2 Program State

The CPA is defined to be in the BEGIN state while being loaded for operations. It then transits immediately to the INITIALIZATION state as execution begins, and thence to the REAL-TIME COMMAND state when directed by manual mode control. REAL-TIME COMMAND may transit either to the END state or back to INITIALIZATION, as directed. Figure 4.2.2-1 shows the state-transition diagram. INITIALIZATION and REAL-TIME COMMAND are referred to herein as the CMSP-I program states.

The INITIALIZATION state prepares the DSS Command Subsystem for command operations. The DSS Command Subsystem assumes the MANUAL command mode during INITIALIZATION and rejects all HSD blocks that are received from remote control centers. INITIALIZATION consists of (1) subsystem identification, (2) CPA function selection, (3) project specification, (4) subsystem calibration, and (5) subsystem validation. When initialization is complete and upon input of the "RUN" directive by the DSS operator, the program exits the INITIALIZATION state and enters the REAL-TIME COMMAND state.

Spacecraft command operations are then supported while in the REAL-TIME COMMAND state. After entering the REAL-TIME COMMAND state, the DSS operator normally places the subsystem in the REMOTE command mode. In case of a command system emergency, the subsystem may be returned to the MANUAL mode. When the DSS operator has placed the CMSP-I program in REMOTE mode, then, at first, all HSD input from the NOCC is accepted, but all HSD input from the MOC is rejected. The NOCC configures and validates the Command Subsystem and, upon completion of these functions, relinquishes control to the MOC to direct further command operations. During MOC operations, the CPA receives spacecraft command data, initiates and maintains radiation of the spacecraft commands, and monitors subsystem operation to ensure the correct and timely transmission of commands. The command program exits the REAL-TIME COMMAND state to the INITIALIZATION state if the DSS operator inputs a "REIN" directive; the program terminates

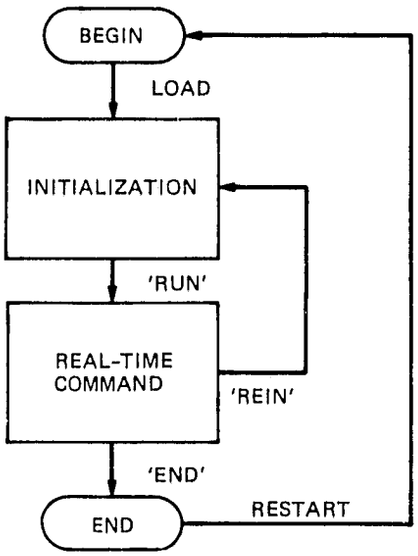


Fig. 4.2.2-1. CMSP-I program state diagram

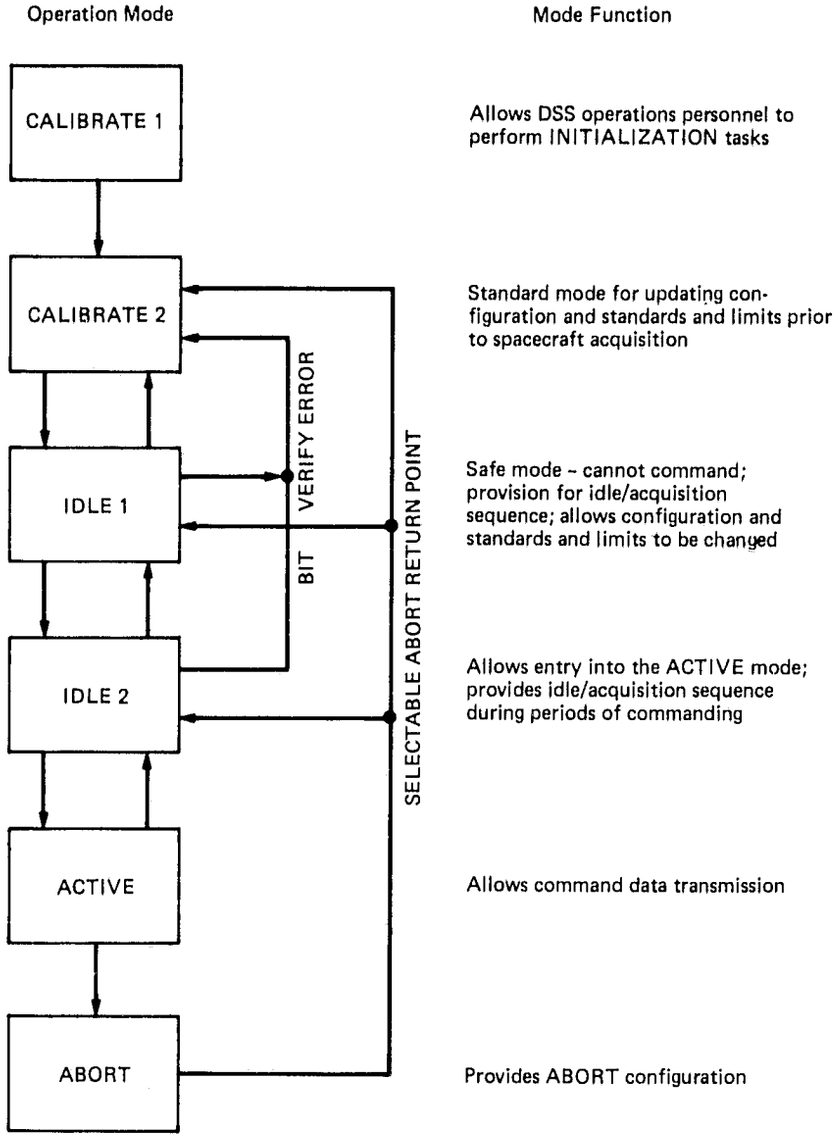


Fig. 4.2.3-1. CMSP-I program operational modes

on the input of the "END" directive. After program termination, the program may be restarted only by reloading the command program.

4.2.3 Operational Modes

The DSS Command Subsystem, at any one time, is in one of the six operational modes; these govern the orderly progression of the subsystem through the initialization, configuration, acquisition, and command radiation sequences that are required to transmit commands to the spacecraft successfully. Figure 4.2.3-1 shows the mode-transition diagram.

The CALIBRATE-1 mode provides the opportunity for the DSS operator to initialize the command software for the support of command operations. The Command Subsystem enters the CALIBRATE-1 mode immediately after the program load. While in this mode, no remote control source may gain access to the Command Subsystem. The command program may exit the CALIBRATE-1 after all initialization has been completed. Then one of two control sources may cause the command program to proceed. The DSS operator may move the Command Subsystem to the CALIBRATE-2, IDLE-1, or IDLE-2 mode prior to selecting the REMOTE operational mode. However, if the REMOTE mode has been selected while in the CALIBRATE-1 mode, NOCC may move the Command Subsystem to the CALIBRATE-2 mode by updating the subsystem configuration. The Command Subsystem cannot radiate commands from the CALIBRATE-1 mode.

(A similar discussion of all operational modes ensues, to end the overview.)

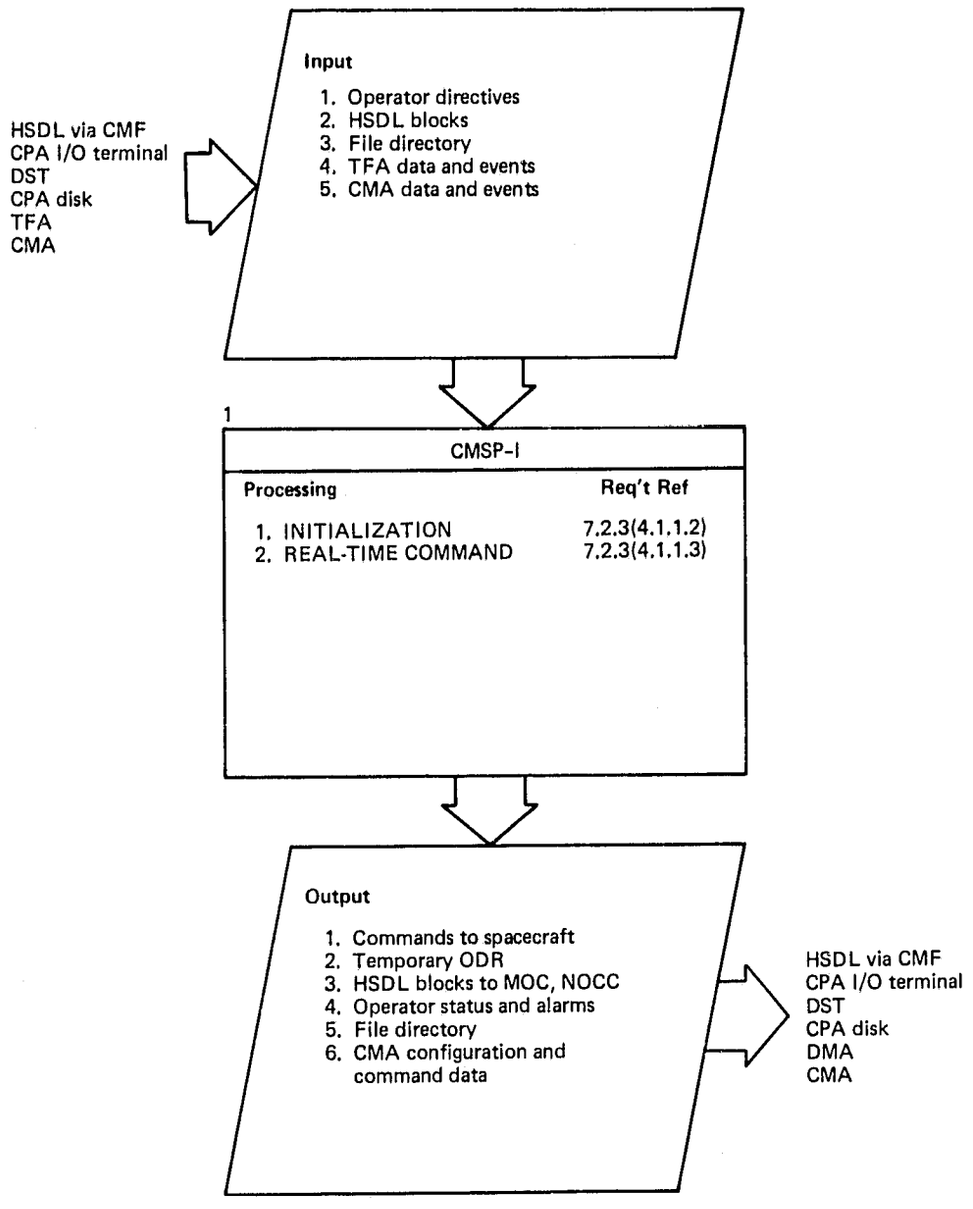
4.3 Detailed Functional Specifications

4.3(1) Detailed Functional Specifications

The program functions are displayed in the input-processing-output diagram 4.3(1), separated into groups of functions by program state. The superposition of command mode and operational modes on these states appears as the heirarchic definition of the functions unfolds. Inputs and outputs are refined by function; composite I/O specifications and format appear in Appendix 7.7.

Input — Processing — Output Table

Description: Command Mission Support Program (Phase I)	Function 1
	Identifier: CMSP-I
Prepared by: Frank Hlavaty Phone: 6533	Date: 5/4/76

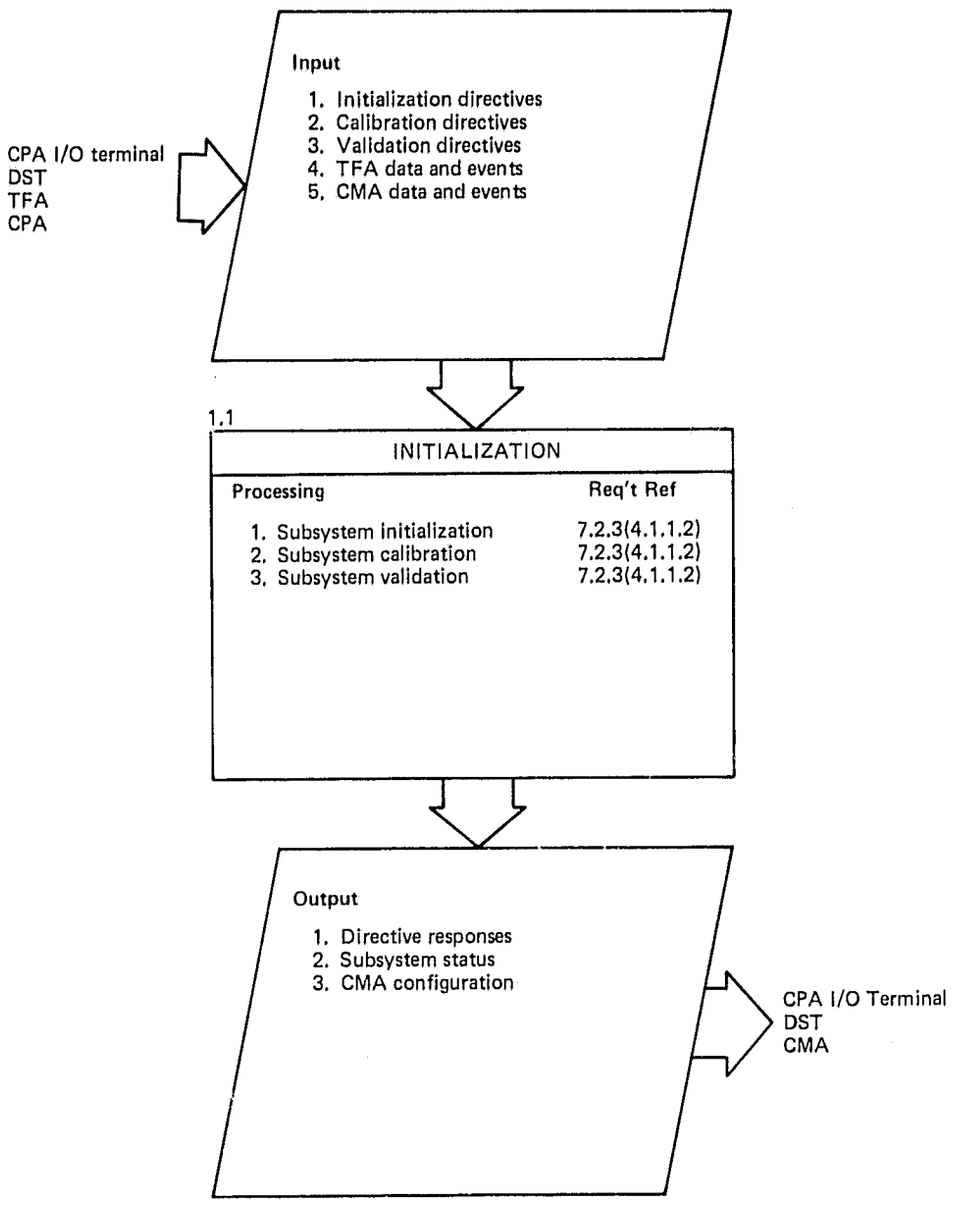


4.3(1.1) Initialization Functions

The CMSP-I program receives initialization, calibration, and validation directives entered by the DSS operator and acknowledges this input with an operator response message. The program outputs subsystem status messages to the DSS operator reflecting the calibration and validation functions being performed. All access to the subsystem via the HSD communication link is rejected until the program exits the INITIALIZATION state, upon receipt of a directive from the DSS operator to proceed to REAL-TIME commanding.

Input – Processing – Output Table

Description: Initialization Function	Function 1.1
	Identifier: INITIALIZATION
Prepared by: Phone:	Date: 5/7/76



4.3(1.2) Real-Time Command Functions

In the REAL-TIME COMMAND state, the CMSP-I program is capable of accepting, acknowledging, and processing either remote (HSD) or manual (DSS) input directives, depending on the operational mode. The CMA is configured to support a particular deep-space mission in accordance with parameters supplied during initialization. The program is then capable of receiving, validating, and storing command data files at the DSS. The program initiates and controls the command radiation while monitoring the subsystem status for abortive and alarm conditions that may affect spacecraft command operations. Subsystem status and alarm conditions, command events, and command radiation status are reported to the remote control centers via HSD lines and are displayed locally for the benefit of DSS operators. A temporary ODR is maintained by the program for all incoming and outgoing HSD blocks in order to provide a data record of all command operations, should communications with the GCF Subsystem at the DSS fail.

The program receives CMA configuration, file maintenance, radiation control, and status request inputs either locally or via the HSD lines. Responses are provided to all input directives. The program also provides subsystem status and alarm, command event, command radiation, and file directory information upon operator request (see SOM) or as a result of processing conditions.

The DSS Command Subsystem may be in either the REMOTE (nominal) or the MANUAL (emergency) mode of operation during the REAL-TIME COMMAND state.

The program exits the REAL-TIME COMMAND state on program termination or reinitialization.

(Hierarchic input, processing, and output descriptions of the program continue. The next level of function 1.2.4 is outlined below.)

Input – Processing – Output Table

Description: Real-Time Command Function	Function: 1.2
	Identifier: REAL-TIME COMMAND
Prepared by: Frank Hlavaty Phone: 6533	Date: 5/7/76

Input

1. CMA configuration data
2. Command file maintenance data
3. Command radiation control data
4. Subsystem, directory, and command status requests
5. TFA data and events
6. CMA status and events

1.2

REAL-TIME COMMAND

Processing	Req't Ref
1. Accept/acknowledge HSD directives	7.2.3(4.1.1.3)
2. Accept/acknowledge DSS directives	7.2.3(4.1.1.3)
3. Configure CMA	7.2.3(4.1.1.3)
4. Maintain command files and directory	7.2.3(4.1.1.3)
5. Initiate/control/monitor command radiation	7.2.3(4.1.1.3)
6. Monitor/report subsystem alarm conditions	7.2.3(4.1.1.3)
7. Generate temporary ODR	7.2.3(4.1.1.3)

Output

1. Input responses
2. Subsystem status and alarms
3. Command events to CMA
4. Command files and directory
5. Command radiation status
6. Command data files
7. Temporary ODR

(See Function 1)

4.3(1.2.4) Maintenance of Command File Directory

Command file directory maintenance refers to the receiving, validating, and storing of command data files at the DSS and also maintaining an up-to-date directory (refer to 7.7.4.1) of all command files currently stored. Directive inputs may be received either via HSD lines and the CMF or through DSS operator terminals, depending on the mode of operation. Command data and file directory information are always available on the CPA disk.

File directory processing includes the functions of initializing the command file directory, validating and storing command data files, erasing of selected command data files listed in the directory, and providing file directory information to an operator upon request. Directive responses are sent back to the input source. Event messages resulting from file maintenance activities are displayed for the DSS operator. Command file directory information are reported to a requesting source. Command data files and directory contents are always maintained on the CPA disk.

Command file directory maintenance via HSD lines is possible only when the DSS Command Subsystem is in the REMOTE operational mode.

HSD data is received and transmitted by the program in the form of SSB standard data blocks (Reference 7.2.31), input via the CMF at the DSS. The format and contents of the HSD command file directives and data inputs are described in Reference 7.2.1, Section COM 4-4. Other input and output formats are specified in this document (see Appendix 7.7).

Data flow in the CMSP-I program is shown for MANUAL and REMOTE modes functionally in Figures 4.3(1.2.4)-1 and 4.3(1.2.4)-2.

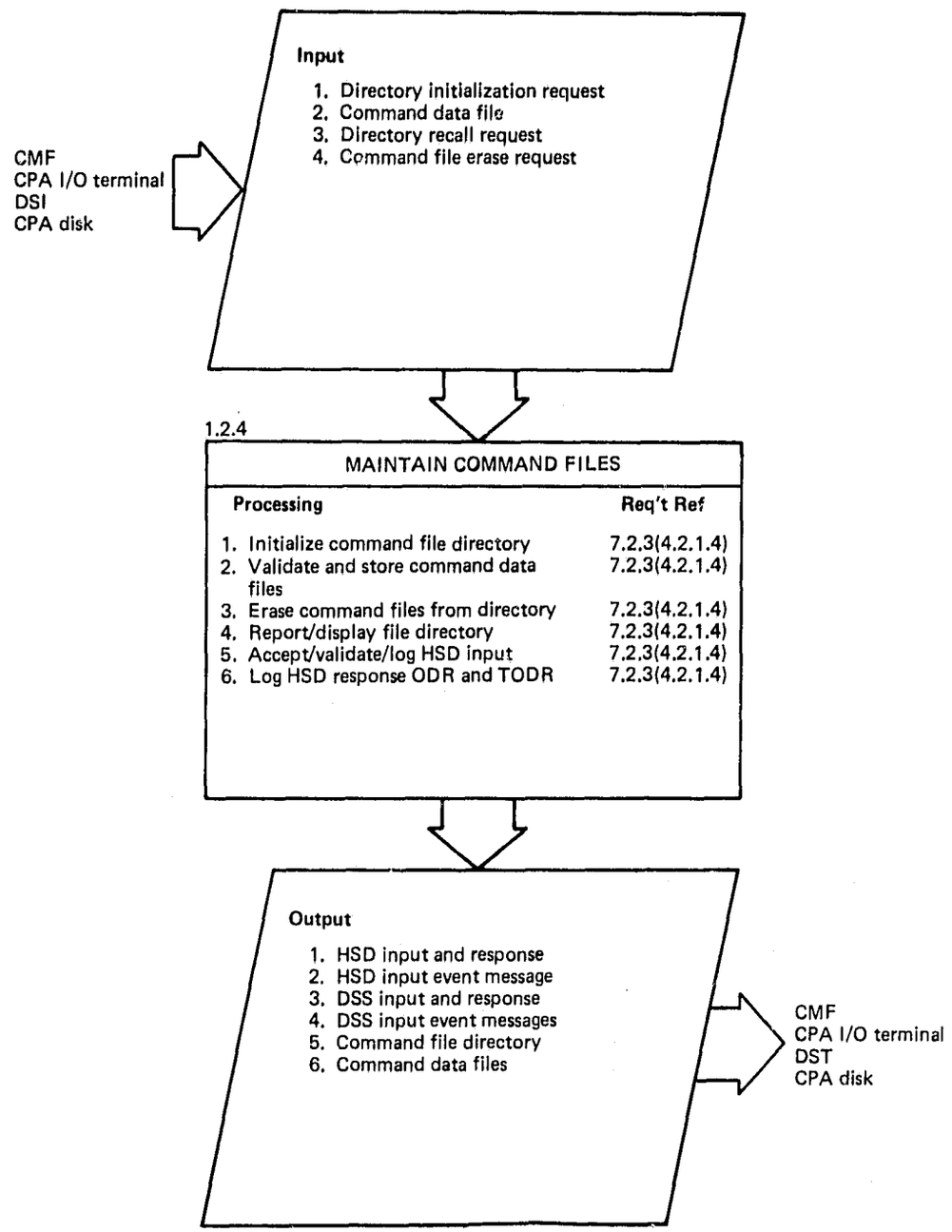
(Hierarchic detailing of function continues to the detail level.)

4.4 Data Base Specifications

(Here, perhaps, would be a description of the Command File Directory, Command Data Files, and ODR File.)

Input – Processing – Output Table

Description: Maintenance functions associated with stored command files and the directory of these files	Function: 1.2.4 Identifier: MAINTAIN COMMAND FILES
Prepared by: Frank Hlavaty Phone: 6533	Date: 6/27/76



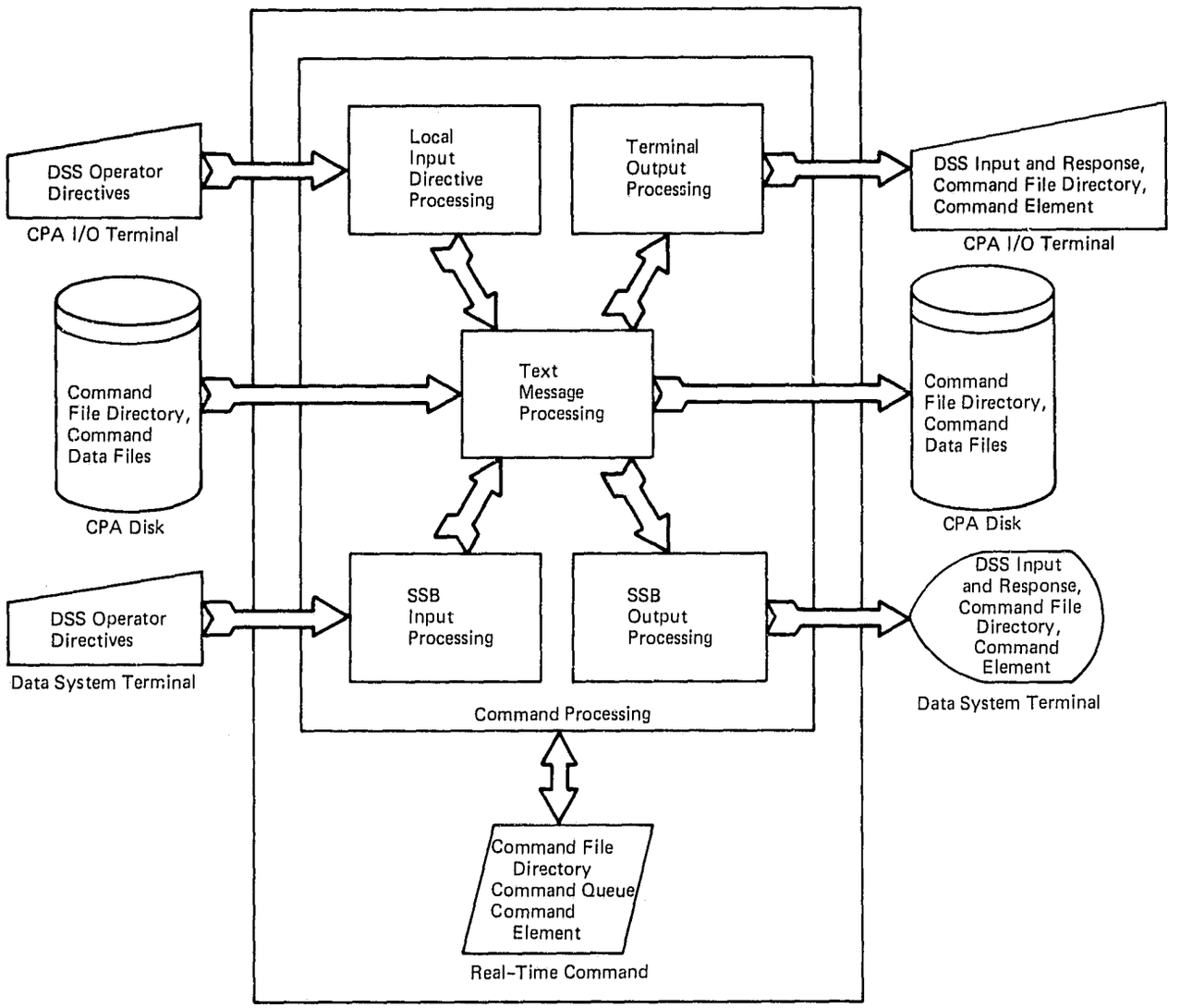


Fig. 4.3(1.2.4)-1. Command file directory maintenance processing data flow, MANUAL mode

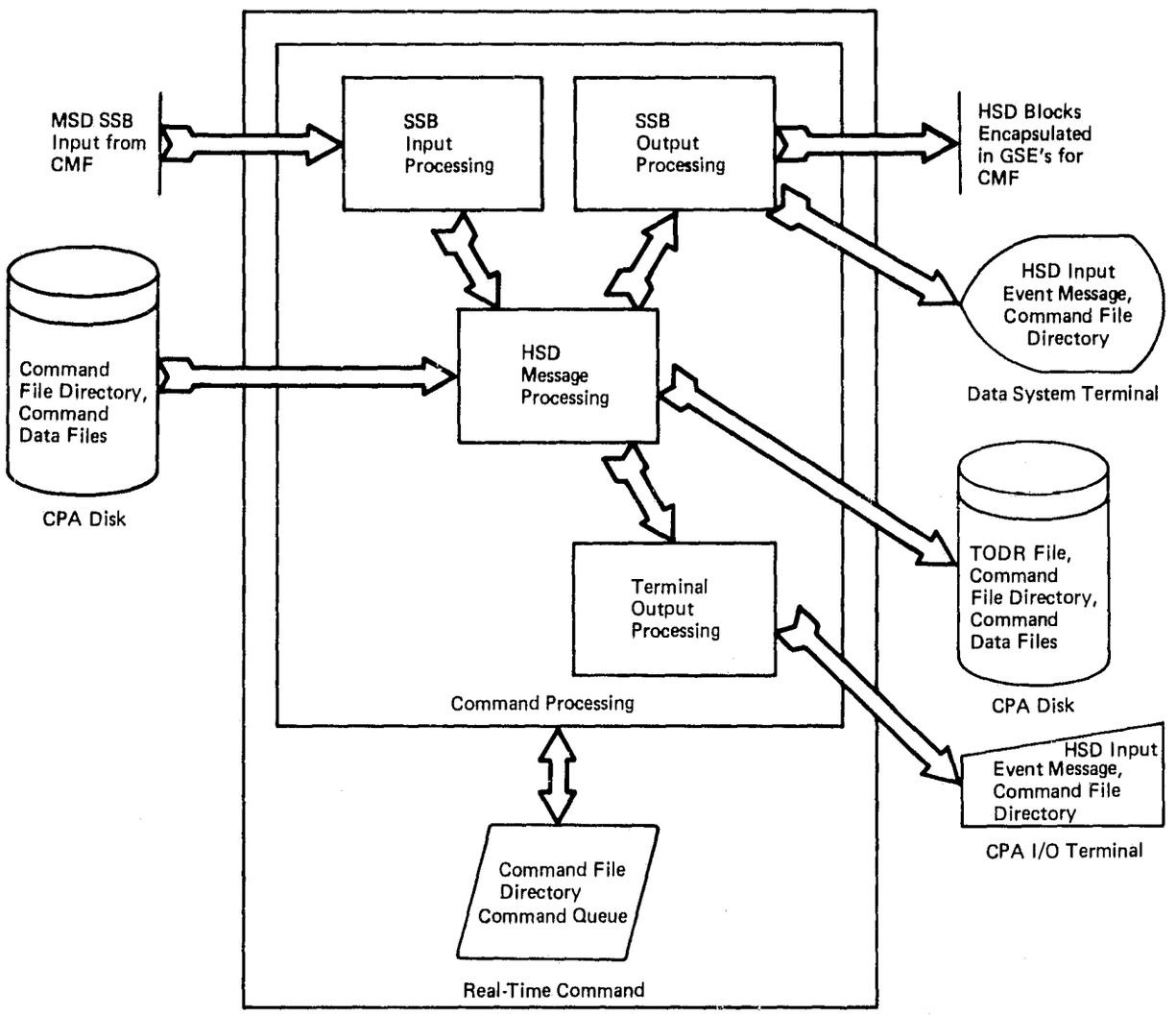


Fig. 4.3(1.2.4)-2. Command file directory maintenance processing data flow, REMOTE mode

A.5 Example of PROGRAM SPECIFICATIONS

5.0 PROGRAM SPECIFICATIONS

5.1 Program Overview

The SCHEDULER program consists of four subprograms that execute in sequence:

BUILD NETWORK	(BUILD)
TOPOLOGICAL SORT	(TOPOSORT)
CALCULATE DATES	(DATES)
DISPLAY SCHEDULE	(DISPLAY)

The representation of the schedule network is the chief item required for understanding the procedures in the remainder of this section. There is a certain set of information that is input or calculated for each task, represented by a NODE, in the network:

information	identifier	type
TASK CODE	CODE	string
TASK TITLE	TITLE	string
TASK DURATION	DUR	integer
EARLIEST START	EST	integer
EARLIEST FINISH	EFIN	integer
LATEST START	LST	integer
LATEST FINISH	LFIN	integer
FLOAT TIME	FLOAT	integer
NUMBER OF PROCESSORS	COUNT	integer
POINTER TO LIST OF SUCCESSORS	TOP	pointer

The first three items are input directly, the next five are calculated after sorting, and the final two are supplied as a result of building the network.

The connectivity of the schedule network is achieved by way of the successor list attached to each node's TOP. Each such list element is represented by a CONNECTION data structure having two fields:

information	identifier	type
SUCCESSOR NODE	SUC	node
NEXT LIST ITEM	NEXT	pointer

The "pointer" data type in the two tables, above, is an index into the SUC:NEXT CONNECTIONs; the "node" data type is an index into the CODE: TITLE:DUR:...:TOP NODES. In this way, information about a node (task) is locatable via the node pointer, and its successor nodes can be found

following the node TOP pointer to SUC, for the first, and thence via NEXT pointers to the remainder. This representation is discussed further in Knuth (Reference 7.2.9). The apparent node representation is shown in Figure 5.1-1.

5.1.1 Building and Sorting the Network

The process of building the network consists of reading the Work Breakdown Structure (WBS) task file, and storing task information in the node arrays. As tasks are read, each task node location is recorded in a topological-sort list, TSORT. Locators of nodes having a zero predecessor COUNT are queued into TSORT from the front; these are already in sorted order. The others are queued at the rear of TSORT. Such a list permits the processing of tasks independently of the way the nodes have been entered into memory. Thereafter, the topological sort procedure considers, in turn, each node in the front segment of TSORT, "removing its edges" in the network by reducing the COUNT of nodes identified as SUCcessors. When a COUNT field of a node hits zero, its node locator is inserted at the rear of the front segment of TSORT. If some nodes still have non-zero COUNTs after all of the front segment of TSORT has been processed, a loop in the network exists (identified as an error).

5.1.2 Calculating Schedule Times

Early start and finish times are calculated by scanning the nodes listed in TSORT in forward order, then late start, late finish, and float values, by scanning TSORT in reverse order (see 4.3 for formulas).

Dates are assigned to the project times by way of the CALenDaR array, filled from the calendar file. The CALDR array is an ordered set of strings indexed by work day. More precisely, when filled, the value of CALDR(0) is the starting date (month/day/year) corresponding to START, read in from the project start milestone. START is an integer that defines the day of year for beginning the calendar file read-in. The size of the CALDR array permits entries indexed 0 through MAXDATE. If the project goes beyond MAXDATE days, only the first MAXDATE days appear in the array. If the calendar file does not extend far enough into the future to provide dates for the project times, then "DAY *n*," where *n* is the project work day, is entered into the CALDR array.

Further details are contained in the programming specifications in the remainder of this section.

5.1.3 Program Tier Chart

The list shown in Figure 5.1.3-1 presents the modular nesting of program elements. Identification in this list denotes nesting of the modules named in call-order. Numbers are Dewey-decimal module codes (e.g., TERMINATOR is found as module 1.2.8).

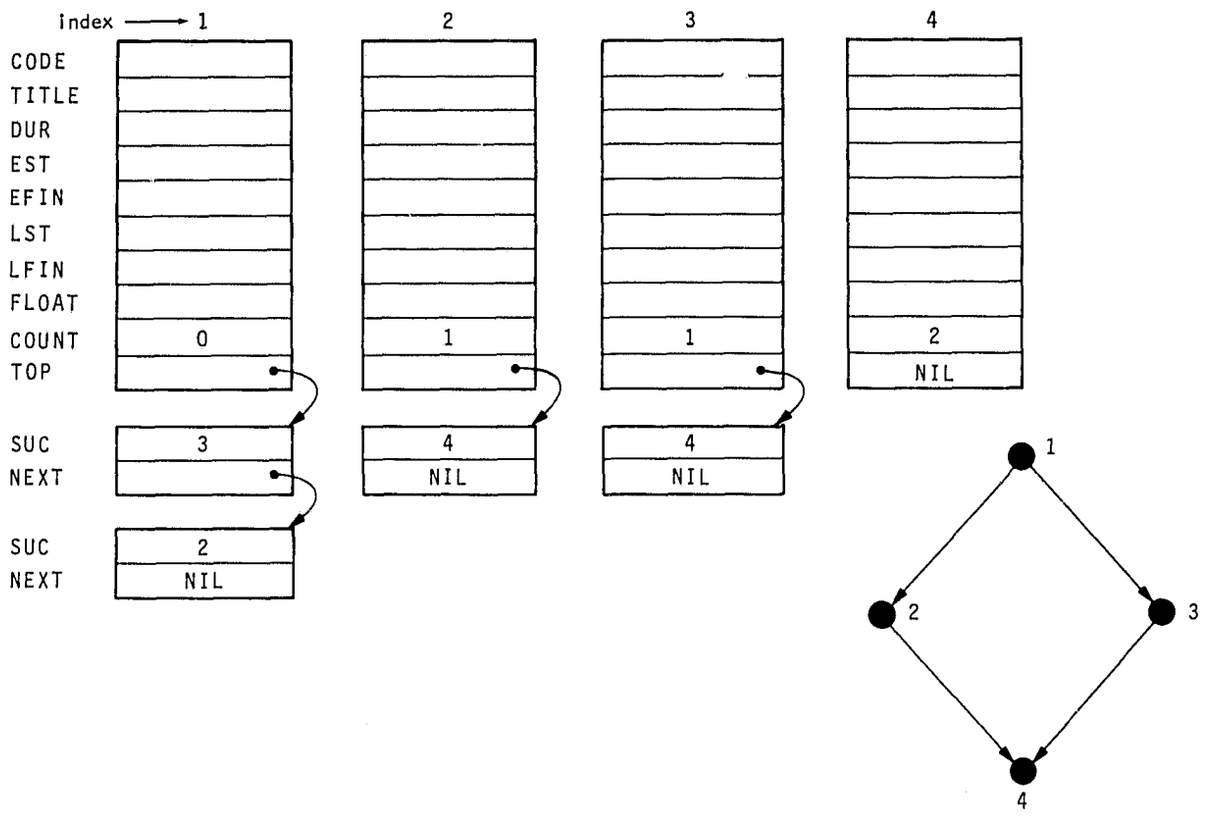


Fig. 5.1-1. A simple precedence graph and its structural representation

```
1       SCHEDULER
        .2      BUILD
                .6      REGISTER
                        .1/S1   SEARCH
                        .6      CONNECT
                                .3/S1   SEARCH
                                .7/E1   ERROR
                .8      TERMINATOR
                        .1/S1   SEARCH
                        .3      CHECK_SUCCESSORS
                                .3/E1   ERROR
                        .6      LINK_TO_FINISH
                                .4.E1   ERROR
        .5      TOPOSORT
                .4      ERASE_EDGES
        .7      DATES
                .1      EARLY_DATES
                        .4      SUCCESSOR_DATES
                .3      LATE_AND_FLOAT
                        .4      TASK_LATE_DATE
        .8      DISPLAY
                .1      CALENDAR
                        .1/E1   ERROR
                        .2/E1   ERROR
                        .6/F1   STR
                        .10/E1  ERROR
        .9      DIAGNOSE
                .1/E1   ERROR
E1      ERROR
F1      STR(STRING FUNCTION)
S1      SEARCH
        .10/E1   ERROR
```

Fig. 5.1.3-1. The SCHEDULER program tier chart

5.2 Main Program Detailed Design

5.2(1) Critical Path SCHEDULER Program

On entry, the program data structures are all uninitialized.

This program is the top-level control procedure that causes all specified program functions to be performed.

On exit, the schedule has been printed and the COUNT field of each node has been destroyed. If no error has occurred, TSORT and the node-connection network are otherwise intact.

.1 All structures and variables are considered global at this point. Prepare HEADER data for the report in step 8.

.2 In building the network, if either the task node arrays or the successor linkage arrays become filled prematurely, print an error message. Add a project termination milestone to the schedule network. Return a flag OVFLOW with false value if the WBS input did not exhaust the network arrays; true, otherwise. NTASKS records the number of tasks entered.

.3 A true value of OVFLOW terminates the scheduler,

.4 printing a message before the program terminates.

.5 If the network was input without overflow, then the TSORT list contains the topological sort. T records the number of sorted items entered into TSORT.

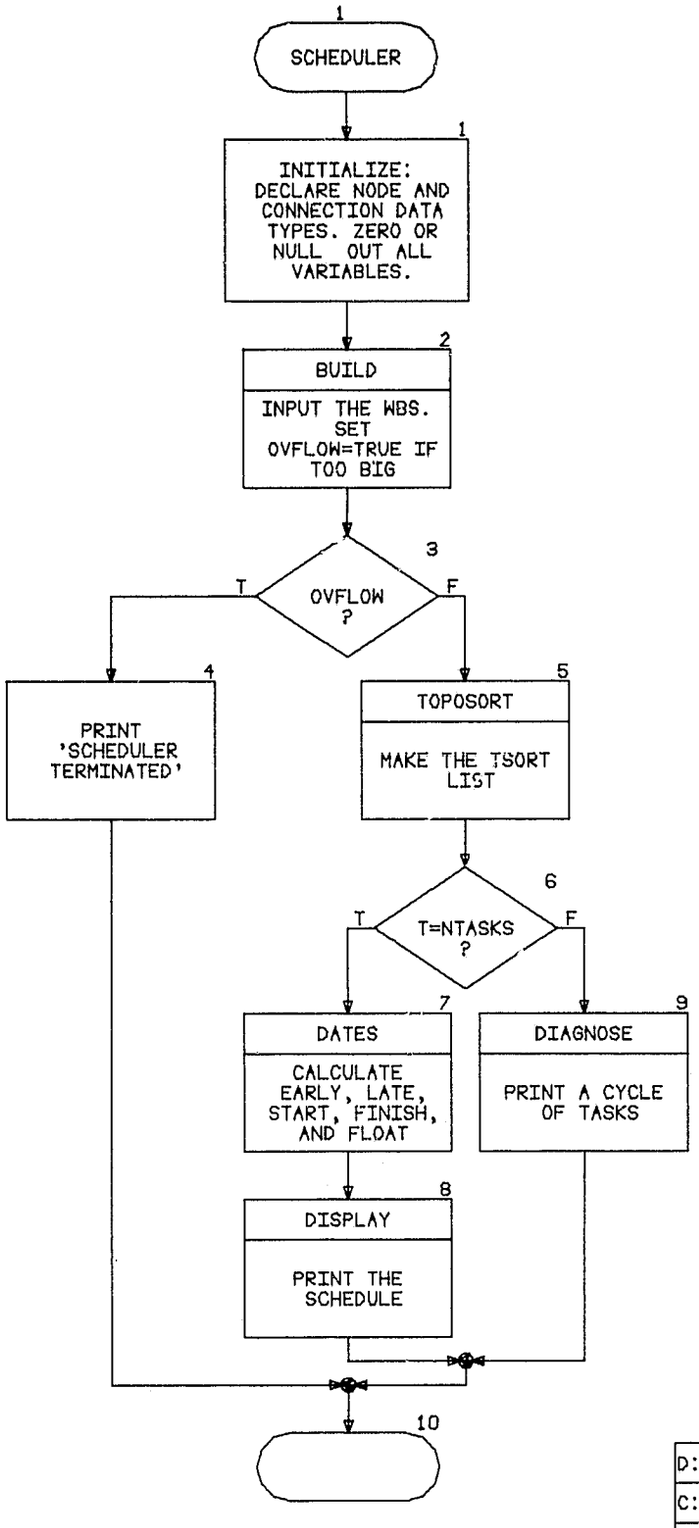
.6 If all tasks are in the list, the WBS is of proper form.

.7 Scan the list forward for early times (see definitions in Section 4.1), and in reverse for late times (Section 4.1). Calculate float times during the second scan as well.

.8 See Sections 4.3 and 7.6 for details of output format and content.

.9 Here a WBS loop has been detected. Print "WBS is circular among items:" and then print the task codes forming one such loop.

.10 Perform any cleanup necessary in the coding language (e.g., closing files) before program termination.



D:	
C:	
A:	17 May 79

MAY 79

(The module hierarchy continues to the level that each flowchart represents about 100 executable lines of source code. Section 5.3 contains similarly documented internal subroutine designs.)

5.4 External Subroutine Interfaces

5.4(F1) STRing Function (STR)

The STR(I) function converts its integer argument, I, to a string value. The returned string is the character representation of the integer, left and right justified (i.e., no spaces). No decimals or commas appear in the format. If the input value is positive, the output string is unsigned; if negative, the output is preceded by a minus sign.

(Other external subroutine interfaces, if any, are similarly described, as needed.)

5.5 Data Structure Definitions

NODE and CONNECTION data structures are described in the program overview (Section 5.1); other variables and values are described in the Glossary (Appendix 7.1).

Data structures defined in this program specification are considered by design to be globally accessible, so as to promote codability. Localized scoping and limited accessibility to data is part of the code design, and may be found in the code listings (Section 8 of this SSD).

5.6 Resource Allocation, Scheduling, and Access

Not applicable.

A.6 Example of VERIFICATION AND TEST SPECIFICATION

6.0 VERIFICATION AND TEST SPECIFICATIONS

6.1 Correctness Test Criteria

During the code checkout activity, input data are used to drive the program through every "flow line" of the procedures at least once. WBS networks are input to (singly) violate each of the network boundary conditions. Such data items include:

(a) Empty WBS.

(b) WBS consisting only of "END" record.

(c) WBS not ending with "END" record.

(d) Sufficient tasks (nodes) to cause overflow.

(e) Sufficient connections to cause overflow.

(f) Tasks with no predecessors.

(g) Tasks with no successors.

(h) Tasks without predecessors or successors.

(i) FINISH task with successors.

(j) WBS with a circular set of tasks.

(k) WBS of moderate size (more than one output page) formatted correctly and within boundary constraints.

The program herein described is deemed ready for acceptance testing provided that these tests demonstrate:

(l) Proper response to tests (a) - (k), above, as determined from functional specifications in 4 and format in 7.2.1-1.

(m) Error-free performance on all of 10 correct user-supplied WBSs ranging in size over the required limits.

(n) The detection of all of nine calibration errors of a random nature inserted into a special test version of the program built for assessing test adequacy. The data in (a) - (m), above, is used, together with other data as required to detect the errors.

Test data (a) - (n), above, are retained in test data files to serve as a regression baseline for maintenance and acceptance test purposes. Each time that step (n), above, is used to qualify a new update of the program, that data, too, is added to the baseline.

A.7 Example of APPENDIX Material

7.0 APPENDICES

7.1 Glossary

This appendix contains the names of program modules, variables, constants, textual acronyms, and special terms used in this document.

AVAIL: AVAILable list pointer. A "pointer" into SUC:NEXT linkages. It contains the address of the next available CONNECTION to be attached to the network.

B: Bottom of unsorted list. An integer index into the TSORT array. In REGISTER/1.2.6, the tasks from B to NTASKS are unsorted.

BUILD/1/2: Procedure name of the module which builds the schedule network.

CALDR: CALenDaR array. Array 0..MAXDATE of date, where date is a string of the form 150477, which prints as a date in the form 15APR77 in 1.8.8.

CALENDAR/1.8.1: Procedure name of module that reads calendar file into CALDR array.

CHECK SUCCESSORS/1.2.8.3: Procedure name of module which prints a list of task codes that have FINISH as a successor.

. . . *and so on.*

7.2 Formats

7.2.1 Scheduler Output Format

Output of the SCHEDULER program is shown by example in Figure 7.2.1-1.

7.3 Memory Map

The memory map of the SCHEDULER program as implemented on the 60K Intel 8085/CPM with memory-mapped VIO is shown in Figure 7.3-1.

7.4 Decision Log

1.(2/12/77) Network and topological sorting will use the method in Knuth, Vol. I, pp. 260-263, including his diagnostic procedure (page 543). Reason: proven and efficient algorithm.

2.(2/13/77) Separate sorted list TSORT will be used in lieu of the depleted COUNT fields, as done in Knuth. Reasons: readability of the design; independence of search/insert method.

PAGE 1

WIDGET PROJECT

WORK BREAKDOWN STRUCTURE

DATE: 12 MAR 77

CODE	TITLE	DURATION	START EARLY DAY	START EARLY DATE	START LATE DAY	START LATE DATE	FINISH EARLY DAY	FINISH EARLY DATE	FINISH LATE DAY	FINISH LATE DATE	FLOAT TIME DAYS
*1	START	0	0	11APR77	0	11APR77	0	11APR77	0	11APR77	0
4	ORDER PACKAGING MATERIALS	10	0	11APR77	35	31MAY77	10	25APR77	45	14JUN77	35
3	FABRICATE USING PARTS ON HAND	20	0	11APR77	10	25APR77	20	9MAY77	30	23MAY77	10
*2	PROCURE NEEDED PARTS	30	0	11APR77	0	11APR77	30	23MAY77	30	23MAY77	0
*5	FABRICATE USING PROCURED PARTS	15	30	23MAY77	30	23MAY77	45	14JUN77	45	14JUN77	0
*6	INTEGRATE AND TEST ASSEMBLY	20	45	14JUN77	45	14JUN77	65	14JUL77	65	14JUL77	0
*7	PRODUCT COMPLETION	0	65	14JUL77	65	14JUL77	65	14JUL77	65	14JUL77	0
*8	PACKAGE AND DELIVER ASSEMBLY	10	65	14JUL77	65	14JUL77	75	28JUL77	75	28JUL77	0
*11	TRAIN PERSONNEL	15	75	28JUL77	75	28JUL77	90	18AUG77	90	18AUG77	0
10	ASSEMBLE TEST GEAR	5	75	28JUL77	80	4AUG77	80	4AUG77	85	11AUG77	5
9	INSTALL AT CUSTOMER FACILITY	5	75	28JUL77	80	4AUG77	80	4AUG77	85	11AUG77	5
12	PERFORM READINESS TESTS	5	80	4AUG77	85	11AUG77	85	11AUG77	90	18AUG77	5
*13	PERFORM ACCEPTANCE TESTS	5	90	18AUG77	90	18AUG77	95	25AUG77	95	25AUG77	0
*14	FINISH	0	95	25AUG77	95	25AUG77	95	25AUG77	95	25AUG77	0
*FINISH		0	95	25AUG77	95	25AUG77	95	25AUG77	95	25AUG77	0

* CRITICAL PATH ITEM

Fig. 7.2.1-1. Output format of the SCHEDULER program illustrated by a hypothetical set of tasks representing a "Widget Project"

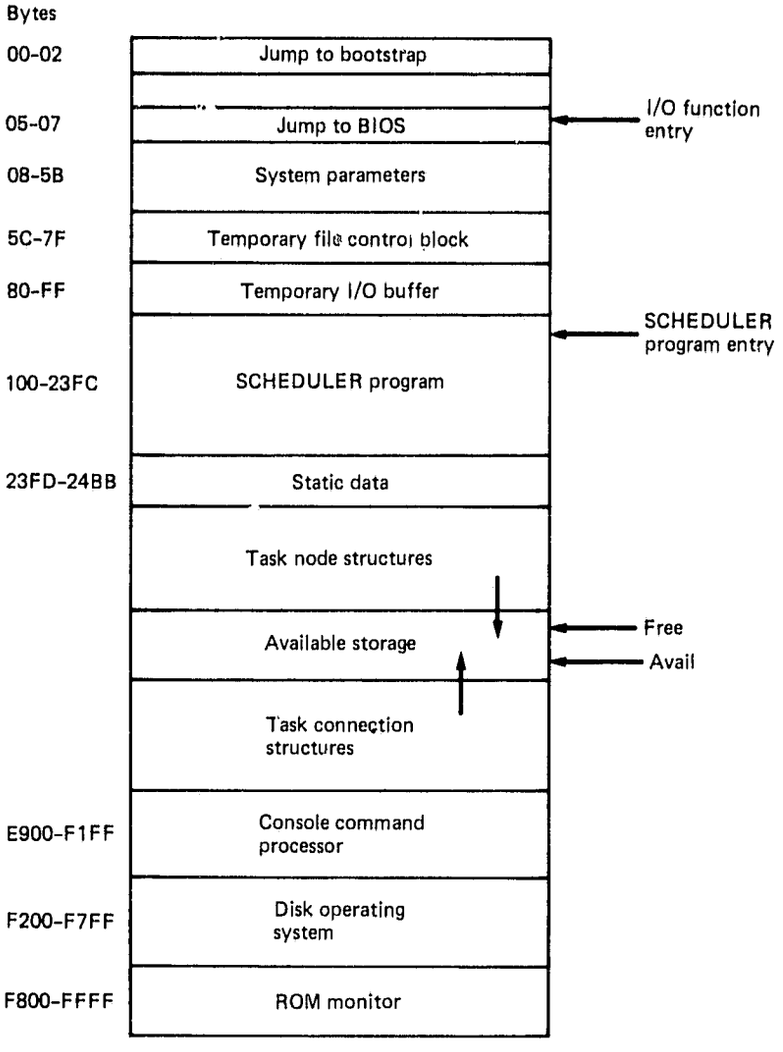


Fig. 7.3-1. SCHEDULER program memory map

3.(2/15/77) DIAGNOSE and ERROR will be stubs, printing only short messages until the main program functions are correct.

4.(2/16/77) SEARCH will be a linear search through task nodes, comparing task codes. This may be a prime subroutine for replacement if it proves too slow. Thus, be careful to structure program so that grabbing a new node for the network (done by SEARCH) is not assumed to be linear.

. . . *and so on.*

A.8 Example of Class B Narrative, Flowchart, and Code

The narrative in Figure A.8-1, below, the corresponding flowchart in Figure A.8-2, and the code they describe (shown in Figure A.8-3) are Class B equivalents to the Class A module descriptions given in Figures B-4 and B-5 of Ref. 2. The reader may note that the flowchart boxes are uncommented in the narrative (due to simplicity of the algorithm) but are commented in the code when that code is not straightforward.

SSD-DOI-5466-SP

Chart 4.4.1
FINDNO
8/7/75
Page 1 of 2

5.2(4.4.1) FINDNO Procedure

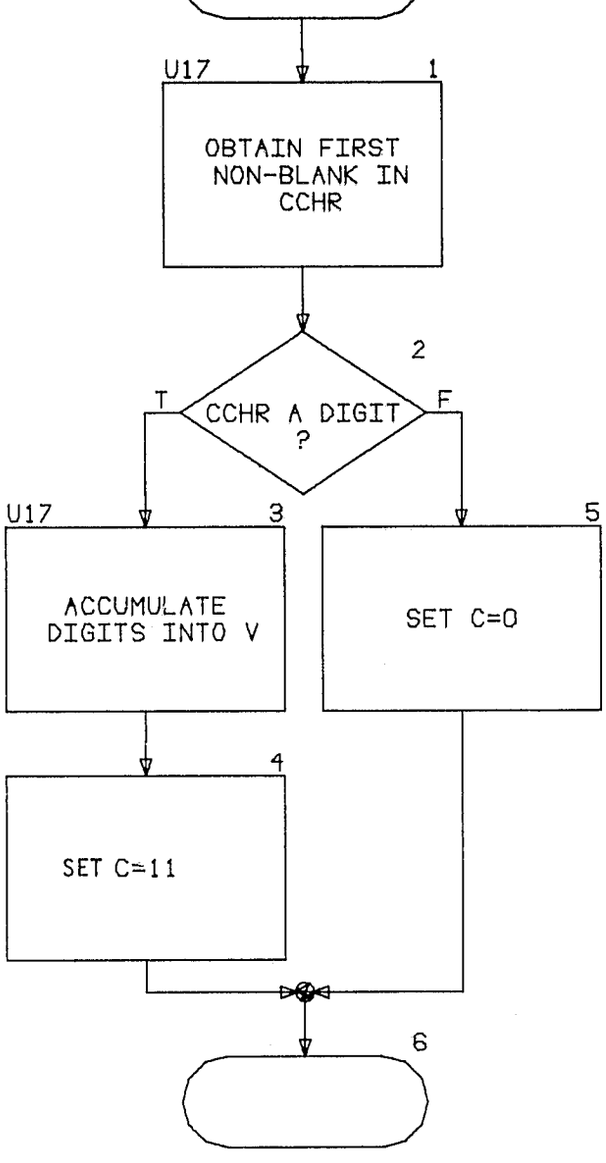
On entry, and input line has been received into an input buffer bia EMSGIN (U14). The LiNe PoinTeR, LNPTR, is positioned so as to extract the characters of the line via calls to GET/U17 (characters are returned in CCHR).

This procedure discards leading blanks, if any, and looks for an MBASICTM statement number. If a statement number is present, the digits are converted to an integer and placed in the Value variable V; the Class variable C is set to 11 to indicate that the first symbol on the line is an integer. If a statement number is not present, C is set to 0, indicating the input line does not have a statement number; V thus retains its entry value. See Table 7.2.2.5 for further symbol class and value definitions. The scan terminates at the first nondigit or when V > 16777215.

On exit, C and V contain the symbol Class and Value values above; the Current CHaRacter variable, CCHR, holds the character which stopped the number scan; and LNPTR points to the next input character to be fetched. A value of V > 999,999 indicates that the statement number is too large.

Fig. A.8-1. Class B narrative for Module 4.4.1 of the MBASIC™* processor which accompanies the flowchart in Figure A.8-2

*A trademark of the California Institute of Technology



D:		
C:		
A:	RBH	8 NOV 78

07 NOV 78

Fig. A.8-2. Flowchart of the FINDNO/4.4.1 module drawn to Class B standards

```
MODCOMP MACRO ASSEMBLY (X)*I  DATE  CCA-DEVELOPMENT...08/21/  PAGE   1
FINDNO.4.4.1                BH 01-20-78

  2                     ************************************************************
  3                     *                                                          *
  4                     *   FINDNO/4.4.1                VERSION: 8/7/75            *
  5                     *                               CODER: RBH                 *
  6                     ************************************************************
  7                           PGM         FINDNO
  8                     *  DEFINE INTERNALS/EXTERNALS
  9                           INT         FINDNO
 10                           EXT         GET
 11                           EXT         RETURN
 12                           EXT         C
 13                           EXT         CCHR
 14                           EXT         V
 15     0000 A 0009  NINE     DEC         9
 16     0001 A 2030  ZRO      DEC         " 0"
 17     0002 A 2020  BLANK    DEC         "  "
 18          R 0003  FINDNO   EQU         $
 19     0003   E780  PT1      BLM.8       GET             .1/U17
        0004 X 0001
 20     0005   E570           LDM.7       CCHR
        0006 X 0004
 21     0007   C770           CRMB.7      BLANK.PT1.PT2
        0008 R 0002
        0009 R 0003
        000A R 000B
 22     000B   E170  PT2      SUM.7       ZRO             .2
        000C R 0001
 23     000D   7670           TBRB.7.0    PT5
        000E R 0013
 24     000F   C770           CRMB.7      NINE.PT3.PT3
        0010 R 0000
        0011 R 0018
        0012 R 0018
 25     0013   6C77  PT5      ZRR.7                       .5
 26     0014   E670           STM.7       C
        0015 X 0003
 27     0016   E700           BRU         PT6
        0017 R 0039
 28     0018   6C66  PT3      ZRR.6                       .3
 29                     *                                ACCUMULATE IN REG 6 & 7. HAS 1ST DIGIT NOW
```

Fig. A.8-3. Coded module corresponding to the flowchart and narrative in Figures A.8-1 and A.8-2

```
MODCOMP MACRO ASSEMBLY (X)*I  DATE  CCA-DEVELOPMENT...08/21/  PAGE   2
FINDNO.4.4.1              BH 01-20-78

 30                       *                                  DOUBLE PRECISION PROD FOR LINE NBRS TO 999999
 31      0019   E780  PT3A   BLM,8        GET                /U17
         001A X 0001
 32      001B   E590         LDM,9        CCHR
         001C X 0004
 33      001D   E190         SUM,9        ZPO
         001E R 0001
 34      001F   7690         TBRB,9,0     PT3C
         0020 R 0033
 35      0021   C790         CRMB,9       NINE,PT3B,PT3B
         0022 R 0000
         0023 R 0027
         0024 R 0027
 36      0025   E700         BRU          PT3C
         0026 R 0033
 37      0027   6D46  PT3B   TRR,4,6                        ACC=8*ACC+ACC+ACC+DIGIT
 38                       *                                  (ACC=10*ACC+DIGIT)
 39      0028   6D57         TRR,5,7
 40      0029   2E63         LAD,6,3                         MULT ACC BY 8
 41      002A   2264         DAR,6,4                         ADD ACC
 42      002B   2264         DAR,6,4                         ADD ACC
 43      002C   6C88         ZRR,8
 44      002D   2268         DAR,6,8                         ADD DIGIT
 45      002E   0846         MBR,4,6                         CHECK OVERFLOW (>16777215 FOR CONVENIENCE)
 46      002F   7D44         TRRB,4,4     PT3C               JUMP IF OVERFLOW
         0030 R 0033
 47      0031   E700         BRU          PT3A
         0032 R 0019
 48      0033   A560  PT3C   SFM,6        V
         0034 X 0005
 49      0035   FD70         LDI,7        11                 ,4
         0036 A 000B
 50      0037   E670         STM,7        C
         0038 X 0003
 51      0039   E700  PT6    BRU          RETURN
         003A X 0002
 52      003B                END
```

Fig. A.8-3 (contd)

A.9 Typical Class B Program Operation Descriptions

The following descriptions of operations are considered Class B. Information within each description may be given simply in narrative form or it may appear on a flowchart (or equivalent), or divided meaningfully between these.

1 Known Algorithms

Let it be assumed in the examples to follow that F(x) is a mathematical function defined adequately elsewhere in the SSD. Then the following are appropriate forms for describing the integration of F(x):

(a) INTEGRATE F(X) FROM 1 to 10 WITH DX = 0.1 USING TRAPEZOIDAL RULE
Note: Narrative or text can be used to contain such qualifying information as:
Flowchart: INTEGRATE F(x)
Narrative: Use the trapezoidal rule; x = 1(0.1)10

(b) INTEGRATE F(X) FROM 1 TO 10 WITH DX = 0.1 USING MODIFIED TRAPEZOIDAL RULE (REF. 1.3.3)
where reference 1.3.3 is:
Handbook of Mathematical Functions, U.S. Dept. of Commerce, National Bureau of Standards, Applied Math Series 55, U.S. Govt. Printing Office, Washington D.C., June 1964, formula 25.4.4, page 885.

(c) INTEGRATE F(X) FROM 1 to 10 WITH DX = 0.1 USING METHOD IN SECTION 7.5.3
where Appendix 7.5 has a subsection 7.5.3 discussing the special method.

2 Custom Algorithms

In cases where the module algorithm is not well known or easily referenceable, then a more explanatory description may be required in order for one to understand the code. The following illustrates Class B descriptions of this sort:

(a) PRINT WBS REPORT IN FORMAT 7.2.5 USING HEADER_DATA, DATE, START_DATE, AND ARRAYS TASK_CODE, TASK_TITLE, DURATION, EARLY_START, EARLY_FINISH, LATE_START, LATE_FINISH, AND FLOAT
where format section 7.2, subsection 5, specifies the desired format.

(b) IF THIS WORK TASK IS ON CRITICAL PATH (PER SEC. 4.3.8)
where the detailed functional specification section 4.3 has a subsection 8 containing the definition of a critical-path work task. This description corresponds to an unstriped decision box (decisions may not be striped).

(c) GET NEXT CHARACTER FROM NAMED SOURCE FILE BUFFER (SKIP MULTIPLE BLANKS, RETURN HEX 1C IF EMPTY)

(d) WAIT UNTIL OPERATOR ENTERS 'GO'

A.10 Equivalent Documentation Forms

The CFG-PDL listings (Figures A.10-1 and A.10-2), HIPO chart, and decision table that follow depict standard formats and typical contents of equivalents of the program specification module documentation shown in example A.8 of this appendix. The CFG-PDL contains both narrative and flow descriptions. However, the HIPO chart describes only functional information and, in this case, requires an accompanying flowchart (Figure A.8-2), a PDL flow segment, or the decision table in Figure A.10-4. Similarly, the decision table requires narrative (Figure A.8-1), a PDL text segment, or HIPO chart (Figure A.10-3).

```
FINDO .. <*01-20-78*> RBH MOD# 4.4.1

REF
PAGE ********************************************************************************************************************
     *                                                                                                                  *
     *  1    .. THIS PROCEDURE LOOKS FOR AN MBASIC STATEMENT NUMBER.                                                    *
     *  2    .. CHARACTERS ARE OBTAINED FORM THE MBASIC INPUT LINE VIA CALLS                                            *
     *  3    .. TO GET/U17, WHICH RETURNS THE CHARACTER IN CCHR (CURRENT                                                *
     *  4    .. CHARACTER).  IF A STATEMENT NUMBER IS FOUND, THE CLASS VARIABLE C                                       *
     *  5    .. IS SET TO 11 (INTEGER IDENTIFIER) AND THE VALUE IS PLACED IN V.                                         *
     *  6    .. IF NO STATEMENT NUMBER IS FOUND, C IS SET TO 0.                                                         *
     *  7    .. ON EXIT C AND V ARE SET, AND CCHR CONTAINS THE CHARACTER WHICH                                          *
     *  8    .. STOPPED THE NUMBER SCAN, A VALUE OF V>999,999 INDICATES THAT THE                                        *
     *  9    .. STATEMENT NUMBER IS TOO LARGE.                                                                          *
     * 10    OBTAIN FIRST NON-BLANK CHARACTER IN CCHR ..1/U17                                                           *
     * 11    IF CCHR IS A DIGIT ..2                                                                                     *
     * 12        DO UNTIL NON-DIGIT OR V>16777216 (OVERFLOW IS ANY NUMBER                                               *
     * 13            >999999. 16777216 IS USED BECAUSE OF EASE OF TESTING                                               *
     * 14            (LEFT BYTE) ) ..3                                                                                  *
     * 15            ACCUMULATE DIGITS INTO V                                                                           *
     * 16        ENDDO                                                                                                  *
     * 17        SET C=11 (INDICATE INTEGER (STATEMENT NUMBER) BOUND) ..4                                               *
     * 18    ELSE                                                                                                       *
     * 19        SET C=0 (INDICATE NO STATEMENT NUMBER FOUND) ..5                                                       *
     * 20    ENDIF                                                                                                      *
     *                                                                                                                  *
     ********************************************************************************************************************
```

Fig. A.10-1. A CFG-PDL flow segment containing narrative, procedure, and comments

```
MODULE DATA INTERFACES

DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
D                                                                                                    D
D 1   CCHR:  CHARACTER FROM MBASIC INPUT LINE.  SET VIA CALL TO GET/U17                              D
D 2   C: CLASS OF IDENTIFIER (SEE TABLE 7.2.2.5)                                                     D
D 3   V: VALUE OF IDENTIFIER (32 BITS) (SEE TABLE 7/2/2/5)                                           D
D                                                                                                    D
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
```

Fig. A.10-2. A CFG-PDL data segment which defines the data interfaces to FINDNO/4.4.1

Decision Table

Description: Algorithm for FINDing the statement number (NO)	Module 4.4.1
	Identifier: FINDNO
Prepared by: R. B. Hartley Phone: 2459	Date: 8/7/75

Conditions	Rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	ELSE	COST
	Prob																
1. Entry?		Y	N	N													
2. Is CCHR a digit? (CCHR=Current CHaRacter)		–	Y	N													
Actions	**Rule**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	ELSE	
1/U17. get nonblank CCHR		1	–	–													
2. test CCHR		2	–	–													
3/U17. accumulate digits in V		–	1	–													
4. set C=11		–	2	–													
5. set C=0		–	–	1													
6. exit		–	3	2													

Sequential Test Procedure

Signature	Date	Signature	Date
RBH	8/7/75		

Fig. A.10-3. HIPO chart form of Figure A.8-1; requires accompanying flowchart, PDL flow segment, or decision table

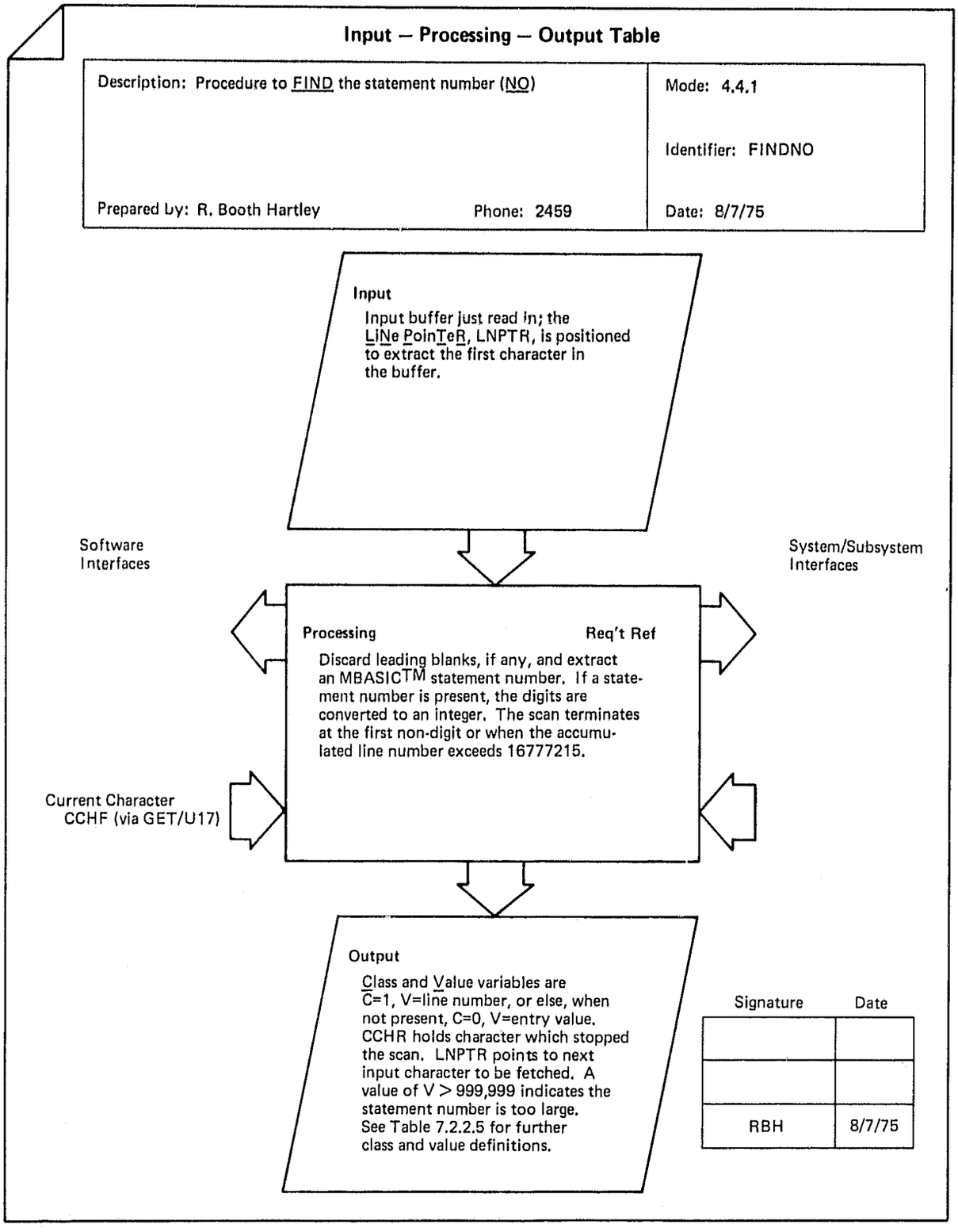


Fig. A.10-4. Decision table form of the FINDNO algorithm; requires narrative, PDL text segment, or HIPO

A.11 Other Useful Documentation Forms

This section contains examples of figures, charts, and formats (Figures A.11-1 through A.11-9), which are often extremely useful in promoting understandability of an SSD. Almost all of these will require some narrative to tell the reader how to interpret the symbols or format. The net effect, however, should be a reduction in the textual material which would otherwise result, had the graphic not been provided, to reach the same level of understanding.

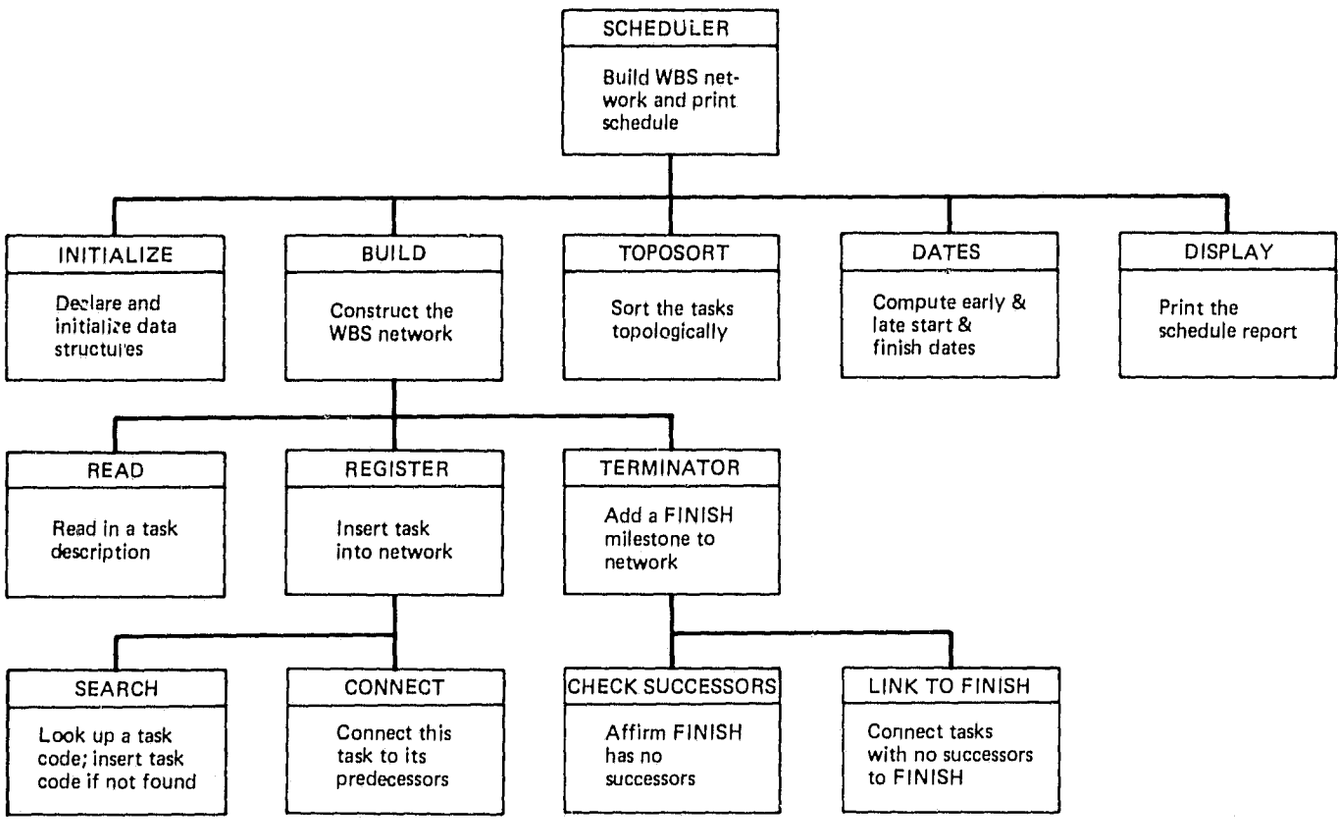


Fig. A.11-1. A partial structure chart, showing modular breakdown of a program into functions (useful for overview description of processing)

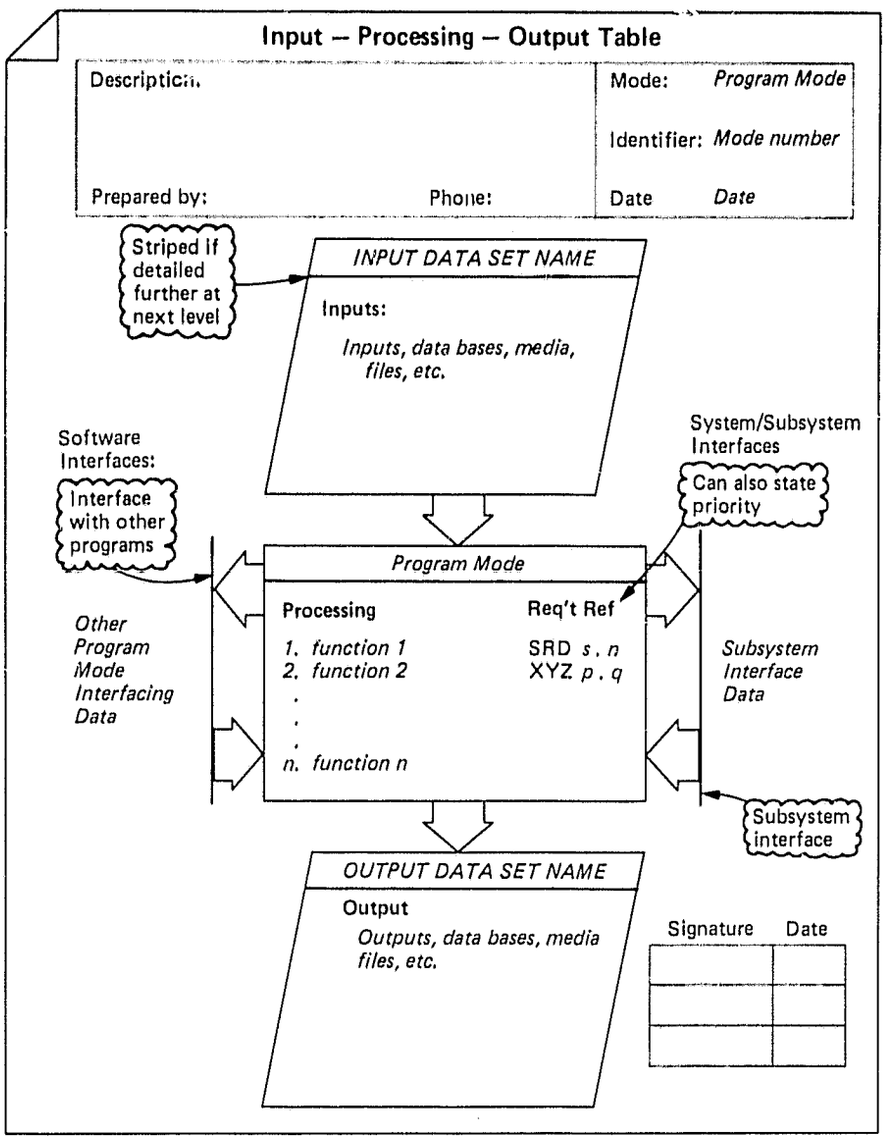


Fig. A.11-2. A useful function description graphic format, showing interfaces to other program modes or other subsystems (use of such graphics is optional; note the use of traceability mechanism employed here to show origin of function requirements)

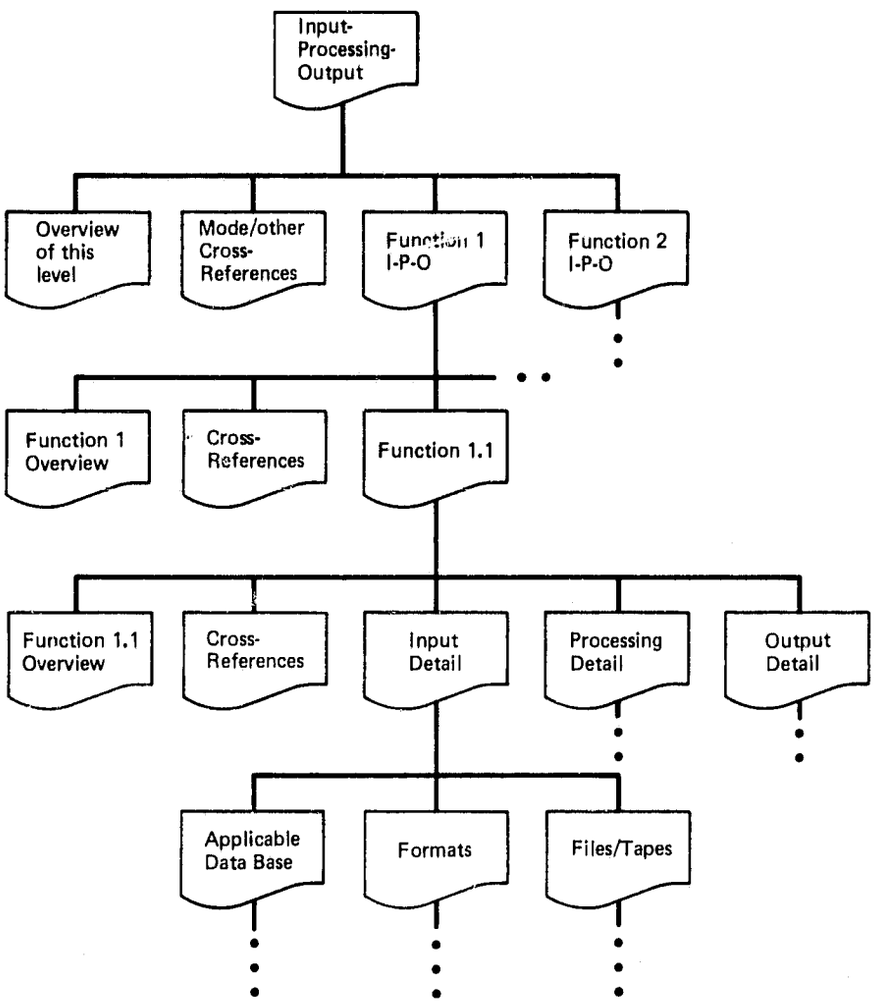


Fig. A.11-3. Suggested input-processing-output description methods for function documentation (two methods of detailing are shown, and others may also be used; cross-references may be useful to connect "cousin" sections of the document)

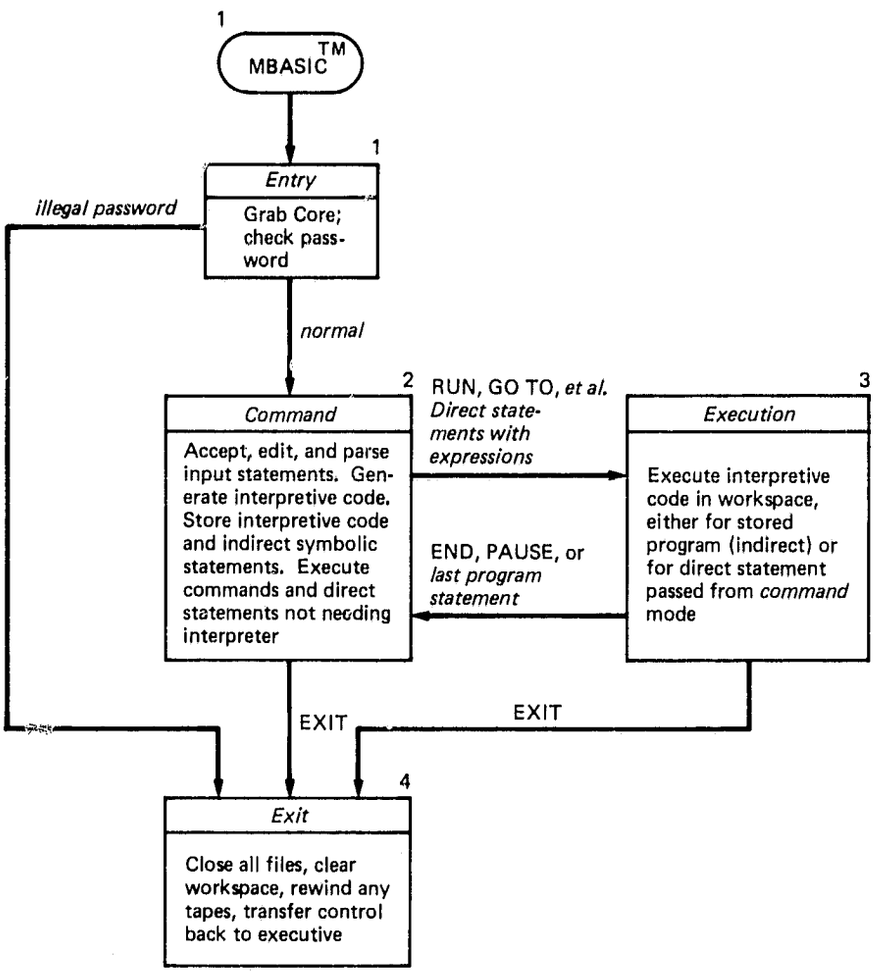
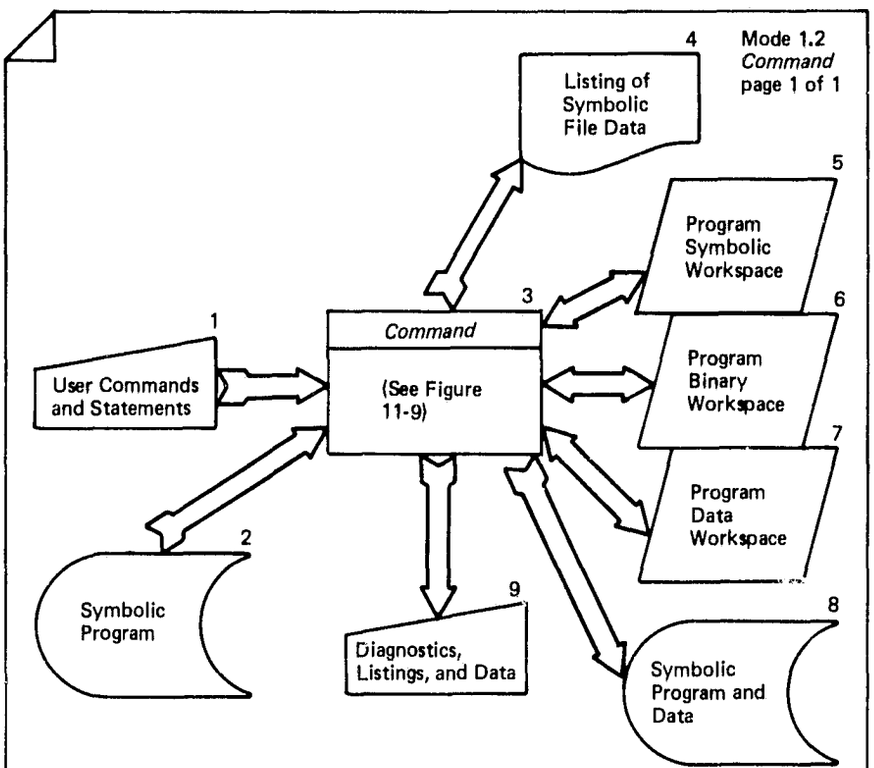


Fig. A.11-4. A simplified mode diagram for the MBASIC™ interactive language processor (boxes show required modes of operation and reasons for transitions between modes as stated in the language specification; this diagram does not show the design of the program into modules)



.1 Primary entry of commands and source program statements in normal mode shall be from user interactive terminal.

.2 Upon commands LOAD and MERGE, symbolic programs shall be taken from standard-format MBASIC or system files. (See also requirement for COPY in language specification [11].)

.3 The command-mode functions are listed in Figure 11-9.

.4 Symbolic files shall be printed on line printer by use of COPY. . .TO HSP.

.5 The current symbolic program shall be held in a form suitable for EDIT, MODIFY, etc.

.6 The current binary (interpretive) code shall be held in a form suitable for execution upon RUN, GO TO, etc.

.7 Data workspace shall hold all current values of variables, or mark that variables have not been initialized.

.8 Symbolic workspace shall be written on a file upon SAVE; symbolic workspace or other files written upon COPY command.

.9 Primary user display medium shall be interactive terminal.

Fig. A.11-5. A brief multi-mode information flow diagram and narrative that lists a few of the top-level requirements for the command mode of an interactive MBASIC™ language processor (later levels refine the separate operational mode requirements within the command mode)

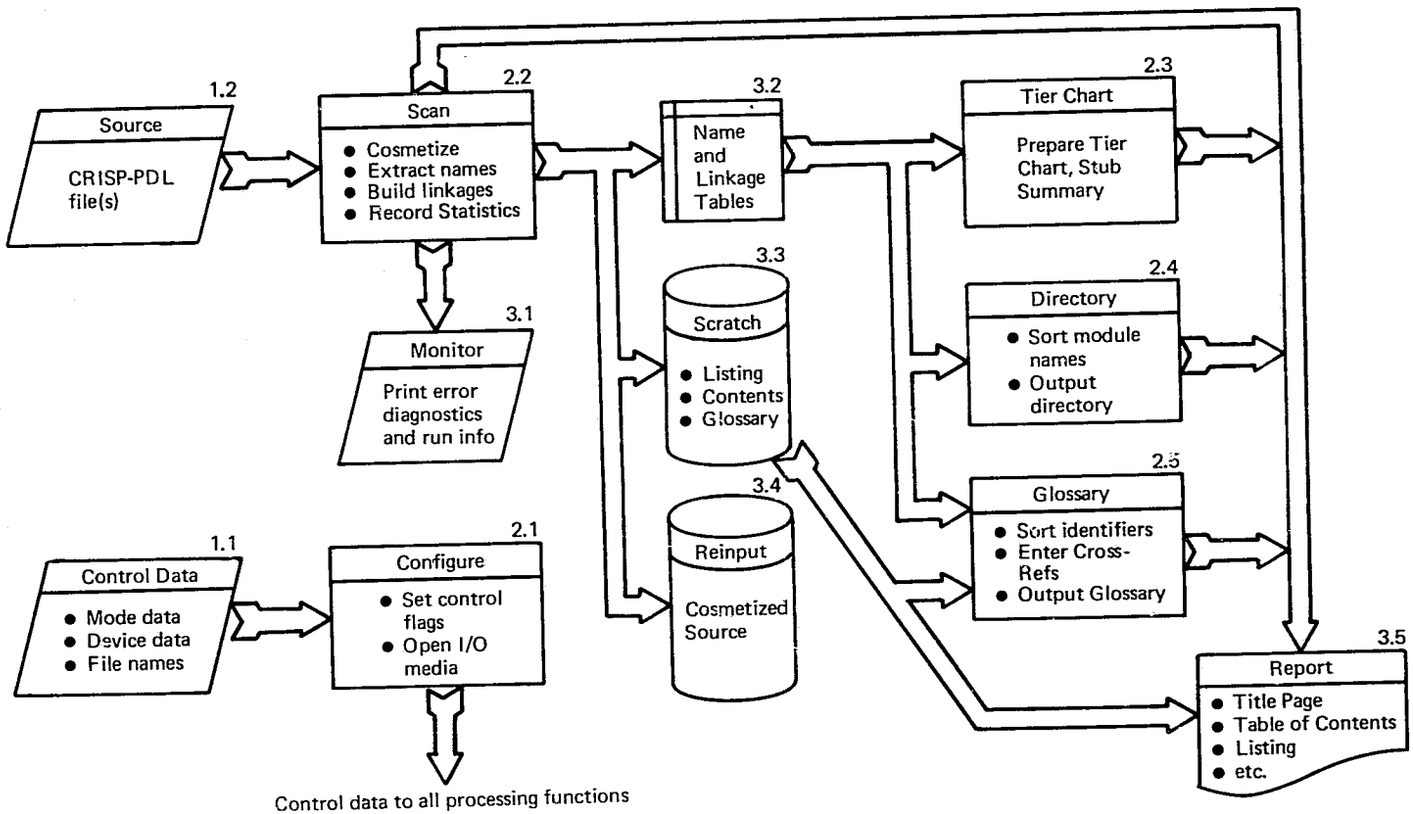


Fig. A.11-6. A data flow diagram showing major components of a CRISP-PDL processor and internal routing of information

SEGNO — SEGment number (NO), flag variable (range: 3-9) first assigned in SYSUP (module 2). Value specifies the configuration currently active, or next to be activated after configuration by USWAP (module U1), as detailed in the table below. SEGNO is active throughout the entire program except for the subprogram EXIT (module 9).

SEGNO = ?	Configure for
3	SYSIZL
4	PARSE
5	RUNIZL
6	RUN
7	BATCHC
8	BATCHR
9	EXIT

Fig. A.11-7. Typical glossary entry of data item

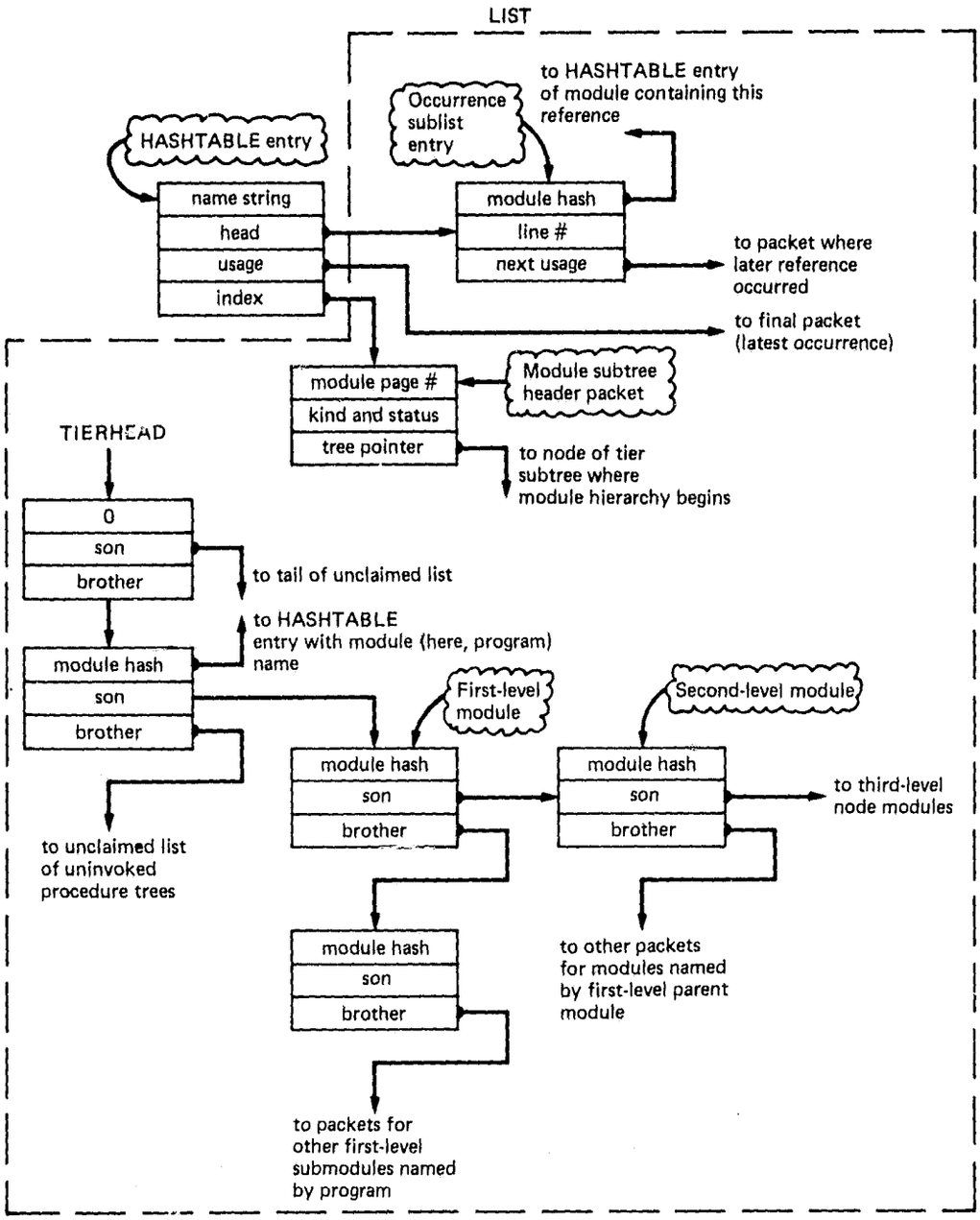


Fig. A.11-8. Example of a data structure layout diagram, showing HASH table and usage cross-reference LIST data structures, including method for representing tier-tree and occurrence-data lists for indexing a main program and its invoked procedures; its subroutines, macros, and functions will have similar tier trees

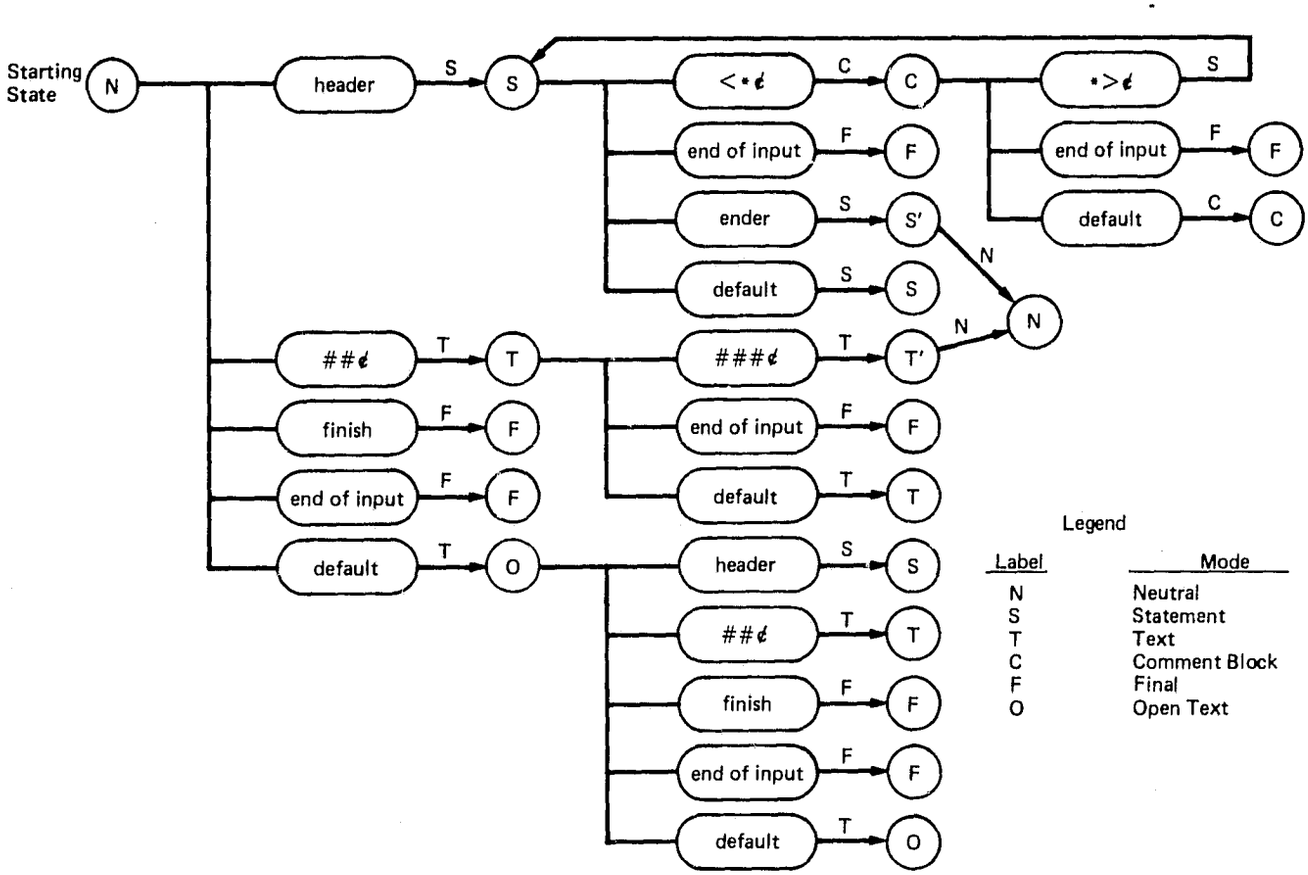


Fig. A.11-9. A finite-state-machine state-transition diagram (transition labels indicate assignments to MODE flag according to legend; entries in ovals are alternative input conditions for state transition

Appendix B
Equivalent Documentation Forms

Besides flowcharts and accompanying narratives, there are other valid techniques to design and display procedures in a two-dimensional structured form and explain what is going on. The Caine-Farber-Gordon PDL (Ref. 6), the JPL SDDL (Ref. 5), CRISPFLOW (Ref. 7), and CRISP-PDL (Ref. 9) are also effective procedure design languages. This appendix provides rules for using these tools as an alternate to flowcharts and narrative in order to qualify as being "equivalent" as required by DSN software standards. The authorized use of alternate forms to flowcharts and narratives is addressed in the SRD/SDD.

Two forms of documentation may be classed as equivalent if a mechanical procedure could conceptually be devised to translate one to the other. For such an equivalence to occur, there must be a one-to-one correspondence among the information items in each, a topological isomorphism among the program structures (routines and data), and the same level of detail (here, Class B) available in each one.

DSN standards require (1) a top-down control-logic structure of routines and Dewey-decimal numbering of routines and steps within routines; (2) a method to describe or rationalize procedural steps apart from the procedural description itself; (3) signature concurrence of design, checking, and approval functionaries; and (4) a configuration control mechanism (e.g., revision number, effective data, or other unique identification of configuration items).

Equivalents to flowcharts and explanatory narratives must, therefore, also display similar traits. The mere use of a tool such as CFG-PDL, SDDL, CRISPFLOW, or CRISP-PDL alone is not sufficient to guarantee equivalence.

Table B-1 is a standard CFG-PDL "deck" that also may be used as a guide for CRISP-PDL and SDDL specifications. This deck contains the entire typical SSD outline of Ref. 2, which can be used (subject to rule 6, below) to produce an entire SSD. In this outline *italicized* items and lines containing the elipsis (. . .) require entry of information concerning the program being described. Other items, not italicized, are to appear as written in the table. Appendix A contains an illustrative example CFG-PDL equivalent to flowchart and narrative.

1. Structure the specification or program algorithms into a hierarchic, top-down syntax using the control language of CFG-PDL, SDDL, or CRISP-PDL superimposed on simple English language constructions.

2. Limit each such specification to one page by inventing named subspecifications for expansion at the next hierarchic level. These named subspecifications are equivalent to "striped modules" on flowcharts.

3. Observe all rules for content level, narrative description, and annotation given in Refs. 1, 2, and this Guide, just as if the descriptions were actually flowcharts and narratives.

4. Use in-line comments to explain steps, if these would normally appear on the flowchart. Use a text segment where flowchart supplementary narrative would normally be supplied. A preamble comment block within a module may be used to document module input, processing, output, constraints, entry assumptions, etc.

5. Use the imperative mood to specify actions taken.

   To promote functional cohesion in the module, strive to use single-verb, single-direct-object statements with modifiers as appropriate to specify input, action, and result, and to indicate source and destination of data. Modifiers should clarify the type and significance of the data and action. Comments or text should be used to clarify the reason or significance of the step.

   For example, rather than merely writing a statement such as "DISPLAY MESSAGE," use the more descriptive form, "QUEUE MESSAGE 'INVALID ENTRY' FOR DISPLAY AT OPERATOR TERMINAL . . INPUT HAS SYNTAX ERROR."

   For relational expressions used in decision statements, state the condition being tested and the data upon which the test is based. Make the outcome branching be clear from the context of the test. Use comments or text to explain the significance of the test.

   For example, rather than the statement, "IF THERE IS A MATCH . . .," use the more descriptive form, "IF CURRENT INPUT CHARACTER MATCHES CURRENT NODE OF LEXICAL GRAPH (INCHAR=LEXCHR(NODPTR)) . . INPUT IS PROPER FOR STATE TRANSITION IN LEXICAL ANALYZER."

6. Use the standard CFG-PDL format given in Table B-1 (or CRISP-PDL or SDDL equivalent).

In cases where SSD material appears divided between this and the SPMC word processor database (a segmented SSD), annotate each segment to show that omitted information resides in the other segment.

Table B-1. Standard CFG-PDL SSD source format

```
%NOSOURCE
%DINDEX
%SINDEX
%TREE
%NOUSCORE
%DATACH _
%TITLE DOC. NO. ssd number REV letter VERSION version number
%TITLE CODE ID NO. 23835
%TITLE SOFTWARE SPECIFICATION DOCUMENT
%TITLE program name
%TITLE subsystem
%TITLE PREPARED BY programmer(s)_____
%TITLE COD DEV ENG cde name_____
%TITLE JET PROPULSION LABORATORY, CALIF. INST. OF TECHNOLOGY, PASADENA, CA
%DATE effective date
%G 1.   INTRODUCTION
%T 1.1  Purpose and Scope of SSD
    . . .
%T 1.2  General Program Description
    . . .
%T 1.3  Applicable Documents
    . . .
%G 2.0  STANDARDS AND CONVENTIONS
    . . .
%G 3.0  ENVIRONMENT AND SYSTEM INTERFACES
%T 3.1  Hardware Configuration and Interfaces
    . . .
%T 3.2  Software Environment and Interfaces
%E 3.2.1  Common Software Routines
    . . .
%E 3.2.2  Special Operating System Services
    . . .
%E 3.     System Integration Specifications
    . . .
```

Table B-1 (contd)

```
%G 4.0  FUNCTIONAL SPECIFICATIONS
%T 4.1  Functional overview
    . . .
%T 4.2  Software Configuration and Modes of Operation
    . . .
%T 4.3  Detailed Functional Specifications (Inputs, Processing, Output)
    . . .
%T 4.4  Data Base Specifications
    . . .
%G 5.0  PROGRAM SPECIFICATIONS
%T 5.1  Design Philosophy, Rationale, Approach, and Organization
    . . .
%T 5.2  Main Program Detailed Design
    . . .
%S module name ..<* design date *> programmer MOD# Dewey decimal
    .. comment section containing
    .. functional description, inputs, outputs, assumptions,
    .. references to functional or algorithmic
    .. descriptions, etc.,
    ..
    procedural description                    .. step number
                .
                .
                .
%T  Text title
    (narrative description of algorithmic steps, rationale, etc.,
    keyed to step number)
%D  Data description
    .. MODULE DATA INTERFACES
    . . .
    .. LOCAL DATA
    . . .
%T 5.3  Subroutine Detailed Designs
    . . .
```

Table B-1 (contd)

```
%E 5.4  External Subroutine Interfaces
    . . .
%D 5.5  Data Structure Definitions
    ..  COMMON DATA
    . . .
    ..  TASK-GLOBAL DATA
    . . .
%T 5.6  Resource Allocation and Access
    . . .
%G 6.0  VERIFICATION AND TEST INFORMATION
%T 6.1  Correctness Test Criteria
    . . .
%T 6.2  Test Specifications
    . . .
%G APPENDICES
%T A.  Glossary
    . . .
%T B.  Formats
    . . .
%T C.  Memory Maps and Overlays
    . . .
%T D.  Decision Log
    . . .
%T E.  Other Tables and Figures
    . . .
%T F.  Source Code Listings
```

Appendix C
Decision Table Standards

Decision tables are another means by which the logical response of a program or subprogram may be described whenever there are multiple factors that determine the response. The following rules set forth a few standards for displaying decision tables.

1. Conform decision tables into the normal format shown in Figure C-1, which also summarizes the remaining rules in this section. The information and arrangement shown in the form is standard; however, the use of the form itself is optional.

2. Limit the size of tables to one page, using hierarchic subtables if necessary.

3. Assign a mnemonic name and Dewey-decimal number to each table.

4. Strive to limit each table to no more than 6 conditions, 12 decision rules, and 15 actions by hierarchic nesting of table entries.

5. Use dashes to indicate immaterial condition entries and ignored actions in a rule. Do not leave blank entries.

6. Use a consistent set of indicators in LEDT condition entries ("Y" or "N"; "T" or "F").

   Use "X" in action entires to indicate single actions or multiple actions where sequence is immaterial. If action-item sequence is material, number such items in the action entry in their order of logical precedence. Assign equal numbers to processes having equal precedence.

7. Number each action item and indicate whether there is further hierarchic development.

   For example, if there is no further detail, merely use the action number *n*; if there is more, but the action is not one common to other charts, then use *n/*; if the action is one common to other charts, described by the cross reference *x*, then write *n/x*.

8. Enter only rules which correspond to true alternatives into the table.

   If the order of rule testing is necessary or pertinent to specify this entry, then state the Sequential Testing Procedure (STP) or give a reference to the procedure elsewhere in the documentation.

9. Unless otherwise stated, control flow of each table will be assumed to merge into a single (proper) exit at the end of the table.

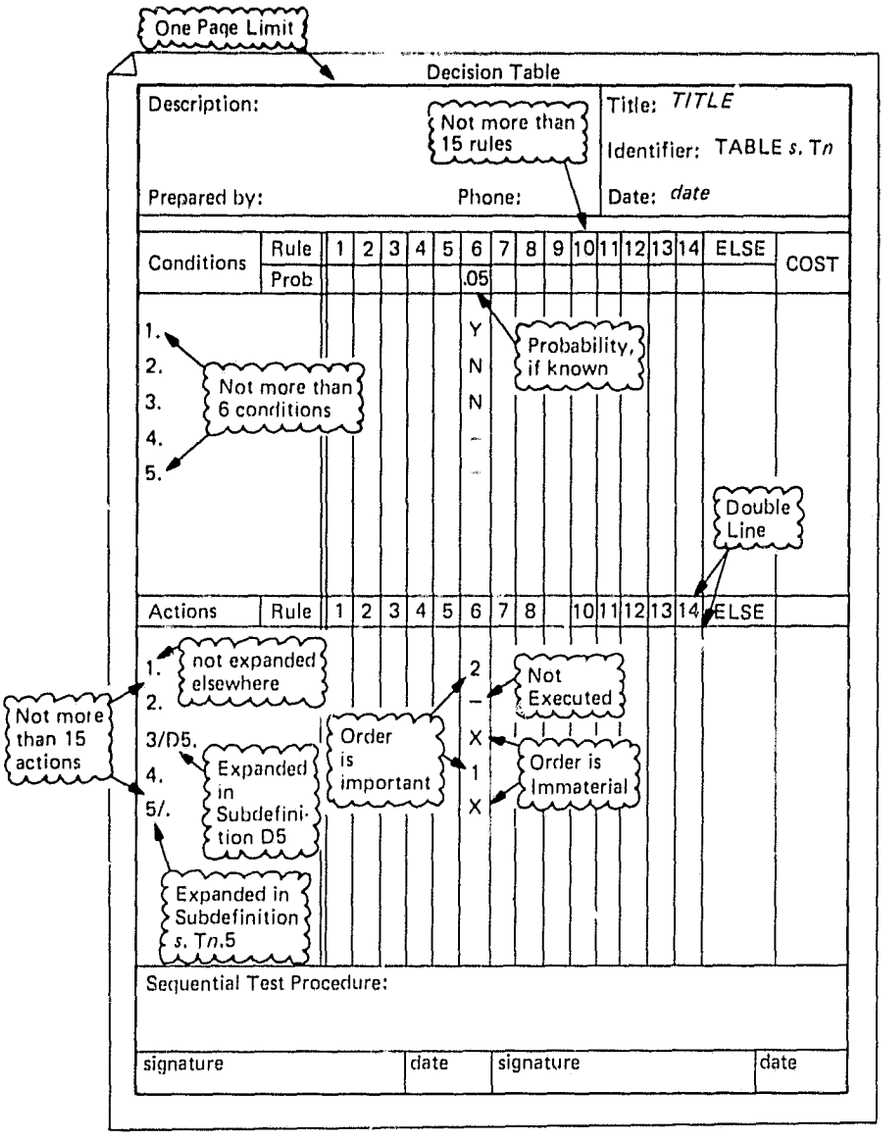


Fig. C-1. Summary of decision table documentation rules

Appendix D
Typical Quality Assurance Audit Criteria

Quality assurance is supposed to be a function that is both effective and economical, in the sense that the expenditure of effort and costs to certify the software represents a justifiable cost savings in the software package life cycle. The general purpose of QA measures is, therefore, to minimize any foreseeable post-transfer operational and sustaining problems by exercising better and tighter controls during implementation than would likely take place if QA measures were not in effect. The QA function is performed by an organization separate from the implementation and operations organizations, so as to obtain an unbiased, dispassionate confirmation that the product has met certain mutually agreed-to requirements for transfer to operations.

A software audit consists of an inspection of the SSD to determine if its parts conform to standards and requirements. The audit is not meant to review the conceptual approach of the design or the efficiency of the code. Rather, it is meant to provide a definite assurance that the SSD is self-consistent, and is in accordance with DSN content and format standards. When inconformities are found, the QA report will explicitly detail the location and type of inconsistency.

This Guide sets forth typical criteria and standards for the audit of Class B DSN software prior to transfer to operations.

1 Definition of QA Audit Discrepancy

In the strict sense, a QA audit reportable item is (1) any inconsistency between parts of the SSD and the operational code, (2) any instance of unwaived nonconformance with standards, (3) any deviation between the SSD and requirements placed on it in the SRD, (4) any omission of information needed to thread functions from the design into the code, and (5) any lack of information needed to understand how the code or design accomplishes its functions.

The technical accuracy and content and the adherence to the standards contained in Refs. 1 and 2 and this Guide are the responsibility of CDE and not QA.

The following items are considered discrepant:

(1) Incomplete SSD

(a) The SSD has not addressed all topics of the standard outline ("not applicable," if appropriate, is considered a response).

(b) Module descriptions and/or code are missing.

(c) Required functions do not appear in the functional specification.

(d) System generation, linkage edit, or disk cataloging code is absent or insufficiently described.

(e) Functional specifications and program design have no connecting threads.

(f) Information needed to thread design module into the code is missing or insufficient.

(g) References are missing where needed.

(2) Inconsistent items

(a) Design or coding errors are apparent.

(b) Control flow logical differences exist between design and code.

(c) Data description or processing differences exist between design and code.

(d) Identifiers and names in the program specification do not match those in the code, and no cross-reference table has been provided.

(e) The code does not agree with the modular structure of the design.

(f) References to figures or to other information within the SSD are inconsistent.

(g) References are incorrect.

(h) Procedural step numbering between design and code is inconsistent.

(3) Inadequate Items

(a) The step number and module numbering are inadequate to locate corresponding design and code.

(b) Module code banners are not provided or are misplaced.

(c) Names are not affixed to each flowchart (or equivalent) and to each code module.

(d) The design and code are not hierarchical, top-down nor structured.

(e) Violation of standards has no accompanying waiver.

(f) Type, units, and scaling of variables and parameters have not been specified.

(g) Incorrect, ambiguous, or unclear functional specifications or program specifications appear.

(h) Programming standards are inadequate to describe conventions used.

(i) Design or code is not understandable from the top down on a module basis.

2 QA Inspection Level

The QA audit of a Class B as-built SSD is oriented primarily toward certifying accuracy, clarity, completeness, relevance, consistency, design/code threadability, and adherence to DSN and special task standards. Such a certification is deemed necessary for the following reasons. Acceptance tests, if based on the functional specification, attest to the end-to-end consistency between the functional specifications and the code, and to the correctness of the code itself (barring undiscovered errors). That correctness is then checked inwardly into the program specification by the QA audit.

QA audit procedures will require that certain comparisons and verifications be made. In each of these, QA personnel are asked to note all errors, exceptions, and discrepancies. The level of QA inspection is typified by the following procedures:

1. Compare the SSD table of contents with written material given in the SSD.

2. Compare the SSD contents with the standard outline given in 810-19. Verify that all of the required SSD topics are addressed.

3. Verify the inclusion of all codes required to build the program, such as system-generation code, linkage-edit code, disk-catalog code, and program source code.

4. Verify the design hierarchy of main program, subprograms, subroutines, macros, etc., from the top down into the code modules for traceability and completeness on a modular scale. Note missing parts, excess parts, incorrect hierarchy, and violations of module numbering and cross-referencing standards.

5. Verify that flowchart boxes (or equivalent) are numbered and that these numbers appear in the code.

6. Compare the logical structure of the flowcharted design with the code logic for each module. (The code will probably be more intricate; however, there should be a correspondence in evidence, in which entry and exit flowlines are identical between design and code.) All decisions in a module design specification must have outcomes determinable without looking deeper into the program hierarchy.

7. Verify that textual descriptions are keyed to the flowchart (or equivalent) by step numbers. Verify that the text given is relevant to the step(s) cited.

8. Verify that comments in the code are relevant to the code to which they are attached.

9. Compare the processing stated in a flowchart box and its narrative accompaniment (or equivalent) with the code for that box; a correspondence should be evident in data operands, operations, etc. The processing should be clearly described. In cases where the code is stated to follow a referenced algorithm, verify that the algorithm function, inputs, and outputs correspond to those provided.

10. Compare data structural descriptions with the data declarations and annotations in the code. Verify a correspondence between descriptions.

11. Compare module names and variable identifiers between design and code. These should match or else correspondences should be defined in a design/code index or glossary.

12. Verify that all references to figures, tables, and other information within the SSD are consistent; verify that references to external documents are valid (document exists, has the correct title, and contains the subject referred to).

13. Verify that each flowchart (or equivalent) and code module have appropriate module names, Dewey decimals, programmer names, and revision number. Each code module should have a correct banner containing this information, properly placed at the module entry point.

14. Verify that units, data type, and scaling appear in the glossary for every variable and parameter.

15. Verify that flowcharts and narrative (or equivalents) conform to style standards in Refs. 1 and 2 and this Guide.

16. Verify that SPMC and special standards and programming conventions have been adhered to.

References

1. *Software Implementation Guidelines and Practices*, DSN Standard Practice 810-13, Aug. 1977 (JPL internal document); also available in *Standard Practices for the Implementation of Computer Software*, JPL Publication 78-53, Chapter 1, edited by A. P. Irvine, Jet Propulsion Laboratory, Pasadena, CA, Sept. 1, 1978.

2. *Preparation of Software Specification Documents*, DSN Standard Practice 810-19, Mar. 1977 (JPL internal document); also available in *Standard Practices for the Implementation of Computer Software*, JPL Publication 78-53, Chapter 4, edited by A. P. Irvine, Jet Propulsion Laboratory, Pasadena, CA, Sept. 1, 1978.

3. *DSN Engineering Documentation Management Plan*, DSN Standard Practice 810-26, Nov. 1976 (JPL internal document).

4. *Standard Classifications of Software Documentation*, Technical Memorandum 33-756, Jet Propulsion Laboratory, Pasadena, CA, Jan. 1976.

5. *SDDL: A Software Design and Documentation Language*, Special Publication (JPL internal publication).

6. *PDL: A Programming Design Language*, Caine, Farber, and Gordon, Inc., Pasadena, CA.

7. *CRISPFLOW User Manual*, SPMC Special Document SOM-DSNSSP-001-B, DSN Data Systems, July 18, 1977 (JPL internal document).

8. "Decision Tables as Programming Aids," in *Standardized Development of Computer Software*, Special Publication 43-29, Part I, Chapter 8, Jet Propulsion Laboratory, Pasadena, CA, July 1976.

9. "CRISP Syntax and Structures," in *Standardized Development of Computer Software*, Special Publication 43-29, Part II, Appendix G, Jet Propulsion Laboratory, Pasadena, CA, Aug. 1978.

NASA—JPL—Coml., L.A., Calif.