

EPAX 8508-0012

EPA

220

1980.1

ADP SYSTEM DOCUMENTATION STANDARDS

U.S. Environmental Protection Agency

Prepared by:

Management Information and Data Systems Division

U.S. Environmental Protection Agency  
Library, Room 2404 PM-211-A  
401 M Street, S.W.  
Washington, DC 20460

February 1980

## TABLE OF CONTENTS

	Page
<u>Section 1 - Introduction</u>	
1.1 Objectives . . . . .	1-1
1.2 Software Development Within EPA . . . . .	1-2
<u>Section 2 - Functional Description</u>	
2.1 Summary . . . . .	2-1
2.2 Table of Contents. . . . .	2-2
2.3 Description. . . . .	2-3
<u>Section 3 - Data Dictionary</u>	
3.1 Summary . . . . .	3-1
3.2 Table of Contents. . . . .	3-2
3.3 Description. . . . .	3-3
<u>Section 4 - User's Guide</u>	
4.1 Summary . . . . .	4-1
4.2 Table of Contents. . . . .	4-2
4.3 Description. . . . .	4-3
<u>Section 5 - Design Document</u>	
5.1 Summary . . . . .	5-1
5.2 Table of Contents. . . . .	5-2
5.3 Description. . . . .	5-3
<u>Section 6 - Implementation and Test Plan</u>	
6.1 Summary. . . . .	6-1
6.2 Table of Contents. . . . .	6-2
6.3 Description. . . . .	6-3
<u>Section 7 - Programmer's Maintenance Manual</u>	
7.1 Summary . . . . .	7-1
7.2 Table of Contents. . . . .	7-2
7.3 Description. . . . .	7-4
<u>Section 8 - Coding Guidelines</u>	
8.1 Introduction . . . . .	8-1
8.2 Choice of Language . . . . .	8-1
8.3 Programming Style. . . . .	8-2
8.3.1 What is Style?. . . . .	8-2

## TABLE OF CONTENTS (continued)

		<u>Page</u>
8.3.2	Good Program Constructs . . . . .	8-3
8.3.3	Consistent Language Usage . . . . .	8-3
8.3.4	Use of Meaningful Mnemonics . . . . .	8-3
8.3.5	Good Commenting . . . . .	8-3
8.4	Program Layout. . . . .	8-4
8.4.1	Prologs . . . . .	8-4
8.4.2	Annotation . . . . .	8-7
8.4.3	Blocking of Statements . . . . .	8-7
8.4.4	Format . . . . .	8-12
8.5	Coding Practices . . . . .	8-12
8.5.1	Efficiency . . . . .	8-12
8.5.2	Naming and Labeling Conventions . . . . .	8-17
8.5.3	Expression and Computational Accuracy . . . . .	8-17
8.5.4	Acceptable Program Structures . . . . .	8-21
8.5.5	Language Usage and Restrictions . . . . .	8-28

### Appendix A - Glossary

### Appendix B - Charting Conventions for System Development

### Appendix C - Bibliography

## List of Illustrations

<u>Figure</u>		<u>Page</u>
1	Prolog (Full Form) . . . . .	8-5
2	Unseparated Code . . . . .	8-8
3	Code Divided Into Blocks and Separated by Headers . . . . .	8-9
4	Code Separated into Blocks . . . . .	8-10
5	Unacceptable Use - Copy Lines of Code . . . . .	8-11
6	Uncommented Poorly Formatted Code . . . . .	8-13
7a,b	Well-Formatted and Commented Code . . . . .	8-14,15
8	Improving Readability . . . . .	8-16
9	Improving Readability . . . . .	8-18
10	Improving Readability . . . . .	8-19
11	Program Structure Types . . . . .	8-22
12a,b	Sequence Structures . . . . .	8-23,24
13	Repetition Structures . . . . .	8-25
14a,b	Case Structures . . . . .	8-26,27

## List of Tables

<u>Table</u>		<u>Page</u>
1	Overview of System Development Phases . . . . .	1-4

## 1. INTRODUCTION

### 1.1 OBJECTIVES

Although the need for documentation has been recognized for many years, it was often felt that annotated program listings and some form of user's guide would satisfy the documentation needs of a program or system of programs. It has recently become evident, however, that such an approach to documentation is so inadequate as to be practically worthless. This is most apparent when maintenance or enhancement is required or when a program or system is transferred to a computer different from that on which it was developed.

There have been many attempts to address this problem by establishing guidelines for documenting programs and systems, and indeed, these guidelines resulted in documentation far superior to previous efforts. They failed, however, in one major aspect - they addressed documentation typically as a final discrete step that was carried out after the subject system or program was completed.

Recent advances in software development methodologies, however, have demonstrated that proper documentation is not only an essential aspect of a system development, but is also an integral part of the development process itself. In light of these advances, the EPA has developed standards (based for the most part on those published by the National Bureau of Standards in FIPS PUB 38) for the key documentation products of Agency software development efforts.

It is the intention of the Agency to include these standards in the EPA ADP Manual either by direct inclusion or by reference. EPA data processing personnel should use these standards when developing information systems in-house, by contract, or by grant. The standards included herein apply to the various system development phases that follow a feasibility study. Each phase has one or more documents associated with it as described below. Standards for each of these documents are presented in subsequent sections in the form of a sample Table of Contents, followed by descriptions of the sections to be included in the document. It is important to note that, although each document is presented section by section, the primary purpose of these standards is to indicate information that should appear in the subject documents. In addition to these standards, program coding guidelines for COBOL, PL/I, and FORTRAN have been included below as a separate section. These guidelines should be used by programmers during the implementation phase. Use of these standards is intended to provide EPA with more effective information systems, and will provide a better basis on which to ensure system viability during computer center changes or upgrades.

## 1.2 SOFTWARE DEVELOPMENT WITHIN EPA

The development and maintenance of software systems within EPA has become a complex activity involving many organizations. Development activities usually span a number of months and the resulting systems may be used in normal production for years. Because of the large investment by the Agency in software, guidelines are essential to ensure that reliable and maintainable software is produced. These guidelines must be flexible enough to be applicable to the wide variety of development activities -- from infrequently used batch reporting systems to high use, on-line interactive systems. In keeping with this, MIDSD has established guidelines for the development of software systems that include the phases described on the following page. Table 1 provides a brief overview of a typical system development effort, including the processes involved and approximate resources required.

- Initiation
- Feasibility  
(Also commonly referred to as a Concept Study, Functional Requirements and Alternatives Analysis, or a Definition Stage)
- System Design  
(Also commonly referred to as Functional Design, General Design, External Design, or System Architecture)
- Program Design  
(Also commonly referred to as Detailed Design, or Internal Design)
- Implementation
- Approval
- Evaluation

The Initiation Phase is begun when a user identifies a problem that requires solving and, together with a system analyst, defines the scope of the problem and the objectives of its solution. The brief written report produced in this phase is commonly referred to as a "System Concept Report."

The purpose of the Feasibility Phase is to define and scope out a particular problem, to define and evaluate various alternative solutions, and to recommend a selected alternative. This work is documented in a Feasibility Study report. Guidelines for preparing and evaluating feasibility study reports are available from MIDSD in a separate document. Please note that, when referring to the "Guidelines for Preparing and Reviewing Feasibility Studies" document, Sections II.7 through II.9, which deal with system design, have been superseded by standards included herein.

Generally speaking, the guidelines presented below complement the feasibility study guidelines, and provide standards for the system development process that would logically begin after completion and management approval of a feasibility study recommending development of an ADP system.

The system development process begins with a need to take the general concepts of the feasibility study and to make them more concrete. This is analagous to an architect's first rendering of a house, and marks the beginning of the System Design Phase for the proposed ADP system. In this phase, the system requirements are documented in a Functional Description, Data Dictionary, and a draft User's Guide. Guidelines for these documents are included herein.

Next comes the Program Design Phase in which the detailed internal structure of the system programs are developed and presented in a Design document. To continue the analogy one more step, this is comparable to the detailed construction drawings for a house. This structure should be developed using an established design methodology (e.g., Jackson, Yourdon, HIPO, etc). The Program Design Phase is completed by the production of a unified Implementation and Test Plan.

The Implementation Phase consists of coding, testing, and demonstrating the system as well as any other steps necessary to produce an operational system (e.g., data conversion, or capturing new data and establishing required manual procedures for the new system). During this phase, the Programmer's Maintenance Manual is produced.

The Approval Phase consists of demonstrating, through separate system demonstrations and/or parallel operations with an old system, that the system operates as expected. The final User's Guide is produced during this phase.

The Evaluation Phase is performed shortly after the system has been placed into full production use. A report should be produced that examines how well the original objectives, budgets, and schedules have been met. Areas that exceed or fall short of expectations should also be included.

Table 1. (1 of 2) Overview of System Development Phases

Phase	Products of Phase	Action at End of Phase	Personnel Involved
Initiation	System Concepts and Objectives Report and Management Plan	Management Decision to Go or No-Go With Feasibility Study	Management Program Analysts System Analysts
Feasibility	Feasibility Study Report	Management Decision to Go or No-Go With System Development	Management Users Program Analysts System Analysts
System Design	Functional Description Data Dictionary Draft User's Guide	System Manager and Users Approve Functional Design and Authorize Further Development	System Manager Users Program Analysts System Analysts
Program Design	Program Design Report Test and Implementation	System Manager and Programming Manager Agree on Design, Plans for Test and Project Control	System Manager Programming Manager Programmers
Implementation (See Note)	Programmer's Maintenance Manual	System Manager Receives a System Ready for Final Test	System Manager Programming Manager Programmers
Approval	Final User's Guide	System Manager and Users Accept System	System Manager Users Programming Manager Programmers
Evaluation	Post-Implementation Audit Report	Management Receives an Evaluation of Project Objectives, Schedules and Budget	Management Program Analysts

Table 1. (2 of 2) Overview of System Development Phases

Phase	Estimated Time and Personnel Resources To Complete					
	Small System			Large System		
	Time (Months)	Work-Months (WM)	%WM	Time (Months)	Work-Months (WM)	%WM
Initiation	1/2 - 1	1	8	2 - 3	6	4
Feasibility	1 - 2	2	17	6 - 9	36	22
System Design	1 - 2	2	17	4 - 6	30	19
Program Design	1 - 2	2	17	3 - 4	24	15
Implementation (See Note)	1 1/2-3	3	25	6 - 9	54	33
Approval	1/2 - 1	1	8	2 - 3	6	4
Evaluation	1/2 - 1	1	8	1 - 2	6	4
Total	6 - 12	12	100	24 - 36	162	100

(Note: The cost of computer time during implementation is commonly estimated as being equal to the implementation personnel cost.)

## 2. FUNCTIONAL DESCRIPTION

The system's Functional Description is produced during the System Design Phase and is used to describe the external characteristics of the subject system. This description provides a basis of understanding between users and designers with respect to the initial problem, system requirements, and development plan.

## FUNCTIONAL DESCRIPTION

### TABLE OF CONTENTS

	<u>Page</u>
<u>Section 1 - Introduction</u> . . . . .	2-3
1.1 Overview . . . . .	2-3
1.2 Environment . . . . .	2-3
<u>Section 2 - System Summary</u> . . . . .	2-4
2.1 Background . . . . .	2-4
2.2 Objectives . . . . .	2-4
2.3 Existing Methods and Procedures . . . . .	2-4
2.4 Proposed Methods and Procedures . . . . .	2-4
2.4.1 Summary of Improvements . . . . .	2-5
2.4.2 Summary of Impacts . . . . .	2-5
2.5 Assumptions and Constraints . . . . .	2-6
2.6 Cost Considerations . . . . .	2-6
<u>Section 3 - Detailed Characteristics</u> . . . . .	2-7
3.1 System Functions . . . . .	2-7
3.2 Performance Characteristics . . . . .	2-7
3.2.1 Accuracy and Validity . . . . .	2-7
3.2.2 Timing . . . . .	2-7
3.2.3 Flexibility . . . . .	2-8
3.3 Inputs . . . . .	2-8
3.4 Outputs . . . . .	2-8
3.5 Data Characteristics . . . . .	2-8
3.6 Failure Contingencies . . . . .	2-8
<u>Section 4 - Operational Environment</u> . . . . .	2-9
4.1 Equipment . . . . .	2-9
4.2 Support Software . . . . .	2-9
4.3 Interfaces . . . . .	2-9
4.4 Security and Privacy . . . . .	2-9
4.5 Controls . . . . .	2-9
<u>Section 5 - Development Plan</u> . . . . .	2-10
<u>Appendixes</u> . . . . .	2-11
<u>Appendix A - Project References</u> . . . . .	2-12
<u>Appendix B - Glossary</u> . . . . .	2-13

## FUNCTIONAL DESCRIPTION

### SECTION 1 - INTRODUCTION

#### 1.1 OVERVIEW

This subsection identifies the agency mission addressed by the system to be developed and summarizes the general nature of the system.

#### 1.2 ENVIRONMENT

This subsection identifies the project sponsor or organization, user, developer (if known), and the computer center or network where the system is to be implemented.

## SECTION 2 - SYSTEM SUMMARY

### 2.1 BACKGROUND

This subsection gives background information about the purpose and uses of the system so as to orient the reader. It explains relationships with other software or procedures.

### 2.2 OBJECTIVES

This subsection contains concise, quantified statements about the major performance requirements of the system. Anticipated operational changes that will affect the system and its use are identified.

### 2.3 EXISTING METHODS AND PROCEDURES

This subsection describes existing methods used to satisfy current requirements. Included is information on:

- Organizational and personnel responsibilities
- Required and available equipment
- Inputs and outputs - their volume and frequency
- Deficiencies/limitations
- Pertinent cost information.

Included also are figures and appropriate narrative to help the reader better understand the existing system. At least one chart should be included which illustrates the overall processing flow for the existing system. It should identify inputs, functions performed, and outputs. Major inadequacies of the current system are described.

### 2.4 PROPOSED METHODS AND PROCEDURES

This subsection describes the proposed system. A chart depicting the proposed data flow and processing should be included with appropriate narrative. Other illustrations should be included as necessary to provide a more thorough understanding of that which is being proposed. Information on the following should also be included:

- Organizational and personnel responsibilities
- Required and available equipment
- Inputs and outputs - their volume and frequency
- Deficiencies/limitations
- Pertinent cost information (developmental as well as operational).

Techniques and procedures borrowed from other systems are identified. Operational functions available to the user are explained.

#### 2.4.1 Summary of Improvements

This subsection gives a summary of the benefits to be obtained from the proposed system. It should include a comparison between deficiencies identified in Section 2.3 and additional capabilities provided by the proposed system. The improvements may be categorized as:

- New capabilities
- Improved existing capabilities
- Timeliness (response time, processing time)
- Elimination of existing deficiencies
- Elimination or reduction of existing capabilities that are no longer needed.

#### 2.4.2 Summary of Impacts

This subsection lists the anticipated impacts and associated costs of the proposed system on the present system as follows:

- Equipment Impacts - Changes to current equipment and buildings.
- Software Impacts - Changes to existing software in order to adapt them to the proposed system. If the proposed system eliminates or degrades any capabilities in the existing system, these capabilities must also be described, as well as the reasons for their elimination or degradation.
- Organizational Impacts - Personnel impacts as a result of implementing the proposed system. Discuss any modifications in positional responsibilities as a result of implementing the new system.
- Operational Impacts - Changes required in both staff and operations center procedures. Discuss impacts on the relationship of the operating center and the user; new data sources; quantity, type, and timeliness of data to be provided; data retention and data retrievals; methods of reporting; and modes of user access. Also include recommended methods for providing input data if these data are not already available.
- Developmental Impacts - User efforts required prior to installation of the system, such as manpower required to develop a data base, and operator and computer time necessary for testing. Also identified are requirements for conversion efforts. Exceptional levels of staffing or computer time required for the parallel operation of the new and existing systems should be discussed.

## 2.5 ASSUMPTIONS AND CONSTRAINTS

Any assumptions and constraints that will affect development and operation of the system will be discussed. Any limitations affecting the desired capability and explicit identification of any desired capabilities that will not be provided by the proposed system are discussed. Examples of assumptions include organizational actions, budget decisions, operational environment and staffing requirements. Examples of constraints include operation environment, operator skill level, human factors (man/machine interface), budget limitations or development schedules.

## 2.6 COST CONSIDERATIONS

This subsection describes resource and cost factors that may influence the development, design, and continued operation of the proposed software. Also discussed are other factors which may determine requirements, such as interfaces with other systems.

### 3 DETAILED CHARACTERISTICS

#### 3.1 SYSTEM FUNCTIONS

This subsection briefly describes all functions and operational capabilities provided by the proposed system. Each is listed with a unique identifying number for easy correlation with individual requirements described in other project references such as the design document and test plan.

#### 3.2 PERFORMANCE CHARACTERISTICS

This subsection describes the specific performance characteristics of the proposed system. Each characteristic should be listed with an identifying number to facilitate referencing from other system documents. The performance characteristics should be organized according to the following categories.

##### 3.2.1 Accuracy and Validity

Briefly describe the accuracy considerations of the proposed system. The following must be considered:

- Accuracy requirements of mathematical calculations
- Accuracy requirements of data
- Data validation requirements.

##### 3.2.2 Timing

Briefly describe the timing characteristics of the proposed system. The following should be considered:

- Throughput time
- Response time to queries and to updates of data files
- Response time of major functions
- Priorities imposed by types of inputs and changes to modes of operation
- Sequencing and interleaving programs and systems (including the requirements for interrupting a program without loss of data).

### 3.2.3 Flexibility

Describe the capability for adapting to changes in requirements, such as:

- Changes in mode of operation
- Operating environment
- Interfaces with other software
- Accuracy and validation timing
- Planned changes or improvements.

### 3.3 INPUTS

This subsection provides examples and explains the various data inputs, specifying medium, format, range of values, accuracy, etc.

### 3.4 OUTPUTS

This subsection provides examples and explains the outputs of the proposed system. Included are quality control outputs that may have been identified, descriptions and examples of hardcopy reports, and display outputs.

### 3.5 DATA CHARACTERISTICS

This subsection discusses the storage of the data elements to be used by the application. It includes information on specific data elements by name and size, if known. It also discusses dictionaries, tables, and reference files, if applicable. It includes estimates of total storage for the data and related components based on a summation of the requirements. It describes the expected growth of the data and related components.

### 3.6 FAILURE CONTINGENCIES

This subsection discusses possible failures of the hardware or software system, the consequences of such failures, and the alternative courses of action that may satisfy the information requirements. It includes, as appropriate:

- Backup - Specify backup techniques. For example, a backup technique for disk would be to periodically copy the disk to a tape.
- Fallback - Explain the fallback techniques, i.e., the use of other means to accomplish some portion of requirements. For example, a fallback technique for an automated system might be manual manipulation and recording of data.
- Recovery and Restart - Discuss recovery and restart techniques, i.e., the capability to resume execution of software from a point at which a problem occurred.

## OPERATIONAL ENVIRONMENT

### 4.1 EQUIPMENT

This subsection describes the equipment capabilities required by the system. Equipment should be described as follows:

- Processor(s), including number of each on/off-line and size of internal storage.
- Storage media, including number of disk units, tape units, etc.
- Input/output devices, including number of each on/off-line
- Communications net, including line speeds.

### 4.2 SUPPORT SOFTWARE

This subsection describes support software with which the computer programs to be developed must interact. Included will be both support software, input and equipment simulators, and test software, if needed. In addition, the language, operating system, and any Data Base Management System (DBMS) used will be identified.

### 4.3 INTERFACES

This subsection describes the interfaces with other systems and subsystems. For each interface, the following shall be specified:

- Operational considerations of data transfer, such as security
- General description of data transfer requirements and procedures to and from the subject system
- Formats and volumes of data to be interchanged.

### 4.4 SECURITY AND PRIVACY

This subsection describes overall security and privacy requirements imposed on the software. If no specific requirements are imposed, a statement to that effect should be made.

### 4.5 CONTROLS

This subsection describes any procedures or authorizations required to access the system.

## SECTION 5 - DEVELOPMENT PLAN

This section discusses the overall management approach to the development and implementation of the proposed system. It includes a list of the documentation to be produced, time frames and milestones for the development of the system, and necessary participation by other organizations to assure successful development.

## APPENDIXES

These sections contain supporting information and lists which for the sake of convenience and readability are left out of the main sections of this document. The two described here (Project References and Glossary) are required. Additional appendixes may be added as appropriate. The order in which the appendixes occur is incidental.

## APPENDIX A - PROJECT REFERENCES

This appendix contains a summary of related pertinent documents or monographs. Examples are:

- Operating system reference manuals
- System processor reference manuals (e.g., Text Editor)
- Requirements documents
- Mission definition documents.

## APPENDIX B - GLOSSARY

The glossary is an alphabetic list of special terms, initials, and acronyms (and their definitions) that are used in this document.

### 3. DATA DICTIONARY

The Data Dictionary is produced during the System Design Phase and is intended to present the various data items required by the system so that they may be readily identified, defined, represented, and structured. It should be noted that the Data Dictionary describes only those data items that enter the system from the outside world or those items that are produced by the system for use outside the system. Data items that exist wholly within the system are of no concern at this time.

DATA DICTIONARY

TABLE OF CONTENTS

	<u>Page</u>
<u>Section 1 - Introduction</u> . . . . .	3-3
<u>Section 2 - Data Dictionary</u> . . . . .	3-4
<u>Appendixes</u> . . . . .	3-9
<u>Appendix A - Group/Category Cross-Reference</u> . . . . .	3-10
<u>Appendix B - Record Cross-Reference</u> . . . . .	3-11
<u>Appendix C - File Cross-Reference</u> . . . . .	3-12
<u>Appendix D - Data Item Cross-Reference</u> . . . . .	3-13

## SECTION 1 - INTRODUCTION

This section describes the content and organization of this document. It also identifies the data base being addressed and describes its organization.

## SECTION 2 - DATA DICTIONARY

This section contains a description for each named data entity in the data base. Each description must not exceed one page in length. There are four general, organizational entities for which description formats are provided. They are the individual data item, the data group or category, the record, and the file. A data item is a single element of data - the smallest unit of data that can be accessed. A data group or category is a number of data items or groups of data items which are grouped together logically but which may be distributed over more than one physical entity of a file or data base. A record is a collection of data items which are physically grouped together and form a constituent entity in a file or in a data base. A file is an organized collection of records.

Some systems allow for the naming of each of these entities and others permit the naming of only a subset of these. Each named entity is described according to one of the formats which follow. They are presented in this section in alphabetic order regardless of entity type. To facilitate rapid reference, the entity name may also appear in the upper right corner of each page.

## DATA ITEM DESCRIPTION

Data Item Name - The name by which the item is most commonly known. In some cases, this name is different from that by which it is referred to in the data base dictionary. The DBMS dictionary name can be mentioned separately or with the aliases mentioned below and flagged in some way to identify it as the DBMS dictionary name.

Alias Names (Synonyms) - A list of other names, in alphabetical order, by which the item is known.

Description - A brief (one- or two-sentence) English language statement describing the meaning of the item.

Type and Length - A designation of the item as being alpha, numeric, alphanumeric, blank, or special character string and the maximum length of the item in number of characters.

Security/Accessibility - Information regarding access to instances of the item in a file or data base.

Source - The source(s) from which the data item is obtained.

Validation - A specification of validation criteria (e.g., a range of values).

Category/Group - The name of a group or set of data items in which the item falls.

File(s) - The name of the file(s) in which the item appears.

Edit Considerations - Information regarding the addition, deletion, and modification of the data item.

### GROUP DESCRIPTION

Group Name - The name by which a group or category of data items is most commonly known. In some cases, this is different from that by which it is referred to in the data base dictionary. The DBMS dictionary name can be mentioned separately or with the aliases mentioned below and flagged in some way to identify it as the DBMS dictionary name.

Alias Names (Synonyms) - A list of other names, in alphabetic order, by which the group is known.

Description - A brief English language statement describing the meaning of the group or category.

Security/Accessibility - Information regarding access to the group of data items in a file or data base.

Source - The source(s) from which items in the group are obtained.

File - The names of files containing data items in the group.

Content - An alphabetic list of data items and/or groups that make up the group.

## RECORD DESCRIPTION

Record Name - The name by which the record is most commonly known.

Alias Names (Synonyms) - A list of other names in alphabetic order by which the record is known.

Description - A brief English language statement describing the record.

Content - An alphabetical list of data items and/or groups contained within the record.

File - The name of the file in which the record occurs.

Source - The source(s) from which data items are obtained for inclusion in the record.

Edit Considerations - Information regarding addition and deletion of the record type and modification of the data within the record.

## FILE DESCRIPTION

File Name - The proper name by which the file is known.

Alias Names (Synonyms) - A list of other names by which the file is known.

Description - A brief English language statement of the purpose of the file.

Security/Accessibility - Information regarding access to and update of the file.

Source - The sources of data for inclusion in the file.

Content - An alphabetic list of named records which make up the file.

Media - Disk, magnetic tape, card, etc.

Organization - Physical organization of the file (i.e., random, sequential, indexed sequential, etc.)

## APPENDIXES

Various cross-references of data entities are presented in the appendixes. They facilitate the identification of related and constituent data entities from the name of a single entity. The cross-references described in the following pages are examples of the types of cross-references that may be developed for a particular system; others may be developed as appropriate.

APPENDIX A - GROUP/CATEGORY CROSS-REFERENCE

This appendix lists in alphabetic order the names of all data groups and gives in alphabetic order the proper names for each data item or subgroup in each group.

## APPENDIX B - RECORD CROSS-REFERENCE

This appendix lists in alphabetic order the names of all records and gives in alphabetic order the names of all data items in each record.

1

APPENDIX C - FILE CROSS-REFERENCE

This appendix lists in alphabetic order the names of all files and gives in alphabetic order the names of all records in each file.

#### APPENDIX D - DATA ITEM CROSS-REFERENCE

This appendix lists in alphabetic order all possible names, proper and alias, for all possible data entities. The proper names have no cross-reference presented. The alias names are cross-referenced to the appropriate proper name.

#### 4. USER'S GUIDE

A draft User's Guide is produced during the System Design Phase and is finalized during the Approval Phase. A prime consideration for producing a draft User's Guide is that it allows the user to determine his/her role in the final system before any serious misconceptions are implemented.

## USER'S GUIDE

### TABLE OF CONTENTS

	<u>Page</u>
<u>Section 1 - Introduction</u> . . . . .	4-3
1.1 Purpose . . . . .	4-3
1.2 User Profile . . . . .	4-3
<u>Section 2 - System Summary</u> . . . . .	4-4
2.1 Overview . . . . .	4-4
2.2 Operational Environment . . . . .	4-4
2.3 System Configuration . . . . .	4-4
<u>Section 3 - System Operation</u> . . . . .	4-5
3.1 Common System Initiation Procedures (if several processes are described) . . . . .	4-5
3.2 Process 1 Description . . . . .	4-5
3.2.1 Security . . . . .	4-5
3.2.2 Initiation Procedures . . . . .	4-5
3.2.3 Operational Procedures . . . . .	4-5
3.2.4 Error and Recovery Procedures . . . . .	4-6
3.2.5 Constraints/Limitations . . . . .	4-6
3.2.6 Data Base . . . . .	4-6
3.2.7 Control Language . . . . .	4-6
3.2.8 Input . . . . .	4-6
3.2.9 Output . . . . .	4-7
3.3 Process 2 Description (each described system process includes the subsections given for Section 3.2)	
<u>Appendixes</u> . . . . .	4-8
<u>Appendix A - Project References</u> . . . . .	4-9
<u>Appendix B - Glossary</u> . . . . .	4-10
<u>Appendix C - Diagnostic Messages</u> . . . . .	4-11
<u>Appendix D - Control Instructions</u> . . . . .	4-12
<u>Appendix E - Input Vocabulary/Query Language</u> . . . . .	4-13

## USER'S GUIDE

### SECTION 1 - INTRODUCTION

#### 1.1 PURPOSE

This subsection briefly describes the purpose of the system being discussed - the reason for its existence. It relates the system to a particular agency mission.

#### 1.2 USER PROFILE

This subsection identifies the organization for which the use of this system is intended. It describes the person(s) for whom this document is intended. It expresses whatever assumptions are made about the users, their capabilities and their level of expertise. It also identifies related systems, programs and/or operational procedures with which the user should be familiar.

## SECTION 2 - SYSTEM SUMMARY

### 2.1 OVERVIEW

This subsection is a high-level description of what the system does. It identifies and summarizes the input and relates the processing that is performed to the output that is produced. Types of input are identified as either required or optional. If a system with several operational capabilities is being described, all such capabilities are briefly described, as are the relationships between them. A system flow diagram is included and the substance of this subsection may consist entirely of a narrative description of that diagram.

### 2.2 OPERATIONAL ENVIRONMENT

This subsection identifies the host computer(s), operating system(s), executive software system, and other major system components with which the system must interface. If it is part of a sequential processing environment, the processing that immediately precedes and/or prepares input for this process is identified, as is the processing which immediately follows.

### 2.3 SYSTEM CONFIGURATION

This subsection briefly relates the computer type and model and the required input and output devices. If the hardware configuration may vary, both ideal and minimal configurations are described. Any support equipment (such as plotters) that may be needed is also included.

## SECTION 3 - SYSTEM OPERATION

### 3.1 COMMON SYSTEM INITIATION PROCEDURES

Procedures that must be followed to initiate system operation will be detailed in this subsection. It may include information such as job request forms, control card formats, or log-on procedures for on-line operations. If these procedures are standard or are detailed in another manual, that manual will be referenced.

For a system with several operational capabilities, this subsection describes preliminary operational procedures that are common to the various system capabilities. If a single program or operational capability is being described in this document, this subsection may be omitted.

### 3.2 PROCESS 1

The function(s) of the first process whose operational characteristics are described in the subsections that immediately follow. Section 3.2 (including 3.2.1 through 3.2.9) will be repeated for each process that is described.

#### 3.2.1 Security

The procedures that should be followed to use input, files, data bases, output, or computer programs to which access is restricted are included.

#### 3.2.2 Initiation Procedures

A step-by-step description of the operational procedures for starting up the program or particular operational capability, including the same type of information discussed in Section 3.1 as it pertains to the program or system capability. Examples of job deck construction, setup information, loading procedures, and instructions for input data preparation would be included.

#### 3.2.3 Operational Procedures

Step-by-step procedures for interfacing with this system capability are presented.

#### 3.2.4 Error and Recovery Procedures

All known anomalous conditions are described with references to the corresponding diagnostics listed in Appendix C. Appropriate recovery procedures for each condition are also described.

#### 3.2.5 Constraints/Limitations

This subsection discusses any restrictions that may be imposed.

#### 3.2.6 Data Base

This subsection describes data base files that are referenced, supported, or kept current by the system. They are referred to in operational or functional terms rather than by program designation. The description includes the type of data in the files, the uses made of the data, and the various keys by which data can be accessed.

#### 3.2.7 Control Language

This subsection describes the control language used to control the initiation and sequencing of runs in either batch or on-line mode and the query language used to reference the data base. The description includes the requirements for, and the preparation of, control cards which may be required by the system or application process. If this information is contained in other support documentation, that source may be referenced. The syntax of a query language is described in detail sufficient for the uninitiated to develop the full range of data base inquiry that is permitted. Summaries of the control language and of the query language may be included as appendixes (see Appendixes D and E) for easy reference.

#### 3.2.8 Input

This subsection discusses all inputs for this process and any interrelationships which may be helpful to its understanding. A clear, detailed description of each valid input is given in separate subsections. Valid input is that set of data which is recognized by the system (although it may contain erroneous fields); it is to be clearly defined. Invalid data is that which is not valid - that is, data which is not recognized by the system and which may either crash the system or produce unpredictable results. Such invalid data should be discussed to whatever extent is both possible and meaningful. Valid data is described in terms of the characteristics listed below. Samples of completed data layout forms and input data formats and sequences will be included. The data characteristics to be included are given below.

- Medium - The physical form of the input
- Length - Maximum number of characters per line or per record
- Content
- Format
- Punctuation - Spacing and use of symbols (e.g., slash, asterick, character combinations) to denote start and end of input, lines, or data groups, etc.
- Sequencing - Order of items in the input
- Labeling - Use of tags or identifiers to denote major data sets
- Combinations - Rules forbidding use of particular characters or combinations of parameters.

### 3.2.9 Output

This subsection describes all nondiagnostic output. Each type of output is described in a separate subsection which contains the following information:

- Purpose - The way in which the output is utilized
- Medium - The physical form of the output
- Format
- Examples
- Security - Distribution and access considerations
- Files - content, organization, record structure
- Data Base - content, key information.

## APPENDIXES

These sections contain supporting information and lists which for the sake of convenience and readability are left out of the main sections of this document. The first three described here (Project References, Glossary, and Diagnostic Messages) are required. Any of the others can be eliminated if not pertinent and additional appendixes may be added as appropriate. The order in which the appendixes occur is incidental.

## APPENDIX A - PROJECT REFERENCES

This appendix contains a summary of related pertinent documents or monographs. Examples are:

- Operating system reference manuals
- System processor reference manuals (e.g., Text Editor)
- Requirements documents
- Mission definition documents.

## APPENDIX B - GLOSSARY

The glossary is an alphabetic list of special terms, initials, and acronyms (and their definitions) that are used in this document.

### APPENDIX C - DIAGNOSTIC MESSAGES

All diagnostic and error messages that are output by the software described in this document are listed in this appendix. Each diagnostic will have a unique identifying reference number by which it can be referred to in other sections of this document.

#### APPENDIX D - CONTROL INSTRUCTIONS

Described in this appendix are specific references of input commands or operational procedures that perform a single function and which might, for the sake of convenience, be better referred to by a single term in Sections 3.1, 3.2.2, or 3.2.3 (Common System Initiation, Initiation, and Operational Procedures).

#### APPENDIX E - INPUT VOCABULARY/QUERY LANGUAGE

This appendix lists the verbs and keywords that are required to interface with the software described in this document. A brief description of the function and/or meaning of each word is included.

## 5. DESIGN DOCUMENT

The Design Document is produced during the Program Design Phase. This document is intended for use by analysts and programmers during the Implementation Phase and contains precise specifications relating to the internal structure of the subject system or program.

## DESIGN DOCUMENT

### TABLE OF CONTENTS

	<u>Page</u>
<u>Section 1 - Introduction</u> . . . . .	5-3
<u>Section 2 - Summary of Requirements</u> . . . . .	5-4
2.1 System Description . . . . .	5-4
2.2 System Requirements . . . . .	5-4
2.2.1 Performance Requirements . . . . .	5-4
2.2.2 Functional Requirements . . . . .	5-4
2.2.3 User Requirements . . . . .	5-4
2.2.4 System Interface Requirements . . . . .	5-4
<u>Section 3 - Environment</u> . . . . .	5-5
3.1 Equipment Environment . . . . .	5-5
3.2 Support Software Environment . . . . .	5-5
3.3 Interfaces . . . . .	5-6
3.4 Security and Privacy . . . . .	5-6
3.5 Controls . . . . .	5-6
<u>Section 4 - Design Characteristics</u> . . . . .	5-7
4.1 Operations . . . . .	5-7
4.2 System Logical Flow . . . . .	5-7
4.3 System Data . . . . .	5-7
4.3.1 Inputs . . . . .	5-7
4.3.2 Outputs . . . . .	5-8
4.3.3 Data Base . . . . .	5-8
<u>Section 5 - Program Specifications</u> . . . . .	5-9
5.1 Subsystem A . . . . .	5-9
5.1.1 Program (Identify) Specification . . . . .	5-9
. . . . .	
. . . . .	
5.1.n Program (Identify) Specification . . . . .	5-9
<u>Appendixes</u> . . . . .	5-10
<u>Appendix A - Project References</u> . . . . .	5-11
<u>Appendix B - Glossary</u> . . . . .	5-12

## DESIGN DOCUMENT

### SECTION 1 - INTRODUCTION

This section summarizes the circumstances and events leading to the development of this design. It identifies the agency mission that is being addressed and the project sponsor or organization. It summarizes the general nature of the system, briefly describing the operational problem that is being solved. Any studies or reports which bear significantly on the design approach are referenced.

## SECTION 2 - SUMMARY OF REQUIREMENTS

This section provides a summary of the system characteristics and requirements. It is an extension of the Functional Description document. Any changes to the characteristics or requirements set forth in the Functional Description must be specifically identified.

### 2.1 SYSTEM DESCRIPTION

This subsection provides a general description of the system to establish a frame of reference for the remainder of the document. Higher order and parallel systems and their documentation will be referenced as required to enhance this general description. Also included will be a chart showing the relationships of the user organizations to the major components of the system and a chart showing the interrelationships of the system components. These charts shall be based on or be updated versions of the charts included in the Functional Description document. The detailed charts included in Section 4 are based on the charts included here.

### 2.2 SYSTEM REQUIREMENTS

This subsection lists the requirements which must be addressed by the system being designed. The requirements must be stated to permit a discernable means of proving that each has been fulfilled. For ease of reference, each requirement should have a unique identifying number. General requirements should be stated and then broken down into appropriate lower levels of detail. A cross-reference table or chart which relates requirements to subsystem, subroutine, module, or component is desirable. The requirements should be categorized. Some possible categories are suggested below.

- 2.2.1 Performance Requirements
  - 2.2.1.1 Timing
  - 2.2.1.2 Accuracy and Validity
  - 2.2.1.3 Flexibility
- 2.2.2 Functional Requirements
  - 2.2.2.1 Function 1
    - 2.2.2.1.1 Subfunction 1
    - 2.2.2.1.2 Subfunction 2
  - 2.2.2.2 Function 2
  - .
  - .
  - 2.2.2.n Function n
- 2.2.3 User Requirements
- 2.2.4 System Interface Requirements

### SECTION 3 - ENVIRONMENT

This section provides an expansion of the environment given in the Functional Description document. Changes in the environment that do not affect the scope of the project as described in the Functional Description and are the result of ongoing analysis and design will be explicitly identified within the appropriate subsections of this section. These changes will be discussed in terms of the impacts on the currently available environmental components (equipment, software, etc.) as well as the impacts on estimates and functions which were based on the original planned environment.

#### 3.1 EQUIPMENT ENVIRONMENT

This subsection provides a description of the equipment required for the operation of the system. Included will be descriptions of the equipment presently available as well as a more detailed discussion of the characteristics of any new equipment necessary. Equipment requirements will be related to the requirements stated in the Functional Description document. A guideline for equipment to be described follows:

1. Processor(s), including number of each on/off-line and size of internal storage
2. Storage media, including number of disk units, tape units, etc.
3. Input/output devices, including number of each on/off-line
4. Communications net, including line speeds.

#### 3.2 SUPPORT SOFTWARE ENVIRONMENT

This subsection provides a description of the support software with which the computer programs to be developed must interact. Included will be both support software and test software, if needed. The correct nomenclature and documentation references of each such software system, subsystem, and program shall be provided. Included must be a reference to the languages (compiler, assembler, program, query, etc.), the operating system, and any Data Base Management System (DBMS) to be used. This description must relate to and expand on the information provided in the Functional Description document. If operation of the computer programs to be developed is dependent upon forthcoming changes to support software, the nature, status, and anticipated availability date of such changes must be identified and discussed.

### 3.3 INTERFACES

This subsection provides a description of the interfaces with other applications programs, including those of other operational capabilities and from other EPA organizations.

### 3.4 SECURITY AND PRIVACY

This subsection describes the overall security and privacy considerations of the system. If none are imposed, a statement to that effect must be made.

### 3.5 CONTROLS

This subsection describes any operational controls imposed on the system and identifies the sources of these controls.

## SECTION 4 - DESIGN CHARACTERISTICS

### 4.1 OPERATIONS

This subsection describes the operational characteristics of the system and the computer centers where the software will be installed.

### 4.2 SYSTEM LOGICAL FLOW

This subsection presents a high-level discussion of inputs, processing performed, and outputs produced. The logical flow of the system is depicted in high-level charts which provide an integrated presentation of system dynamics, entrances and exits, interfaces with other programs, support software, controls, and data flow.

If the scope of this document is the design of a system, this subsection will describe the orderly set of subsystems that make up the system. If the scope of this document is an individual subsystem or program, this subsection relates lower level functions to an orderly set of named routines.

### 4.3 SYSTEM DATA

This subsection describes the inputs, outputs, and data used. Optionally, these may be described with the individual programs to which they relate.

#### 4.3.1 Inputs

Each input will be described as follows:

- Title and tag
- Format and acceptable range of values
- Number of items
- Means of entry and input initiation procedures
- Expected volume and frequency, including special handling (such as queuing and priority handling) for high-density periods
- Priority (e.g., routine, emergency)
- Sources, form at source, and disposition of source documents
- Privacy and security considerations
- Requirements for timeliness.

#### 4.3.2 Outputs

Each output will be described as follows:

- Title and tag
- Format, including headings, line spacing, arrangement, totals, etc.
- Number of items
- Preprinted form requirements
- Means of display (e.g., CRT, printer, typewriter, projector, alarm type, internal)
- Expected volume and frequency, including special handling (such as queuing and priority handling) for high-density periods
- Priority (e.g., routine, emergency)
- Timing requirements (e.g., response time)
- Requirements for accuracy
- User recipients and use of displays, such as notification, trends, or briefings
- Privacy and security considerations.

#### 4.3.3 Data Base

Each data file, table, dictionary, or directory will be described as follows:

- Title and tag
- Description of content
- Estimate of number of records or entries
- Storage, including type of storage, estimates of amount of storage and, if known, beginning and ending addresses
- Privacy and security considerations
- Data retention.

## SECTION 5 - PROGRAM SPECIFICATIONS

This section identifies and describes the individual software units constituting the system, subsystem, or program. If the scope of this document is the design of a system, the program descriptions which follow are grouped according to subsystem. Utility routines are to be grouped as a utility subsystem.

### 5.1 SUBSYSTEM A

This subsection describes the aggregate functions performed by the programs or subroutines that are described in the subsections which follow. It also identifies the requirements listed in Section 2.2 that are addressed by this software. If the scope of this document is the design of a system, this and the following subsections are repeated for each subsystem. Programs which do not fall into the category of any specific subsystem should be grouped under a general category such as Library Routines or Utility Routines.

#### 5.1.1 Program (Identify) Specification

This subsection identifies the functions performed by the program and describes the procedures employed to provide those functions. The types of information to be provided are as follows:

- Calling sequence and argument description
- Inputs/outputs
- Program logic
- Error conditions and their disposition
- Significant design considerations

#### 5.1.n Program (Identify) Specification

A subsection, as described above, is provided for each program (1 through n) within the system.

## APPENDIXES

These sections contain supporting information and lists which for the sake of convenience and readability are left out of the main sections of this document. The two described here (Project References and Glossary) are required. Additional appendixes may be added as appropriate. The order in which the appendixes occur is incidental.

## APPENDIX A - PROJECT REFERENCES

This appendix contains a summary of related pertinent documents or monographs. Examples are:

- Operating system reference manuals
- System processor reference manuals (e.g., Text Editor)
- Requirements documents
- Mission definition documents.

## APPENDIX B - GLOSSARY

The glossary is an alphabetic list of special terms, initials, and acronyms (and their definitions) that are used in this document.

## 6. IMPLEMENTATION AND TEST PLAN

High quality software is easy-to-use, accurate, robust, understandable, simple and efficient. This list may not be complete (it also may not be minimal) but, rough achievement of these characteristics would have far reaching impact on EPA system and mission.

The ease-of-use of a system has to do with the "friendliness" of the user interface. Friendly interfaces insulate the user from many of the vagaries of the computer system. Such interfaces, which are written in user terminology, reduce the number of errors committed by the user and make the process of getting the user's work more reliable. A side benefit is that users have less vested interest in the system and are able to change from computing system to computing system more freely.

The accuracy of a system is the degree to which it satisfies its intended requirements. Not only does an accurate system contain few errors, but its design supports the requirements, and its documentation is also accurate. In other words, it solves the right problem correctly. An accurate system embedded in a good working environment makes for high reliability.

Robustness of a system is the degree to which small changes in the requirements result in small changes in the system. Robust systems are easy to maintain. Robustness is achieved primarily by designing the system as a model of the problem environment as seen through the eyes of the user.

Understandability of a software system is achieved through good documentation, adherence to good coding standards, and programming in well-known languages.

Simplicity is a virtue in its own right but is implied by the other characteristics above.

Efficiency is by far the most abused notion on this list. Consequently it has become fashionable to put all efficiency considerations aside while in reality we want the efficiency we can afford. In other words, we want that amount of efficiency that will minimize (not lessen) the system life cycle cost.

The Implementation and Test Plan addresses the last five of these considerations

## IMPLEMENTATION AND TEST PLAN

### TABLE OF CONTENTS

	<u>Page</u>
1.1 Overview. . . . .	6-3
2.1 Implementation Plan . . . . .	6-3
2.1.1 Hardware Environment. . . . .	6-3
2.1.2 Software Environment. . . . .	6-3
2.1.3 Implementation Strategy . . . . .	6-3
2.1.4 Configuration Control . . . . .	6-3
2.1.5 Change Control. . . . .	6-3
2.1.6 Schedule. . . . .	6-3
2.1.7 Resource Utilization. . . . .	6-3
2.1.8 Reporting . . . . .	6-4
2.2 Test Plan . . . . .	6-4
2.2.1 Testing Overview. . . . .	6-4
2.2.2 Types of Failures . . . . .	6-4
2.2.2.1 System Design Failures. . . . .	6-4
2.2.2.2 Detailed Design Failures. . . . .	6-4
2.2.2.3 Program Failures. . . . .	6-4
2.2.2.4 System Failures . . . . .	6-5
2.2.2.5 Performance Failures. . . . .	6-5

## IMPLEMENTATION AND TEST PLAN

### 1.1 OVERVIEW

The purpose of this document is to describe those implementation and testing efforts designed to result in high-quality software.

### 2.1 IMPLEMENTATION PLAN

#### 2.1.1 Hardware Environment

In this section include a description of the hardware that is available for the implementation of this system.

#### 2.1.2 Software Environment

In this section, describe any software tools that will be used such as pre-processors, library systems, code analyzers, test data generators. Also state language selections and data base management systems.

#### 2.1.3 Implementation Strategy

Describe the implementation strategy, "Will the system be developed 'bottom-up' with lower level components being combined to form higher-level components? Will it be developed 'top-down' with high level components being expanded in terms of lower-level components? Will the system be developed in versions with an ever-increasing set of implemented features or will only the final version be delivered?"

#### 2.1.4 Configuration Control

Describe procedures for ensuring that completed components are "frozen" and protected from future change.

#### 2.1.5 Change Control

Describe change control procedures. These procedures should allow for two kinds of changes. Those that originate from the user in the form of changed requirements and those that originate during the implementation in the form of errors or weaknesses in the detailed design. These procedures should contain provisions for decision-making and record keeping that are associated with these changes.

#### 2.1.6 Schedule

Include in this section a list of all components (modules, programs) of the system as defined by the detailed design in the order that they must be completed. Include resource estimates for both the implementation and testing of each component, including the system testing that goes with each delivered version.

#### 2.1.7 Resource Utilization

Present estimates of computing resources required along with procedures for controlling computing resource usage.

### 2.1.8 Reporting

Progress reporting procedures will be described in this section including chart formats, rules for declaring components complete and reporting frequency. Progress should only be reported in terms of completed activities.

## 2.2 TEST PLAN

### 2.2.1 Testing Overview

The ultimate objective of any testing process during implementation is to attain a certain level of confidence that the subject of the test is performing according to specifications. We reach this objective by designing a set of test procedures that will demonstrate the existence of errors. Note that we do not try to find all errors in a program or system (that being much like trying to catch the last fish in a lake without knowing how many fish there are or how good our bait is); rather, we try to find those errors which are most likely to occur or which would be most costly should they occur.

In light of the above comments, this section should describe the overall testing strategy for this system as well as which kinds of errors are most important to detect for this system.

### 2.2.2 Types of Failures

A system may fail (i.e., not satisfy user requirements) in the following ways:

- The System Design (Functional Description, Draft User Guide, Data Dictionary) does not accurately reflect requirements.
- The Detailed Design is not a faithful elaboration of the System Design. (Frequently referred to as a specification error.)
- A program within the system fails to meet its specifications. This failure can occur in two ways: (1) Wrong operations are performed on program path; (2) The path taken is the wrong one for some data.
- The performance (response time, turn around time, etc.) of the system may not be as specified.
- The system design may not be implementable in the chosen environment.

This section should indicate those types of failures considered applicable and the following subsections should discuss the steps taken to detect these errors.

#### 2.2.2.1 System Design Failures

The conformance of the System Design to the requirements will have been certified by a design review prior to the beginning of Detail Design. This section should discuss the findings of this design review in terms of the robustness and simplicity of this system.

#### 2.2.2.2 Detailed Design Failures

The Detailed Design can best be shown to be a faithful representation of the System Design by a technical review team made up primarily of data processing personnel. This section will describe procedures for performing that review and will dictate the makeup of the team.

#### 2.2.2.3 Program Failures

Of the two kinds of program failures mentioned above, the first kind is best detected through the use of "coverage" tests which ensure that each statement of the program has been executed and the result observed (either explicitly or implicitly). Failures of the second kind are best detected through a formal review of the subject code by disinterested programmers (also ensuring adherence to coding guidelines).

This section should, therefore, describe all applicable test cases (including how to run them, their expected results, and procedures to follow should they fail) and describe the findings of the code review.

#### 2.2.2.4 System Failures

This section will describe procedures for conducting the system test. It will consist of three parts: (1) a review of the user documentation to ensure that it faithfully represents the System Design and the Detailed Design; (2) a re-run of all program tests on the completed system or version; and (3) a set of randomly selected tests to determine if there has been a gross misunderstanding of the problem. All test cases will be developed from user documentation and will be documented prior to running with the expected results. There must be a system test for every delivered version.

#### 2.2.2.5 Performance Failures

This section will describe procedures for doing a volume test to determine timings and to estimate operational costs.

## 7. PROGRAMMER'S MAINTENANCE MANUAL

The Programmer's Maintenance Manual is produced during the Implementation Phase. This document is intended for use by analysts and programmers when performing maintenance or enhancement activities in order to increase their understanding of the subject program. Even when programs are written according to current standards, a well written maintenance manual will invariably decrease the amount of time required to understand both the program and its role within the system.

## PROGRAMMER'S MAINTENANCE MANUAL

### TABLE OF CONTENTS

	<u>Page</u>
<u>Section 1 - Introduction</u> . . . . .	7-4
1.1 Summary Overview . . . . .	7-4
1.2 History . . . . .	7-4
1.3 Environment . . . . .	7-4
1.4 Document Organization . . . . .	7-4
 <u>Section 2 - System Overview</u> . . . . .	 7-5
2.1 Application . . . . .	7-5
2.2 Requirements . . . . .	7-5
2.3 Software Functions . . . . .	7-5
2.4 Program Hierarchy . . . . .	7-5
2.5 Inputs . . . . .	7-5
2.6 Processing . . . . .	7-5
2.7 Outputs . . . . .	7-5
2.8 Interfaces . . . . .	7-6
 <u>Section 3 - Program Descriptions</u> . . . . .	 7-7
3.1 Identification of Program 1 . . . . .	7-7
3.1.1 Software Function . . . . .	7-7
3.1.2 Inputs . . . . .	7-7
3.1.3 Processing . . . . .	7-7
3.1.4 Outputs . . . . .	7-8
3.1.5 Interface Requirements . . . . .	7-8
3.1.6 Data Description . . . . .	7-8
3.1.7 Program Execution Requirements . . . . .	7-8
3.2 Identification of Program 2 . . . . .	7-8
: . . . . .	
: . . . . .	
 <u>Section 4 - System Environment</u> . . . . .	 7-9
4.1 Hardware Configuration . . . . .	7-9
4.2 Software Configuration . . . . .	7-9
4.3 Data Base Requirements . . . . .	7-9

TABLE OF CONTENTS (Cont'd)

	<u>Page</u>
<u>Section 5 - Maintenance Tools and Procedures</u> . . . . .	7-10
5.1 Configuration Control and Management . . . . .	7-10
5.2 Programming Conventions . . . . .	7-10
5.3 Documentation Conventions . . . . .	7-10
5.4 Test Tools and Procedures . . . . .	7-11
5.5 Error Procedures . . . . .	7-11
5.6 Special Maintenance Procedures . . . . .	7-11
 <u>Appendixes</u> . . . . .	 7-12
 <u>Appendix A - Project References</u> . . . . .	 7-13
 <u>Appendix B - Glossary</u> . . . . .	 7-14

## PROGRAMMER'S MAINTENANCE MANUAL

### SECTION 1 - INTRODUCTION

#### 1.1 SUMMARY OVERVIEW

This subsection introduces the reader to the software system described in this manual. The general nature and application of the processing performed by the software to be maintained are presented. Any security or privacy requirements are included in this subsection.

#### 1.2 HISTORY

This subsection presents the history of the development of the software system. All major events are summarized and the applicable reference material is identified and discussed, including all significant source documents. (All references will be identified in Appendix A, Project References.)

#### 1.3 ENVIRONMENT

This subsection identifies the environment of software described in this document. Items such as the following are included:

- Program sponsor
- Software developer
- Computer center or network where software is implemented
- Normal user(s).

#### 1.4 DOCUMENT ORGANIZATION

This subsection presents the purpose of the Program Maintenance Manual, the intended users, and a summary statement on the contents of each section of the manual.

## SECTION 2 - SYSTEM OVERVIEW

### 2.1 APPLICATION

This subsection describes the purpose of the software system and the function it performs. This description is presented in terms of user-oriented requirements as opposed to software functions performed within the system.

### 2.2 REQUIREMENTS

This subsection lists all the user requirements fulfilled by the software system. Requirements are related to programs in order to aid the individual making modifications in relating the program structure to specific requirements (see the following subsections).

### 2.3 SOFTWARE FUNCTIONS

This subsection summarizes all software functions and relates the software functions to the requirements fulfilled by the software system.

### 2.4 PROGRAM HIERARCHY

This subsection presents a top-down structure of the software system and relates each software function to the individual program(s) satisfying the software function.

### 2.5 INPUTS

This subsection lists all software inputs and provides a summary description of the inputs, including examples, formats, or references to other documents for details about the input.

### 2.6 PROCESSING

This subsection summarizes all processing performed by the software system and relates all inputs and outputs to the given process.

### 2.7 OUTPUTS

This subsection lists all software outputs and provides a summary description of the outputs, including examples, formats, or references to other documents for details about the output.

## 2.8 INTERFACES

This subsection describes all interfaces with other software systems, and lists references for additional information about each interface.

## SECTION 3 - PROGRAM DESCRIPTION

### 3.1 IDENTIFICATION OF PROGRAM 1

This subsection identifies the program by name, tag or label, and programming language. A version number of the program is also included as appropriate.

#### 3.1.1 Software Function

This subsection identifies the software function(s) performed by this program. A description of the problem and corresponding solution is given. The solution is given in terms of the methods used within the program.

#### 3.1.2 Inputs

This subsection describes in detail the inputs used by this program and includes:

- All data used by the program during its execution
- Data types and location(s) required to begin program execution
- All entry requirements concerning program initialization

#### 3.1.3 Processing

This subsection describes all processing performed on the input listed in Section 3.1.2. The description of the processing includes:

- Description of each major operation of the program
- Major branching or decision points within the program flow
- Constraints or limitations on the processing
- Program exit requirements for terminating execution
- Transition to the next program element
- Error processing
- Output media and types
- Storage requirement
- Execution time requirements.

#### 3.1.4 Outputs

This subsection describes in detail all outputs produced by this program and includes:

- All data modified by this program during execution
- All additional data generated
- Data types and location(s) resulting from program execution.

#### 3.1.5 Interface Requirements

This subsection describes all interface requirements to execute this program if it is not a stand-alone program, including other programs, special test drivers, and data generators, etc.

#### 3.1.6 Data Description

This subsection describes all data items used within the program, including data arrays, tables, COMMON locations, temporary work areas and buffers, etc. The following will be provided for each data item:

- Tag, label, or symbolic name
- Purpose of data item
- Other programs that use data item
- Logical divisions within data
- Basic data structure.

#### 3.1.7 Program Execution Requirements

This subsection summarizes the operating procedures to run the program, including compiling, loading, operating, terminating, and error handling. Any unique run features not included in the Operations Manual are included in this subsection.

### 3.2 IDENTIFICATION OF PROGRAM 2

A subsection, as described above, is provided for each program within the system.

## SECTION 4 - SYSTEM ENVIRONMENT

### 4.1 HARDWARE CONFIGURATION

This subsection provides a description of the equipment required for operating the system. The hardware requirements are related to each program described in Section 3, and include the following items:

- Internal storage required for processor
- On-line or off-line storage, media, form, and devices
- Input and output devices, both on-line and off-line
- Data communications devices.

A hardware diagram will also be included.

### 4.2 SOFTWARE CONFIGURATION

This subsection provides a description of all support software required for the execution of each program. Areas covered are as follows:

- Operating System - Identification, version or release, and any unusual features used for each program
- Compiler/Assembler - Identification, version or release, and any unusual features used for each program
- Other Software - Identification of any other support software used, including data base management systems, utilities, macros, report generators, etc.

### 4.3 DATA BASE REQUIREMENTS

This subsection provides a description of each data base, file, table, or data dictionary used by each program. The following information will be included:

- Format
- Units of measurement
- Codes
- Range of values

If the information required is extensive, a reference may be used for details about each data base, file, table or data dictionary used by the program.

## SECTION 5 - MAINTENANCE TOOLS AND PROCEDURES

### 5.1 CONFIGURATION CONTROL AND MANAGEMENT

This subsection presents all formal procedures to be followed in making modifications to the system, including all applicable forms and documentation.

### 5.2 PROGRAMMING CONVENTIONS

This subsection summarizes and/or references all applicable programming conventions used in developing the software system. Items included are

- Coding conventions
- Documentation within the code, such as prologs and inline comments
- Design scheme for mnemonic identifiers and labels
- All symbols and conventions used in preparing the code, such as charts, listings, serialization of cards, abbreviations used in statements and remarks, etc.

### 5.3 DOCUMENTATION CONVENTIONS

This subsection will contain, or provide a reference to, the location of the program listings. All documentation conventions used within the listings to describe the code will be explained or reference will be made to the appropriate software standards document. An example will be included if applicable. All programs will have two levels of comments: one in the form of a prolog and the other as inline comments. These levels are summarized below:

- Prolog - The prolog will contain as a minimum (1) the program name, EPA organization responsible for the program, author's name, and name of the system of which the program is part; (2) the relationship of the program to other programs, files, and control cards; (3) a description of all principal symbolic names used in the program, including the meaning of different values of switches, etc.; and (4) a description of the overall logic of the program, including method of solution used.
- Inline comments - The comments, located throughout the program at major sections, shall contain (1) a description of the logic of the section in more detail than given in the system-level comments, including the symbolic names affected; (2) a statement as to which conditions cause the section to be entered; and (3) a statement as to which section or subsections follow the execution of this section.

#### 5.4 TEST TOOLS AND PROCEDURES

This subsection describes all test tools and procedures used to verify the execution of the program at the unit level. This information includes:

- Test procedures and expected results
- A description of required test data, test drivers, simulators, etc., including test data ranges, limits, and error conditions for each
- A description of all test tools available, including special options such as debug statements and dumps, execution counters, timing devices, etc.

#### 5.5 ERROR PROCEDURES

This subsection describes all error conditions, including identification of the error, explanation of the source, and recommended methods to correct the error. Each error will be related to the program(s) in which it can occur.

#### 5.6 SPECIAL MAINTENANCE PROCEDURES

This subsection contains an inventory and description of any special programs used to maintain the system and all related maintenance procedures. Procedures include requirements, processing, and verification necessary to maintain system input/output components, such as data base and system library support.

## APPENDIXES

These sections contain supporting information and lists which for the sake of convenience and readability are left out of the main sections of this document. The two described here (Project References and Glossary) are required. Additional appendixes may be added as appropriate. The order in which the appendixes occur is incidental.

## APPENDIX A - PROJECT REFERENCES

This appendix contains a summary of related pertinent documents or monographs. Examples are:

- Operating system reference manuals
- System processor reference manuals (e.g., Text Editor)
- Requirements documents
- Mission definition documents.

## APPENDIX B - GLOSSARY

The glossary is an alphabetic list of special terms, initials, and acronyms (and their definitions) that are used in this document.

## 8. CODING GUIDELINES

### 8.1 INTRODUCTION

The source code is one of the most important forms of documentation for any system or program; the quality of that code can depend, to a large extent, on the time and effort spent during the preceding phases of system or program development. Since other types of documentation used for software maintenance (such as programmer's manuals) are frequently out of date within six months of their release, the source code, which is usually current, can enjoy undue prominence. The relative excellence, however, of important coding characteristics - such as efficiency; freedom from errors; and the degree of difficulty posed by maintenance, modification, and/or enhancement - can be traced, at least in part, to earlier stages in software development, and in particular to the design stages. In turn, good documentation of the earlier phases provides a valuable historical record of the software design that may facilitate the debugging, maintenance, modification, or enhancement of the program or system. Each successive phase of software development should be thought of as part of an ongoing process contingent upon the preceding phases, and not as a discrete set of problems, choices, and decisions confronted and resolved without regard to the previous and subsequent phases. In short, careful development and review of software design should result in source code that is efficient, free of errors, easy to maintain, and flexible enough to be compatible with any future modifications or enhancements.

The principal beneficiaries of software documentation are the software maintenance personnel. The maintenance programmer relies heavily on the source code for the most accurate, detailed information about the software. Consequently, it is not sufficient for the source code listings to be simply a list of the source statements. If all necessary information is in the code itself and if the code is readable, maintenance and modification become much easier. A well-written, intelligible, and readable program is usually more reliable and free of errors. The gross inefficiencies and errors inherent in sloppy, confused logic are eliminated. The logical structure of a readable program is apparent, which facilitates the recognition of inefficient code and the detection of errors.

This standard is intended to provide suggestions and guidelines for developing source code of good quality that will serve as an effective documentation for

### 8.2 CHOICE OF LANGUAGE

The selection of a source language is usually either made by the system designer or imposed by the user. Whoever makes the selection should choose the language that is best able to express the functions of the problem in terms which lead to efficient machine execution. The greatest influence on machine execution is program design. If an efficient design has been formulated, the language used for coding is usually of secondary importance.

High-level languages are preferred over assembly language code (ALC). Programs written in high-level languages are generally easier to debug, maintain, and modify. They contain logical and control structures that facilitate programming. With respect to source code as a documentation form, they are more readable. The current tools available for analyzing and optimizing source code have relegated the use of assembly languages to extreme situations. Assembly language should only be used for utilities which provide operating system or hardware interfaces that are unavailable in high-level languages.

### 8.3 PROGRAMMING STYLE

#### 8.3.1 What is Style?

Style is typically defined as a characteristic or distinctive form of expressing thought in writing or speaking which is often considered exemplary in nature. This definition holds equally well for programming - the coding phase of a project should be nothing more than writing a previously defined algorithm in some programming language.

The most important result of good style is program readability. Since most production programs are read by considerably more people than write them, program readability and understandability are of primary importance. Program readability also becomes of paramount importance when one considers that program source listings are usually the best technical documentation available to the maintenance programmer.

Although programming style will vary from programmer to programmer, the basic components of all good styles are:

- Good program constructs
- Consistent language usage
- Use of meaningful mnemonics
- Good comments.

It is considered good programming practice to use the simplest, most straightforward coding possible, even if this is not the most efficient coding from a hardware point of view.

If efficiency becomes a major concern, a program evaluator should be used after the program is debugged and working. The relevant areas of code should then be optimized, comparing the results with the original, working version. The use of short cuts or reliance on the side effects of instructions should be avoided. Similarly, clever or tricky coding is shunned by these guidelines because it obscures processing, causes problems in maintenance, and can create hard-to-detect side effects. Writing a simple, straightforward, understandable program is an intellectual and creative challenge. Although initially more difficult than the alternative, it will result in easier debugging and maintenance and greater reliability.

### 8.3.2 Good Program Constructs

Good program constructs are largely a result of sound initial design and consist of graphically presenting a program in a form similar to that of the design. This includes effective use of modularization and appropriate indenting of source code. For example, at the beginning of a program,

```
PERFORM INITIALIZE.  
PERFORM PROCESS-RECS UNTIL END-OF-INPUT.  
PERFORM CLEANUP.
```

is far more readable than

```
OPEN INPUT FILE-IN.  
OPEN OUTPUT FILE-OUT.  
MOVE 0 TO RECS-IN RECS-OUT.  
LOOP.  READ FILE-IN AT END GO TO DONE.  
      .  
      .  record processing  
      .  
      GO TO LOOP.  
DONE.
```

Acceptable program constructs are discussed further in Section 8.5.4.

### 8.3.3 Consistent Language Usage

Consistent language usage calls for coding similar tasks in a similar manner. Doing so not only eases the burden of the maintenance programmer, but also improves the reader's ability to understand the intent of a piece of code. For example, the use of binary integers for all counters underscores the use of these items as counters better than if some were character strings. Use of dissimilar approaches to similar problems in the same program is frequently an indication that the original programmer may not have fully understood what he was doing.

### 8.3.4 Use of Meaningful Mnemonics

Use of meaningful mnemonics for item names cannot be overemphasized. It is sufficient to note that INPUT-REC-COUNT is far superior to CNT1 and that PROCESS-ERROR is clearer than OOPS.

### 8.3.5 Good Commenting

Good commenting is absolutely necessary if the time spent by the maintenance programmer in understanding a program is to be minimized. Program comments (in particular the Prolog) should correctly state the intent of a block of code, thereby simplifying the debugging process if errors exist in the code.

Good comments describe what is being done in a block of code - not how a block of code works. If the programmer feels it necessary to describe how a block of code works, the code in question is unnecessarily complicated, long, and/or obscure.

Good commenting implies quality, not quantity, of comments. Comments on every line of code detract from program readability.

#### 8.4 PROGRAM LAYOUT

##### 8.4.1 Prologs

A prolog is a block of comments that precedes a unit of source code. It contains essential information for debugging, maintenance, and modification and greatly facilitates these activities. Having this information easily accessible in the source code is much more useful than having it scattered among isolated notes, manuals, and other individuals. Writing a prolog before starting to code helps clarify program requirements for the programmer. When programming is finished, the prolog must be checked for accuracy, since it is one form of the final documentation of the program. Because other programmers will use the prolog for interface information, it should be checked and updated each time the routine is compiled.

A full prolog, as illustrated in Figure 1, should be included for every software unit capable of being separately compiled. An abbreviated form can be used for internal subroutines. An outline of a prolog is presented below. A blank line (beginning with the comment character) separates each item. The items in a prolog are as follows:

1. IDENTIFICATION LINE - This line indicates the name of the program to which the routine belongs and possibly an index or version number.
2. ENGLISH NAME - The English name of the routine should be indicated.
3. LANGUAGE USED - The computer model, compiler or assembler version (number), and operating system version or level number should be noted.
4. PURPOSE OF ROUTINE - Describes the net effect of one execution of the code, not an explanation of how the routine accomplishes its effect. A routine should have one purpose.
5. CALLING SEQUENCE - Formal arguments or other linkage should be described.
6. NOTES - These include restrictions, requirements, and references. A description of access conditions or execution environment should be provided.

```

SUBROUTINE TAPTYP(EQCOD, MODSET, TRKS, DENSTY)
C*****SUBR: TAPTYP  VERSION: 2  SAT: SMS-GOES  PROGRAM: VISSR *****
C
C  ENGLISH NAME: TAPE ASSIGNMENT TYPE
C
C  LANGUAGE:  UNIVAC 1108 FORTRAN V VERSION E3.  EXEC8 LEVEL 33.
C
C  PURPOSE: DETERMINE DENSITY AND NUMBER OF TRACKS FOR A MAGNETIC TAPE
C           ASSIGNMENT.
C
C  CALLING SEQUENCE:
C           ARGUMENT I/O  DESCRIPTION
C           EQCOD    I    SIX-BIT EQUIPMENT CODE FROM FITEM PACKET
C           MODSET   I    EIGHTEEN LOW ORDER BITS CONTAIN THE MODE SETTINGS
C                       FROM THE FITEM PACKET
C           TRKS     O    NUMBER OF TAPE RECORDING TRACKS, BINARY
C           DENSITY  O    TAPE RECORDING DENSITY (FRAMES PER INCH, BINARY)
C
C  NOTES: 1. VALID RESULTS ARE ONLY RETURNED FOR UNISERVO 12/16/20
C           TAPE ASSIGNMENTS.
C           2. CALLING PROGRAM OBTAINS FITEM PACKET FROM SUBROUTINE UFITEM.
C           3. SEE PP. 7-8, 7-9 OF 1108 P.R.M. ON FITEM PACKET FOR
C           DESCRIPTION OF EQUIPMENT CODES, MODE SETTINGS.
C
C  VARIABLES: (LOCAL)
C           DMASK = BIT MASK FOR DETERMINING DENSITY.
C
C  ERROR HANDLING: IF 'EQCOD' DOES NOT INDICATE A UNISERVO 12/16/20
C                   TAPE UNIT THEN 'TRKS' AND 'DENSTY' ARE SET TO ZERO.
C
C  CALLS: NONE.
C
C  CALLED BY: TAPHND.
C
C  METHOD: TAPTYP USES THE EQUIPMENT CODE TO DISTINGUISH BETWEEN
C         7 AND 9 TRACK UNITS.  USING BIT MASK DMASK, THE DENSITY IS THEN
C         EXTRACTED FROM THE MODE SETTINGS.
C
C  PROGRAMMER: S. SANDERS, CSC, NOVEMBER 1975.
C
C  MODIFIED: R. BIERI, CSC, JULY 1977.  ERROR HANDLING CHANGED.
C
C*****

```

Figure 1. Prolog (Full Form)

7. **VARIABLES** - Information should include descriptions of variable functions and indications of which variables in COMMON blocks are changed, which variables are local, and which are input and output. File usage may also be described. Calling arguments are described under CALLING SEQUENCE (item 5, above). Some items to be included are:

- a. Itemized lists and descriptions
- b. Summaries (if a whole COMMON block is used, the block should be identified)
- c. Lists of exceptions
- d. References to the source code location of the information (some COMMON block listings include detailed variable descriptions; local variables might be flagged in the code).

Another programmer must be able to easily determine the function of each variable used in the routine.

8. **ERROR HANDLING** - The types of errors that are detected and how they are handled should be noted. If no checking is performed, "None" should be indicated.
9. **CALLS** - Required subroutines (or entry points to large utilities) should be listed. If no subroutines are required, "None" should be indicated.
10. **CALLED BY** - Routines which call this routine should be listed. Most routines are called by only one or two other routines. If this is a utility (called by many other routines), "Utility" should be indicated.
11. **METHOD** - The specific software operations that are used to accomplish the purpose of the routine should be indicated. A short general description should be given. The method description, along with the comments in the code, should allow another programmer to easily understand the processing algorithm. Program design language (PDL) could be substituted for narrative.
12. **PROGRAMMER AND DATE** - The programmer or analyst, location, and date (month/year) should be noted.
13. **MODIFIED** - This section, used to list important changes, should include references to change control procedures and the programmer's name.

A bound set of prologs can make up most of the detailed design and program documentation for a system. By putting this information in the code where it is most useful, much of the detailed material in formally published documents can be eliminated. Documentation costs are substantially reduced, and the published documentation, which contains only high-level descriptions, is less subject to change.

#### 8.4.2 Annotation

Annotating code is not a simple process. The objective is to give enough information so that another programmer can understand the routine and at the same time avoid unnecessary comments. The nature of high-level languages is that they are descriptive of the very processes they perform. However, they are frequently inadequate to describe what is accomplished by the process. It is in this context that comments can be used most effectively - to supply meaning to source code when the source language falls short.

#### 8.4.3 Blocking of Statements

High-level languages, while being more readable than assembly language, are not so readable as to be self-documenting.

However, the technique of blocking source code statements can improve the readability of high-level languages (see Figures 2, 3, and 4) by approximating paragraphs. A block is a group of related statements which perform a specific function. Within these guidelines, subroutines must be divided into blocks of related statements. Examples of large blocks would be initialization, input/output, error processing, or alternative processing sections. Smaller blocks within these sections might be loops, a few arithmetic statements which make a single computation, or a conditional statement with its alternative groups of instructions. If these groups are large enough, they might become smaller blocks within the conditional block. Examples of blocks are presented in Figures 3 and 4.

Monolithic program listings do not group statements into blocks and do not set the blocks apart with blank comment lines, indentation, or other techniques described in this section. Such undivided listings are difficult to read (see Figure 2). The smallest blocks in a routine should be set off with a blank comment line before and after; larger blocks can be set off with a number of blank lines and prominent comments or headers (see Figure 3).

A block of any size should have only one purpose. A few lines of code should be duplicated (copied in line where they will be sequentially executed) if this will bring together all statements related to one function. The use of GO TOs or jumps to pick up several isolated statements (to avoid copying them in line) leads to logic errors and creates confusion (see Figure 5).

A block should have only one entry point (at the top) and one exit point (at the bottom), with the exception of error and loop exits.

```

IF PWS-NORMAL-CAPACITY = '$
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE Ø TO BLD-C186
ELSE IF PWS-NORMAL-CAPACITY NOT = SPACES
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE PWS-NORMAL-CAPACITY TO BLD-C186;
IF PWS-DESIGN-CAPACITY = '$
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE Ø TO BLD-C187
ELSE IF PWS-DESIGN-CAPACITY NOT = SPACES
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE PWS-DESIGN-CAPACITY TO BLD-C187.
IF PWS-EMER-POWER = $
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE Ø TO BLD-C188
ELSE IF PWS-EMER-POWER NOT = SPACES
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE PWS-EMER-POWER TO BLD-C188.
IF PWS-S TOR-CAPACITY = '$
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE Ø TO BLD-C189
ELSE IF PWS-S TOR-CAPACITY NOT = SPACES
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE PWS-S TOR-CAPACITY TO BLD-C189.
IF PWS-MAX-PROD = '$
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE Ø TO BLD-C19Ø
ELSE IF PWS-MAX-PROD NOT = SPACES
MOVE DEFAULT-ACTION TO BLD-INV-ACT
MOVE PWS-MAX-PROD TO BLD-C19Ø.

```

Figure 2. Unseparated Code  
8-8

```

*
* Update PWS-NORMAL-CAPACITY (C186 in database)
*
    IF (PWS-NORMAL-CAPACITY = '$      ')
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE Ø              TO BLD-C186

    ELSE IF (PWS-NORMAL-CAPACITY NOT = SPACES)
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE PWS-NORMAL-CAPACITY TO BLD-C186.

*
* Update PWS-DESIGN-CAPACITY (C187 in database)
*
    IF (PWS-DESIGN-CAPACITY = '$      ')
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE Ø              TO BLD-C187
    ELSE IF (PWS-DESIGN-CAPACITY NOT = SPACES)
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE PWS-DESIGN-CAPACITY TO BLD-C187.

*
* Update PWS-EMER-POWER (C188 in database)
*
    IF (PWS-EMER-POWER = '$      ')
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE Ø              TO BLD-C188
    ELSE IF (PWS-EMER-POWER NOT = SPACES)
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE PWS-EMER-POWER TO BLD-C188.

*
* Update PWS-STOR-CAPACITY (C189 in database)
*
    IF (PWS-STOR-CAPACITY = '$      ')
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE Ø              TO BLD-C189
    ELSE IF (PWS-STOR-CAPACITY NOT = SPACES)
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE PWS-STOR-CAPACITY TO BLD-C189.

*
* Update PWS-MAX-PROD (C190 in database)
*
    IF (PWS-MAX-PROD = '$      ')
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE Ø              TO BLD-C190
    ELSE IF (PWS-MAX-PROD NOT = SPACES)
        MOVE DEFAULT-ACTION TO BLD-INV-ACT
        MOVE PWS-MAX-PROD TO BLD-C190.

```

Figure 3. Code Divided Into Blocks and Separated by Headers

```

C
C   SETUP BEGIN AND END POINTER
C   IB=ISTART
C   IE=IEND
C
C   SEARCH FOR IDP BETWEEN TABLE (IB),..., TABLE (IE)
1002  INDEXP=IB
      IF (IDP .EQ. IPLANT(IB)) RETURN
C
C   IDP FALLS IN FRONT OF IPLANT (IB), INSERT IDP INTO IPLANT.
C   IF (IDP .LT. IPLANT(IB)) GO TO 1099
C
C   IDP FALLS BEHIND IPLANT (IB)
C
C   IF IB IS THE SAME AS IE, INSERT IDP.
C   IF (IB .EQ. IE) THEN
      INDEXP=IE+1
      GO TO 1099
    END IF
C
C   IB AND IE ARE ADJACENT TO EACH OTHER
C   IB1=IB+1
C   IF (IB1 .EQ. IE) THEN
      INDEXP=IE
      IF (IDP .EQ. IPLANT(IE)) RETURN
      IF (IDP .LT. IPLANT(IE)) GO TO 1099
      INDEXP=IE+1
      GO TO 1099
    END IF
C
C   OTHERWISE, DETERMINE THE MIDDLE POINT OF IB and IE.
C   IMID=(IB+IE)/2.
C   IF (IDP .EQ. IPLANT(IMID)) THEN
      INDEXP=IMID
      RETURN
    END IF
C
C   IDP FALLS INTO UPPER HALF OF THE TABLE, RESET IE AND SEARCH AGAIN.
C   IF (IDP .LT. IPLANT(IMID)) THEN
      IE=IMID
      GO TO 1002
    END IF
C
C   IDP FALLS INTO LOWER HALF OF THE TABLE, RESET IB AND SEARCH AGAIN.
C   IF (IDP .GT. IPLANT(IMID)) THEN
      IB=IMID
      GO TO 1002
    END IF
1099 CONTINUE

```

Figure 4. Code Separated Into Blocks

poor usage to avoid code duplication:

```
IF(FIRST .EQ. 1)GO TO 20
10 ... block A
IF(FIRST .EQ. 1)GO TO 30
20 ... block B
IF(FIRST .EQ. 1)GO TO 10
30 ... block c
FIRST=0
```

duplication improves understanding

```
IF(FIRST .EQ. 1)GO TO 10
... block A
... block B
GO TO 20
10 ... copy of block B
... copy of block A
FIRST=0
20 ... block C
```

Figure 5. Poor Usage - Copy Lines of Code

#### 8.4.4 Format

As discussed in the previous subsection, routines consist of blocks of related statements that are linked together in some logical way. Figures 6 and 7 illustrate the techniques of blocking combined with indentation.

Horizontal indentation is most useful for languages like FORTRAN. Statements or comments on the same logical level must be evenly indented. A consistent number of spaces should be used, either four or five for each level. This enables programmers to determine which statements are within the range of a loop. Debugging statements or logically subordinate statements can be further indented from more important code. Comments (other than inline) should precede the code they describe and be indented to the right of the code. Exceptions would be headers or very important comments, which should be set off with a border or with leading asterisks or other characters. It should be easy to distinguish the executable instructions from the comments (Figures 6 and 7).

If code is indented to show logic structure (whether by the programmer or by structured programming macros), the indentation must be consistent. Assembly language code must also be separated vertically into blocks.

Part of the art of programming lies in the aesthetics of a clean program listing. Although a clean layout does not guarantee a correct program, a program with a clean, readable format is more often correct than one with a messy, confusing format (Figure 8).

### 8.5 CODING PRACTICES

#### 8.5.1 Efficiency

Efficiency, which was a major concern with second-generation hardware, has been the cause of more badly written code than any other factor. Various studies have shown that fewer than four percent of the statements in a typical FORTRAN program account for more than half of the execution time. Worrying about minor savings in individual code statements or instructions is a waste of the programmer's time. The fact that execution time is critical does not mean a program must be written in assembly language. The greatest influence on execution time is program design. If an efficient design has been formulated, the language used for coding is of secondary importance. If a higher level language is used for its readability and programming efficiency and the program written in this language executes fast enough, the problem is solved. If the object program is too slow, the bottlenecks can be located and improved. If the original design is inefficient, however, no language will compensate for this deficiency.

The proper method for writing an efficient program involves constructing an efficient design and then writing a logically sound and readable program.

```

S2: DO K=N TO 2 BY -1;
MAXV=AR(1); MP=1; DO I=2 TO K BY 1;
IF AR(I)>MAXV THEN DO: MAXV=AR(I); MP=I; END;
END; TMP=AR(K); AR(K)=AR(MP); AR(MP)=TMP; END S2;

```

VALEAF-SETUP.

```

    IF VALEAF-REC-HELD = 1 MOVE SPEC-EAF-HOLD TO VAL-EAF
    GO TO NO-READ.

```

```

    READ VALEAF AT END MOVE 1 TO EAF-EOFEOF.

```

NO-READ.

```

    MOVE 0 TO VALEAF-REC-HELD EAF-RECS-BCNT.

```

SAVE-LOOP.

```

    IF EAF-EOFEOF = 1 OR VALEAF-REC-HELD = 1 GO TO SETUPDUN.

```

```

    IF VEAF-ID < SAVE-ID MOVE VEAF-ID TO RPT-PWS

```

```

        MOVE EAF-ID TO RPT-REC

```

```

        MOVE EAF-TYPE TO RPT-TYPE

```

```

        GENERATE DETL

```

```

    ELSE GO TO NO-READ2.

```

```

    READ VALEAF AT END MOVE 1 TO EAF-EOFEOF.

```

```

    GO TO SAVE-LOOP.

```

NO-READ2.

```

    IF VEAF-ID > SAVE-ID MOVE 1 TO VALEAF-REC-HELD

```

```

        MOVE VAL-EAF TO SPEC-EAF-HOLD GO TO SAVE-LOOP.

```

```

    ADD 1 TO EAF-RECS-BCNT.

```

```

    MOVE 1 TO SUMM-EA-UPD.

```

```

    ADD 1 TO SUMM-EA-CNT.

```

```

    IF EAF-RECS-BCNT > EAF-RECS-BMAX PERFORM WRITE-EAF-EXCESS

```

```

    ELSE MOVE VAL-EAF TO VAL-EAF-BUF (EAF-RECS-BCNT).

```

```

    READ VALEAF AT END MOVE 1 TO EAF-EOFEOF.

```

```

    GO TO SAVE-LOOP.

```

SETUPDUN. EXIT.

Figure 6. Uncommented Poorly Formatted Code

```

/*          */
/* SORT ARRAY AR */
/*          */
SORTAR: DO K= = N TO 2 BY -1;
/*          */
/* FIND GREATEST VALUE IN AR(1)...AR(M) */
/*          */
        MAXVAL = ARR(1);
        MAXPOS = 1;
        DO I = 2 TO K BY 1;
            IF AR(I) > MAXVAL THEN DO; MAXVAL = AR(I)
                                     MAXPOS = I;
            END;
        END;
/*          */
/* MOVE GREATEST VALUE TO END OF AR(1)...AR(M) */
/*          */
        TEMP      = AR(K);
        AR(K)      = AR(MAXPOS);
        AR(MAXPOS) = TEMP;
END SORTAR;

```

Figure 7a. Well-Formatted and Commented Code

VALEAF-SETUP SECTION.

SECT-ENTRY.

IF VALEAF-REC-AVAILBL

MOVE SPEC-EAF-HOLD TO VAL-EAF

ELSE

PERFORM READ-NEXT-VALEAF.

MOVE 0 TO VALEAF-REC-AVAILBL-INDIC.

MOVE 0 TO EAF-RECS-BCNT.

PERFORM SAVE-VALEAF-RECS UNTIL EAFEOF

OR VALEAF-REC-AVAILBL.

SECT-EXIT. EXIT.

\*

\* THIS SECTION SAVES THE NEXT VALEAF REC INTO COR OR ONTO

\* THE SCRATCH FILE EAF-EXCESS IF THE CORE BUFFS ARE FULL.

\* EAF-RECS-BCNT IS USED TO COUNT THE TOTAL NUMBER OF

\* RECORDS SAVED FOR THIS PWS.

\* VALEAF-REC-AVAILBL-INDIC IS USED TO INDICATE THAT A RECORD NOT

\* BELONGING TO THIS PWS HAS BEEN FOUND.

\* THIS SECTION ALSO ACCUMULATES SUMMARY INFO FOR SUBSEQUENT

\* INASERTION INTO PWS-SUMMARY.

\*

SAVE-VALEAF-RECS SECTION.

SECT-ENTRY.

IF VFAF-ID < SAVE-ID MOVE VFAF-ID TO RPT-PWS

MOVE EAF-ID TO RPT-REC

MOVE EAF-TYPE TO RPT-TYPE

GENERATE DETL

PERFORM READ-NEXT-VALEAF

GO TO SECT-EXIT.

IF VFAF-ID > SAVE-ID MOVE 1 TO VALEAF-REC-AVAILBL-INDIC

MOVE VAL-EAF TO SPEC-EAF-HOLD

GO TO SECT-EXIT.

MOVE 1 TO SUMM-EA-UPD.

ADD 1 TO EAF-RECS-BCNT.

ADD 1 TO SUMM-EA-CNT.

IF EAF-RECS-BCNT > EAF-RECS-BMAX

PERFORM WRITE-EAF-EXCESS

ELSE

MOVE VALEAF TO VALEAF-BUP (EAF-RECS-BCNT).

PERFORM READ-NEXT-VALEAF.

SECT-EXIT. EXIT.

\*

READ-NEXT-VALEAF SECTION.

SECT-ENTRY.

READ VALEAF AT END MOVE 1 TO EAF-EOFEOF.

SECT-EXIT. EXIT.

\*

Figure 7b. Well-Formatted and Commented Code

# IMPROVING READABILITY

UNCLEAR:

MOVE 0 TO ACCT-TIME-TOTAL ACCT-CAU-TOTAL ACCT-ALL-CNT.  
MOVE ACCOUNT-KEY TO CURRENT-ACCT.  
PERFORM PTERM UNTIL EOF-ACCOUNT OR ACCOUNT-KEY NOT = CURRENT-ACCT.  
PERFORM CLEANUP.  
STOP RUN.

PTERM.

MOVE 0 TO TERM-CNT.  
MOVE TERM-KEYX TO CURRENT-TERM.  
PERFORM PREC UNTIL EOF-ACCOUNT OR ACCOUNT-KEY NOT = CURRENT-ACCT  
OR TERM-KEYX NOT = CURRENT-TERM.

CLEARER:

MOVE 0 TO ACCT-TIME-TOTAL  
ACCT-CAU-TOTAL  
ACCT-ALL-CNT.  
MOVE ACCOUNT-KEY TO CURRENT-ACCT.  
PERFORM PTERM UNTIL EOF-ACCOUNT  
OR ACCOUNT-KEY NOT = CURRENT-ACCT.  
PERFORM CLEANUP,  
STOP RUN.

PTERM.

MOVE 0 TO TERM-CNT.  
MOVE TERM-KEYX TO CURRENT-TERM.  
  
PERFORM PREC UNTIL EOF-ACCOUNT  
OR ACCOUNT-KEY NOT = CURRENT-ACCT  
OR TERM-KEYX NOT = CURRENT-TERM.

Figure 8

Programmers should be aware of the optimizations that are done automatically by the compiler. This information is in the programmer's reference manual for the language. Along with program evaluation tools, this knowledge will enable programmers to write more straightforward source code. Misapplied optimization frequently leads to unclear code.

#### 8.5.2 Naming and Labeling Conventions

All names and labels in a program should be used to convey information such as the function of the variable, routine, or label. They may also convey order or indicate logical relationships to other variables, routines, or labels. Some examples are as follows:

<u>Type of Name</u>	<u>Informative</u>	<u>Uninformative</u>
Flag	PARITY	OOPS
Variable	FILENO	XX
Label	INLOOP	\$-3
Entry point	EDIT	THERE

Mnemonics should be meaningful to other programmers. In the examples provided above, OOPS is probably meaningful to the original programmer, but it is meaningless to others. Use of 6-letter names in FORTRAN is encouraged; other languages allow 12-, 32-, and even 132-character names. Acronyms should be defined somewhere in the code or external documentation. Without a definition, SRDPPR is almost as meaningless as XXXXXX.

If mnemonic statement labels are permitted, they should suggest the function of adjacent code or the reason for referencing the statement. In assembly languages, all transfers of control should be directly to a label. Relative addressing should not be used because changing the number of instructions alters program logic.

#### 8.5.3 Expression and Computational Accuracy

Arithmetic, logical, and FORMAT statements should be properly punctuated and written for human readability (Figures 9 and 10). Expressions such as those in the unclear examples do not conform to these guidelines for the following reasons. First, different compilers evaluate logical expressions in different orders; parentheses should be used to make the order of evaluation explicit. Similarly,

# IMPROVING READABILITY

UNCLEAR: BMOFF (1)=(JTIME-IBTAB(1))/(IBTAB(2).IBTAB(1))\*(BMTAB(I+9)-BMTAB(1D)

CLEARER:  
 1 BMOFF (1) = (JTIME - IBTAB(1)) / (IBTAB(2) - IBTAB(1))  
       \* (BMTAB(I+9) - BMTAB(1))

UNCLEAR: Y(1)=C(1,1)\*Y(1)+C(2,1)\*Y(2)\*Y(2)+C(3,1)\*Y(3)

CLEARER:  
       Y(1) = C(1,1) \*Y(1)  
 1      + C(2,1) \*Y(2) \* Y(2)  
 2      + C(3,1) \*Y(3)

UNCLEAR: IF CNT1 = 3 OR CNT2 < D AND CNT4 > CNT5 AND FLG > 0 GO TO DONE

CLEARER:  
       IF (CNT2 < D) AND (CNT4 >CNT5) AND (FLG > 0)  
       OR (CNT1 = 3)  
           GO TO DONE

Figure 9

# IMPROVING READABILITY

UNCLEAR:

```
IF I <= 1 THEN P = 0; ELSE IF I = 2 THEN P = 1;
ELSE DO; S = SORT(I); J = 3;
DO WHILE (J <= S); IF MOD (I,J) = 0 THEN DO; P = 0; GO TO DONE;
J = J + 2; END; P = 1; END;
```

DONE:

CLEARER:

```
IF I <= 1 THEN
    P = 0;
ELSE
    IF I = 2 THEN
        P = 1;
    ELSE
        DO; S = SORT(I);
        J = 3;
        DO WHILE (J <= S);
            IF MOD (I,J)=0 THEN
                DO; P = 0;
                GO TO DONE;
            END;
            J = J + 2;
        END;
        P = 1;
```

Figure 10

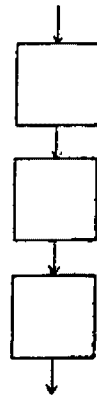
defaulting to precedence rules in arithmetic and logical statements inevitably causes errors. These can be avoided simply by writing the expression in standard mathematical notation. Also, because of binary hardware, floating point arithmetic is not associative in all cases;  $(A + B) + C$  does not always equal  $A + (B + C)$ . When operations on either very large or very small numbers are involved, programmers must specify an order of evaluation to avoid numerical errors. Proper use of parentheses increases computational accuracy. There is also a much greater chance of typographical and logical errors when arithmetic, logical, or FORMAT expressions are packed together. Without spaces between operators and operands, floating point logical expressions can be very difficult to read. It is much easier to verify the correctness of the properly formatted expressions, as shown in Figures 9 and 10.

Problems arise when quantities of greatly differing orders of magnitude are combined. Some of these problems result because many decimal fractions do not have exact binary representations. Also, differences in machine accuracy (due to differing word length and internal floating point representations) make many high-level language programs machine dependent. It should be noted that there are two representations for zero on one's complement computers (such as the UNIVAC 1100 series). Parentheses should be used to specify an order of evaluation, giving the best accuracy. Users of certain types of mainframes have experienced problems with mixed-mode arithmetic involving double-precision variables. These problems can be eliminated by using explicitly declared double-precision variables instead of relying on the compiler to propagate double-precision results through mixed-mode expressions.

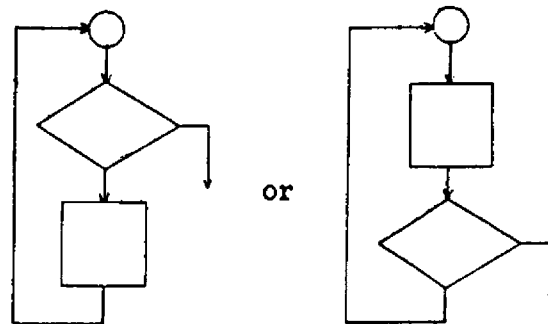
#### 8.5.4 Acceptable Program Structures

The formalization of programming disciplines has produced three basic program control structures by which any program function can be performed. It is recommended that all programs follow these constructs so as to promote uniform coding practices and avoid the difficulties so frequently associated with unstructured code. The three basic constructs - sequence, repetition, and case - are illustrated in Figure 11 and are represented by COBOL, PL/1, and FORTRAN examples in Figures 12, 13, and 14.

SEQUENCE:



REPETITION:



CASE:

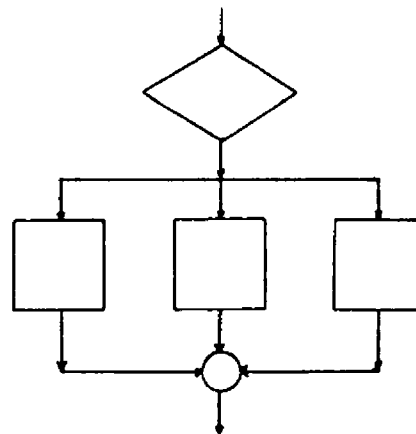
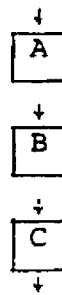


Figure 11. Program Structure Types



### COBOL

```

      .
      .
      .
PERFORM  PROCESS-A.
PERFORM  PROCESS-B.
PERFORM  PROCESS-C.
      .
      .
      .
  
```

### FORTRAN

```

      .
      .
      .
aa01 statement a1
      .
      .
      .
  
```

The prefixes aa, bb, and cc are used to uniquely identify each block.

```

aa99 CONTINUE
bb01 statement b1
      .
      .
      .
  
```

The suffix 01 indicates the beginning of a sequence block.

```

bb99 CONTINUE
cc01 statement c1
      .
      .
      .
  
```

The suffix 99 indicates the end of a block.

```

cc99 CONTINUE
      .
      .
      .
  
```

### PL/1

```

      .
      .
      .
CALL  PROCESS-A;
CALL  PROCESS-B;
CALL  PROCESS-C;
      .
      .
      .
  
```

Figure 12a. Sequence Structures

COBOL

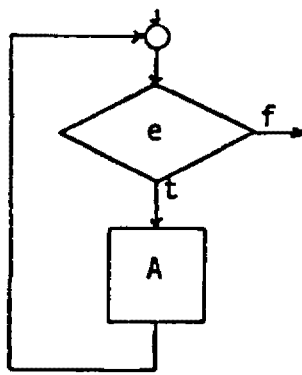
```
A-BODY-SEQ.  
    statement 1  
    .  
    .  
    .  
    statement n  
A-BODY-END. EXIT.  
*  
B-BODY-SEQ.  
    statement 1  
    .  
    .  
    .  
    statement n  
B-BODY-END. EXIT.  
*  
C-BODY-SEQ.  
    statement 1  
    .  
    .  
    .  
    statement n  
C-BODY-END. EXIT.
```

PL/1

```
A_BODY_SEQ: DO; statement 1;  
    .  
    .  
    .  
    statement n;  
END A_BODY_SEQ;  
  
B_BODY_SEQ: DO; statement 1;  
    .  
    .  
    .  
    statement n;  
END B_BODY_SEQ;  
  
C_BODY_SEQ: DO; statement 1;  
    .  
    .  
    .  
    statement n;  
END C_BODY_SEQ;
```

These alternate coding sequence schemes are provided for those cases when it may be considered undesirable to branch to a separate part of the program.

Figure 12b. Sequence Structures



#### COBOL

PERFORM PROCESS-A UNTIL NOT e.

```

A-BODY-ITR.
  IF NOT e GO TO A-BODY-END.
  statement 1.
  .
  .
  statement n.
  GO TO A-BODY-ITR.
A-BODY-END. EXIT.
  
```

#### PL/1

```

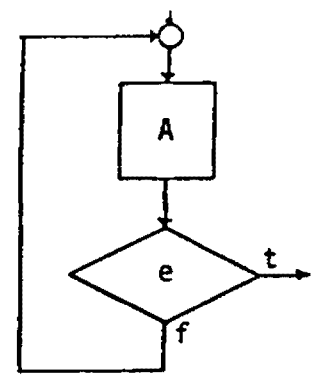
DO WHILE (e);
  CALL PROCESS_A;
END;

P_BODY_ITR:  IF (e) THEN
              DO; statement 1;
              .
              .
              statement n;
              END;
              GO TO P_BODY_ITR;
P_BODY_END:
  
```

#### FORTRAN

```

aa02  IF(.NOT. e)GO TO aa99
      statement 1
      .
      .
      statement n
      GO TO aa02
aa99  CONTINUE
  
```



PERFORM PROCESS-A.  
PERFORM PROCESS-A UNTIL e.

```

A-BODY-ITR.
  statement 1.
  .
  .
  statement n.
  IF NOT e GO TO A-BODY-ITR.
A-BODY-END. EXIT.
  
```

```

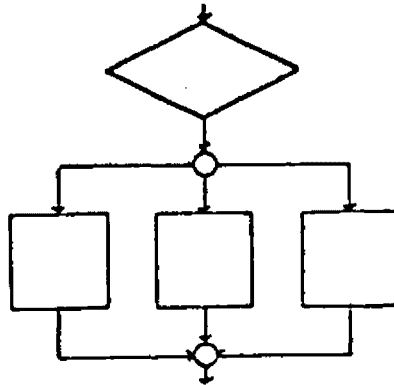
CALL PROCESS_A;
DO UNTIL (e);
  CALL PROCESS_A;
END;

P_BODY_ITR: DO; statement 1;
            .
            .
            statement n;
            END;
            IF (.NOT. e) THEN
              GO TO P_BODY_ITR
P_BODY_END:
  
```

```

aa02  statement 1
      .
      .
      statement n
      IF(.NOT. e) GO TO aa02
aa99  CONTINUE
  
```

Figure 13. Repetition Structures



# COBOL

```

IF (case-1-condition) PERFORM-CASE-1.
IF (case-2-condition) PERFORM-CASE-2.
IF (case-3-condition) PERFORM-CASE-3.

```

or

```

A-BODY-SEL.
  IF (NOT case-1-condition)GO TO A-BODY-ALT-01.
    (body of case 1)
  GO TO A-BODY-END.
A-BODY-ALT-01.
  IF (NOT case-2-condition)GO TO A-BODY-ALT-02.
    (body of case 2)
  GO TO A-BODY-END.
A-BODY-ALT-02.
  IF (NOT case-3-condition)GO TO A-BODY-END.
    (body of case 3)
  GO TO A-BODY-END.
A-BODY-END. EXIT

```

(Note: inclusion of this GO TO facilitates the addition of future alternatives.)

# PL/1

```

IF (case-condition-1) THEN CALL PROCESS_CASE_1;
IF (case-condition-2) THEN CALL PROCESS_CASE_2;
IF (case-condition-3) THEN CALL PROCESS_CASE_3;

```

or

```

A_BODY_SEL:  IF(case-1-condition) THEN
              DO; statement 1;
                (body of case 1)
              statement n;
              END;
              ELSE IF (case-2-condition) THEN
              DO; statement 1;
                (body of case 2)
              statement n;
              END;
              ELSE IF (case-3-condition) THEN
              DO; statement 1;
                (body of case 3)
              statement n;
              END;

```

A\_BODY\_END:

Figure 14a. CASE Structures

FORTRAN (1977 ANSI Standard)

```
aa03      IF          (case-1-condition) THEN
           .
           .
           .
           ELSE IF    (case-2-condition) THEN
           .
           .
           .
           ELSE IF    (case-3-condition) THEN
           .
           .
           .
           END IF
aa99      CONTINUE
```

FORTRAN (alternative)

```
aa03      IF (.NOT. case-1-condition) GO TO aa04
           .
           .
           .
           GO TO aa99
aa04      IF (.NOT. case-2-condition) GO TO aa05
           .
           .
           .
           GO TO aa99
aa05      IF (.NOT. case-3-condition) GO TO aa99
           .
           .
           .
           GO TO aa99
aa99      CONTINUE
```

Figure 14b. CASE Structures

### 8.5.5 Language Usage and Restrictions

There are a number of features and programming liberties provided or allowed by high-level and assembly languages which have caused a disproportionate number of errors and which tend to cause new bugs when compiler or operating system versions change. In the interest of more dependable systems, which are easier to debug and maintain, the use of many of the practices or features discussed herein is discouraged.

It is considered poor programming practice to rely on defaults provided by the language. Although these seem to speed up initial development (by saving the few seconds it takes to write something out fully), they may not be obvious to the maintenance programmer, and default parameters change. Debugging and maintenance are simplified if all processing procedures and parameters are completely written out. The general rule is to be explicit. This includes parenthesizing arithmetic and logical expressions to force the desired order of evaluation. All subscripts should be written out even though some languages allow them to be omitted. The arguments passed between routines must agree in number (none should be omitted), order, and type.

#### 8.5.5.1 FORTRAN

The following considerations should be taken when coding in FORTRAN. Most of these points apply as well to PL/I.

- Subscripts - All subscripts must be explicitly written out; none should be omitted. Exceptions include situations in which an entire array is referenced in an output list, data statement, or calling sequence.
- Internal subroutines - With some compilers the definitions of local and global entities are complicated in internal subroutines. Use of this feature may degrade reliability; therefore, it should be used with caution.
- Multiple subroutine entry points and nonstandard returns - Use of this feature confuses program logic because control flow between subroutines is obscured, which results in systems that are several times more difficult to debug. A unit of code should have only one entry point and one exit point. Non-standard entry points and returns should not be used.

- Branching statement - The FORTRAN logical IF statement, as in

```
IF (IFILES .LT. LIMIT) CALL MASTHD
```

is superior to the arithmetic IF or any of the nonstandard variations currently on the market because it clearly states decision criteria and has simple, predictable branching effects. The arithmetic IF, as in

```
IF (NTERMS) 10,6,23
```

is not self-explanatory, and it forces at least two GO TOs and two more statement labels on the programmer. It is unnecessary and should not be used.

The assigned GO TO, as in

```
ASSIGN 6 TO SWITCH
```

```
•
•
•
```

```
GO TO SWITCH
```

confuses program logic, complicates debugging, and can easily be abused without generating compiler diagnostics.

The computed GO TO, as in

```
GO TO (10,20,30,40), INDEX
```

may be used to implement decision tables and case structures. When the index is negative, zero, or out of limits, the results may be unpredictable. For this reason, computed GO TOs should be preceded by a test for a valid index.

Every upward GO TO in a program creates a potential loop. Therefore, they should be used solely for the purpose of implementing loops and not for picking up a few lines of code or for patching faulty logic.

- PAUSE and STOP - The FORTRAN PAUSE statement should not be used because it requires operator intervention. STOP statements must have associated messages that identify the program elements to which they belong.

- **LOOPS** - All program loops should be explicit; that is, the loop body should be separated from surrounding code and should be preceded by a prominent comment. All FORTRAN DO loops should end with a CONTINUE statement, one for each loop. All programmer-implemented loops (DO WHILE, DO UNTIL) should begin with a CONTINUE statement. The contents of a loop are indented one level (four or five spaces) for each level of nesting (see Figures 4 and 7).
- **IMPLICIT** - The variables affected by this feature must be consistent for every routine in a program.
- **PARAMETER** - There are other means of parameterization in FORTRAN; again, unless used consistently (the user should be wary of COMMON), it can lead to confusion. Before using this feature, the FORTRAN manual should be read carefully, giving attention to the numerous important restrictions. It may be desirable to use another method, even if it takes more work initially.
- **MIXED-MODE EXPRESSION** - These should be made explicit. Mode conversions should be performed by calls to the appropriate conversion routine so as to avoid misunderstandings with implied conversion.
- **EQUIVALENCE** - This is a standard feature which can be easily abused. It must not be used to extend the length of arrays. Use of EQUIVALENCES can also complicate debugging because subscripts must be mapped into their corresponding locations. The FORTRAN manual should be read carefully before using this feature.

#### 8.5.5.2 COBOL

##### 8.5.5.2.1 General COBOL Guidelines

- Use ANSI Standard COBOL as much as possible - this will greatly facilitate transporting the program between computer facilities as well as improve understanding on the part of programmers from other installations.
- Division headers should appear at the top of the page and should be separated from the body of the division by one or more blank lines.

- There should be at least one blank line between a section header and the body of a section.
- Use of columns 1-6 and 73-80 should be consistent throughout all programs and subroutines of a system.

#### 8.5.5.2.2 IDENTIFICATION DIVISION Guidelines

- This division should include the PROGRAM-ID, AUTHOR, INSTALLATION, and DATE-COMPILED paragraphs.
- This DIVISION should contain the prolog described in Section 4.1 with appropriate modifications pertinent to COBOL.
- Once a program has been placed into production, every modification should include comments that contain the following:
  - a. an index of the current modification
  - b. a description of and the rationale for the changes made
  - c. the date and person making the change.
- The AUTHOR paragraph should list the names of all persons who have made changes to the program.

#### 8.5.5.2.3 ENVIRONMENT DIVISION Guidelines

- Internal file names, the ASSIGN TO, and external file names within SELECT statements should be aligned in columns.
- Each clause subordinate to a SELECT/ASSIGN TO statement should appear on a separate line underneath the ASSIGN TO, as follows:

SELECT MASTER-IN    ASSIGN TO MASS-STORAGE OLD-MASTER.

SELECT MASTER-OUT    ASSIGN TO MASS-STORAGE NEW-MASTER.

SELECT SUMM-RPT    ASSIGN TO PRINTER    SUMMARY.

SELECT XREFFILE    ASSIGN TO MASS-STORAGE XREFFILE

ORGANIZATION IS INDEXED

ACCESS IS RANDOM

ACTUAL KEY IS XREF-ID.

#### 8.5.5.2.4 DATA DIVISION Guidelines

- Clauses within an FD should appear on separate lines under the FD file name and FDs should appear in the same order as the corresponding SELECT statements.

FD MASTER-IN

LABEL RECORDS ARE STANDARD

RECORDING MODE IS F

BLOCK CONTAINS 153 RECORDS

RECORD CONTAINS 473 CHARACTERS.

- Each 01 level item should be preceded by one or more blank lines.
- Item names, PICTURE, USAGE, and VALUE clauses for items at the same level should all be aligned in columns.
- If an item has both USAGE and VALUE clauses, align the VALUE clause beneath the USAGE. If a VALUE literal will not fit on one line, it should be placed on a separate line beneath the item name.
- Each level in a data structure definition should be indented four spaces from the previous level. This greatly facilitates the understanding of the structure.
- String lengths within PICTURE clauses should always be specified in parentheses - even if the length is a single character. This reduces the probability of error and eases the checking of group lengths.
- The use of 88 level items is highly encouraged. This simplifies the understanding of Procedure Division code.
- The use of 77 level items should be avoided.
- Constants should appear at the beginning of WORKING-STORAGE along with their associated VALUES. Variables should not be initialized with VALUE clauses - they should be initialized by the program.

```

01  PROG-VERSION      PIC X(4)          VALUE 'V3.2'.
*
01  RECS-IN           PIC 9(10)         USAGE COMP.
*
01  RECS-OUT          PIC 9(10)         USAGE COMP.
*
01  HOLD-REC.
    02  HOLD-REC-ID    PIC 9(4).
    02  HOLD-REC-TYPE  PIC X(1).
    02  HOLD-REC-DATE.
        03  HOLD-REC-YY    PIC 9(2).
        03  HOLD-REC-MMDD  PIC 9(4).
    02  HOLD-REC-QUANT PIC 9(5)V9(1).

```

#### 8.5.5.2.5 PROCEDURE DIVISION Guidelines

- Multiple objects of a single statement should be lined up in columns with one object per line.

```

OPEN INPUT  MASTER-IN
              PARM-FILE
              XREF-ADDR-FILE.
MOVE 0 TO REC-CNT
          LINES-OUT
          TAB-PTR.

```

- Operators and operands in continued statements or groups of similar statements should be aligned in columns.

```

MOVE OLD-ID      TO PRT-LINE-ID.
MOVE ID-TOTAL    TO PRT-LINE-TOTAL.
MOVE ZEROES      TO ID-TOTAL.

COMPUTE YEARLY-TOTAL = QTR1-TOTAL + QTR2-TOTAL
                      + QTR3-TOTAL + QTR4-TOTAL.

```

- IF statements should not be nested deeper than three levels. Deeper nesting is usually very difficult to understand. If deeper nesting is required by the problem at hand, then the lower level nesting should be performed elsewhere.
- Each ELSE should be aligned with its corresponding IF.
- Each condition of compound IFs should appear on a separate line. Parentheses should be used to simplify understanding of the relationships between the conditions.
- Subjects of IFs and ELSEs should be aligned in columns after the object conditions.

```

IF      (EMP-TENURE > 40)
AND     (EMP-AGE    < 60)
                                ADD 1 TO TYPE1-CNT
                                PERFORM PROCESS-TYPEA
ELSE
                                PERFORM CHECK-TYPES-B-F.

```

- STOP RUN should appear only in the driver SECTION.
- PERFORM THRU should not be used.
- The VARYING FROM, BY and UNTIL clauses on a PERFORM should appear on separate lines aligned under the PERFORMed section name. The VARYING index should not, of course, be altered by the PERFORMed section.

```

PERFORM LOAD-REC-BUFF
      VARYING BUFF-PTR FROM 1 BY 1
      UNTIL BUFF-PTR > BUFF-SIZE
           OR REC-ID-CHANGE
           OR END-OF-MASTER.

```

- All I/O operations should be isolated into separate SECTIONS. This makes the details of such operations more transparent to the program and simplifies the reading and modification of the program.
- All arithmetic computations involving more than one operation should be performed with the COMPUTE verb. The only exception is that the DIVIDE verb may be used if a remainder is required. In this case, DIVIDE BY should be used and not DIVIDE INTO.
- Parentheses should be used to simplify the understanding of complex arithmetic statements or wherever confusion may arise as to the order of evaluation of a statement.

```

COMPUTE ROOT = (DISC - B) / (A + A).

```

```

COMPUTE ADJUSTED-INT = BASE-INT * (NUM-PDS - 1)
                     + BASE-VAL / (NUM-PDS + 1).

```

### 8.5.5.3 PL/1

#### 8.5.5.3.1 General Guidelines

- Each PROCEDURE should appear at the top of a page. If a particular implementation of PL/1 does not allow this, then sufficient blank lines should be used to clearly separate the PROCEDURE from other PROCEDURES.
- Each PROCEDURE should be followed by a prolog as described in Section 7.4.1, followed by the procedure's DECLAREs, followed by the body of the procedure.
- Nested statements should be indented several spaces for each level of nesting. The degree of indenting should be chosen so as to align the indented statements in a reasonable manner.
- Avoid the use of columns 1 and 73-80 for program text.

#### 8.5.5.3.2 DECLARE Guidelines

- DECLARE procedure arguments before local variables and in the order they appear in the PROCEDURE statement.
- Constants should appear before variables. Constants should be initialized in their DECLARE statements whereas variables should be initialized in the body of the procedure.
- Unrelated items should be placed in separate DECLARE statements
- Related items appearing in the same DECLARE should be placed on separate lines with item names and attributes aligned in columns.
- Structured definitions should appear with one item per line and each level of the structure indented an appropriate number of spaces.

```
DECLARE PI          FLOAT      INITIAL(3.14159),
          TAB_SIZE   FIXED      INITIAL(139);
DECLARE 1 TIME_CARD,
          2 NAME      CHARACTER(30),
          2 SSN        FIXED(9)
          2 HOURS_CHARGED,
            3 STRAIGHT  FIXED(3),
            3 OVERTIME  FIXED(3),
          2 BASE_PAY   FIXED(2,2);
```

- Avoid the use of DECLAREs with factored attributes:

```
DECLARE (I, J, K) FIXED;
```

#### 8.5.5.3.3 PROCEDURE Body Guidelines

- Each procedure should be entered only at its top and exited at its END. Do not use ENTRYs.
- Each END should appear on a line by itself and should be aligned under the beginning of the statement with which it is associated .
- Avoid the use of BEGIN blocks. Use separate procedures instead. Since a BEGIN block is logically a discrete procedure, the text of the BEGIN block can only impede understanding the program flow.
- Avoid the use of subscripted labels. They may appear to be a convenient feature, but their use is often unfathomable by the maintenance programmer.
- Do not use local variables whose names duplicate global variables.
- Do not nest IF statements deeper than three levels. If deeper nesting is required by the problem at hand, separate IFs or procedures should be used instead.
- Each IF THEN ELSE should have the THEN and ELSE aligned under the IF, as shown below.

```
IF cond1
THEN DO;
    statement11;
    .
    .
    .
    statement1n;
END;
ELSE DO;
    statement 21;
    .
    .
    .
    statement 2n;
END;
```

APPENDIX A

GLOSSARY

## APPENDIX A - GLOSSARY

Accuracy - Precision or exactness, or freedom from error, of the results of a computation or a program.

Component, Hardware - Any of the main constituent parts of the physical equipment included in the computer system.

Component, Software - A discrete set of program instructions which may be handled as a unit for compilation and loading.

Configuration - A particular combination of hardware (CPU, disks, tapes, etc.) and software (operating system, utilities, etc.). Typically used in "minimal configuration", i.e., the minimal hardware and software requirements for a particular application.

Control Language - A language used to specify job level processing requests to an operating system. This language will vary from computer to computer (e.g., ECL on UNIVAC 1100, JCL on IBM 360/370).

Data Base - A collection of data, usually integrated and organized for ease of retrieval and maintenance.

Data Base Management System - A collection of software that provides a wide variety of data manipulation capabilities that allow a user to organize data in a meaningful and useful manner.

Data File - A collection of related data records, usually organized to meet a specific purpose and often sequenced in accordance with a key contained in each record.

Data Group - A collection of related data items within a data record.

Data Item - A specific unit of information to which meaning can be assigned, represented in signs or symbols which can be interpreted by a person or by a machine.

Data Record - A collection of related data items treated as a unit, specifically for input-output purposes.

Design Methodology - An orderly, logical procedure or approach to the design of a computer system.

Developmental Impact - The effect of a system change, enhancement, or redesign on the implementation schedule of a system or program.

Diagnostic - A message from a program (user or system) indicating that the program has detected a possible error condition.

## APPENDIX A - GLOSSARY (continued)

Documentation - A collection of written or printed items that aid in the understanding of a system or program.

Enhancement - An extension and improvement in the features and capabilities of an existing computer program or system.

EPA ADP Manual - The document issued by the Environmental Protection Agency that details data and information processing policies and procedures.

FIPS (Federal Information Processing Standards) - Publications of the National Bureau of Standards relating to automated data systems.

Hardware - The physical components (mechanical, magnetic, electronic, or electrical) of a computer system.

Host Computer - The primary computer used by a particular system.

Interactive - A mode of communication between a computer system and a person via a terminal, in which a query/response "dialogue" is conducted.

Interface - Linkage or other arrangement or convention established for communication between hardware and/or software components, such as program-program, computer system-operator, and computer system-terminal user.

Job - A specific amount of work to be performed by a computer, involving one or many processes, programs, etc.

Key - One or more characters used to uniquely identify a data item or a data record within a data file.

Log-On - The process of establishing a communications link with a computer and identifying oneself for accounting purposes.

Macro - A block of source code text that is used by certain system utilities as a template for generating source code.

Maintenance - Performance of support functions to modify, update, and correct computer programs; also to update, correct, and purge data files; also to maintain hardware on a scheduled or corrective basis.

Medium (Media) - The physical manifestation of stored data or programs such as punch cards, magnetic tape, disk, paper tape, cassettes, etc.

Mnemonic - A device or technique intended to help the memory.

Modularization - The organization of a program into sets of instructions which may be treated as discrete units.

## APPENDIX A - GLOSSARY (continued)

- Module - A discrete set of program instructions which may be treated as a unit.
- On-Line - The direct communication of a particular hardware component with another, especially with one under control of the other; also access to a computer system by a person via a terminal.
- Operating Environment - The conditions under which a program or system operates. This includes the physical state of hardware as well as conditions such as job and user mix.
- Operating System - The software components which provide control of all elements of computer system functions, including executive, call, and loading routines; hardware assignment; multiprocessing; and input-output operations.
- Operational Characteristic - A distinguishing feature or quality of the method by which a data processing function is performed.
- Operational Controls - Procedures, usually manual, which effect the continued processing of a system, such as approval procedures, confirmation of successful completion of one program before initiation of the next, and file cycle validation.
- Operational Function - A defined action by the computer system which controls data recording, transmission, interpretation, or other processing.
- Operational Impact - The effect of a system change, enhancement, or redesign on the way in which a program or system is used.
- Operational Procedure - The steps and actions performed in accordance with user policies in the functioning of a computer system.
- Organizational Impact - The effect of a system change, enhancement, or redesign on the personnel structure under which a system or program functions.
- Output, Display - A visual representation of information, such as on a CRT screen, usually read and acted upon before a subsequent action is to take place.
- Output, Graphic - Data written in representational or pictorial form, such as mathematical curves, histograms, contour plots, or maps.
- Portability - The ease with which data or programs may be transferred from one computer system to another.

## APPENDIX A - GLOSSARY (continued)

Program - A sequence of data processing instructions or statements in a form acceptable to, and intended for execution on, a computer.

Query Language - A language used to specify interactive processing requests to a Data Base Management System.

Reference File - A data file used by a program for reference purposes such as validating data items or retrieving detailed information about a data item.

Report Generator - A program whose primary purpose is to produce a report or reports based upon provided input parameters. Most computer sites have a report generator utility that can produce reports from many types of input files.

Response Time - Elapsed time between the sending of a signal or message and the receipt of an answering signal or message. This is typically a measure of how quickly an operating system can react to a processing request from an on-line terminal.

Responsiveness - The ability of the computer system to meet the needs of the user on a timely basis, especially in interactive mode.

Software - Computer data, information, programs, compilers, routines, etc., that work in conjunction with the hardware to accomplish data processing functions.

Software, Application - Computer software used for specific user needs, such as payroll, accounting, scientific calculations, or inventory control.

Software, System or Support - Computer software used to provide common data processing functions to many users.

Source Code - The symbolic text of a program. This is the form in which programs are written and read by programmers.

Subroutine - A set of coded instructions that performs a certain function within a program or that exists in a library and may be referenced by other programs for incorporation therein.

Subsystem - A related set of programs that function together to perform a portion of a data processing application; linked with other subsystems to form a system.

System - An integrated set of programs or subsystems that function together to perform a data processing application.

## APPENDIX A - GLOSSARY (continued)

System Flow Diagram - A pictorial representation of the overall flow of data and control through the system; its purpose is to graphically represent the relationships between programs, subsystems, files, and major I/O functions.

Testing Tools - Utility programs or semi-automated procedures intended to aid in the testing or debugging of a system or program. These include items such as test data generators, program execution monitors, and dump analyzers.

Throughput - The amount of work produced by a computer system, normally measured in jobs per day or per hour.


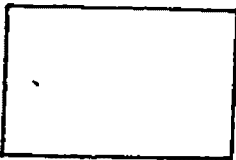
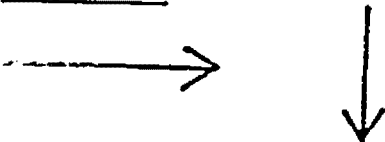


Top-Down - A method of design in which the general design is presented first, followed by successive refinements of detail, until a final precise design is reached. This is also termed "step-wise refinement".

User - A person or organization who employs or applies, for his own needs, the hardware or software components of a computer system.


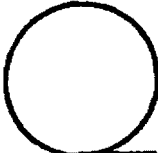

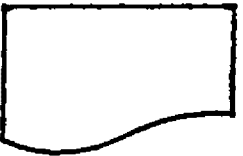

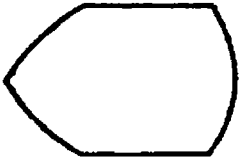


Utilities - Programs in the system library that are frequently used by most of the system's users. These include items such as sort, copy, and dump utility programs.

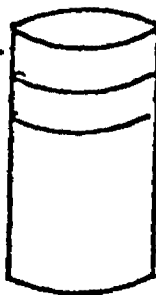


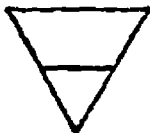
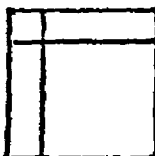
## CHARTING CONVENTIONS FOR SYSTEM DEVELOPMENT

- A. BASIC SYMBOLS. A basic symbol is established for each function and can be always used to represent that function. Specialized symbols are established which may be used in place of a basic symbol to give additional information.

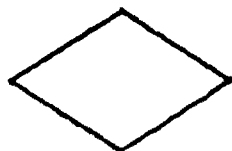
SYMBOL	USE
1. <u>INPUT/OUTPUT</u>	This symbol represents the input/output (I/O) function, i.e., the making available of information for processing (input) or recording processed information (output).
	
2. <u>PROCESS</u>	This symbol represents the processing function i.e., the process of executing a defined operation or group of operations resulting in a change in value, form, or location of information or in the determination of which of several flow directions are to be taken.
	
3. <u>FLOWLINE</u>	This symbol represents the flowline function, i.e., the indication of the sequence of available information and executable operations.
	
4. <u>ANNOTATION.</u>	This symbol represents the annotation function, i.e., the addition of descriptive comments or explanatory notes as clarification.
	
5. <u>KEYING</u>	This symbol represents an operation using a key driven device - such as punching, verifying, typing.
	



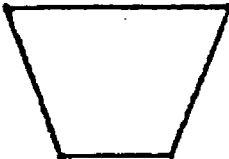




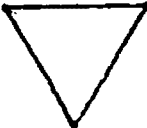
- B. SPECIALIZED I/O SYMBOLS. Specialized I/O symbols may represent the I/O function and, in addition, denote the medium on which the information is recorded or the manner of handling the information or both. If no specialized symbol exists, the basic I/O symbol may be used.

SYMBOL	USE
1. PUNCHED CARD 	For an I/O function in which the medium is punched cards, including mark sense cards, partial cards, stub cards, etc.
2. <u>MAGNETIC TAPES.</u> 	For an I/O function in which the medium is magnetic tape.
3. <u>PUNCHED TAPE.</u> 	For an I/O function in which the medium is punched tape.
4. <u>DOCUMENT.</u> 	For an I/O function in which the medium is a document.
5. <u>MANUAL INPUT.</u> 	For an I/O function in which the information is entered manually at the time of processing, by means of online keyboards, switch settings, push buttons, card readers, etc.
6. <u>DISPLAY.</u> 	For an I/O function in which the information is displayed for human use at the time of processing, by means of online indicators, video devices, console printers, plotters, etc.
7. <u>COMMUNICATION LINK.</u> 	For an I/O function in which information is transmitted automatically from one location to another.
8. <u>MAGNETIC DRUM.</u> 	For an I/O function in which information for processing (input) or the recording of processed information (output) is stored online on magnetic drum. B-2

SYMBOL	USE
<p>9. <u>MAGNETIC DISK.</u></p> 	<p>For an I/O function in which information for processing (input) or the recording of processed information (output) is stored online on magnetic disc.</p>
<p>10. <u>TRANSMITTAL TAPE</u></p> 	<p>Proof or Adding Machine tape, or other batch control information.</p>
<p>11. <u>ONLINE STORAGE.</u></p> 	<p>For an I/O function utilizing auxiliary mass storage of information that can be accessed online, e.g. magnetic drums, magnetic discs, magnetic tape, automatic magnetic card systems, or automatic chip or strip systems.</p>
<p>12. <u>OFFLINE STORAGE</u></p> 	<p>For any offline storage of information, regardless of the medium on which the information is recorded.</p>
<p>13. <u>CORE.</u></p> 	<p>Represents an I/O function in which the medium is magnetic core.</p>

C. SPECIALIZED PROCESS SYMBOLS. Specialized process symbols may represent the processing function, and in addition, identify the specific type of operation to be performed on the information. If no specialized symbol exists, the basic process symbol may be used.

SYMBOL	USE
<p>1. <u>DECISION.</u></p> 	<p>Represents a decision or switching type operation that determines which of a number of alternate paths it is to be followed.</p>

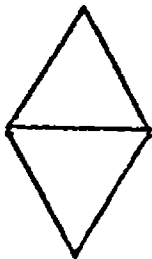
SYMBOLS	USE
2. <u>PREDEFINED PROCESS.</u>	Represents a named process consisting of one or more operations or program steps that are specified elsewhere. e.g., subroutine or logical unit.
	
3. <u>AUXILIARY OPERATION.</u>	Represents an offline operation performed on equipment not under direct control of the central processing unit.
	
4. <u>MANUAL OPERATION.</u>	Represents any offline process geared to the speed of a human being.
	
5. <u>CONNECTOR.</u>	Represents a junction in a line of flow.
	
6. <u>TERMINAL.</u>	Represents a terminal point in a system or communication network at which data can enter or leave. e.g., start, stop, halt, delay or interrupt.
	
7. <u>PREPARATION.</u>	Represents modification of an instruction or group of instructions which change the program itself, for example, set a switch, modify or index register, and initialize a routine.
	
8. <u>COLLATE.</u>	Represents merging with extracting, that is, the formation of two or more other sets.
	
9. <u>MERGE.</u>	Represents the combining of two or more sets of items into one set.
	

## SYMBOLS

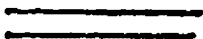
## USE

10. EXTRACT.

Represents the removal of one or more specific sets of items from a single set of items.

11. SORT.

Represents the arranging of a set of items into a particular sequence.

12. PARALLEL MODE.

Represents the beginning or end of two or more simultaneous operations.

APPENDIX C  
BIBLIOGRAPHY

## APPENDIX C

### BIBLIOGRAPHY

1. Auerbach Publishers, Inc., "Objectives of Structured Programming", Auerbach Programming Management, Section 14-02-01, 1974
2. N. Wirth, "Program Development by Stepwise Refinement", Communications of the ACM, vol. 14, no. 4, April 1971
3. J. Aron, The Program Development Process (Parts I and II). Reading, Massachusetts: Addison-Wesley, 1974
4. C. McGowan and J. R. Kelly, Top-Down Structured Programming Techniques. Reading, Massachusetts: Addison-Wesley, 1975
5. S. H. Caine and E. K. Gordon, "PDL - A Tool for Software Design", Proceedings of 1975 AFIPS National Computer Conference
6. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design", IBM Systems Journal, no. 2, 1974
7. D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, vol. 15, no. 12, December 1972
8. D. L. Parnas and D. P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems", Communications of the ACM, vol. 18, no. 7, July 1975
9. E. Yourdon, Techniques of Program Structure and Design. Englewood Cliffs, New Jersey: Prentice Hall, 1975
10. B. W. Kernighan and P. J. Plauger, The Elements of Programming Style. New York, New York: McGraw-Hill, 1974
11. G. Lowrimore, Applications Systems Development: A Better Way. EPA DATA TALK, May 1978
12. M. H. Weik, Standard Dictionary of Computers and Information Processing. Rochelle Park, New Jersey: Hayden Book Company, Inc., 1977
13. M. A. Jackson, Principles of Program Design. New York, New York: Academic Press, 1975