UCID-20643

1714-0

User Systems Guidelines for Software Projects

Coordinated by
Jeanne Martin

With Contributions from

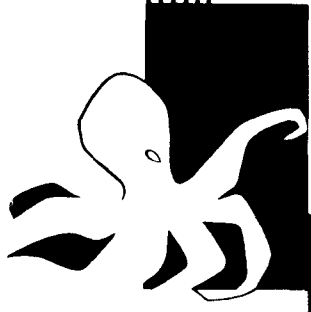
Karl Dusenbury
Barbara Herron
Jeanne Martin
Keith Grant
Peter Keller
Kelly O'Hair

Edited by
Lila Abrahamson

1 April 1986

UCID--20643

DE86 010490



Livermore Computing Systems Document

MASTER

University of California
Lawrence Livermore National Laboratory

DISCLAIMER

Contents

CONTENTS

Contents

Contents

Contents

Contents

DISCLAIMER

Introduction

INTRODUCTION

A committee within the User Systems Division (USD) of the Computation Department at Lawrence Livermore National Laboratory (LLNL) developed these guidelines for generating software between 1981 and 1984. Members of the committee were

Barbara Atkinson
Karl Dusenbury
Keith Grant
Terry Heidelberg
Barbara Herron
Pete Keller
Jeanne Martin, Chair
Kelly O'Hair
Roger Skowlund

Karl Dusenbury, Keith Grant, Barbara Herron, Pete Keller, Jeanne Martin, and Kelly O'Hair contributed chapters to the committee's final report and, thus, to this document.

The USD guidelines for software standards were developed so that software project-development teams and management involved in approving the software could have a generalized view of all phases in the software production procedure and the steps involved in completing each phase.

A typical development cycle is illustrated in Figure 1. A project that follows this cycle will have certain milestones indicated by the triangles and by the box representing the code. Although many of the indicated phases proceed simultaneously, the milestones would normally be reached in the following order.

- Project definition
  -- Project plan
  -- Specifications
- User-interface design
- Design specifications
  -- Data-flow diagrams
  -- Data dictionary
  -- Structure charts
  -- Pseudocode
- Test plan

- Design walkthrough (not shown in Figure 1)
- Code
- User documentation
  -- Reference manual
  -- Summary sheets
  -- By-example guide
  -- Tutorial
- Test history

In addition, the project manager should maintain a history of the project. The history should contain, or reference, all the documents mentioned above and should be kept current throughout the maintenance phase of the software life-cycle. Major enhancements to the software should be developed in a way similar to that of the initial development cycle.

This report is divided into six chapters, each addressing the criteria developed for a specific phase. Examples are given when appropriate, and available and nice-to-have support tools that could be used to maintain the criteria are listed.

Introduction

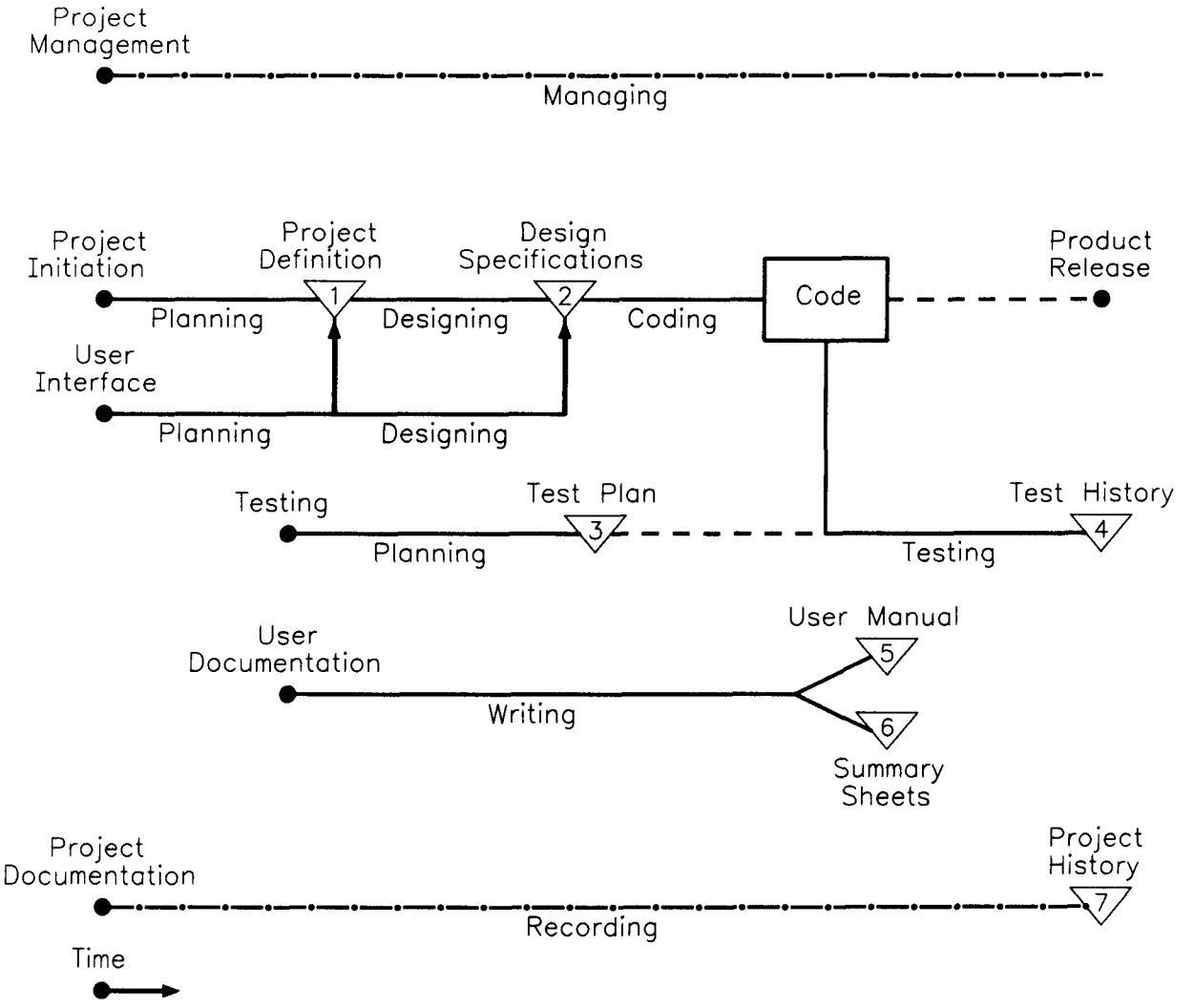


Figure 1. Software development cycle. Milestones are indicated by the triangles and by the the box representing the code. Many of the outlined phases proceed simultaneously.

PROJECT DEFINITION

Overview

USD software serves project, user, and managerial communities. Therefore, the preparation of a well-conceived project definition (PD), written without the technical jargon of a particular discipline and addressing the needs of the three groups, is an essential step in developing new software or revising an existing software product. The PD is usually generated during the initial project-development phase and functions as a combined planning, requirements, and specifications document. At that time, for example,

- It announces to the community at large that serious work is planned in a specific area
- It can be used to help management decide to proceed with the work
- Prospective users can provide feedback and help refine the project and define the potential audience
- It can serve as the project team's development base by defining goals, deliverables, milestones, and detailed requirements and specifications

The PD may undergo several revisions as the project evolves. A revision history, or audit trail, should be kept so that if the product needs to be modified, the modifications can be evaluated in light of the original and subsequent definitions. Knowing that an alternative was considered and rejected, and why, can be very valuable when changes are considered.

To generate the initial PD and any subsequent revisions, the project must be described. Each software project can be described basically in terms of its purpose, justification, goals (including milestones), resources, and maintainability.

In this chapter, we will first present guidelines for generating a PD; second, we will give examples of how to implement them; and third, we will outline some tools to support generation of the PD.

Project Definition/Guidelines

Project-Definition Guidelines

When you prepare a PD, there are some sections that are considered essential contributions and others considered less important, but helpful, contributions. In Table 1 is a summary of PD guidelines in outline form containing the different sections, a brief description of the contents of each section, and an indication whether the information is essential or helpful. Following the table is additional material about each section of the PD and the rationale for including each section. A list of questions that may be important in your PD is appended.

Table 1. Summary of project-definition guidelines for software projects at Lawrence Livermore National Laboratory.

Section	Contribution, essential (E) or helpful (H)	Content of section
Purpose(s)	E	Brief purpose(s) of the project
		Description of the product to be delivered
Justification	H	Why the project is required
		Users of the product
		How the product will benefit users
Description	E	What the product does, as opposed to how it does it
		The specific input and output
		A diagram showing the interrelationships among the different phases of the project
		An overview of the user interface

Project Definition/Guidelines

Table 1. Summary of project-definition guidelines for software projects at Lawrence Livermore National Laboratory. Continued.

Section	Contribution, essential (E) or helpful (H)	Content of section
Goals	H	The end result(s) of a successful implementation of the project. For example, Conditions that make the product efficient, interactive, user friendly, and compatible with other USD products Adherence to broad-based internal and external standards Minimum size for maximum end products Flexibility
Previous Relevant Work	H	Summary of previous and current research and dates
Milestones	E	A list of the intermediate deliverables The expected life span of the present iteration of the project, if pertinent
Resources	E	Rough estimates of person hours, hardware, software, and time needed to complete the project

Project Definition/Guidelines/Purpose

Table 1. Summary of project-definition guidelines for software projects at Lawrence Livermore National Laboratory. Continued.

Section	Contribution, essential (E) or helpful (H)	Content of section
Constraints	H	Dependencies on hardware or other products
		Limitations regarding speed or size, etc.
Accountability	E	A list of PD authors
		An account of each substantive change that includes the date of the change, the responsible author or contact, and the problems encountered in implementing the change

Purpose

The information in the purpose statement is considered essential. Therefore, the purpose is placed first in the PD. The order of the rest of the material is based on the author's priorities.

Most software development projects are concerned with product development, and you should mention this and the proposed product in the first sentence. If product development is not the reason for undertaking the project, more explanatory material may be required when you state the purpose. In general, one or two sentences should adequately describe the purpose.

Justification

If the product is in great demand by nearly all users, this section can be omitted. If one of the objects of the PD is to gain management approval for a project or to locate potential users for a proposed product, this section should be more extensive.

Assess your audience(s), and write the justification in terms of how the group(s) you are addressing will use both the end product and the PD.

For example, users would appreciate knowing what is being made available and of what value it is to them. Management needs to know what the project does for the users and why it should be added to the system. Project personnel should have justifiable references to use at critical points in the design phases.

Description

The description is the heart of the PD and is essential. For a large or complex project, two or more documents will be needed to describe what is to be done. The first document of a multiple-document PD should be a project plan (PP) that includes all the information suggested for a PD, except the description section would be less detailed. The PP should contain only enough detail for understanding and credibility. The detailed requirements and specifications should be contained in supporting documents. The supporting documents would be used as input to the design phase and are intended for use by project personnel only. These documents might not be of particular interest to prospective users or management, but they are essential to further development of the project.

For a small or less complex project, the description section in the single PD document could serve as the specifications document. In either case, graphic representations such as data-flow diagrams (see Glossary, page 137) are more effective in communicating the function of the project than words. As long as a diagram doesn't require specialized knowledge to understand, it will enhance the effectiveness of the PD. An important diagram to include in the description is a picture of how the product will interface with the product users. If the project can be broken into parts or distinct operations, the input to each part and the output from each part should be shown.

The picture could be hand drawn; it doesn't have to to be automatically produced from some machine-readable description, although a machine-readable description is easier to update and maintain than is a hand-drawn picture. Some tools are available to help produce pictures (see page 29).

Goals

Goals are the reasonable expectations for the product(s), or other results, produced on completion of the project. They are derived from user surveys, perceived functional requirements, and hardware and system capabilities. The goals may include such considerations as efficiency, size, interactivity, adherence to a standard, friendly user-interfaces, early completion of the product, compatibility with other USD products and

with previous versions of the same product, adaptability to changing conditions, ability to run on different machines, and extensibility.

It is almost inevitable that some goals will be incompatible with others, and an attempt should be made in the project-definition phase to assess the goals in terms of priorities. If you set priorities for the goals and for possible constraints, you will better determine the requirements for the project.

Constraints

If there are overwhelming constraints that drive the project, state them in a separate section. Such constraints may be performance criteria or the need for compatibility with specific hardware, environments, other products, or industry-wide standards.

Previous Relevant Work

If research was done before the PD was prepared, the results and the relevancy of these results should be reported in this section. Pertinent questions and answers that could be included are,

- Do similar facilities exist here or elsewhere that could be used?
- Is there a body of experience with similar products that could be drawn upon for guidance?

Milestones

Some thought should be given during the definition phase to the expected life cycle of the product. What types of changes will likely occur? What functions will likely be replaced or removed? For any software product, some changes are easier than others. Guidance in the PD will help the designer assure that the easier changes correspond to the most likely changes.

Early in the project life-cycle, it's important to establish intermediate products as well as the end products. The intermediate products are called deliverables or milestones. The first deliverable is the PD. For typical projects, the deliverables are the items represented by the rectangle and triangles in Figue 1 (page 3).

If the development of the product associated with the project differs from the typical development cycle, the differences should be noted, and the list of deliverables revised. The differences and any changes should be reflected in the milestone charts for the project. If the product development is atypical (such as that in Example 3, page 22), it should be carefully described; otherwise, a typical development cycle with typical deliverables will be assumed.

Resources

Milestone Charts will be required by management once the project is initiated, so it's reasonable that they be required in the PD as well. However, the charts furnish a limited amount of information, so additional comments may be needed.

Accountability

The author's name must appear somewhere in the PD. A revision history will provide an account of each substantive change and includes the date of the change, the responsible author(s) and designer(s), and the problems encountered in making the change.

Some Questions and Other Information

There may be problems, topics, and questions peculiar to the project that would interest users or management. These deserve consideration if relevant. For example, if there are unforeseen problems, delays, or resource cut-backs and the list of goal priorities changes, alter the PD to reflect these. The updated PD will alert management and users to the changes and the reasons for them.

An example of a topic that would be of interest to users or management is possible enhancements.

Following are some questions to consider as part of the PD.

- Can an old procedure be adapted to meet these needs or must a new one be developed?
- Must the product run on more than one machine?
- Should the use of a high-level language be restricted to a portable subset so the code can be more easily transported?

Project Definition/Guidelines/Questions

- Will it be necessary to organize the code into memory overlays?
- Will dynamic memory-management be necessary?
- Should table sizes be allowed to vary drastically?
- Should the program be able to terminate gracefully on an outside signal? Should it be able to restart?
- Is any run-time support required?
- Are there acceptance criteria below which the product will not be used--for example, response time, memory capacity, running time, etc.
- Should provision be made for future extension and modification? What is the growth potential?
- Are there special input/output needs?
- Will other supporting software be needed? Does other supporting software exist, or should another project be started?

Under optimum conditions, the output of each developmental phase of a project would be tested and validated before the project enters its next phase. Currently, no tools exist that perform this service. However, before it's released for review, the PD can be looked over by its author(s) with the following in mind:

- Is it complete? Are all the necessary parts described?
- Is it consistent? Is there any redundancy?
- Is the project feasible?
- Is all the information provided correct?

Project-Definition Examples

Three sample project-definitions follow. The first example (PD1) describes a specific, but very simple, file converter. This file converter will be used in subsequent chapters to illustrate other phases of software development. The second example (PD2) is for a fairly typical project, the object of which is to produce a product--a very small one-person project. The third example (PD3) is a PD for a research project that has as its principal objective not a software product but a report describing the methods and strategies to be used in later projects.

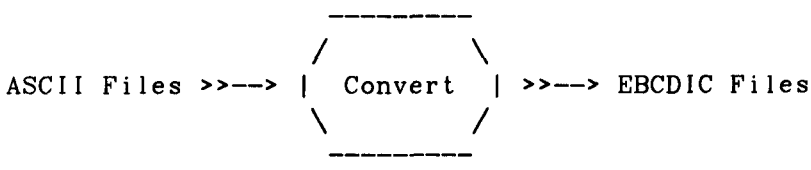
Example PD1. ASCII to EBCDIC File Converter

Project:

ASCII to EBCDIC File Convertor

Description:

- This program should accept LLNL ASCII files and convert them to EBCDIC files. There should be no limit to the size of the file except that imposed by the system. There will be a limit of 200 characters per line.



- There are no options.
  The first filename should be the name of the ASCII file to convert. The second filename will be the name of the EBCDIC file to create. If the second filename exists it will be destroyed before the new EBCDIC file is created.
- A line is defined as FORTLIB's RDLINE routine defines a line. All undefined control characters will be thrown away, unless they map to an EBCDIC character without loss of meaning.

Project Definition/Examples/ASCII to EBCDIC File Converter

User Interface:

There are no options to this program. The user will provide two filenames on the input line as follows:

PROGRAM-NAME Input-File Output-File

Error messages should be minimized but when needed will be one line and look like the following:

*** Text describing the error.

Old copies of *Output-File* are destroyed before conversion takes place. The input line must contain two filenames and the first file must exist, anything else is an error.

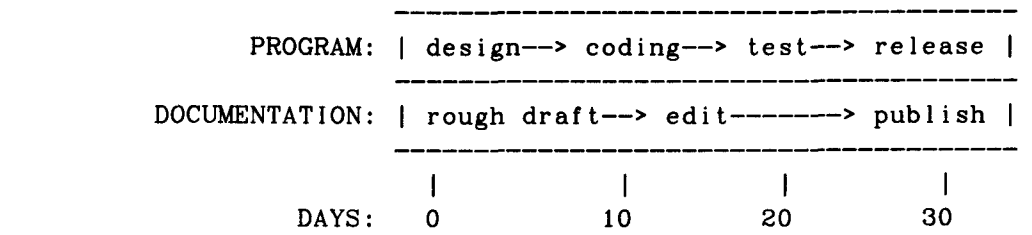
Errors on the input line should trigger the help package to be printed out at the terminal. The help package should contain the same information as the Summary Sheets.

Milestones:

- Expected life: 10 years or more
- Deliverables: --Design Document in one week
  --Finished Code in one month
  --User Manual in one month
  --Summary Sheets in one month

Resources:

- 1 FTE for one month, access to LCC network for one month, and 5 minutes of CRAY CPU time per day for one month.
- Milestone Chart



Accountability:

Author: Kelly O'Hair USD
Status: Unchanged as of 4/9/84
Changes: J. Martin, 9/85

1. User interface changed to allow omission of EBCDIC file name, making the interface more user friendly.
2. Limit of characters per line changed to 132. This number seems adequate, and larger lines are detected by RDLINE.

Example PD2. Module Template Tool

Project Definition

Module Template Tool

Purpose

To provide a tool to support the USD standard for the information content of source modules. A tool will provide ease of conformance to the standard and assure uniformity throughout the division.

Justification

Software products usually consist of a collection of modules, where a module is any separable part (subroutine, procedure, macro, etc.). If the division is to build easily maintainable products, then certain information must be available with each module in every product. It has been determined that the source module itself is the best place to maintain this information because it is the most accessible and the most likely place to be updated when changes occur.

If all products provide the same information in the same format, it will be considerably easier for programmers to move from one project to another. They will know what information they can expect to be provided and where to find it. A tool that supports this standard will contribute greatly to the achievement of the goal of producing self-documenting code.

Description

This product will produce a machine-readable source module. To do this it draws upon a template (or skeleton) which it then fills out to create the required module. (In the past the term module prolog has also been used, but this implies limiting the template to the first part of a module.)

Project Definition/Examples/Module Template Tool

The product will operate in an interactive mode. The template contains default information that the user may easily modify to define the attributes of a particular module.

As the template is being built, it will be displayed on the user's TMDS. Figure 1. illustrates the environment in which the tool operates.

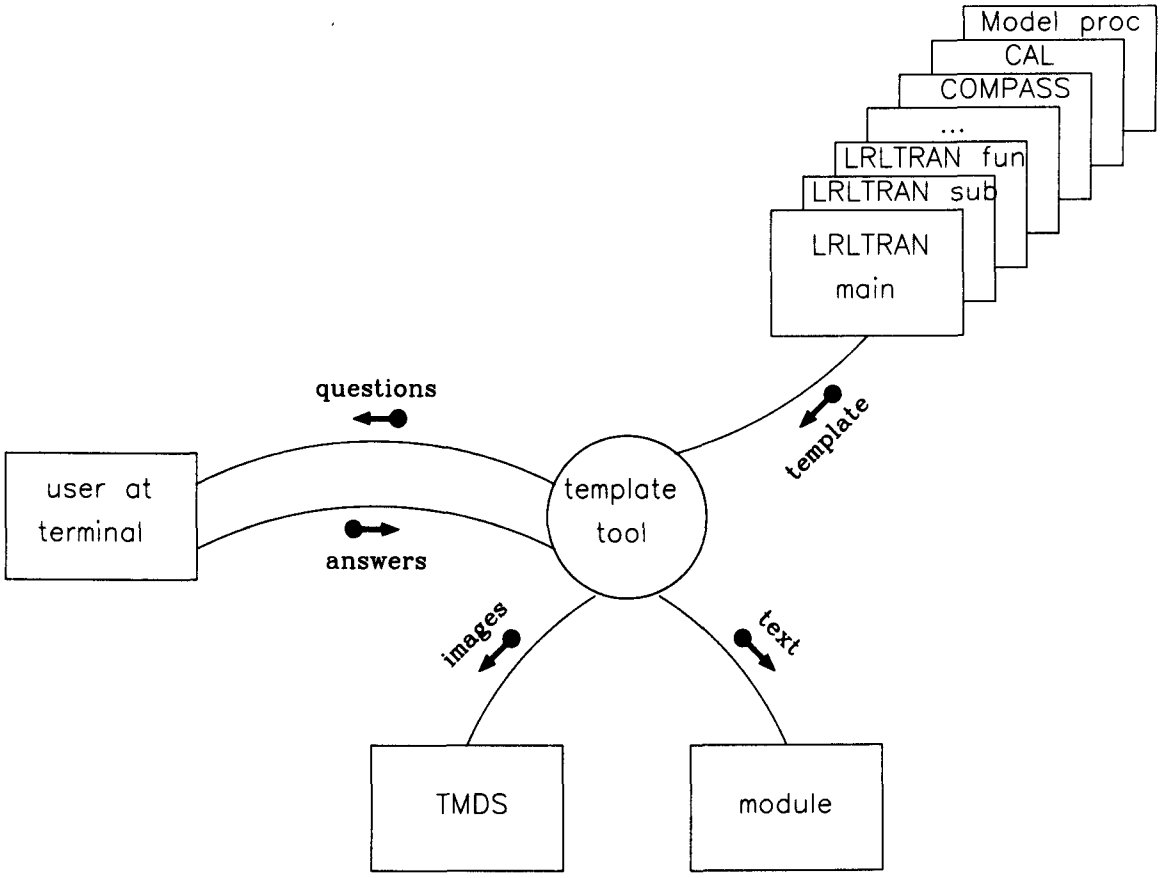


Figure 1. The Environment of the Template Tool

A user at a terminal initiates the template tool and gives it the module name and type. The tool will prompt for this information if it is not supplied on the execute line. Based on this information, the tool selects a template (or skeleton) from a closet of skeletons which it displays on the user's TMDS and copies into a private file with the supplied module name.

Project Definition/Examples/Module Template Tool

As the user fleshes out the module skeleton, the new information is displayed on the TMDS and reflected in the text file. At the conclusion of the session the file containing the module will reside in the user's file space.

The final machine readable form of the template will contain the following module attributes in the order indicated:

1) Module Name
2) Calling Form
3) Expanded Module Name
4) Description: What is done
5) Author Name and Date
6) Category
7) Input Arguments
8) Input Cliche Variables
9) Output Arguments
10) Output Cliche Variables
11) Procedure: How it's done
12) Local Variable Definition
13) Local Parameter Definition
14) Module Body
15) Module Exit: Normal
16) Module Exit: Error

The physical representation must be highly consistent between modules. This will aid in visual apprehension and also, in the extraction of module attributes into a data base. This data base can later be used to analyze, print reports and reconstruct the module from its attributes.

The following scenario will typify the use of this product.

1) A programmer designs a module. A design typically yields: data-flow diagrams, a data dictionary, structure charts and structured English.

2) The template tool is invoked. The design documents (structure chart and structured English) are used as aids in supplying the information.

3) The module body (code and comments) are then inserted, using a standard editor, directly into the module.

Project Definition/Examples/Module Template Tool

Goals

Project:

The goal of the project is to provide support for the USD standard for the information content of code modules, regardless of the type of the module or the language in which it is written. When the tool is fully implemented, every newly written module for a division-supported product should conform to the standard. When already existent modules undergo major modification, they should also be made standard conforming.

Product:

1) The first prototype of the product should be available for division-wide use in four months. Because of this time constraint, not all module types for all languages need be provided for the first version of the tool. The first version will provide templates for the following LRLTRAN modules: main, subroutine, function. Eventually it will provide templates for additional LRLTRAN modules: block data and cliche, for CAL modules, COMPASS modules, and Model modules: procedure.

2) The tool should be coded with portability in mind. The initial version should function on the 7600 and CRAY machines, but subsequent versions may be required for smaller machines, including intelligent terminals.

3) The tool should be easy to use.

    1) The tool should supply, without asking, all information that is accessible to it, such as the date, the programmer's name, TMDS number, etc.

    2) It should require the minimum amount of input from the programmer, by providing whenever possible a menu of choices for selection by typing a single key, etc.

4) The tool should encourage the user to supply information.

    1) The default information may be in a form that would encourage the programmer to supply his/her own.

    2) When the user signals the end of a session, the tool may remind the user of any information not supplied and give the user a second chance to supply it.

5) The tool should be designed so that it does not depend on a rigid format for

the template skeletons. Then the skeletons can be extended or modified slightly to tailor them for particular projects or groups within USD.

6) The tool should be easily modifiable and capable of extension.

Possible Enhancements

Initially the tool will be used in conjunction with a text editor for making corrections and modifications. In the future this division of labor may not be as pronounced.

Future possibilities are:

1) use of color to set apart different kinds of information

2) sophisticated aids for building the module body

    1) templates for language constructs (such as IF-THEN-ELSE)

    2) syntax checking of code as it is entered

Previous Work

1) USD/SIG - Documentation Standards for URLIB Routines, Karl Dusenbury

2) USD/SIG - Programmer's Template, Karl Dusenbury, 3/5/81

3) USD/DMG - FRAMIS Coding Style Sheet, Steve Jones, Aug 8 1979

4) USD/LG - OPTIM : POOLMEET, Wetherell, 3/20/78

5) USD/CGG - GRAFLIB Programming Standard, Kelly O'Hair

6) CRAY RESEARCH - Software Development Standards Manual, SM-0053

7) NASSI : Steve Wong

8) ATA prolog : Allyn Saroyan

9) NSSD - PSG Tools : Routine Prolog, John Henry, Jan 9 1981

Project Definition/Examples/Module Template Tool

---

10) NSSD - APD II Software Engineering Guidelines, 5 June 1981

Project Phases

This template tool will be a supported product, so it follows the typical life cycle. The only deviation is that there will be an extended period for user testing as a result of which the product may be modified.

This is a small project. The major milestones are 1) the completion of the Project Definition, 2) the development of the initial version of the product. The use of the product should be self-explanatory, so that very little user documentation is required. The product will be enhanced as experience is gained with its use, and extended as needs become evident.

Resources

There will be a brief period for the USD to review and revise the Project Definition. During this period, it is hoped that project personnel will be assigned.

The project will require one FTE full time for four months to develop the initial version and quarter time for an indefinite future period for extensions and maintenance.

Once the design is developed, a design walk through will be held with the project personnel and SPSC.

After the prototype is tested, division members will have a chance to use and evaluate the product before any enhancements are added.

Project milestones are shown in Figure 2.

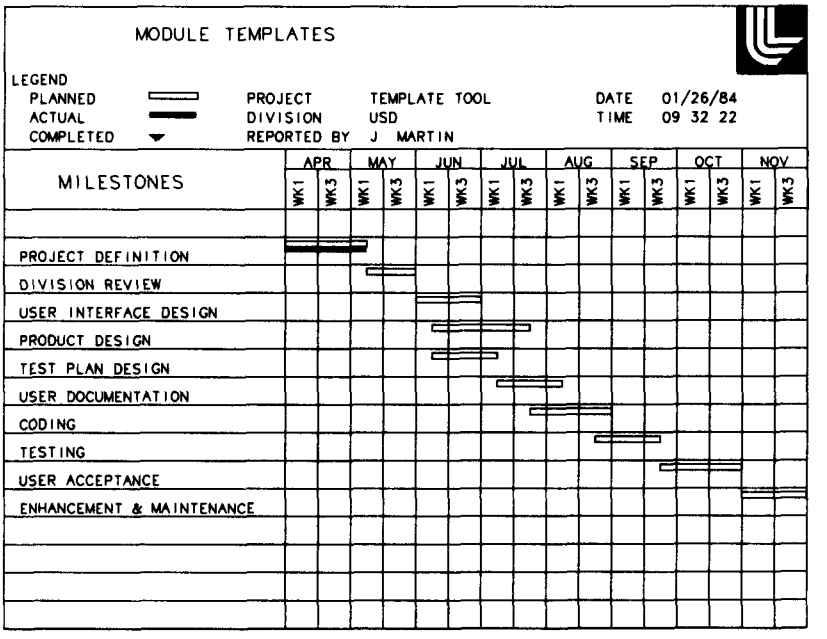


Figure 2. Milestone Chart

Revision History

Version 1	Initial cut	Apr. 7, 1982
Version 2	Description added	Apr. 29, 1982
Version 3	Prepared for review of SPSC	May 2, 1982
Version 4	Revised based on SPSC comments	May 6, 1982
Version 5	Revised based on PD guideline	Jan. 22, 1983

Authors

Roger Skowlund and Jeanne Martin

The unformatted source for the Project Definition may be found at .569800:PSC:TEMPLATEDF

The input for the Milestone Chart may be found at .569800:PSC:TOOLPDMC

Project Definition/Examples/Parallel Processor Project

Example PD3. Parallel Processor Project--Languages and Compilers, Basic Compiler Optimization Project

Project Definition

Parallel Processor Project
Languages and Compilers
Basic Compiler Optimizations Project

purpose:

There is a need to identify language-independent and machine-independent techniques to optimize the use of parallel and vector computers. As a tool to investigate possible implementation techniques, we propose an experimental compiler for a small subset of the Ada language.

justification:

The lab has always been interested in using the most powerful scientific computers available. Currently the most powerful machines are vector processors such as the CRAY-1 and the CDC Cyber 205. The vector machines are however reaching the limits of their capabilities. To get the kind of performance increase that we are interested in we will probably have to acquire some kind of parallel processor. One does not, however, see the performance increase when the hardware is installed. The software has to be modified to take advantage of the increased capabilities of the hardware. Our past experience with the STARs illustrates this. To do the job correctly, software should be completely rewritten with parallel algorithms in mind. The software should be rewritten in a language in which the parallelism inherent in the algorithm is easily expressible. This is a long and involved process. During this transition period it should be possible for existing software to exploit the new architecture as much as possible. One way to do this is to provide optimizations within the compilers for the new machine that attempt to utilize the vector and parallel hardware in as efficient a manner as possible. Thus, existing software

could in general enjoy marginal speedup by incurring no more cost than that of recompilation.

description:

the language:

For the purpose of this investigation, a small but typical set of sequential language features are required. These features should include standard structured control constructs such as if-then-else and for-do, simple data types such as integer, real, and boolean, and multi-dimensional arrays. It is possible to isolate a small subset of Ada to fill this requirement.

the compiler:

The compiler will consist of three sections. Figure 1. is a high-level data-flow diagram showing the input to and output from each section. The first section makes use of well-known technology to generate a language specific intermediate representation of the program being compiled. For the second section, a method of internal representation that is both language-independent and machine-independent must be devised. Such a method does not currently exist, and will be of use for future projects other than this one. The first two sections are required to produce the input to the third section which is the portion of the compiler that is of most interest in this investigation. This is where parallelism may be detected and operations that can be vectored may be discovered. The output from this section will be a language-independent, machine-independent, representation of the program in which the parallelism and vector operations are made explicit.

goals:

goals for the project:

1. To discover techniques that can be used to transform programs written in standard sequential languages into code that is optimized for parallel and vector machines. (Much can be learned from reading papers and reports on this subject, but there is a great deal of knowledge that can only come from an attempt to implement the techniques. Actual implementation will clarify and reinforce information culled from papers and reports.)

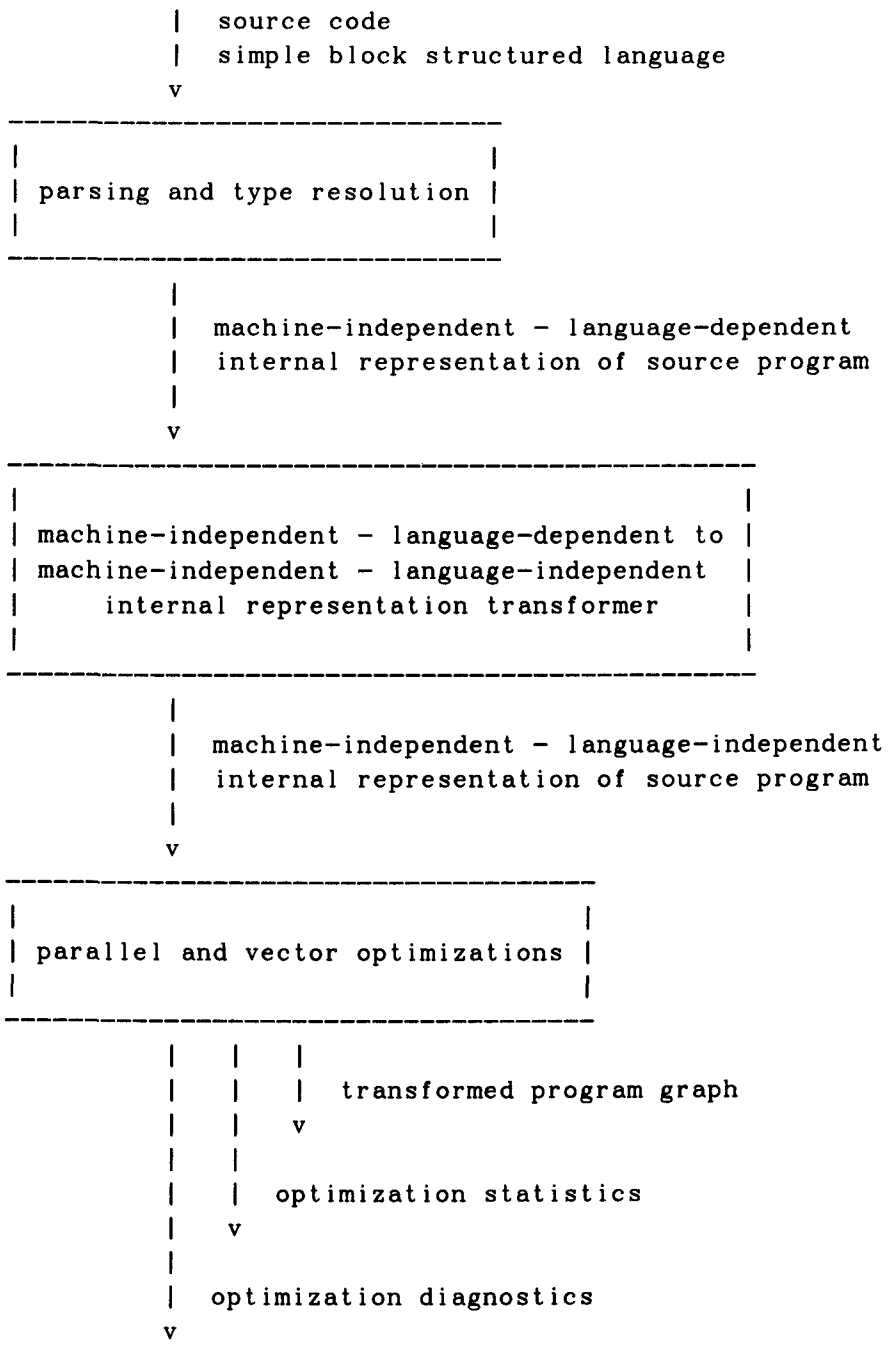

source code
simple block structured language
parsing and type resolution
machine-independent - language-dependent
internal representation of source program
machine-independent - language-dependent to
machine-independent - language-independent
internal representation transformer
machine-independent - language-independent
internal representation of source program
parallel and vector optimizations
transformed program graph
optimization statistics
optimization diagnostics

FIGURE 1.

2. To discover how difficult it is to implement these techniques in a language-independent and machine-independent manner. (At this point we do not know what kind of parallel processor we will be likely to acquire. It would be nice to develop techniques now that can be useful on whatever kind of machine we will acquire and that will not have to be reproduced each time another new machine is acquired.)

3. To identify the characteristics of these techniques that are language-independent or machine-independent. (This should help us identify those characteristics in machines and languages that detract from our optimization efforts. It may also help us to decide what kind of problems are best solved using certain language constructs on certain kinds of machines.)

4. To investigate intermediate representations. (A compiler built of modules most of which are language-independent, machine independent, or both, would be of great future benefit. Can we find intermediate representations that are sufficiently powerful to allow us to do this and still produce compilers that generate highly optimized code?)

goals for the experimental compiler:

1. Since the compiler is primarily a research vehicle, it is essential that it be highly modular and easy to modify, so that different strategies may be investigated.

2. Since it is an experimental compiler that does not even produce code, normal goals for compilers (such as reliability, efficiency, well-engineered, user-oriented diagnostics, etc.) can be slighted.

relation to previous work:

Dr. David Kuck and others at the University of Illinois have done a lot of work in the area of optimization techniques for vector and parallel computers. They have mainly looked at the problem of vectorizing FORTRAN do loops by creating and then analyzing data dependency graphs built from a representation of the subject program. They have also looked at methods to detect when do loops can execute in parallel rather than in a sequential iterative mode. This project will

basically be a vehicle to gain an understanding of their techniques and to try to identify any machine or language dependencies in the techniques. The attempt to implement them in a machine-independent and language-independent way should indicate the basic set of information that is required in the intermediate representation of a program in a compiler in order to perform these optimizations.

project phases:

The life cycle of this project is atypical since the compiler to be developed will not be a supported product. User interface and documentation can be ignored. However, testing and project documentation are highly significant. The major task with regard to project management is to keep good records of the decisions made and the optimizations that were effective.

There are three major milestones: 1) completion of the project definition and description, 2) determining the machine-independent, language-independent internal representation of the code, and 3) arriving at a basic set of optimizations. There may be several iterations of the latter two milestones. Only two are shown in the milestone chart below.

manpower estimates:

A very rough estimate of of the time required is 0.5 man-years. A large portion of the project is the initial definition of the machine independent language-independent internal representation. It is hoped, however, that the effort to develop the definition can be shared with the modular compiler project. That may cause a delay in the project but then the optimizations developed should be usable in the modular compiler, making any delay worth the wait. A milestone chart is included. See Figure 2. General milestones are given along with rough time estimates. These time estimates are based on one person working on the project half-time with the exception of the initial machine-independent language specification. That estimate is based on four people working half-time.

Project Definition/Examples/Parallel Processor Project

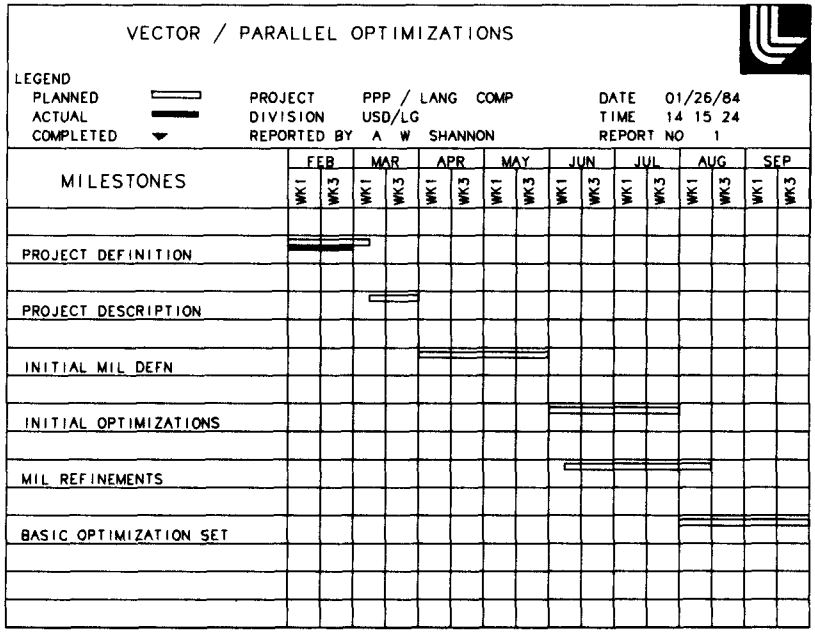


FIGURE 2. Milestone Chart

revision history:

02 Feb 1982 - document origination
16 Feb 1982 - draft released to T. Axelrod for use in meeting with Woodruff
26 Feb 1982 - draft released to D. Seberger for review and release in USD 5-year planning
- draft released to J. Martin for Project Standards Committee review as a sample project definition
05 Mar 1982 - final draft
23 Mar 1982 - revised by J. Martin to be used as an example in the description of Project Definitions in the guidelines document

Project Definition/Examples/Parallel Processor Project

22 Jan 1983 - 2nd revision as example of a Project Definition based on revision of the Project Definition Guideline

author: Al Shannon

unformatted text for PD is at
.569800:PSC:SHANNONS:PPPWBYJM

input for the milestone chart is at
.569800:PSC:SHANNONS:PPPMCHRT

Project-Definition Tools

Available Tools

There are no tools included in this section that directly support writing a PD. Instead, there are tools for general document preparation and picture drawing.

Problem-definition languages (SADT, PDL, PSA/PSL, etc.) and processors for these languages are specialized--targeted for very complex problems. These languages and processors are not generally available at LLNL. They are intended for use with detailed requirements and specifications, and their output would be part of the supporting documentation for a multiple-document project definition.

The Minimal Ada Programming Support Environment (MAPSE) is presently too unstable to support this phase of software development [Ref. 1]. If these tools become available, they will be recommended for inclusion in USD long-range plans.

The available tools are listed below and are described in two groups--documentation preparation and drawing. Each tool is referenced by a document and reference number, or by a directory pointer, or both a document and pointer. In addition to the tools listed here, see User-Documentation Tools on page 135.

UNLESS NOTED OTHERWISE, THESE TOOLS ARE AVAILABLE ONLY ON A CDC 7600

Document Preparation

The RED Dialect of TRIX

Document: H. Moll, *The Trix Report Editor*, LCSD-818 (1984) [Ref. 2].

To get a printed copy, log onto a CDC 7600 and enter

trix ac!print!nip trixred *ann id*

Use: The RED dialect of TRIX generates, updates, and formats reports in an interactive mode from a terminal. Because the RED dialect is an extension of the AC dialect, all editing features of AC are available. Source text can be formatted and viewed on the Television Monitor

Display System (TMDS). The RED dialect is loaded by issuing the AC command RED (enter: red). To send and receive the 7-bit ASCII character-set, issue the command RED7 (enter: red7).

REDPP (Post processor for RED dialect of TRIX)

Documents: J. C. Beatty, *REDPP -- A Postprocessor for the TRIX/RED Report Editor*, UCID-30125 Rev. 1 (1977) [Ref. 3].

K. O'Hair, *Computer Graphics by Example, Part 3 -- REDPP: A Post Processor for TRIX/RED*, UCID-30166 (1978) [Ref. 4].

Copies of the above documents are available at the Computer Documentation Library, T2106, Room 1001, Ext. 2-0592.

Use: REDPP accepts as input reports formatted by TRIX RED. The output produced is for different devices, one at a time. The devices include TMDS, Remote Job Entry Terminal (RJET), a printer (NIP), etc. Text can be printed in different fonts, and pictures may be intermixed with text.

SPELLING

Document: SPLREPORT. This document can be retrieved through XPORT from

.502500:reports:splreport [Ref. 5].

To get a printed copy, log onto a CDC 7600 and enter

trix ac!print!nip splreport box *ann ide*

Use: The SPELLING program is a spelling checker. It checks all alphabetic words in your input file against a database of alphabetic words. If you wish, you may supply a database of your own. Your database will be accessed and checked in addition to the one supplied in the program. A file called ERRORS will be generated. ERRORS will contain all the words in the checked file that did not match a word in the supplied databases. After you run the spelling checker, print a copy of ERRORS if you need to refer to it.

To execute SPELLING, type

gap spelling *filename* [*your database*]

For a brief explanation, type

gap help spelling

Drawing Preparation

PICTURE

Documents: J. C. Beatty, *PICTURE -- A Picture-Drawing Language for the TRIX Report Editor*, UCID-30156 Rev. 1 (1979) [Ref. 6].

K. O'Hair, *Computer Graphics by Example, Part 4 --PICTURE: A Picture-Drawing Language for TRIX Report Editor*, (1978) [Ref. 7].

Copies of the above two documents are available in the Computer Documentation Library, T2106, Room 1001, Ext. 2-0592.

Use: PICTURE is a set of commands for drawing diagrams that are to be plotted by REDPP. PICTURE commands may be included in TRIX RED source files or may be separately compiled and displayed by REDPP.

SCI (Used to draw structure charts and available only on Cray computers)

Document: C. Streeter, *SCI (Structure Chart Interface) Users Manual* (1982) [Ref. 8].

You can obtain a copy of the document from the author or from Al Liebee.

The controllee can be retrieved by entering

xport!read .522575:tools:sci

Use: This drawing tool is targeted more toward the design phase of software development. SCI allows a user to construct a structure chart on the TMDS. Once the chart is constructed, it can be extended, modified, and plotted on different media.

MCHARTSC (A program that can be used to modify milestone charts)

Document: J. S. Chin, *MCHARTSC: A Program that Creates and Modifies Milestone Charts* (1978) [Ref. 9].

To obtain a copy of this document, log onto a CDC 7600 and enter

trix ac!print!nip ucid30165 box *ann id*

To use the program, type

gg mchartsc *options*

Use: To include a milestone chart using MCHARTSC in a document, type nk before you output the chart (this keeps the UX80 file). Then, ask for the large size by typing l (ell). The UX80 file can then be included in your text file. This was described in the March 1982 *Tentacle*, page 16 [Ref. 10]. To see the results of using MCHARTSC, see the two project definition examples in this section.

FTE (A resource-allocation program for managers)

Document: P. Keller, *FTE: A Resource-Allocation Program for Managers* (1977) [Ref. 11].

Use: This tool is targeted more toward the later phases of project management.

Ideal Tools

There are at least four types of tools that would aid in writing a PD. These are a project-definition template, tools for the requirements definition, integrated document-preparation tools for a PC, and an environment with data-base management facilities.

A Project-Definition Template

Until a project-definition template is available, the unformatted text of the two examples in this section my be used as a guide. To obtain the source, refer to the Author Information section in each example. If a widely used template tool is developed, a project-definition template tool based on the same model could be very useful.

Tools for the Requirements Definition

More sophisticated tools for the requirements definition are needed. The tools would be used to validate the definition and assure that the design, coding, and documentation fulfill the requirements. An integrated tool set would justify expending effort to learn, use, and maintain a new language, and its processors, devoted solely to project development.

Integrated Document-Preparation Tools on a PC

An integrated set of tools available on a personal computer (PC) could provide offline and online support for all documentation, including the PD. The tool set would include processors that check spelling and suggest improvements in writing style, picture processors that have power comparable to PICTURE, and project-management aids similar to MCHARTSC and FTE.

An Environment with Data-Base Management Facilities

An environment, such as proposed in the *Programming Environment Project Definition* [Ref. 12] and with access to data-base management facilities, is needed. It could provide a project-definition base for maintaining all project documents, including the PD.

Project Definition/Bibliography

Project-Definition Bibliography

Anonymous (Stoneman), *Requirements for Ada Programming Support Environments*, Department of Defense, Washington, DC (1980).

R. L. Glass, "A Minimum Standard Software Toolset," *ACM-SIGSOFT Engineering Notes*, 7(4) (1982).

K. L. Heninger, "Specifying Software Requirements for Complex Systems," *IEEE Transactions on Software Engineering*, SE-6(1), 2-13 (1980).

R. C. Houghton, "Software Development Tools: A Profile," *Computer*, 16(5), 63-70 (1983).

W. E.Howden, "Life-Cycle Software Validation," *Computer*, 15(2), 77-78 (1982).

Jet Propulsion Laboratory, *Standard Practices for the Implementation of Computer Software*, California Institute of Technology, Pasadena, CA, JPL 78-53 (1977).

B. L. Meek and B. A. Heath, *Guide to Good Programming Practice* John Wiley and Sons, New York, 1980).

M. Page-Jones, *The Practical Guide to Structured System Design* (Yourdon Press, New York, 1980).

L. J. Osterweil, *A Software Life-Cycle Methodology and Tool Support*, NTIS, Atlanta, GA, AD-A076 335 (1979).

D. T. Ross, "Reflections of Requirements (Software Engineering), *IEEE Transactions on Software Engineering*, SE-3(1), 2-5 (1977).

D. T. Ross and K. E. Schoman Jr., "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, SE-3(1), 6-15 (1977).

USER INTERFACE

Overview

An interface is defined as a common boundary between systems, devices, or programs. A user interface (UI) is that portion of a software package specifying how the person running a program and the program software communicate. To the user, the UI is the most visible part of a software package; and if the interface is well programmed, it is possible to use the software without knowing about its internal workings. A product is often judged on how well the UI software handles error situations and how helpful it is in accomplishing the user's task.

In this chapter, we will first present guidelines for building a UI and provide steps to take that will help you prepare a well-designed UI; second, we will give a you an example of how to build a UI; third, we will suggest material to include in specific UIs; and fourth, we will list and describe some tools you might use to implement interface generation. A bibliography for this chapter is also provided on page 51. You may find the glossary on page 137 useful.

Some aspects of a UI are not controlled by the USD interface-designers and standards for these are not covered in this chapter. The material not included here are the operating systems (LTSS, UNIX, NLTSS, etc.), specific hardware devices, characteristics of the devices (baud rate, resolution, focus, flicker, color, etc.), and the physical environment.

General Guidelines for Building a User Interface

When you build a UI, there are some phases or areas that are considered essential and others that are considered less important but helpful to cover. In Table 2 (page 36) is a summary of the essential phases in writing a UI and a brief description of each in outline form. Each item listed is then expanded. A section describing voluntary design considerations follows.

For both the essential and less important areas of the UI, there are commonly-used terms for commands, keywords, specifiers, etc. We suggest that if you use any of these, you allow for them to be typed in upper- and lowercase combinations and, possibly, misspelled. Refer to the Glossary on page 137 for how to use the terms END, HELP, TV, TTY, BOX, etc. and the control functions in menus and documents.

Essential Phases

Table 2. Summary of the essential phases in writing a user interface for software generated at Lawrence Livermore National Laboratory.

Phase	Description
Write the project definition for the user interface	The project definition is written before designing the user interface. It should include the essential contents--that is, purpose, description, resources, and accountability (refer to Table 1, page 6).
Design the user-interface module	The user interface should be designed before other parts of the software package. If the interface is designed as a stand-alone package or as a module, it could easily be changed or updated to reflect the needs of the users.
Write the help package	A detailed help package should be included in the interface module.
Prepare a user manual	Write the portion of the user manual that descibes the interface before conducting the design review.
Conduct a design review	The user interface should pass a design review before building the product. The design review could be conducted either by building an interface prototype for prospective-user experimentation or by asking prospective users to review the interface documentation.

Write the Project Definition

Before designing a UI, write a project definition (PD). Use the guidelines in the chapter on PDs (page 5), with particular emphasis on profiles and needs of the intended users--that is, the audience.

You may want to interview potential users and find out directly how they use computers or what they would like the interface to do.

Once you identify the potential users, you can group them and note the requirements for each group. For example, most experts prefer terse commands, while novices prefer the question and answer, menu, or form-filling formats. Novices are generally tolerant of extra typing, but managers are not. If the users will be both novices and experts, or you expect the novices will quickly become expert, you may wish to provide two or more modes of interfacing with the main product. The users could choose the most comfortable mode if you write a description of how each mode works. This description may also help you and potential users identify awkward designs.

Design the User-Interface Module

The user interface should be designed both before other parts of the software product and as a separate unit or module. The user interface and the functional portion of the software project will influence each other. However, if you postpone the design of the interface, other concerns may inadvertently take priority, and the success of the entire project may be reduced.

If the user interface is designed as a module, it easily can be replaced or updated. For example, if your project plan includes the use of a prototype interface, you could substitute the prototype without reworking the entire software package. Also, new features requested by users could be incorporated in an interface module without disturbing the rest of the project.

Write the Help Package

The UI should contain a help package. Whenever the user needs to make a decision or choose among several options, he should be able to request additional information that explains the options with words or diagrams.

The help package could be integrated into the UI design. For example, an error could be treated as an implicit command for help, and successive errors could be treated as implicit commands for more detailed explanations. Another example is an interface that only allows legal displayed choices--other choices become illegal. That is, all keys except those listed in the displayed (legal) list are disabled.

It is theoretically possible to provide a generalized help package for a class of software products. This could be used as a stand-alone tool, or it could be accessed from individual products. In the latter case, the individual products would not have to include a help package. However, the project team would have to supply information on the help requirements of its product to the person maintaining the help tool.

Write a User Manual

The UI user manual should be written before the design review. This would help to consolidate ideas and to identify problems and resolve them before the code is written. If problems are discovered early, rewriting the code could be avoided.

A well-conceived user manual should have a description or an illustration of what the user can expect at each point in the interaction. All information needed to perform the task(s) should be provided in standard terms and in a form both the novice and expert users can access. This may mean that, as with the interface itself and the help package, there be two or more ways to access information.

Conduct a Design Review

The user interface should pass a design review before building the product. The review could be done either by the prospective users experimenting with a UI prototype or reviewing UI documentation. If the prospective users misinterpret the instructions, you could then modify the interface to reduce the error probability.

Other Considerations

There are additional items to consider in building a UI. Although these might not apply to every situation, you might find the items below of value and wish to evaluate your UI in the context of each.

Assess Requirements

Each package will have unique requirements, but there are some important requirements you should think about as you prepare the interface. A comprehensive, but necessarily incomplete, list follows.

- List necessary performance criteria. For example, list response time, error handling, and the possible user learning-curve.

- Do not make the interface the same for all terminals. This would penalize the users of advanced terminals if you prepared the interface for less advanced ones. However, use the advanced features when you can.

- Assume that everyone has a screen for output and a keyboard for input. Allow the input to appear immediately at the work station. If you're expecting the users to work with pointing devices, allow for easily adding pointing and selecting input. Consider voice input.

- Identify tasks the product must perform. Pay special attention to high-frequency tasks. Can the high-frequency tasks be automated so that no keystrokes are needed. Can the number of keystrokes or movements be minimized? Tasks can be broken into subtasks for easier analysis.

- Consider how user performance can be enhanced. Will default values reduce needed input? Will short commands reduce keystrokes? Is a common task merely a sequential invocation of commands? If so, then create a new command to invoke the sequence.

- Identify the methods a user will perform to accomplish tasks. List the functions that should be available. Describe typical scenarios for different user tasks.

Maintain Simplicity and Consistency

Keep the interface simple. Consider removing options that only a few people would use, and resolve all design and implementation trade-offs in favor of simplicity and consistency within your interface and among others. For example, use the one word ERASE to remove characters, lines, sentences, and paragraphs rather than a different word to remove each. At every decision point or menu listing, keep the number of menus small (seven or fewer) and the number of choices within a menu even smaller (three or fewer). See page 43 for more information on menu-driven interfaces.

To maintain consistency, you could model your interface after a successful one and employ commonly-used commands. However, if you plan to deviate from common usage, justify it in the initial document. Then, be consistent within your program, use common conventions, and choose true

opposites for opposite operations. An example of a common convention is to use a left to right or top to bottom format. STOP and GO are opposite choices rather than STOP and CONTINUE.

Organize for Quick Comprehension

If your program is simple and consistent, it should be usable with little or no documentation. To assure this, avoid jargon, use vocabulary and structure consistent with your audience, and include access to help facilities.

Determine the way you wish to control the interface--that is, what dialogue you use. Choose one from the following list.

Question and answer
Form filling
Menu selection
Function keys
Command language
Natural language
Interactive graphics--for example, light pen, mouse, etc.

For whatever dialogue type you select, let the user be in control by allowing him a way to restart, reset, abort, skip, etc. Separate the programming language from the task language by letting programming commands be executed through the control-key commands.

Provide Working Environments

For editorial tasks, provide the same text editor for each task. Don't change text editors within the project. However, do allow different windows for different tasks.

Establish Execute Lines

Execute lines should be similar to those of other programs. In addition, they should be terse and use file input rather than long lines. Input should be allowed in any order and omitted items provided for by defaults. If there are no defaults for omitted items, provide prompts--for example, prompt the user for a TV monitor number.

User Interface/General Guidelines for Building a User Interface/Other Considerations

Present Models or Analogies

Present the user with a model or analogy of the interface to help define what the interface will accomplish. For example, if your application is building pictures interactively with the program, you might choose the analogy or model of an artist at an easel or a draftsman at a drawing board. If your program doesn't lend itself to a direct analogy, present the user with a familiar model that will work with the interface. For example, compare your program with a calculator or microwave oven with its buttons. You can then relate function buttons and menu items in the software to the model(s).

Provide Error Tolerance

The interface should allow for user error at several levels. For example, the program should

- Check and correct user input
- Provide a restart feature
- Provide a command to escape or undo user error
- Allow the user to edit input discovered to be in error

Provide Error Messages

Error messages should be meaningful and not criticize the user. For example, systax error responses should identify what was expected and where it was expected. Indicate what the product was trying to do and suggest alternate paths for the user. Cryptic output, such as ERROR 1 1 10 ...., may force the user to read a manual or call a consultant unnecessarily.

Establish Cues

Color, blink, and sound (a bell) provide powerful cues and can be used for highlighting and calling attention to aspects of the interface. Be careful to use these cues only for particular purposes. If the cues are used indiscriminately, a user may disregard them or be confused.

Prepare a Pleasing Layout

Design the interface layout with eye appeal in mind. Jan White [Ref. 12] suggested that you use the following graphics art rules.

- Let the form or layout be eye appealing
- Highlight important output
- Avoid clutter
- Avoid extraneous output

Guidelines for Specific User Interfaces

If your interface falls within the interactive, batch, or graphics category, there are other guidelines you may wish to include. These are outlined below.

Guidelines for Interactive Products

- Error messages should be only one line.
- Use multilevel help messages. For example--if the user asks for help, provide a terse help message first; but allow the user to ask for more help in the form of more detailed information.
- Allow the user to edit previous input. The user can then correct any errors and not have to re-input material.

- Remember to remain interactive as you prepare the program.
  -- Keep the product small
  -- Consider using overlays
  -- Consider using controller/controllee capabilities
  -- Consider using interactive machines or interactive modules

Guidelines for Menu-Driven Products

- Keep the number of menus in the entire interface small--seven, plus or minus two is the preferred number.
- Keep the number of choices per menu small--three or fewer.
- Display only legal choices with minimum information
- Use meaningful icons (see Glossary, page 137) for the choices. For example, use a red-octagon icon for STOP.
- Include instructions for selecting items on the menu.
- Use terminal pointing-hardware, if available, to select menu items.
- Use multilevel menus where needed for additional information. For example, multilevel menus in a help package allow for different levels of information detail.
- Display something on the screen rather than leave it blank. You can display status (where I am), history (where I was), or options (where I can go next).
- Select graphics formats following graphics arts guidelines. [Ref. 13].

Guidelines for Batch Interfaces

- Allow five or more lines for error messages if needed.

Guidelines for Graphic Interfaces

- Choose symbols that can be seen easily. For example, use filled circles rather than dots or unfilled circles.
- Choose symbols that are easy to differentiate. For example, use circles and triangles rather than circles and octagons.
- Choose appropriate colors. For example, use green for go, safe, continue, etc. and red for stop, danger, etc.
- Select highly legible fonts. Roman serif is considered the most legible.

User Interface/Example/ASCII to EBCDIC File Converter

User Interface Example--Building a User Interface

The UI example described in this section covers building an interface for an ASCII to EBCDIC file-converter. The material here could be used also in writing the overall project definition and the overall design-guidelines for the file-converter.

Example UI1. ASCII to EBCDIC File Converter

The Audience

This product will be used occasionally by system programmers. They will tolerate cryptic I/O messages but prefer that the messages be documented. A photocopy of work notes will be enough documentation for them. However, if the programmers need the documentation to execute the software, they probably won't use the program. That is, the programmers don't want an interface that insulates them from what is actually occurring in the system. On the other hand, they don't want their hands held or too cute icons--for example, an icon that looks like a garbage can for "delete files, but don't destroy them" is cute, but it might not mean the same thing to the systems programmer as to the builder.

Review of User-Interface Requirements

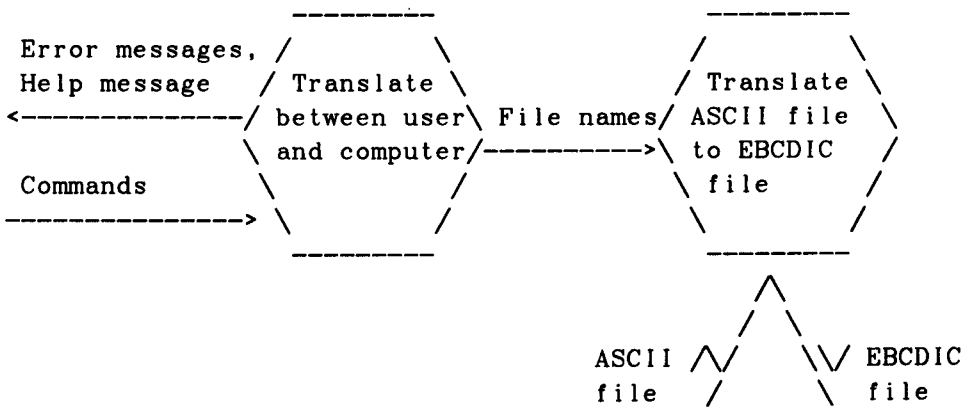
After reviewing the requirements, the intended users agreed that the following features are needed to use the interface.

- Executes with fewer than two dozen keystrokes
- No external-documentation requirement
- Learning time of less than two minutes
- Utility is fail-safe and will not destroy the input file
- Error messages are explicit

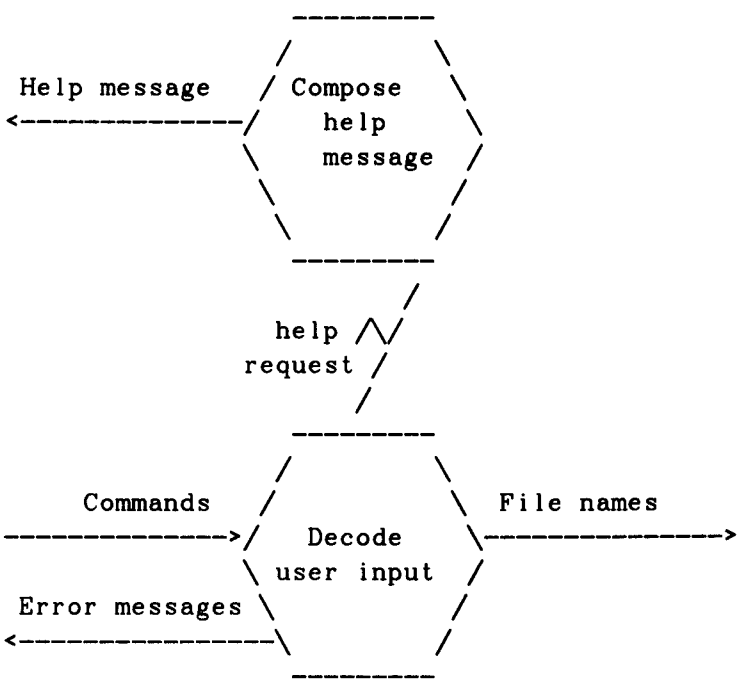
User Interface/Example/ASCII to EBCDIC File Converter

User-Interface Design

Data-Flow Diagram



The user interface part ("Translate between user and computer") may be further broken down into:



User Interface/Example/ASCII to EBCDIC File Converter

Data Dictionary

Error messages	Brief explanations of what is wrong
Help message	A short tutorial, usually on a very specific topic
Help request	A command or flag used to select a tutorial
Commands	Some prompt that instructs the computer to do something
File names	Names of the ASCII and EBCDIC files

Design Analysis

We analyzed the design by thinking about alternative user-interface design, writing down the designs, and selecting the design that best fits the project requirements. For the alternative interfaces below, we assumed that the user is already logged in. All user interfaces described work with the same process--translate ASCII file to EBCDIC file.

The different interfaces represent different dialogue techniques. For example, alternatives 1 and 3 are command language; alternative 2 is a menu; and alternative 4 is a question/answer format. Each alternative is a separate module, and other modular user-interfaces could be considered.

NOTE: User response can be typed in as seen or in any variation of lower- and uppercase letters.

Alternative 1

User types: CONVERT AFILE BFILE
Program responds: ALL DONE

On completion, BFILE contains an EBCDIC conversion of the contents of AFILE. If BFILE already exists, it will be destroyed and recreated. If only AFILE appears on the execute line, a file with a name constructed by prefixing the current suffix to $EBCDIC will contain the EBCDIC conversion of AFILE. If no file names appear on the execute line, the user gets a help message.

Alternative 2

Assume that the user normally uses IMP (U, or some other environment).

User types: C 1 (convert file 1)
IMP displays: ALL DONE

On completion, the contents of the selected files are changed to EBCDIC. The user gets a help message if the file selection is omitted.

User Interface/Example/ASCII to EBCDIC File Converter

Alternative 3

```
User types:  CONVERT FROM.  AFILE TO.  BFILE
Program responds:  ALL DONE
```

On completion, BFILE is the EBCDIC version of AFILE. The user gets a help message if he makes a mistake (this is a form of error handling).

Alternative 4

```
User types: CONVERT
Program responds: TYPE ASCII FILE NAME
User types: AFILE
Program responds: TYPE EBCDIC FILE NAME
User types: BFILE
Program responds: MORE? TYPE: YES or NO
User types: NO
Program responds: ALL DONE
```

On completion, BFILE is the EBCDIC version of AFILE. The user will get a help message if he types HELP.

Design Selection

We selected Alternative 1 as the best fit for user requirements.

Usage Scenarios

There are three possible scenarios that the user could employ. Each is outlined below.

Scenario 1

```
User types:  CONVERT AFILE BFILE
Program responds:  ALL DONE
```

Scenario 2

```
User types:  CONVERT
Program responds:
     This program will convert a LLNL ASCII file to a IBM
     360-compatible EBCDIC file.  To run this program, type:

                          convert name1 name2
```

where name1 is the name of your Ascii file and name2 is the name of the EBCDIC file you want created to contain the converted text. If name2 is omitted, a file will be created with the name ?$EBCDIC, where ? is the current suffix.

filename(s)?

User types: AFILE
Program responds: ALL DONE

Scenario 3

User types: CONVERT ANONE
Program responds: CANNOT OPEN ANONE
This program will convert a LLNL ASCII file to a IBM 360-compatible EBCDIC file. To run this program, type:

convert name1 name2

where name1 is the name of your Ascii file and name2 is the name of the EBCDIC file you want created to contain the converted text. If name2 is omitted, a file will be created with the name ?$EBCDIC, where ? is the current suffix.

filename(s)?

User types: END
Program responds: ALL DONE

Error Detection

The following errors should be detected: input file does not exist; output file can't be created; more than two symbols on the input line.

Available Tools

The implementation tools listed below are available but are not necessarily recommended for your specific application. Refer to the sections on tools in each chapter for other, possibly relevant, programs.

URLIB

Document: J. Minton, et al., *URLIB-Part2*, M-048 (1979) [Ref. 14].

To get a printed copy, log onto a CDC 7600 and enter

trix ac!print!nip urlib *ann id*

Use: URLIB is a utility-routine library containing many subroutines. The subroutines in this library can be used to receive input that is consistent with other utility routines.

LR

Document: K. O'Hair, *LR System User Manual*, LCSD-313, Draft (1985) [Ref. 15].

To get a printed copy, log onto a CDC 7600 and enter

trix ac!print!nip lcsd313 *ann id*

The LR system is a collection of processes and skeleton parser source-files that generate complete parsers for a grammar known as a Backus-Naur form (BNF). A parser is a code that accepts and decodes input languages or sets of commands.

Use: The LR system accepts a user interface defined in BNF and produces a source code that will decode the input of the user interface. The generated source-code is complete and can be used to test your interface or be included with some action routines, making it the permanent user-interface coding.

User-Interface Bibliography

R. W. Bailey, *Human Error in Computer Systems* (Prentice-Hall, Englewood Cliffs, NJ, 1983).

L. Borman and B. Curtis, Eds., *CHI'85 Conference Proceedings--Human Factors in Computing Systems*, Association for Computing Machinery Special Interest Group on Computer and Human Interaction (The Association for Computing Machinery, Inc., New York, 1985).

S. K. Card, T. P..Moran, and A. Newell, *The Psychology of Human-Computer Interaction* (Lawrence Erlbaum Associates, Hillsdale, NJ, 1983).

J. D. Foley, "The Design and Implementation of User-Computer Interfaces," *SIGGRAPH Tutorial* (Association for Computing Machinery, Inc., NY 1982).

J. D. Foley, "Managing the Design of User-Compiler Interfaces," *Computer Graphics World*,

T. Gilb, *Humanized Input* (Winthrop Publishers, Cambridge, MA, 1977).

J. D. Grimes and H. R. Ramsey, "Psychology of User-Computer Interfaces," *SIGGRAPH Tutorial* (Association for Computing Machinery, Inc., NY, 1983).

B. Huckle, *Man-Machine Interface* (Savant Research Studies, Carnforth, Lancashire, England, 1983).

C. Wetherell and A. Shannon, *LR Automatic Parser Generator and LR(1) Parser*, Lawrence Livermore National Laboratory, Livermore, CA, UCRL-82926 (1979).

C. Wetherell and A. Shannon, "LR-Automatic Parser Generator and LR(1) Parser," *IEEE Transactions on Software Engineering* SE-7(3), 274 (1981).

DESIGN

Overview

The design phase of software development involves transforming the project definition (PD) into a design packet (DP). The PD can be thought of as what is needed and the DP as a plan for achieving how the software will accomplish the defined task. The design phase can also be thought of as the solution phase, i.e., someone states the problem (PD) and a solution is described (DP). The design is, thus, a description of how the software will be implemented--it does not include the coding. For information on coding, see page 71 in this document.

In this chapter, we will describe guidelines for developing a design based on the principles of structured design supported by the Yourdon group, specifically by Page-Jones [Ref. 16]. There are other acceptable ways to specify a design. Refer to the Design Bibliography on page 69 for material on other approaches.

First, we will present guidelines for preparing a DP and for its preview and review. The preview and review, referred to by Page-Jones and here as the design walkthrough or walkthrough, are considered necessary for every design. Second, we will give a simple example of a DP and its walkthrough. Third, we will outline some tools to support generation of the DP.

Design Guidelines

Input to the design phase is the PD, and the results are an approved DP. The package may contain such details as data-flow diagrams, a data dictionary, structure charts, and a pseudocode in natural language or narrative prose (refer to the glossary on page 137). The exact contents of the DP are usually specified and deemed necessary by the walkthrough group at the preview. Major changes in the package or in design, such as new specifications or correction of errors discovered after the design phase review, require that the walkthrough group reevaluate the entire DP.

The design walkthrough-group may consist of one person (not an author) or as many as seven people, depending on the importance and size of the project. The group leader or functional manager should designate the group size.

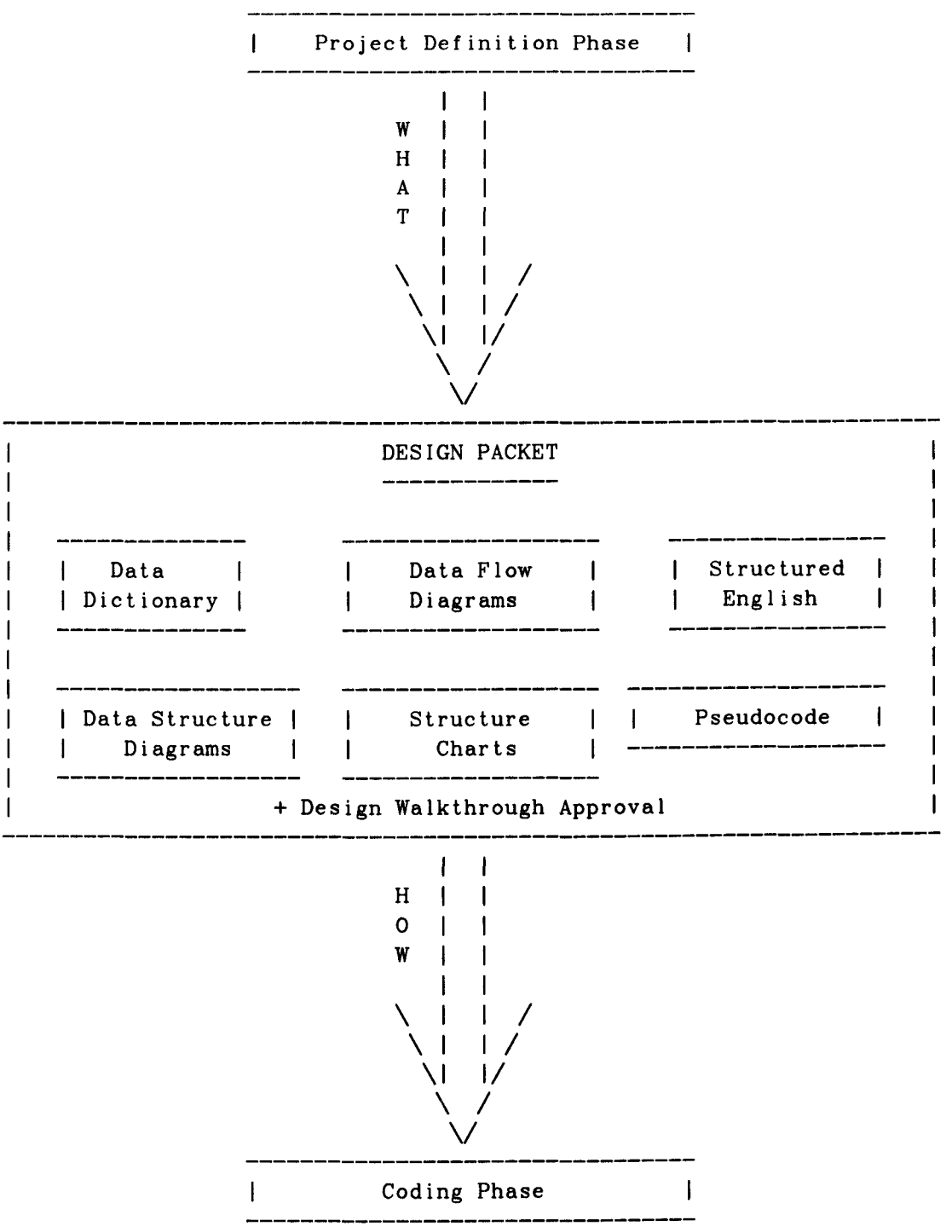


Figure 2. Design-phase flow diagram. The design phase could also be called the solution phase, i.e., someone states the problem (Project Definition) and a solution is rendered (Design Packet).

Data-Flow Diagrams

A data-flow diagram (DFD) is a network representation of the problem and solution, showing the active components and their interfaces. DFDs are used to partition the system into manageable pieces. A DFD is also known as a bubble chart, because the processes are represented as circles (bubbles) where data flows into a process, is transformed, and flows out of the process. A DFD should represent the data flow of the original problem as much as possible.

The movement of data between processes is represented with arrows going from one process to another (see the diagram in the example on page 61). The movements are one-way paths. The bubbles of the DFD are the processes that accept data flowing in and send data out to other processes or sinks. The originator of the data is the source, and the receiver of the data is the sink.

There is also a data store or storage place where no transformation of the data is performed. In the example on page 61, the data store isn't shown; however, it's usually drawn as a rectangle with no lateral sides, only horizontal lines.

DFDs must be kept simple and uncluttered; for this reason, you should always level your diagrams so that each diagram has between five and nine processes. Leveling a DFD means that each process or bubble in a chart is really a DFD in itself. By limiting the number of processes in each individual DFD, you keep the overall picture simple, and details are kept in the lower-level DFDs. You will always have one master DFD that describes the entire system, possibly one DFD for each process in the master DFD, and possibly even more DFDs for the processes in the second-level DFDs.

All data flow must have an origin--that is, a process, source, or data store--or be considered input to the DFD. All data flow must also have a destination--that is, a process, sink, or data store--or be considered output from the DFD.

Data Dictionary

The design data-dictionary lists the different data-element types that flow through your system and are seen in flow diagrams. Therefore, the data-flow diagrams need a data dictionary, and these two go hand-in-hand. The data-structure diagrams are used to describe complex data elements in the data dictionary.

For a more general definition of a data dictionary, refer to the glossary on page 137.

Structured English

Structured English is simple English with short sentences and well-understood verbs. English is a very complicated language and by limiting yourself to simple sentences, eliminating adjectives and adverbs whenever possible, and using elements from the data dictionary to describe the processes in the data-flow diagram, you are using structured English.

Use structured English to describe the purpose of each process in your data-flow diagram. In the data dictionary, use it to describe how the different data elements are used.

Data-Structure Diagram

The data-structure diagram graphically details the contents of complex data items in the data dictionary. The diagrams describe the various fields and components of data elements in terms of understood standard-data-components, i.e., integers, reals, words, bytes, strings, characters, etc.

Structure Charts

Structure charts illustrate how a system is partitioned into independent modules or black boxes and the relationships among all modules in the system. Thus, structure charts show the overall structure of a program plus the hierarchy of the program in terms of modules with well-defined tasks and interfaces.

The three basic elements of a structure chart are: the module, the communication, and the connections. The module of a structure chart is an independent piece with a well-defined function and interface. The connections in a structure chart represent the calling of one module from another--the communication--and define the system hierarchy.

Communication between modules can be accomplished with couples. In the example on page 64, there are two types of couples: control and data. Data couples represent communication of a data entity, i.e., module A passes module B the data I. Control couples represent control communication, i.e., module A calls module B, and B returns an error flag.

Types of Coupling

There are other types of coupling modules, and each determines how independent the modules are. There are some control coupling modules, but most are data coupling. Several types are listed below and are ranged from bottom to top as bad to good. Note that ordinary I/O routines to open, create, read, write, and close files may imply common or content coupling.

	Type of Coupling	Meaning
Good	Data	Communicating through parameters only. Modules must be at least data coupled or they are not communicating. This is the way true black boxes communicate, they are passed only the necessary information to do their job and return only what they are supposed to return, nothing more.
	Stamp	Communicating with larger data elements than necessary. Information overkill, i.e., passing an entire record when only one field is needed.
	Control	Communicating information that determines the action to be performed by a particular module; for example, passing the parameters IOPT, A, and B to a routine. When IOPT equals 1, it does one thing, A=A+B. When IOPT equals 2, it does another, A=A*B.
	Common	Communicating through global data area. Any use of FORTRAN COMMON is a good example of this.
	Content	Communicating through the state of the module; for example, how many times a module has been called. In most FORTRANs the values of local variables are retained from call to call. If a FORTRAN routine relied on this information, it would be content coupled. This is sometimes called state memory.
Bad		

Types of Cohesion

How a module is put together and how its internal tasks relate is defined as cohesion. In the chart below, the types of cohesion are described and are ranged from from bottom to top as worst to best.

	Type of Cohesion	Meaning
Best	Functional	Modules relate functionally. There is true black-box interaction.
↑	Sequential	Modules relate in a sequential manner. A definite order is required in the calling of modules. For example, Initialize, Terminate, ...
|	Communicational	Modules relate through some common input or database. For example, Clear record, Fill record, ...
|	Procedural	Modules relate only in that they help to perform a procedure; each performs partial actions of the procedure.
|	Temporal	Modules relate only because they operate in the same time frame.
|	Logical	Modules perform many different actions in a similar manner, but with a different result for each action.
↑	Coincidental	Modules that perform many different actions in different ways, and these actions do not relate at all.
Worst		

Pseudocode

Pseudocode is an informal program-like notation containing natural-language text. It is used to describe and clarify the functioning of a procedure or program. Pseudocode is used as a design aid, and some people believe it to be better than a flow diagram, because it allows for easier top-down (basic to more complex) design. When compared with structured English, pseudocode is more detailed and more code-like.

Design Walkthrough and Review

The design walkthrough is a review of a design by a small group of people. In the walkthrough, the reviewers try to locate any problems before the actual coding begins. Errors discovered during the design phase are much easier to correct than those discovered in later phases.

There are some basic rules for design walkthroughs.

- Keep it short
- Find errors, not corrections for errors
- Review the product, not the authors
- Use between one and seven reviewers
- Do not invite managers--however, they should be should be kept informed
- Look for a willing outside reviewer to get a new point of view
- Keep it as informal as possible
- Don't get hung up on details; let the authors iron out the details

For each project, you should have a checklist of common items to go over in the walkthrough. Refer to *The Practical Guide to Structured Systems Design* [Ref. 16], page 297, for a sample checklist. Below, are a few items for a possible checklist; however, each design will probably need a different one depending on what the goals of the system are. For example, if you are concerned about portability more than anything else, your list should include portability checks first.

Sample Checklist

- Is each interface between modules implemented cleanly?
- Can each module actually be implemented, with its given interfaces? In other words, are all the necessary calling parameters present?
- How is error reporting handled?
- Does any module have unnecessary state memory?

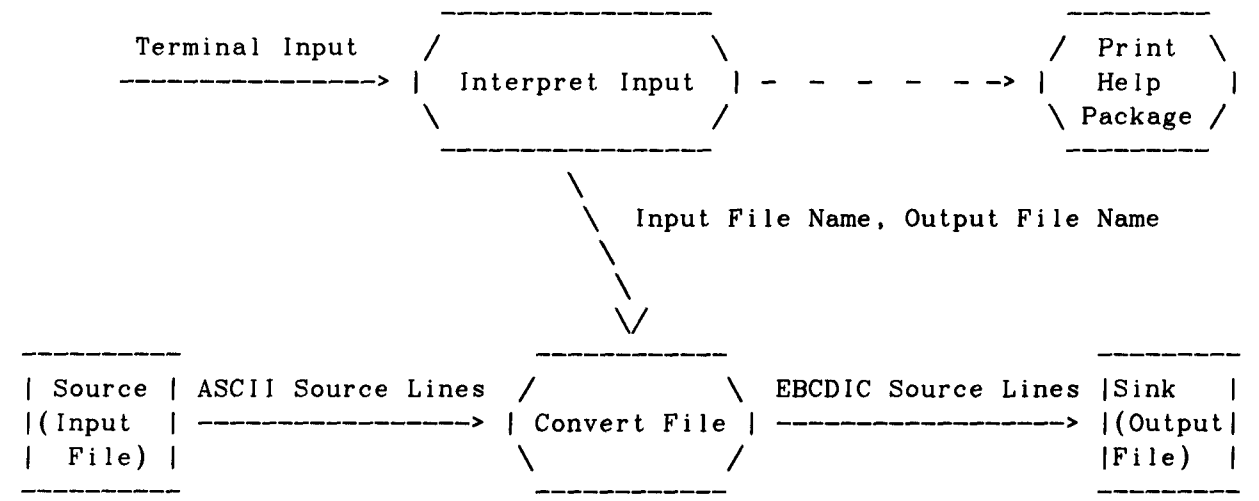
- Are any modules over-general?
- Lastly, WILL IT WORK?

If a major flaw is discovered in the design, the author would be required to have another walkthrough. The members of the walkthrough group would determine what constitutes a major flaw.

Example D1. ASCII to EBCDIC File Converter

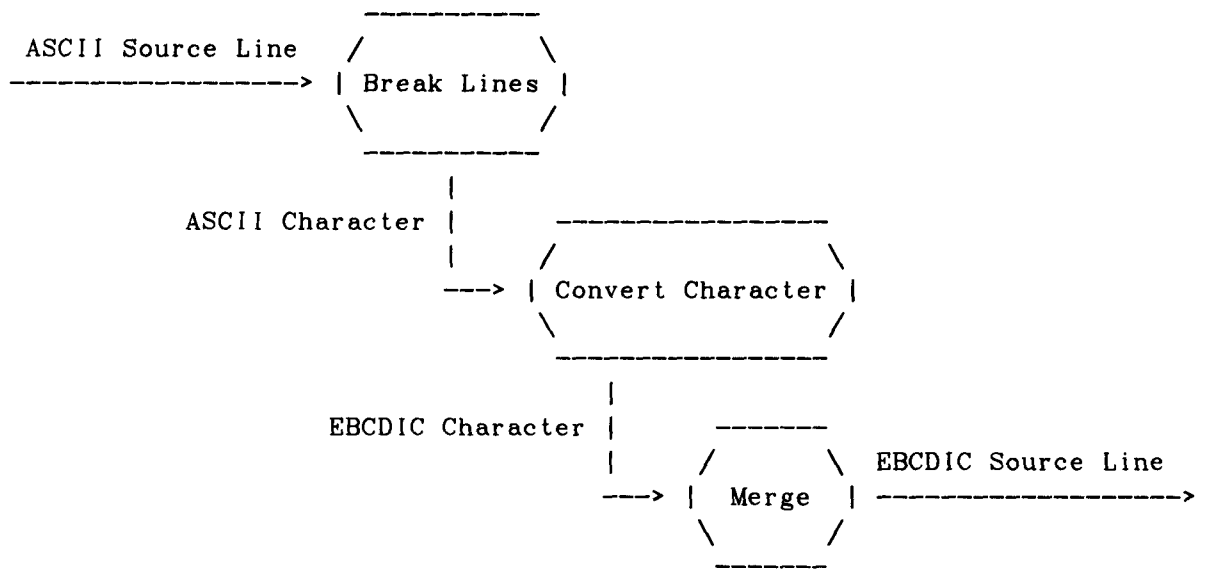
The Data-Flow Diagram

An Overall View



Convert File

The convert file may be further broken down into:



Design/Design Example/ASCII to EBCDIC File Converter

Data Dictionary

Terminal input	Characters received from user's terminal.
File name	System-dependent format for name of a file.
ASCII source line	String of characters in ASCII code that represent a text line from a file.
EBCDIC source line	String of characters in EBCDIC code that represent a text line from a file.
ASCII character	One ASCII character.
EBCDIC character	One EBCDIC character.

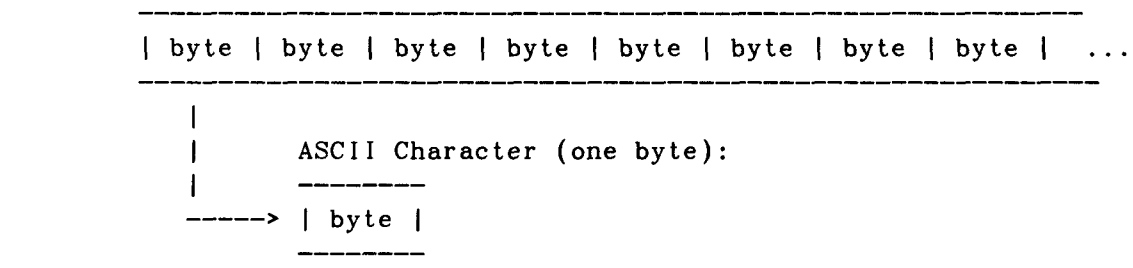
Structured-English Definitions

Interpret input	Interprets the terminal input line.
Print help package	Prints the help package at the user's terminal.
Convert file	Translates ASCII source lines into EBCDIC source lines.
Break lines	Breaks up ASCII source lines into ASCII characters.
Convert character	Converts an ASCII character into a EBCDIC character.
Merge	Merges EBCDIC characters into EBCDIC source lines.

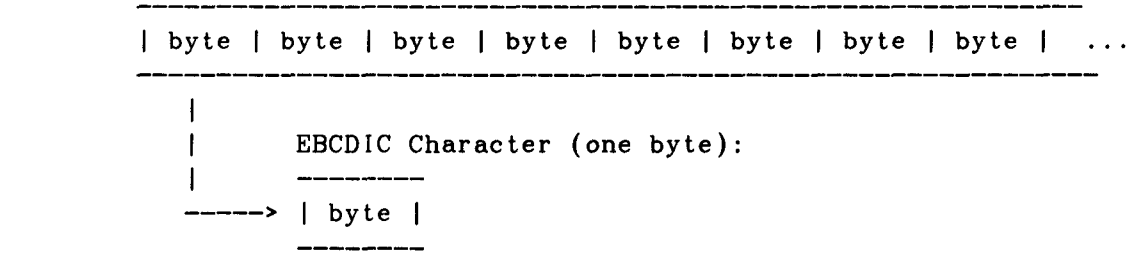
Design/Design Example/ASCII to EBCDIC File Converter

Data-Structure Diagrams

ASCII Source Line (stream of bytes)



EBCDIC Source Line (stream of bytes)



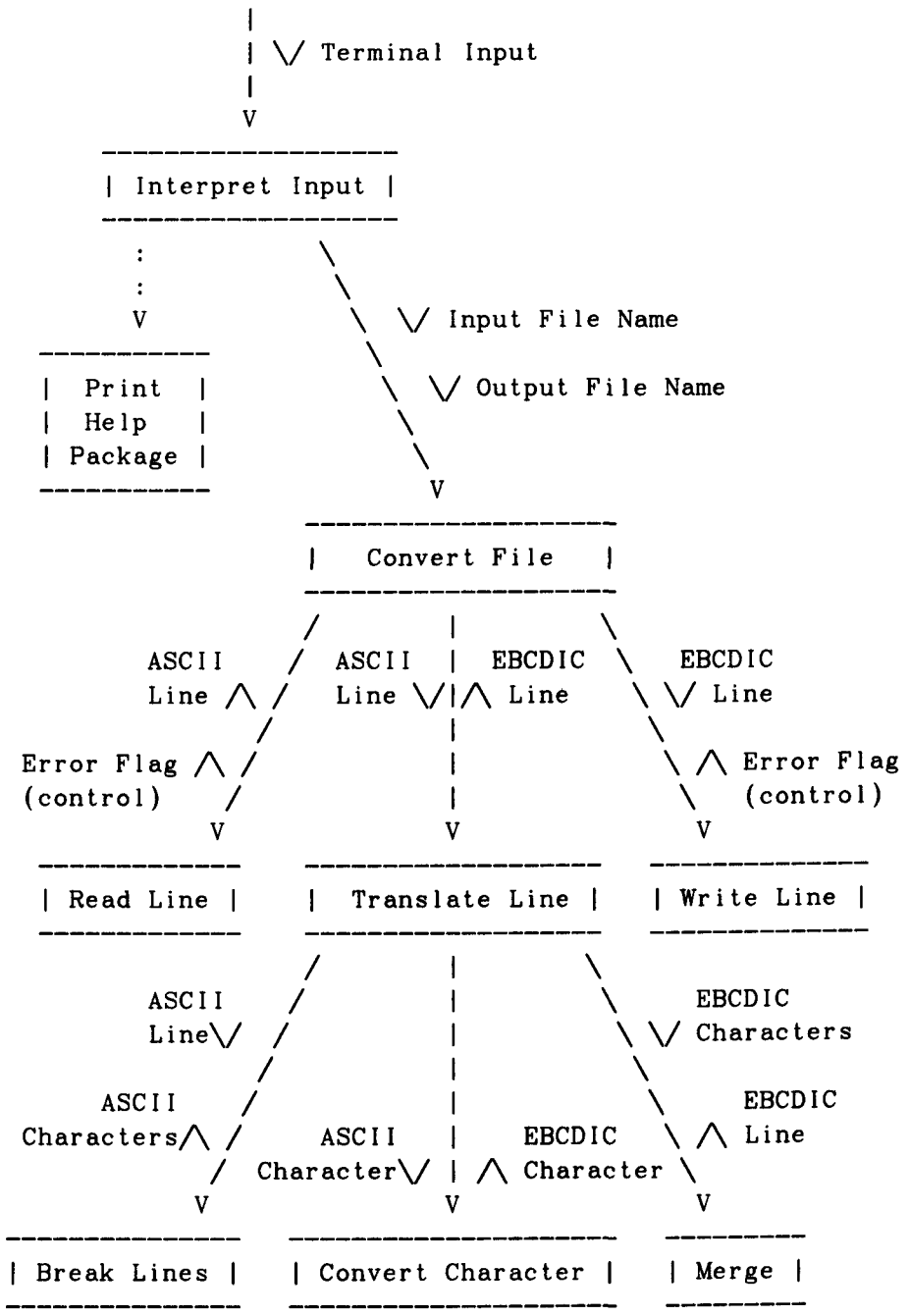
Structure Charts

The communication between modules is done with couples of two types: control and data. Data couples represent communication of a data entity, i.e., module A passes module B the data I. Control couples represent control communication, i.e., module A calls module B who returns an error flag.

In this system, most of the modules show functional cohesion; however, the Read Line and Write Line modules more likely show communicational cohesion (refer to page 58).

What coupling the module has will determine how independent the module is. Here, there is some control coupling, but there is mostly data coupling. Note that the use of standard I/O routines to open, create, read, write, and close files may imply common or content coupling.

Design/Design Example/ASCII to EBCDIC File Converter



∧ = data being sent up

∨ = data being sent down

Pseudocode

```
Procedure Interpret Input ( terminal-input )
  Break terminal-input into symbols
  If ( more than 2 symbols ) Then
    Generate error: Too many symbols on input line
    Print Help Package
  Else
    filein = first symbol
    If ( filein doesn't exist) Then
      Generate error: Input file doesn't exist.
      Print Help Package
    Else
      fileout = second symbol
    If ( fileout isn't present ) Then
      Create name for fileout
      Destroy fileout if it exists
      Convert File ( filein , fileout )
    EndIf
  EndIf

Procedure Convert File ( filein , fileout )
  Do till End Of File ( filein )
    Read ASCII Line from filein ( Error Flag )
    If ( Error Flag is positive ) Then
      Generate error: Cannot Read Input File
      Return
    EndIf
    Translate Line ( ASCII Line , EBCDIC Line )
    Write EBCDIC Line to fileout ( Error Flag )
    If ( Error Flag is positive ) Then
      Generate error: Cannot Write Output File
      Return
    EndIf
  EndDo

Procedure Translate ( ASCII Line , EBCDIC Line )
  Break Line ( ASCII Line , ASCII Characters )
  Do For Each Character in ASCII Characters
    Convert ( ASCII Characters[i] , EBCDIC Characters[i] )
  EndDo
  Merge ( EBCDIC Characters , EBCDIC Line )
```

Design/Design Example/ASCII to EBCDIC File Converter

```
Procedure Break Lines ( ASCII Line , ASCII Characters )
  Do For Length of ASCII Line
    Store Character into ASCII Characters array
  EndDo

Procedure Convert Character ( ASCII Character , EBCDIC Character )
  Define Table to be array mapping ASCII to EBCDIC.
  EBCDIC Character = Table ( ASCII Character )

Procedure Merge ( EBCDIC Characters , EBCDIC Line )
  Do For Each Character in EBCDIC Characters
    Store EBCDIC Character into EBCDIC Line
  EndDo
```

Walkthrough

All the above design information and the testing information (see page 110) were gathered together into a design packet (DP) and delivered to each member of the prechosen design-walkthrough-group one week in advance of the walkthrough date. At the walkthrough it was pointed out that

- The Input file was not opened and the Output File was not Created. These files also need to be Closed.
- Read Line and Write Line will probably need more input, possibly a unit specifier or a file name.
- An additional functional test would be to submit the file to a utility routine that expects EBCDIC input.

These changes were considered minor and would be made by the author. The walkthrough group approved of the design unanimously.

Design Tools

Available Tools

There are not many tools available for the DP. Unfortunately, what few tools there are don't work very well together.

The biggest help for the DP are the many usable, or reusable, routines available through libraries such as BASELIB, URLIB, FORTLIB, MATHLIB, STACKLIB, GRAFLIB, GRAFCORE, etc. A little research into what is available in these libraries may eliminate the need for you to design many modules, since the modules you need may already exist in the public libraries. These libraries aren't really helping you design as much as making your designing job simpler by eliminating some of the module design you have to do. Take advantage of this reusable software.

The available tools listed in the Project Definition chapter (page 29) are also useful for generating modules in the DP. Refer specifically to the structure-chart program SCI [Ref. 8] on page 31 of this document.

In addition to the tools listed previously, is the VW command in TRIX AC.

TRIX AC, VW Command (A screen editor for a general text editor)

Document: A. Cecil, H. Moll, and J. Rinde, *TRIX AC--A Set of General-Purpose Text-Editing Commands*, LCSD-808, Draft (1985) [Ref. 17].

To obtain a copy of this document, log onto a CDC 7600 and enter

trix ac!print!nip lcsd808 box *ann id*

Use: The VW command in TRIX AC is a screen editor. TRIX AC is available on all LCC machines at LLNL, and the VW command can be called up on HP terminals or on HP terminal-emulators. All the diagrams in this chapter were generated using the VW command.

Design/Tools/Ideal Tools

Ideal Tools

The ideal tool for the DP is one that accepts output, in machine-readable form, from the PD and automatically generates data-flow diagrams, a data dictionary, data structures, and structure charts, also in machine-readable form. However, more realistic tools might be ones that

- Accept DFDs from the PD phase and allow you to modify, create, and store machine-readable DFDs.
- Allow the user to create a data dictionary that integrates into the DFD tool and the structure-chart tool. This tool could also keep data structure information.
- Generate structure charts from DFDs automatically and also allow the user to modify these charts, create new ones from scratch, and store all these charts in machine-readable form.
- Generate skeleton pseudocode from structure charts and allow for modifying and storing this pseudocode. A similar tool could also generate skeleton code for the coding phase.
- Use a general database program that keeps all your design information together, verifies interfaces, notes inconsistencies, and keeps records. This tool could replace much of the work done by the members of the walkthrough group.

These tools are, of course, all interactive, fast, of high quality, etc.

Design Bibliography

T. DeMarco, *Concise Notes on Software Engineering* (Yourdon Press, New York, 1979).

G. J. Meyers, *Software Reliability--Principles and Practices* (Wiley-Interscience Pub., New York, 1976).

E. Yourdon, *Structured Walkthroughs* (Yourdon Press, New York, 1978).

E. Yourdon and L. L. Constantine, *Structured Design* (Yourdon Press, New York, 1978).

CODING

Overview

A software programmer's goal is to prepare software that can be read and understood by people as well as by machines. If a code is to continue to perform its function correctly and efficiently, the people maintaining the code must be able to understand it and change it readily. Software should be clear, readable, and consistent. If it is, the software itself will be one of the programmer's best means of documentation. There are several ways to prepare software and several things to include in the documentation to achieve these goals.

The structured-programming method enables programmers to prepare a code easily. If the structured code is then reviewed and the suggested changes incorporated where possible, there should be fewer than usual logic errors discovered during the testing phase.

A well-structured self-documented modular code can be easily understood and revised. When changes are necessary, and if the modules were constructed using the structured-programming method, the software can be kept in its original healthy and robust state.

Some documentation should be a part of the comment information in with the program. These comments should include where to obtain the source file and what compile and load procedures to use. Also, if any special tools are required, include information on where to get the tools and where to get information about using them. In addition, standards and assumptions detailed in the project definition should be included if relevant, because they are considered part of the project documentation.

In this chapter, we will first present coding guidelines that include information on naming conventions, commenting, and variable declarations followed by material on the code body, coding reviews, and maintenance; second, we give examples of codes that adhere to these guidelines; and third, we will outline available and ideal tools for writing and maintaining software.

Coding Guidelines

It has been written that quality software is not only reliable and correct but also easy to use and maintain [Ref. 18]. The key to achieving this is to provide tools and techniques that can be used to improve the process of software production. A side effect of quality software-production is the reduction of support costs.

Naming Conventions

A routine name or a variable name should be as descriptive as possible so the users can easily recall its purpose. Once the purpose has been defined, use the routine or variable only for that unique purpose.

The first or last few characters of a variable name could be special symbols and form a naming structure. Use consistent, well-documented naming structures to denote such things as whether the variable is defined locally, is an input or output argument, or is from a common block. Naming structures can also denote type--for example, integer, real, logical, character, scalar, array, or vector, etc. For an example of naming structures, see Example C3 on page 87.

A comment explaining each name is sometimes necessary, and always helpful. Descriptive names in a code help make the code self-documenting in an easy way.

Comments

Comments are used to help anyone reading the code understand the details of the implementation. The author of the code may forget the details over time and may need to be reminded of them when rereading the code, while others will probably be reading it for the first time. To aid the user, you should write consistent and meaningful comments. A way to provide consistency between modules is to have a prologue for each module.

Prologue

The prologue should consist of comments giving the module name, calling form, and expanded module name. In addition, the purpose should be stated as well as the author and date. There should be a description of input and output and information on modifications--i.e., the author, date, and reason for the modification. It would be helpful if the code reviewer(s) was named and there was information about the algorithm, limitations, restrictions,

and assumptions. A description of anything unusual in the module would also be useful.

Many individuals have written their own tools to assist in producing a prologue, and we hope to have a similar division-wide tool soon. See Example C2 (page 84) for a prologue produced from the TEMPLATE tool. See Examples C3 and C4 (pages 87 and 90, respectively) for two manually generated prologues.

Comments Among Code Lines

When writing variable declarations, include a comment for each variable stating its expanded variable name and purpose. If the variable purpose is known, the reader will better understand where the variable occurs and how it is used. Preceding each logical block of code, include a block of comments describing the purpose of that block and its expected input values and resulting output values. This aids reading the information by physically separating each logical block.

In general, remember to avoid comments that restate the lines of code. Aim for clarity, not redundancy--a comment should logically state what is being done, not how it is done. It is unnecessary to comment on every line of code, and doing so usually has the effect of making the code less readable. However, you should use comments to explain anything not obvious from reading the code.

Variable Declarations

Every variable should be declared with an explicit variable-type statement. In LRLTRAN [Ref. 19], you are encouraged to use the **implicit none**, because it points out those variables not explicitly declared. Note: **implicit none** is <u>not</u> standard Fortran. However, <u>do not use</u> the **all integer** statement in LRLTRAN. This statement allows you <u>not</u> to declare variables and allows errors to creep in that are difficult to detect. Explicit declarations help the user read the code, and they give the programmer the perfect opportunity to comment on the meaning of each variable name and its intended use.

You should order module arguments uniformly to specify input, output, and input-output arguments by their positions within the argument list. Precede all other type declarations with the type declarations for the argument list, preferably in the order they appear as arguments. If global and local variables are being used, you should make that apparent by the order of their type declarations or descriptions. This will give more

information at a glance to the code reader and make variable declarations easier to locate.

The Code Body

The code body is that part of the code after the declarations. It consists of executable statements and should be separated into distinct and logical blocks. Each block should have a purpose and, usually, some input and output values. The blocks should be separated with blank lines or blank comments. The separated blocks of code will enhance readability and show the sequential flow of the program. Each block of code should be no more than one page in length.

Indent and Nest

To help improve readability, you should indent block constructs (**if-then-else**, **do**, **case**, etc.) and any other logical block. If you indent uniformly, you can visually represent the logic of the program.

You shouldn't nest block constructs too deeply; the deeper the nesting, the more difficult the code is to follow. No more than three to four levels of nesting should be considered, in the interest of writing a more easily understood piece of code.

Loop and Go-To Statements

Every loop statement (**do**, **repeat**, etc.) should have its own corresponding endloop statement (**continue**, **until**, etc.) for terminating the loop. More specifically, in Fortran each **do** statement should refer to a unique **continue** statement. If this is done, it will be easier to find the beginning and the end of each loop, and the number and nesting level of the loops will be obvious.

go to statements should be used sparingly and eliminated unless really needed. This should keep the program flowing from top to bottom smoothly; because it's easier to read from top to bottom, rather than back and forth.

Macros and Cliches

Macros and cliches are powerful tools for pieces of code that will be expanded in several places within a program. They are very convenient for

segregating all machine dependent parameters or packaging declaration statements to be uniformly used in many places within a program, for example.

A macro or cliche should consist of a collection of related statements, or perform a distinct action. Some thought should be given to whether that action would be more appropriate in a subroutine or function. Misuse of macros and cliches can result in code that is choppy and difficult to read, debug, and maintain.

Module Length

A module should not be too long or too complex; about two pages of code should be the maximum length. There should be only one entry to a module, and when possible there should be only one exit. This will make testing, debugging, and maintenance much easier because the program flow will be easier to see.

A Transportable Code

Try to adhere to the standards in the programming language you will use. This will help you achieve a fully transportable code. Put machine or environmental dependencies in one place to make finding and changing these things easier as the code is transported.

Coding Reviews

The coding should be reviewed, formally or informally, by at least one person. Although you're familiar with the code and know what it does, someone else should be able to read it and infer what it does. That is, another person should read the coding, understand it, and be able to add to it or modify it without difficulty; because in the life cycle of most codes, several people will work on the code you're now writing.

A reviewer should verify the program and see that the standards specified within the project have been followed. If you, as the coder, decide that it makes good sense to do something other than what the standards specify, the reviewer should confirm your judgment. Refer to *Structured Walkthroughs* by E. Yourdon [Ref. 20] for information on holding a code review or walkthrough.

Coding walkthroughs can benefit the reviewers as well as the coders. For example, reviewers are exposed to different coding styles, and errors

and standards that were inadvertently overlooked can be corrected before the program is made public.

Maintenance and Modification

Maintenance provides an opportunity to better your code. Ideally, before modifying any module, you should be certain you understand the module and all its side effects. Realistically for an old code, this may not be possible. Therefore, it is very important to document all changes and to test them thoroughly. Modifications should always make the module clearer--that is, more understandable--and should be consistent with the rest of the code.

You should implement small modifications very carefully and one at a time. For large modifications, consider the possibility and time effectiveness of rewriting entire modules. Document anything you had trouble understanding when you first worked with a module so that next time someone works with it, this module will be easier to understand and modify.

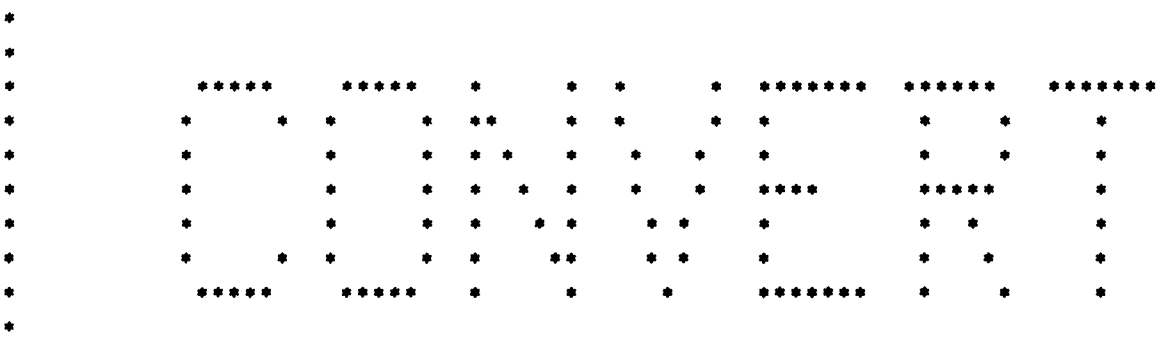
Module Modification

When a module is modified, the comments and prologue information must be checked to update any outdated or incorrect information. The modification must be documented in the prologue with the name of the modifier, the date, and a brief description of any changes.

Coding/Examples/ASCII to EBCDIC File Converter

Coding Examples

Example C1. ASCII to EBCDIC File Converter



```
* This program converts a Cray 8-bit packed ASCII file to a packed EBCDIC
* file. The program consists of a main code and three subroutines (getfiles,
* helppkg, and translte).  The compile line is
*
*                    CIVIC P=SCONVERT X=CONVERT
*
* It was programmed by Jeanne Martin - 9/85.
*
* To use the program, type one of the following:
*
*                        CONVERT
*                        CONVERT HELP
*                        CONVERT input-file-name
*                        CONVERT input-file-name output-file-name
*
* If no output-file-name is supplied, the name of the output file will be
* ?$ebcdic, where ? is the current suffix.
*
* The input file is assigned to unit 2.
* The output file is assigned to unit 3.

      implicit none          $$$ all variables are declared

* Local Variables

      integer line(50)       $$$ working buffer (holds one line)
      integer nochar         $$$ no. of chars in input line
      integer noretc         $$$ no. of chars in output line
```

Coding/Examples/ASCII to EBCDIC File Converter

```
* Executable Code

      call dropfile(0)

* Obtain the input and output file names and associate them with units 2
* and 3 respectively.

      call getfiles

* Process each line

      do
        call rdline (2,line,nochar)
        if (nochar.eq.-1) exit              $$$ test for end of file
        call translte (line, nochar,noretc)
        call wrline (3,line,noretc)
      repeat

* Terminate the program

      call close(2)
      call close(3)
      call exit
      end
*
      subroutine getfiles
*
*
*        ******  *******  *******  *******  ***  *        *******   ******
*       *        *           *     *         *   *        *        *
*       *        *           *     *         *   *        *        *
*       *   ***  ****        *     ****      *   *        ****      *****
*       *     *  *           *     *         *   *        *              *
*       *     *  *           *     *         *   *        *              *
*        ******  *******     *     *        ***  *******  *******  ******
*
*
* Purpose:
*   This is the user interface.  It interprets a command from the user, or
*   controller.  If no command is waiting, it calls a help pachage routine,
*   which supplies information and prompts for a command.  The command may
*   provide file names, ask for help (HELP) or terminate the program (END).
*   It opens the input file and associates it with unit 2. It creates the
*   output file and associates it with unit 3.  Error messages are sent to
*   unit 1, the terminal (or controller).
*
```

Coding/Examples/ASCII to EBCDIC File Converter

```
* Calling Form:  CALL GETFILES

      implicit none          $$$ all variables are declared

* Local Variables:

      integer msg            $$$ flag for waiting message
      integer line(50)       $$$ user command buffer
      integer len            $$$ no. of chars in user command
      integer files(10)      $$$ storage for symbols in user command
      integer type(10)       $$$ must be same size as files
      integer nfiles         $$$ no. of symbols in user command
      integer suffix         $$$ current running suffix
      integer j              $$$ placeholder for unused information
      integer i              $$$ flag for opened file
      data (i=0)

* Executable Code

      call msglink(1,1)              $$$ establish unit 1 as controller
      call msgflag(msg,2)            $$$ is a command waiting?
      if (msg.eq.0) call helppkg     $$$ if not, call help package
getmsg call msgfrr (line,len,200)    $$$ get command
      call mprompt(0,0)              $$$ turn off prompt
      call getsymb(files,type,nfiles,line,len,200)
                                     $$$ break command into symbols
      if(nfiles.gt.2) then
        write(1,"(""Too many symbols on input line"")")
        go to errret
      end if
      if(files(1).eq."help") go to errret
      if(files(1).eq."end") go to errx
      call open (2,files(1),0,i)             $$$ open input file
      if(i.lt.0) then
        write (1,"(""Cannot open "",a)") files(1)
        i=0
        go to errret
      end if
      if(nfiles.ne.2) then
        call userinfo(j,j,j,suffix)          $$$ create name for output file
        files(2)=" $ebcdic".un.suffix
      end if
      call create(3,files(2),2,-1,nocrea)    $$$ (destroy and) create
                                             $$$   output file
      return
```

Coding/Examples/ASCII to EBCDIC File Converter

```
*
nocrea write (1,"(""Cannot create "",a)") files(2)
errret call helppkg
      go to getmsg
*
errx  call exit
      end
*
      subroutine helppkg
*
*
*          *     *  *******  *        ******   ******   *    **   ******
*          *     *  *        *         *    *   *    *  *  **    *
*          *     *  *        *         *    *   *    *  ***       *
*          *******  ****     *         *****    *****   *  *      *   ***
*          *     *  *        *         *        *       *   *     *     *
*          *     *  *        *         *        *       *    *    *     *
*          *     *  *******  *******   *        *       *     *   ******
*
*
* Purpose:
*   To send help information to the controller and prompt for a command.
*
* Calling Form:  CALL HELPPKG

    write(1,"(/""This program will convert a LLNL ASCII file to an"")")
    write(1,"(""IBM 360 compatible EBCDIC file.  To run it type: ""/)")
    write(1,"(""                  convert name1 name2""/)")
    write(1,"(""where name1 is the name of your ASCII file and name2 is"")")
    write(1,"(""the name of the EBCDIC file you want created to contain"")")
    write(1,"(""the converted text. If name2 is omitted, a file will be"")")
    write(1,"(""created with the name ?$ebcdic, where ? is the current "")")
    write(1,"(""suffix.""/)")

    call mprompt ("filename(s)? ",13)

    return
    end
*
```

Coding/Examples/ASCII to EBCDIC File Converter

```
      subroutine translte (line,noac,noec)
*
*
*      *******  ******     ***    *     *   ******  *        *******  ******
*         *      *    *   *   *   **    *  *        *           *     *
*         *      *    *  *     *  * *   *  *        *           *     *
*         *      *****   *******  *  *  *   *****   *           *     ****
*         *      *  *    *     *  *   * *        *  *           *     *
*         *      *   *   *     *  *    **        *  *           *     *
*         *      *    *  *     *  *     *  ******   *******     *     ******
*
*
* Purpose:
*
*   To transform the characters in dummy argument line from ASCII characters
*   to EBCDIC characters.  Characters that are not representable in EBCDIC
*   are dropped so the number of characters input (NOAC) may be greater than
*   the number of characters output (NOEC).
*
* Calling Form:  CALL TRANSLTE (LINE,NOAC,NOEC)
*
* Input-Output Argument:
*
*     line - character buffer for a single line - contains ASCII line on
*            input and EBCDIC line on output
*            Note: There is a change of type across the subroutine boundary.
*
* Input Argument:
*
*         noac - no. of characters input (ASCII)
*
* Output Argument:
*
*         noec - no. of characters output (EBCDIC)

      implicit none           $$$ all variables are declared
```

Coding/Examples/ASCII to EBCDIC File Converter

```
* Dummy Arguments

      char line
      dimension line(132)
      integer noac
      integer noec

* Local Variables

      char table                $$$ conversion table
      dimension table(0:255)
      integer i                 $$$ loop index
      parameter (nil=#7f)       $$$ indicator for non-representable character

* Executable Code

* RDLINE detects records longer than 132 characters, so there is no chance
* NOAC will be > 132.

      noec=0
      do (i=1,noac)
        if (line(i).eq.nil) cycle    $$$ drop non-representable characters
        noec=noec+1
        line(noec)=table(line(i))
      repeat
      return
```

Coding/Examples/ASCII to EBCDIC File Converter

```
* Conversion Table

    data table(#00)/#00,#01,#02,#03,#37,#2d,#2e,#2f,#16,#05,#25,#0b,#0c,#0d/
    data table(#0e)/#0e,#0f/
    data table(#10)/#10,#11,#12,#13,#3c,#3d,#32,#26,#18,#19,#3f,#27,#1c,#1d/
    data table(#1e)/#1e,#1f/
    data table(#20)/#40,#5a,#7f,#7b,#5b,#6c,#50,#7d,#4d,#5d,#5c,#4e,#6b,#60/
    data table(#2e)/#4b,#61/
    data table(#30)/#f0,#f1,#f2,#f3,#f4,#f5,#f6,#f7,#f8,#f9,#7a,#5e,#4c,#7e/
    data table(#3e)/#6e,#6f/
    data table(#40)/#7c,#c1,#c2,#c3,#c4,#c5,#c6,#c7,#c8,#c9,#d1,#d2,#d3,#d4/
    data table(#4e)/#d5,#d6/
    data table(#50)/#d7,#d8,#d9,#e2,#e3,#e4,#e5,#e6,#e7,#e8,#e9,#ad,#79,#bd/
    data table(#5e)/#41,#6d/
    data table(#60)/#e0,#81,#82,#83,#84,#85,#86,#87,#88,#89,#91,#92,#93,#94/
    data table(#6e)/#95,#96/
    data table(#70)/#97,#98,#99,#a2,#a3,#a4,#a5,#a6,#a7,#a8,#a9,#8b,#6a,#96/
    data table(#7e)/#a1,nil/
    data table(#80)/#20,#21,#22,#24,#1a,#15,#09,#87,#88,#89,#04,#34,#36,#06/
    data table(#8e)/#08,#0a/
    data table(#90)/#35,#1b,#2b,#3b,#23,#28,#29,#2c,#30,#31,#33,#38,#39,#3a/
    data table(#9e)/#3e,#ff/
    data table(#a0)/#43,#4f,#fa,#44,#45,#46,#4a,#47,#48,#49,#51,#52,#53,#54/
    data table(#ae)/#55,#56/
    data table(#b0)/#57,#58,#59,#62,#63,#64,#65,#66,#67,#68,#69,#70,#80,#8a/
    data table(#be)/#c0,#bc/
    data table(#c0)/#8d,#8e,#8f,#90,#9a,#d0,#9c,#9d,#9e,#9f,#a0,#aa,#ab,#ac/
    data table(#ce)/#ae,#af/
    data table(#d0)/#b0,#b1,#b2,#b3,#b4,#b5,#b6,#b7,#ec,#ce,#b8,#b9,#ba,#bb/
    data table(#de)/#5f,#bc/
    data table(#e0)/#be,#bf,#ca,#cb,#cd,#cf,#da,#db,#dc,#dd,#de,#df,#e1,#ea/
    data table(#ee)/#eb,#ed/
    data table(#f0)/#71,#72,nil,#73,#74,#75,nil,#77,#78,#ee,#ef,#fb,#fc,#fd/
    data table(#fe)/#fe,#07/
    end
```

Coding/Examples/Subroutine pop

Example C2. Subroutine pop (stkptr, item, rtn)

Note the use of **template** . For a description of the **template** tool look in the section on Code Tools, page 95.

```
c
c  Name --          pop        - pop from a stack
c
c  Function --      to pop an item out of a stack.
c
c  Calling Form -- call pop (stkptr, item, rtn)
c
c  Author, Date -- John Doe (03/17/83)
c
c  Modified by --  Jane Doe (04/09/83) Fixed bug - Updated stack pointer
c
c  Input --
c     Arguments -
c         stkptr - Pointer into stack within scm array
c
c     Common Blocks -
c         common /parstk/ - Parser stack
c            pdata - Offset from stack (stkptr) of first word of data.
c            plen  - Offset from stack (stkptr) of length of item.
c            pnum  - Offset from stack (stkptr) of number of items.
c            stack - The scm dynamic workspace containing the stack.
c
c  Output --
c     Arguments -
c         item - The item just popped out of the stack, if successful
c         rtn  - The return indicator.
c                = 0 , ok
c                = -1 , stack underflow
c
c     Common Blocks -
c         common /parstk/
c            Same as common /parstk/ input (see above)
c
c  Algorithm --
c       1.  Check to see if the stack is in an underflow state.
c       2.  Copy the top item off the stack into the given "item".
c       3.  Decrease the number of items in the stack.
c
c       Note : A stack is a one-dimensional arry and its header
c              contains the information about its type, size, and
```

Coding/Examples/Subroutine pop

```
c               the current stack pointer
c
c  ------------------------ Declarations --------------------------
c
c
c  Arguments --
c
      Integer stack, item (1), rtn
c
c  Local Variables --
c
       integer length, numit, offset, i
c          length - Length of each item.
c          numit  - Number of items.
c          offset - Stack pointer for first piece of stack item.
c          i      - Loop index variable.
c
c  Common --
c
       integer pdata, plen, pnum, stack
       common /parstk/ pdata, plen, pnum, stack(1)
c
c -------------------- Start Execution -------------------------
c
c               Get number of entries in the stack.
c
       numit = stack(stkptr + pnum)
c              Check for underflow condition
       if  ( numit .le. 0 ) then
           rtn = -1
       else
c
c              Extract the last item and decrease
c              the number of entries by one.
c
           length = stack(stkptr + plen)
           numit = numit - 1
           offset = stkptr+pdata + numit*length
c
           do 1600 i + 1,length
              item(i) = stack(offset + i-1)
 1600      continue
c
```

Coding/Examples/Subroutine pop

```
c               Store new number of entries.
c
            stack(stkptr + pnum) = numit
            rtn = 0
c
         endif
c
c
         return
         end
```

Coding/Examples/Subroutine ttyread

Example C3. Subroutine ttyread (iprompt, iplen, iline, ilen)

Note the use of naming structures and a pic prologue that is manually generated.

```
c ********************************************************************
c
c  *******  *******  *     *  ******   *******    ***    ******
c     *        *      *   *    *    *  *         *   *    *     *
c     *        *       * *     *    *  *        *     *   *     *
c     *        *        *      *****   ****     *******   *     *
c     *        *        *      *  *    *        *     *   *     *
c     *        *        *      *   *   *        *     *   *     *
c     *        *        *      *    *  *******  *     *  ******
c
c
c> Author: Kelly O'Hair, LLNL Computer Graphics Group
c          L-73, (415)-422-4296, BLDG 116, Room 2645.
c
c> Purpose:
c   Read message from TTY. If message already there then
c   do not prompt, just return message. If no message is
c   there then use prompt supplied and return message.
c   It is assumed that the array nline is long enough to
c   hold the maximum length message from the tty.
c
c> Calling Form:
c     call ttyread (iprompt, iplen, iline, ilen)
c
c> Input Arguments:
c      iprompt     ASCII text of prompt
c      iplen       length in characters of text in iprompt
c
c> Input Common:
c    g4mess:
c      kames     array containing current message from TTY
c      klen      length of kames (if 0 then no message)
c      koffset   offset into kames (0-origin) of next
c                logical message
c
c> Output Arguments:
c      iline     ASCII text of message read from TTY
c      ilen      length in characters of text in iline
c
c> Output Common:
c    g4mess:
```

Coding/Examples/Subroutine ttyread

```
c       kames    array containing current message from TTY
c       klen     length of kames (if 0 then no message)
c       koffset  offset into kames (0-origin) of next
c                logical message
c
c> Notes:
c       Common g4mess is local to this routine.
c
c ***************************************************************************

c****** get machine dependent parameters from global macro
c******        pline        maximum word length of tty message
c******        peol         left-justified end of line (LF)
c******        pleol        length in characters of pleol
c******        peom         left-justified end of message (EOT)
c******        pleom        length in characters of pleom
c******        pnchw        number of characters per machine word
      macro mparams(spic)
      use mparams

c****** declare arguments
      integer iprompt , iplen , iline ,ilen
      dimension iprompt(pline) , iline(pline)

c****** declare locals (g4mess is a local common block,
c******           only used by ttyread)
c****** put outstanding messages into common g4mess
c******   kmess is the array containing the outstanding messages
c******   klen is the current character length of kames
c******   koffset is the current 0-origin character position
c******           of the next logical message out of kmess
      dimension kmess(pline)
      integer kmess,klen,koffset
      common/g4mess/klen,koffset,kmess
      integer jeol , jbytes

c****** declare baselib function integer
      integer zskeybyt

c****** data load messages
      data klen /0/
      data koffset /0/

c****** initialize returned message
      call zvxmits ( iline , " " , pline )
      ilen=0
```

Coding/Examples/Subroutine ttyread

```
c****** see if we have any messages already in the buffer
 lmes continue
      if(klen.gt.0)then
        jeol=zskeybyt(kmess,koffset,klen-koffset,peol,pleol)
        call zmovechr(iline,0,kmess,koffset,jeol)
        ilen=jeol
        koffset=koffset+jeol+pleol
        if(koffset.ge.klen)then
          klen=0
        endif
        go to rtn
      endif

c****** do we need to initialize the tty message buffer?
c******     (no prompt message in buffer already)
c******     (remember a null message means return a null message)
      koffset=0
      if ( izgetmr(kmess,pline*pnchw,1,jbytes).eq.0 ) then
        klen=zskeybyt(kmess,0,pline*pnchw,peom,pleom)
        if(klen.eq.0)go to rtn
        go to lmes
      endif

c****** prompt for message and wait for message
c******     (remember a null message means return a null message)
      if(iplen.gt.0)then
        call ttytext(iprompt,iplen)
      endif
      if ( izgetmr(kmess,pline*pnchw,0,jbytes).ne.0 ) then
        call ttytext("*** cannot get message from tty ***",35)
        call izexit(1)
      endif
      klen=zskeybyt(kmess,0,pline*pnchw,peom,pleom)
      if(klen.eq.0)go to rtn
      go to lmes

c****** return label
  rtn continue
      return
      end
```

Coding/Examples/Subroutine symenter

Example C4. Subroutine symenter (iname, itype, avalue, index)

```
c ***********************************************************************
c
c  ******  *     *  *     *  *******  *     *  *******  *******  ******
c *         *   *   **   **  *        **    *     *     *         *    *
c *          * *    * * * *  *        * *   *     *     *         *    *
c  *****      *     *  *  *  ****     *  *  *     *     ****      *****
c       *     *     *     *  *        *   * *     *     *         *  *
c       *     *     *     *  *        *    **     *     *         *   *
c ******      *     *     *  *******  *     *     *     *******   *    *
c
c
c> Author:
c    Kelly O'Hair, Computer Graphics Group LLNL
c
c> Calling Form:
c    call symenter ( iname , itype , avalue , index )
c
c> Purpose:
c    Enter new symbol into symbol table.
c
c> Arguments:
c    Input:
c        iname      ascii name of symbol
c        itype      type of symbol
c        avalue     initial value for symbol
c    Output:
c        index      returned position in symbol table
c
c ***********************************************************************

c****** get symbol table common block
c****** get machine parameters

c****** machine flags
c******    i7600       =0 not 7600 coding, =1 7600 coding.
c******    icray       =0 not cray coding, =1 cray coding.

c****** machine independent parameters global to everyone
c******    ptime      timing flag, =0 off, =1 on (timer/tally run).
c******    pmaxline   maximum length of a line in characters
c******    pblanks    a word worth of blanks
c******    pspace     a right adjusted blank in a word
```

Coding/Examples/Subroutine symenter

```
c******    pquote     a right adjusted double quote in a word
c******    papost     a right adjusted apostrophe in a word
c******    prbrac     a right adjusted right square bracket in a word
c******    plbrac     a right adjusted left square bracket in a word
c******    period     a right adjusted period in a word
c******    plowere    a right adjusted lowercase e in a word
c******    pindef     pattern that separates polygons for gppg2d
c******    polymax    the length of the x and y arrays in g4store common
c******                        (at least 100)
c******    pmaxstr    maximum length of a text string in characters
c******    ptypenum   type identifier for a number (always stored real)
c******    ptypestr   type identifier for a string (stored as string id)
c******    ptypeund   type identifier for a undefined variable

c****** machine dependent parameters
c******    pnchw        number of characters per word
c******    plwdb        number of bits per word
c******    puppere      uppercase e
c******    peom         left-justified end of message
c******    pleom        character length of peom
c******    peol         left-justified end of line (LF)
c******    pleol        character length of peol
c******    pcr          left-justified carriage return character
c******    plcr         character length of pcr
c******    padjust      adjustment to addresses for baselib's zgetarec
c******    pline        maximum word length of a line
c******    pmaxlin      maximum length of record for zgetarec
c******    pstring      maximum length of a text string in words
c****** symbol table common block
c******   pmaxsyms  maximum allowable variables in table
c******   ksymt     type of symbol (number, string)
c******   ksymv     value of symbol
c******   csymv     value of symbol (real number access)
c******   ksymn     name of symbol
c******   ksymg     global flag (global=1, local=0)
c******   kcount     count of symbols currently in table
      parameter ( pmaxsyms = 56 )
      dimension ksymt(pmaxsyms), ksymv(pmaxsyms), ksymn(pmaxsyms)
      dimension csymv(pmaxsyms), ksymg(pmaxsyms)
      integer ksymt,ksymv,ksymn,ksymg,kcount
      real csymv
      common/g4symtab/kcount,ksymn,ksymt,ksymg,ksymv
      equivalence (csymv,ksymv)

c****** declare arguments
      integer iname, itype, index
```

Coding/Examples/Subroutine symenter

```
      real avalue

c****** add symbol to table
      if ( kcount .ge. pmaxsyms ) then
         kcount = kcount - 1
         call comerror ( "symbol table overflow" , 21 , 0 )
      endif
      kcount = kcount + 1
      ksymn ( kcount ) = iname
      csymv ( kcount ) = avalue
      ksymt ( kcount ) = itype
      ksymg ( kcount ) = 0
      index = kcount
      return
      end
```

Coding/Examples/Subroutine bufline

Example C5. Subroutine bufline (xfrom, yfrom, xto, yto)

Note the use of the GRAFLIB prologue. The prologue is manually generated.

```
* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
*w
*w Author:
*w    Kelly O'Hair   LLL Computer Graphics Group
*w        x24296  l-73 b116 rm2645
*w
*n Routine Name:
*n          /bufline/
*n
*f Calling Form:
*f   call bufline(xfrom,yfrom,xto,yto)
*f
*p Purpose:
*p   Buffer (save up) this line into the arrays in the common
*p   block linebuf.  A call to dump will empty the buffer.
*p   This routine may call dump if the buffer is full.
*p
*r Arguments
*r   Input:
*r      xfrom,yfrom      <x,y> position to start line
*r      xto,yto          <x,y> position to end line
*r   Output:
*r      None.
*r
* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

c****** declare arguments
      real xto,yto,xfrom,yfrom

c****** declare common area
      common/linebuf/n,x(64),y(64)
      real x,y
      integer n

c****** data load common
      data n /0/

c****** store away line coordinates
      x(n+1)=xfrom
      y(n+1)=yfrom
```

Coding/Examples/Subroutine bufline

```
      x(n+2)=xto
      y(n+2)=yto
      n=n+2

c****** dump buffer?
      if (n.ge.64)call dump

      return
      end
```

Coding Tools

Available Tools

Templates

DEFITS

Availability: DEFITS is available on a CDC 7600 in the LIX file SEJLIB within the public file NEWCP.

Use: DEFITS is a tool to generate a subroutine skeleton with a fairly complete descriptive prologue.

To execute, log onto a CDC 7600 and enter

```
bcon defits
```

DEFITS prompts for all input. Output is in a file with the name you give for the subroutine. DEFITS is not being supported.

TEMPLATE

Availability: TEMPLATE is available on both the CDC 7600 and Cray computers. The author of the subroutine is Denise Sumikawa.

The CDC 7600 version can be retrieved through XPORT as follows.

```
xport
.rd .871406:cdc:template
```

The Cray version can be retrieved in a similar way.

```
xport
.rd .871406:cray:template
```

Use: TEMPLATE is a subroutine prologue that documents programmer information about the source code. To execute, log onto either a Cray or CDC 7600 computer and enter either the Cray or CDC 7600 form of TEMPLATE, respectively.

```
template
```

TEMPLATE prompts for its input and is designed to be used without additional instructions or manuals. The end product of an interaction with TEMPLATE is an ASCII file that may be incorporated into a source listing.

SWRITER

Availability: SWRITER is available on Cray computers. The author of the subroutine is Tokihiko Suyehiro.

SWRITER can be retrieved through XPORT as follows.

```
xport
rd .000029:swriter
```

Use: SWRITER is a tool to assist in making modules with a prologue and source code in the SLATEC format. It prompts for all information as needed and provides instructions on how to enter input. The SLATEC format requires a specific prologue form with uppercase and lowercase characters. Refer to *SLATEC Common Mathematical Library Source File Format* (UCRL-53331) [Ref. 21] for more information on SLATEC format.

Editors

TRIX AC and TRIXGL

Availability: TRIX AC and TRIXGL are public files on the LCC CDC 7600 and Cray computers. See the section on tools in the project-definition chapter for information on TRIX AC (page 29). TRIXGL is no longer supported, but is available for use. Documentation for both files is available from the LCC online documentation system. A summary of useful commands is also in *Summary Sheets* [Ref. 22].

To retrieve the documentation for TRIX AC, for example, enter

```
trix ac!print!nip trixac box ann ident
```

Use: There are commands to simplify creating the source code for different languages--for example, **fed**, **ced**, **bed**, **ned**. You can use **ged** to simplify editing for other languages and the esc and cesc commands to define abbreviations and redefine the escape characters as appropriate.

The **bf** command is useful to make comments stand out, and the **red** commands **fill** and **jfill** can be used to keep the prologue or other blocks of comments together. The **fi** (Fortran index) command is helpful in indexing subroutines and functions.

Other Coding Tools

CHATITS

Availability: CHATITS is available on the CDC 7600 computers in the LIX library SEJLIB, within public file NEWMCP.

Use: This subroutine is a tool for use in maintaining source and binary libraries for reasonably large codes. It controls compilation, updates source and binary libraries, outputs listings, controls loading, saves files and more.

To execute, enter

```
bcon chatists
```

This produces CHATIT, which can then be executed. CHATIT is not supported, however, documentation can be obtained from the USD consulting office.

CIVIT

Availability: CIVIT is available on the Cray computers and can be obtained from the storage directory.

Enter

```
xport
.rd .525050:bcon:civit
```

Use: CIVIT is essentially a Cray version of CHATIT, using the CIVIC compiler rather than CHAT. CIVIT is not being supported.

Coding/Tools/Available

FOCAL

Availability: FOCAL is available in public files on the CDC 7600 and Cray computers. Documentation is available from the LCC online-documentation system as **focal** or **lcsd1630** [Ref. 23]. A summary of FOCAL commands is also in *Summary Sheets* [Ref. 22].

Use: FOCAL produces a list of Fortran source code plus a global cross-reference containing all symbols used in the source file. The cross reference is indexed by source line numbers where a symbol appears.

KLEAN

Availability: KLEAN is available on the CDC 7600 computers. KLEAN is documented in *KLEAN: Clean Up Messy Fortran* [Ref. 24] and through a help package obtained by executing KLEAN without input arguments. Jeff Rowe is the author of the subroutine, and it can be retrieved from his storage directory.

Enter

```
xport
.rd .768350:klean
```

Use: KLEAN is a program designed to clean up messy Fortran programs. It reduces multiple-statement lines to one statement per line, relabels statement labels in an orderly manner, indents **do**-loops and **if-then-else statements**, and more.

Be aware that statement relabeling uses simple pattern replacement, which may occasionally catch more than just labels. KLEAN is minimally supported.

KWIKLIST

Availability: KWIKLIST is a public file on the CDC 7600 and Cray computers within the public file BOOK2 (a LIX library file) as KWICLIST. Documentation is available through the LCC online-documentation system as **kwiclist** or as **lcsd1647** [Ref. 25].

Use: KWIKLIST generates a global cross reference map with a complete listing of the input source-code sequenced by overlay number, subroutine name, and line number within the subroutine. KWIKLIST is currently <u>not</u> being supported.

LBL Software Tools

Availability: LBL Software Tools are available on the Cray. Documentation can be retrieved from storage, but consulting help is not. available.

Enter

```
xport
.rd .318388:stdoc[st.doc progman]
```

Use: The LBL tools include a command line interpreter, on editor, and a text formatter plus several other tools.

MCE

Availability: MCE is available on CDC 7600 and Cray computers. There is documentation [Ref. 26] that can be retrieved from the LCC online documentation-system as either **lcsd5241** or **mce**. A summary of MCE commands is also in *Summary Sheets* [Ref. 22].

Use: MCE facilitates modifying a few or all the subroutines of a code by controlling the execution of compilers, loaders, library manipulation, file storage, and more.

NLTSS-Controller Family of Routines and Support Routines

Availability: The NLTSS-controller family of routines and their support routines are a group of related processes. They are all available on the Cray computers, but only part of this group of routines is available on the CDC 7600 machines. Documentation is available from the USD consulting office. The authors are J. Minton and J. Donnelley.

Use: The NLTSS group of routines is designed to help provide synchronized access to libraries and processes. They also help remove much of the tedium found in our day-to-day operations, such as writing files to storage, outputting listings, running compilers and loaders, counting lines of code, updating the data base, etc. The routines are very specialized for NLTSS use, but could be of possible use to individuals wanting to build their own specialized development tools.

Coding/Tools/Available

NPUT

Availability: NPUT is available on the CRAY computers. The documentation can be retrieved from storage.

Enter

```
xport
.rd .177891:put:putdoc
```

You can get hardcopy using the TRIX AC **print** command.

Use: NPUT merges source files similar to the way BUILD merges relocatable binary-files. NPUT is used to update modules or to add new modules to the source-code file.

To use it, enter

```
exe use nput x
```

NGET

Availability: NGET is available on the CDC 7600 computers in public file ANEW (a LIX file), and on the Cray machines in public file LASLIB (a LIB file). Documentation is available from storage.

Enter

```
xport
.rd .846950:nget:ngetwup
```

You can get hardcopy using the TRIX AC **print** command.

Use: NGET will extract source routines from an input file and write them to an output file. It will extract a single routine or all routines bounded by two specified routines.

NOGOTOS

Availability: NOGOTOS is in public file LASLIB. It also can be accessed from storage by entering

```
xport
rd. .489062:nogotos.
```

Documentation is in storage and can be retrieved by entering

```
xport
rd. .489062:nogotodoc
```

If you want hardcopy of the documentation, you can use the TRIX AC **print** command. The author of NOGOTOS is Jim Kohn.

Use: NOGOTOS is a file containing a small set of macros that can be used to write structured Fortran programs without GOTO commands. NOGOTOS makes looping constructs and case selection constructs readily available. The Fortran source must be processed by PRECOMP (a macro processor) to turn it into code acceptable to CIVIC, CHAT, CFT, and other Fortran compilers.

PASCAL Tools

Availability: PASCAL tools are available on the CDC 7600 computers using SLOPE2. Documentation is in storage on the LCC Octopus system and can be retrieved by entering

```
xport
.rd .381388:pascaldocs[pascalwu pasclibwu toolswu]
```

You can get a brief description of what is available in another document you retrieve from storage. After entering **xport**,

```
.rd .381388:pascal:pascald
```

If you want hardcopy of the documentation, you can use the TRIX AC **print** command. The consultant for the PASCAL tools is Terry Heidelberg.

Use: The PASCAL tools consist of a consist of a disassembler, source file formatter, and more.

Coding/Tools/Available

PRECOMP

Availability: PRECOMP is available in public files on the CDC 7600 and Cray computers. Documentation is retrievable from the LCC online-documentation system as either **precomp** or **ur920** [Ref. 27]. A summary of PRECOMP commands is also in *Summary Sheets* [Ref. 22]. The PRECOMP consultant is Bob Hughes.

Use: PRECOMP is a macro processor and translator. It accepts Fortran-like source code and produces output to be compiled by CHAT, CIVIC, FENIX, and CFT compilers.

RELABEL

Availability: RELABEL is available on the CDC 7600 computers in library file USE. Documentation is retrievable from the LCC online-documentation system as either **relabel** or **ur214** [Ref. 28] A summary of RELABEL commands is also in *Summary Sheets* [Ref. 22].

Use: The subroutine relabels Fortran statements in a source file with alphanumeric sequenced statement labels. Be aware that statement relabeling uses simple pattern replacement, which may occasionally catch more than just labels.

SML

Availability: SML is available on the CDC 7600 and Cray computers. Documentation is retrievable from the LCC online documentation-system as **ucid18887** or **sml** [Ref. 29].

Use: SML sorts and merges files from a LIB library. It makes a library from one program and separates the library one file per module. SML can also do partial copies.

VERIFY

Availability: VERIFY is available on the CDC 7600 computers only. The tool is available from storage and can be retrieved by entering

```
xport
.rd .295701:verify
```

The code is imported and not supported. Documentation can be obtained from Anne Greenbaum.

Use: VERIFY checks a Fortran program for adherence to a portable subset of the 1966 ANSI Fortran- standard. It verifies not only individual program units, but also interprogram-unit communication via COMMON and argument lists. For individual program units, it produces error diagnostics, symbol tables, and cross references. The interprogram-unit information includes, for each program unit, a listing of the arguments, the COMMON regions, the called program-units, and the program units that call it. A list of global common definitions is also produced. The source file to be verified must be named INPUT, and the output will be in a file named HVERFO. The last line of INPUT must contain a single period (.) in the first position. To run the program, simply type **verify** .

WORK

Availability: WORK is available in public file GG on CDC 7600 and Cray computers. The author of WORK is Kelly O'Hair.

Use: WORK expands and manipulates module prologues and subroutines written in the GRAFLIB standard format (see Example C5 on page 93). The tool operates on modules such as subroutines, functions, cliches, or blockdatas with commands to sort modules, extract modules, make a catalog of modules, delete modules, replace modules, and more.

Ideal Tools

We don't know what is the best user interface for a template tool, but we would like to try a menu-interface. The design for a user-interface tool should be flexible so that the tool could be modified or changed easily. The template tool itself should allow for some flexibility, especially in what information is used and how the information is ordered. In addition, the tool should interface to the available text editors and be portable to current and future workstations.

Coding/Bibliography

Coding Bibliography

Anonymous, *Nuclear Software Systems Division Software Engineering Guidelines*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-19228 (1981).

J. A. Clapp, "Designing Software for Maintainability," *Computer Design*, 197 (1982).

James L. Elshoff and M. Morcotty, "Improving Computer Program Readability to Aid Modification," *Computing Practices*, 25(8), 512 (1982).

B. Herron, "Software Project Standards Preview: Coding Practices," *Tentacle*, 2(4) 21 (1982).

D. D. McCracken and G. M. Weinberg, "How to Write a Readable FORTRAN Program," *Datamation*, 73 (1972).

J. Minton, D. Schnabel, J. Huskamp, G. Whitten, and K. Dusenbury, *URLIB - A Subroutine Library for Writing Utility Routines*, Lawrence Livermore National Laboratory, Livermore, CA, M-048 (1979).

J. B. Munson, "Software Maintainability: A Practical Concern for Life-Cycle Costs," *Computer*, 103 (1981).

G. J. Myers, *Software Reliability Principles and Practices* (John Wiley and Sons, Inc., New York, 1976).

P. R. Newsted, W.-K. Leong, and J. Yeung, "The Impact Of Programming Styles On Debugging Efficiency," *Software Engineering Notes*, 6(5), 14 (1981).

M. Page-Jones and E. Yourdon, *The Practical Guide To Structured Systems Design* (Yourdon Press, New York, 1980).

k. O'Hair, *CRAFLIB Programming Standards*, Lawrence Livermore National Laboratory, Livermore, CA, Unpublished (1983).

R. Skowlund and J. Martin, "Software Project Standards Preview, Common Attributes of a Module," *Tentacle*, 1(5), 8 (1981).

A. I. Wassermam and S. Gutz, "The Future of Programming," *Communications of the ACM*, 25(3), 196 (1982).

TESTING

Overview

Testing is the process of running a software program with sample data-sets to find errors. Therefore, testing can only detect errors, not show program correctness. Error detection is done by using sets of test data that cover all possible input (exhaustive testing) and yet are small enough for practical use. The test-data sets should come from an analysis of the product at each stage of its development life-cycle: definition, design, and coding. Only in this manner can it be assured that the specifications for the program are even testable, let alone correct.

Historically, software-products testing has been done informally, and testing for changes in critical software products was often improvised as the changes were made. Test cases were created, run from a terminal, and discarded when successfully completed. Despite a software developer's best intentions, errors would be introduced during enhancement and remain undetected until the entire program was used in a project.

Although software developers generally agree that testing is important, rigorous testing is difficult and complicated by the tendency to test for success rather than failure. It helps to have a somewhat disinterested person work with the developer to set up conditions under which errors will be produced. Peer review and formal walkthroughs are also useful in detecting errors early in the development life-cycle when easiest and cheapest to correct. Even a shift of viewpoint from trying to prove a program correct to trying to find errors can greatly help a developer. To restate Myers' testing principles [Ref. 30]:

> **Testing is the process of executing a program with the intent of finding errors.**
>
> **A good test case is one that has a high probability of detecting an as-yet undiscovered error.**
>
> **A successful test case is one that detects an as-yet undiscovered error.**

There are methods to assist development of test cases, but it's recognized that producing a theory of testing leading to fully automated systems is unlikely. While formal verification techniques have been developed, they need substantial tool-support to implement. Testing is

still the most easily applied technique for verification, and the tester must still use his ingenuity and problem-oriented knowledge for detecting a variety of specific error-types. To quote Myers:

> **It is probably true that the creativity required in testing a large program exceeds the creativity required in designing that program.**

In this chapter, we will consider the guidelines for accomplishing the testing and the test plans for the ASCII to EBCDIC converter and for a module designed to solve a quadratic equation. Refer to the glossary on page 137 for terms associated with testing with which you are not familiar.

Testing Guidelines

We recommend that testing be a part of the project development life-cycle. That is, testing should be done for the project-definition phase, project-design phase, and coding phase. In addition, a test history should be kept.

Testing should not be confined to the final stage of development. Careful test planning is as important as careful project definition and design. It's necessary to choose test data, predict the expected results of the tests, and decide how to compare the results with the desired behavior. If you combine the structural and functional characteristics of a program, you will get a basis for determining criteria for test-data sets. Functionally, the test data should reflect properties of the program and its range and cover extreme and transitional values for input, output, and control. Structurally, the test data should use the program to meet a given level of coverage--for example, all statements, all branches, and all decision-to-decision paths.

However, a set of test data that is logically sufficient may not be practical to use. Therefore, find a test-data set that covers the material to be tested but is small enough for practical use. The following criteria have proven helpful.

- Test data should reflect the special properties of the most extreme values of the system
- Test data should reflect the special properties of the function the program is supposed to implement--for example, the special property of program values that lead to extreme function values

- Test data should employ the program is a specific way--for example, the test data should cause all branches or all statements of the program to be executed

A summary of the guidelines for the testing phases is seen in Table 3 with a brief description. Each item is expanded further in the sections following the table.

Table 3. Summary of the phases for testing software programs generated at Lawrence Livermore National Laboratory.

Phase	Description
Prepare a functional test-plan for the project definition	The functional test-plan treats the product as a black box, associating specific product output-sets with specific input data-sets.
Create a modular test-plan for the project-design phase	The modular test-plan delineates sets of test data for testing module interfaces and functionality. It should specify specific inputs to a module and expected outputs from the module.
Construct a structural test-plan for the project-coding phase	The structural test-plan includes program instrumentation and test-data sets designed to cross and verify specific decision paths within the program.
Keep a test history	The test history includes a record of test set-executions and results in enough detail to allow for reproducing the tests at a later time. The execution of *ad hoc* tests, without sufficient records, are considered part of code debugging rather than as acceptance testing.

Testing/Guidelines/Functional Testing

Functional Testing

A software program can be viewed as a function and can be tested for the degree of faithfulness with which implementing the program reproduces the function specified in the project definition. Therefore, functional testing involves applying test data derived from specified functional requirements and can be accomplished without regard to the final program structure. The act of creating functional tests helps to insure that the requirements are specific and can be measured for testing.

Input functional elements are called valid; other input elements are invalid. A program may be considered incorrect, given either a valid or invalid input, because it does not do something it's supposed to do or it does something it's not supposed to do.

A simple type of functional test can be derived from specifying that a program module receive as input the three coefficients a, b, and c for the quadratic equation $ax^2+bx+c=0$ and return as output the roots. Refer to the example on page 113 for other tests.

Structural Testing

Structural testing is a method in which the test data are derived solely from the program structure. The structure of the program is determined during the design and coding phases of the development life-cycle. The program is divided into a hierarchy (a graded series) of modules connected by specified interfaces and data flows. Within each module, decision points are chosen to implement alternative actions specified in the requirements. Then functional tests for individual modules and module interfaces can now be designed and then tested (see section on functional testing, above), because the data sets will be composed of specific inputs to modules and expected outputs.

It is also now possible to design data sets for structural testing by selecting test data that put all branches into play, execute all statements, or cover all decision-to-decision paths through a program. Structural testing is helped by inserting additional code into the program to collect information about program behaviour during program execution. Code insertion is called instrumentation and is a way of learning about the effect(s) individual tests have on a program. Instrumentation also provides an empirical method of obtaining a measure of the test coverage when a series of test cases is analyzed.

Testing/Guidelines/Structural Testing

Test History

Because of the time required to create, execute, and archive each test case, it is very important that test data are recorded. Then, the tests can be rerun when a product is next modified (regression testing). These records are the test history.

This history is needed for the test-data sets run, the specifics of execution, and the comparison of outputs. It is also needed for the entire software environment on which the program depended when it was last compiled, loaded, and executed. All this information can be used to trace discrepancies in outputs of different executions, using the same test data, back to their sources. Among the possible causes for such discrepancies, look for new errors introduced while making corrections, compiler changes, changes to binary libraries, and changes to other processes or controllees executed by the program.

A comprehensive test-history, sufficient to allow a new program-maintainer to rerun the previous tests, will greatly help recertification of software programs following changes.

Testing Examples

Example T1. ASCII to EBCDIC File Converter

Three tests are run for the ASCII to EBCDIC file converter, i.e., a user-interface test, a functional test, and an exhaustive test. These are summarized below.

Test Plan

User Interface Test

Write the user interface first, using stubs (see Glossary on page 137) for the working parts of the program. Create an input file **inputa** and try the following execute lines.

convert

convert help

convert inputa
If you run **convert inputa** on suffix **a**, check to see if there's an output file **a$ebcdic** in your disk space.

convert inputa outpute

convert inputa, outpute

convert inputa , outpute

convert inputa ,

convert inputa inputa

convert none
There is no input file called **none** .

convert inputa outpute none

Repeat the same commands after receiving prompts from the help package.

Testing/Examples/ASCII to EBCDIC File Converter

Functional Test

After the working part of the code is written, try the following procedures.

- Send the output file **outpute** to a CDC 7600 computer via **xport**, and run the following utility.

```
format outpute bactoa ebcray
```

Then, send **bactoa** to a Cray computer and compare it with the original file **inputa**.

- Create a file with lines longer than 132 characters, and use it as input.

Exhaustive Test

Create a file containing each of the 256 ASCII characters. Then, run **convert** under **ddt** with breakpoints set, so it's possible to check for the correct translation of each character.

Test History

User-Interface Test

All input lines functioned as expected.

Functional Test

- The source for **convert** was used as an input file. When the output file was reconverted by **format** on the CDC 7600, a casual inspection showed that the conversion table contained an incorrect code for the character . (period). This was fixed.

- When lines longer than 132 characters were used as input, **rdline** detected them. An error message, included in the code to detect the longer line, was removed.

- The test suggested by the walkthrough team, while reasonable, was not performed, because an EBCDIC utility was not available.

Testing/Examples/ASCII to EBCDIC File Converter

Exhaustive Test

This test was not performed, because it was tedious and boring, and the use of **convert** was not critical enough to warrant the effort.

Example T2. Quadratic-Equation Test Plan

The following are test cases suggested by Gruenberger [Ref. 31] to test a module designed to solve the quadratic equation $ax^2+bx+c=0$. These are not the only test cases possible; others, for example, are designed to determine the arithmetic limits and precision of the module.

Test A. Normal Case

a=6, b=1, c=-2

Test B. Normal Case with One Zero Root

a=3, b=7, c=0

Test C. Square Root of Zero

a=7, b=0, c=0

Test D. Linear Equation

a=0, b=5, c=17

This is a linear equation instead of a quadratic. How does the module handle it? Does a=0 (or nearly zero) result in an overflow or indefinite?

Test E. Complex Roots

a=3, b=2, c=3

Does the module handle the case where the roots are complex?

Test F. Invalid Equation

a=0, b=0, c=10

The invalid equation is 10=0. Is an appropriate error-return generated?

Testing/Examples/Quadratic-Equation Test Plan

Test G. Degenerate Equation

a=0, b=0, c=0

The equation degenerates to 0=0 and cannot be solved. What does the module do?

Testing Bibliography

W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky, *Validation, Verification, and Testing of Computer Software*, NBS Institute for Computer Science and Technology, U.S. Dept. of Commerce, Washington DC, Special Publication 500-75 (1981).

M. A. Branstad, J. C. Cherniavsky, and W. R. Adrion, "Validation, Verification, and Testing for the Individual Programmer," *Computer* 13(12), 24-30 (1980).

USER DOCUMENTATION

Overview

User documentation (UD) should enable anyone to access software easily and to use the routines to their full potential. Therefore, the goal is well-prepared and timely documentation that is accessible to every user at every level of expertise.

However, not everyone has the same time available to learn to use a program or wants to know the same amount or kinds of material before or during a working session [Ref. 32]. Therefore, we propose preparing at least three categories of general documentation to satisfy the needs of most users. These are listed in the order of priority.

- A reference document
- A by-example document
- A tutorial document

A reference document is a complete description of the software product. It explains all commands, options, alternatives, errors, methods, etc. A by-example document takes one or more examples of typical usage and shows the input with the corresponding output of each example. A tutorial document presents the information as a teacher would in explaining the software to a class of students.

The reference document must always be provided--it is essential. However, the by-example document is likely to be the most heavily used, as many software-product users are interested only in typical cases, especially at first. Only later, when some special option is needed, will users refer to complete specifications in the reference document. The tutorial should be provided for novice or first-time users.

In preparing the documentation, you should include some of the same information in each of the three document categories. A user should be able to use a document without constantly referring to another. Documentation in each category will change as the program or routine for which it was written is updated or revised.

In this chapter, we will first present general guidelines for when and how to prepare and update user documentation. Second, we will present

guidelines for how to prepare documents in the three categories noted above. Third, we will describe some additional mechanisms for communicating documentary information. Fourth, we will provide templates for the different types of documentation. Fifth, we will cite examples. Sixth, we will outline some tools for both presenting and preparing user documentation at all levels.

General Document-Preparation Guidelines

The time at which you prepare user documentation for USD software is as important as how you prepare it. Some users believe the time of UD preparation is more important, because the product will not be used effectively without accompanying documentation. Therefore, both new or revised software and the UD should be available and announced together.

Below is a list of general guidelines that includes information on timeliness, announcements, and content of user documentation for USD software.

- Whenever USD software is created or modified, user documentation for the software should be updated and given to the document coordinator for placing into the LCC online system
- Whenever a document is created or updated, keywords pertaining to the document should be assembled or amended, so users can find a selected portion of the writeup for online viewing
- Whenever a document is created or updated, a revision history should be appended or updated so that the user can locate changes by description and page; also, changes should be indicated with change bars in the text
- Whenever a document is created or updated, there should be an announcement placed in the daily *Octogram*
- Whenever documentation for a software product is updated, pertinent portions should be updated in the *Summary Sheets* entry [Ref. 22] and given to the document coordinator for placing into the LCC online system
- Whenever USD software is released before the completed documentation is edited, the draft documentation should be given to the document coordinator for placing into the LCC online system and be immediately available

- For each USD product, there should be a reference document of commands, tools, and fundamental structures and concepts
  -- Start-up examples should be provided for immediate access to the product

- For each USD product, a step-by-step demonstration document with examples for every command should be provided
  -- In addition, start-up examples should be available for immediate access to the product

- For each USD product released, a tutorial should be prepared
  -- Start-up examples should be provided for immediate access to the product

Guidelines for Preparing User-Documentation Categories

The guidelines in this section are based on documentation preparation at Los Alamos National Laboratory [Ref. 32]. We will discuss three categories of general documentation--reference, by-example, and tutorial documents--and the possible need for start-up examples in each category.

The Reference Document

Description

The reference document is terse, but accurate, and detailed. It should cover what each tool is, what each command does, and what the fundamental structures and concepts of the product are. It should cover all computing features, such as syntax information, calling sequences, error correction, and other supplementary information.

It should also include quick-reference and cookbook information and detailed examples to provide everything a user may need to effectively use the software in question. It may have to have some tutorial information. For example, reference documentation for an obscure tape-utility might include tutorial information on importing, converting, and exporting tapes.

Audience

The primary audience is the user who knows the program and can find the needed reference material. The secondary audience is the user who wants to

learn more about a program or who is familiar with the generic capability of a program but wants a complete description of it.

The By-Example Document

Description

A by-example document contains general rules and examples to apply to what the user is working on. The examples are considered generic rather than specific. The necessary commands and functions of common tasks are described step-by-step and then expanded. The user gets semi-instant gratification.

Audience

The audience includes anyone who wants immediate information on how to generally apply a specific program or the user who completed the tutorial.

The Tutorial Document

Description

A tutorial document contains information about how software or hardware functions and is usually prepared for heavily-used computer capabilities such as filing systems, file management, output generation, and text editors.

Audience

The audience for a tutorial document is the user who is a novice or just beginning to become familiar with the capability of the software and has the time and inclination for learning. Therefore, the tutorial should cover introductory topics, general-reference material, cookbook-type directions using specific rather than generic examples, as well as detailed material to reference as the user advances. Although a tutorial can get a user started by introducing him to fundamental concepts and methods used to run a program, it can also be a supplement to regular classes and should provide learning choices.

User Documentation/Guidelines for Preparing User-Documentation Categories/Start-Up Examples

Start-Up Examples

Description

Start-up examples are task-oriented how-to procedures and should be designed to help a person use software quickly. Therefore, there should be limited explanations of the examples and perhaps no supplemental information. The material included in the examples must be tested and assured before release, because users have no tolerance for examples that won't work. User documentation for commonly used programs, such as TRIX AC [Ref. 17], might have start-up examples for each UD category.

Audience

Start-up examples are for users who have little time to learn a program, who need a memory jog at the beginning of a document, or who would be more comfortable having more examples before them.

Documentation Aids

There are services available through USD that help software-project teams communicate with users and among themselves. These services may not be the usual forms of user documentation, but they do function in a similar way--that is, they inform users of what is new and what is happening. Among the different services are formal (and usually regular) publications, informal (and often irregular) publications, courses, meetings, and consulting.

Formal Publications

Among the formal USD publications are the LCC Routine Summaries (*Summary Sheets*), the monthly *Tentacle*, a glossary, the daily *Octogram*, and introductory cards (introcards).

LCC Routine Summaries

The LCC Routine Summaries are notebook-sized pages condensed from complete routine writeups. Most of the summaries are in *Summary Sheets* [Ref. 22], which provides an overview of heavily-used utility routines. The types of routines found in *Summary Sheets* are programming-language

processors and loaders, batch processors and job controllers, and text-editing software.

Tentacle

Tentacle is the monthly news magazine for Computation Department users. It contains information about new products, department and division activities, and articles of general interest. Daily *Octogram* notices of new or revised software and the longer *Octopus Communique* writeups are also included.

Octogram

The *Octogram* is a single-sheet announcement that is published daily and distributed widely within LLNL. It, therefore, reaches most computer users at the laboratory.

Glossary

The *Glossarium* [Ref. 33] is an LCC-specific glossary and provides definitions for many computer terms used at LLNL.

Introductory Cards (Introcards)

Introductory cards, also called introcards, are fan-fold brochures that give general information about computing facilities. They are designed primarily for users with little to moderate knowledge of LCC operations.

Informal Publications

Among the informal publications are memos about Computation Department meetings and information letters from various divisions in the department. These are distributed through lists generated by the Mail Room. There are two lists that the Computation Department uses--the Cheshire-46 list for general Computation distribution and the Cheshire-36 list for people interested in information on graphics.

Educational Services

Courses

You can take courses in several ways at LLNL.

- Attend a class with others and have an instructor in the room with you. These so-called live classes can be structured or unstructured, depending on the needs of the participants, and interaction with the instructor is encouraged. These courses are often taped for future use or review.

- Attend a previously taped course. While there can be no interaction with the instructor, there is usually a proctor available to answer questions and go over any homework.

- Take a self-paced individual course. This might be a previously taped structured-class, an online tutorial, or a hardcopy tutorial.

Catalogues of available courses are sent regularly to employees at LLNL. Taped classes, produced at LLNL or elsewhere, are available for individual review or use by calling the television studio at Ext. 2-8990.

Computer Documentation Library

The Computer Documentation Library is located Trailer 2106, Room 1001. You can call Ext. 2-0592 for information or to have a document sent to you. Most of the documents in this library are online; and if you're on the Octopus system, you can retrieve them directly. However, some Computation Department documents are available only as hardcopy and found only in this library. Here, you will also find documentation from Cray Research, Inc. and SLOPE2 documentation from Control Data, Inc.

Consulting Services

The USD consulting office is located in Trailer 2106, Room 1004. You can reach a consultant at Ext. 2-3724. The consultants provide advice on software and programs to users every working day. In addition, they collect statistics on the types of questions asked, so general answers may be published in *Tentacle* on a timely basis. A log of reported bugs has been started, and the types of system or program anomalies can now be analysed and solutions sought.

Meetings

If you read the daily *Octogram* and your mail, both electronic and hand delivered, you'll be informed of all the meetings of interest to you. Among these may be

- Computation Department Technical Seminar series
- Major-users group meetings,
- Specific computer-users group meetings
- Twice yearly Computation Department report to the members by the staff
- Special-interest group (SIG) study sessions
- Problem-solving committees

User-Documentation Templates

The templates in this section are for a reference document, a by-example guide, a tutorial document, and summary sheets. Refer to *A Guide for Software Documentation* [Ref. 34] for more information.

Template UD1. Reference Document

Identification

- Purpose
- Available on machines
- Programmed by
- Documented by
- Memory requirements
- Revision history
- References

General Description

Definition of Terms

- List alphabetically all terms used in this report
- Define each term

Options

- Input
- Edit

- Output
- File
- Other

Summary of Usage Forms

- List or simply describe basic instructions for using the routine
- Display the user-routine dialogue with necessary variable terms and options to illustrate the mechanics of one or more usage forms--for example, mention what continuation characters are used in the routine

Usage Examples

- Display examples of how the routine is used

Detailed Information

- List each command if the product is interactive
- Display each icon if the product is menu driven
- If the product is a compiler or translator,
  -- Provide syntax descriptions
  -- List all keywords or statements with their parameters
- List methods employed by the product if it would aid the user
- Provide enough information for the user to choose among alternate methods provided by input options
- Explain how to run the product as the controllee of another or how to use the product as a controller, if possible
- List the effects of combinations of options

Help Facilities

- Briefly state the help provisions the user can access while executing the routine
- Mention when the user can type **help**
- If no help facilities are available, state it

Restrictions

- List restrictions such as maximum number of parameters, reserved words, word format, etc.

Error Messages

- Divide error indications into fatal and nonfatal messages
- Sort alphabetically the error messages in each category
- Special characters used in error-message formats should be defined in terms of
  -- Character
  -- ASCII code

Controller Message-Formats

- List alphabetically messages to the controller
- List all messages in their exact character-by-character description

Revision Histories

Provide revision histories for both the program and the documentation. Changes in the documentation, of course, can provide information on changes in the program.

User Documentation/Templates/Reference Document

Comment Sheet

- Attach a comment sheet for user comments. Allow room for comments and use a fill-in-the blank list that includes, for example,
  -- Typographical errors
  -- Omissions
  -- Unmentioned quirks or peculiarities of the product
  -- Bugs
  -- Nice-to-have options

- Encourage users to make comments and send the sheet to the author

Template UD2. By-Example Guide

A by-example guide is relatively terse, and explanations, although present, are minimal. The guide should include the following information.

Preface and Contents

- Table of contents
- Abstract containing a brief description of the software product
- Information on the effective use of the by-example guide

General Information

- Introduction
- Requirements for successful execution of the software product
  -- Hardware and software environments
  -- Other related products

Example(s)

- Operating-system control statements
- User input data
- Product execution
- All dialogue between the user and the software product
- Error messages
- Any interruptions

- Recovery
- Output
- Termination
  -- Normal
  -- Abnormal

References

- List numerically those documents referenced by number in the guide
- List aphabetically in a separate section those documents not referenced in the guide

Revision Histories

Provide revision histories for both the program and the documentation. Changes in the documentation, of course, can provide information on changes in the program.

Template UD3. Tutorial

The user manual should be prepared in a conversational tone at every level. There may be times when you'll need to use unadorned lists or commands, but the conversational tone can be resumed after these.

Preface and Contents

- Prepare a conversational preface that includes information on
  -- Where and when to use the software or program
  -- How to use the manual effectively
- Prepare a table of contents

Introduction

You should include the following in the introduction.

- Purpose
- Functions performed
- Limitations
- Additional background

General Information

You should provide the user with the following information.

- Acceptable input-data units
- Available processing
- Restrictions
- Generated output-data
- The meaning of operating-system-control statements
- The meaning of installation- or program-control statements

- Organization of the input stream

How to Execute the Job

- Initiating operations
- Loading the program
- Starting the program
- Entering input data
- Error procedures
  -- The user program
  -- The operating system
- Changing the input data
- Interrupting the program
- Obtaining output data
- Ending the program
- Terminating operations

Revision Histories

Provide revision histories for the program and the documentation. Document revisions, of course, reflect program revisions.

Template UD4. Summary Sheets

The summary sheets provide a quick reference to pertinent material in LCC utility routines. It is assumed that the user has some familiarity with the routine. The summaries should provide the following material for the user.

- Reference document
- Purpose
- Availability
- Execution lines
- Definitions
- Options
  - -- Input options
  - -- Alteration options
  - -- Output options
- Other material
  - -- Defaults
  - -- Commands
  - -- Interrupts

User Documentation/Examples/Document Summary Sheet

Examples

There are many documents of the three types we discussed above, plus the LCC Routine Summaries [22], in the Computer Documentation Library (see page 123). *DDT* [Ref. 35] (LCSD-1620) is a reference document, the seven parts of *Computer Graphics by Example* [Ref. 36-42] (UCID-30166 Part 1 through Part 7) is a by-example guide, and *Introduction to FRAMIS* [Ref. 43] is a tutorial.

The example we will use here is the summary sheet for the EBCDIC converter, used as an example in the other chapters of these guidelines.

Example UD1. EBCDIC Summary Sheet

LCSD-0000 CONVERT

PURPOSE	Convert a Cray 8-bit packed ASCII file to a packed EBCDIC file.
AVAILABILITY	Private files on Cray-1 computers.
EXECUTION LINE	CONVERT *infile outfile / t v*

Notes

1. If no *outfile* is given, the name of the output file will be ?$EBCDIC, where ? is the current suffix.
2. The HELP package is available by typing

   CONVERT HELP

DEFINITIONS

infile	The name of the disk file to be converted.
outfile	The name of the output file created. This file holds the converted results.

User-Documentation Tools

Available Tools

The available tools listed in the Project Definition chapter (page 29) and the Design chapter (page 67) are also available for preparing UD.

The LTERM terminal emulator and the typesetting system T_EX are available in addition to the tools listed in previous chapters.

LTERM (A terminal emulator)

Document: S. Sparks, N. Smith, and G. Ledbetter, *LTERM Tutorial, Version 2.0*, LCSD-3, (1985) [Ref. 44].

To obtain a copy of this document, log onto a CDC 7600 and enter

trix ac!print!nip lcsd3 box *ann id*

Use: The LTERM terminal emulator allows you to use an intelligent terminal as if were a terminal on the LLNL Octopus System mainframe. The LTERM Upload and Download functions enable you to copy files back and forth between Octopus worker computers and your PC. Thus, you can prepare a document offline, and then upload it to the mainframe and make it available to other users through the USD online retrieval system.

T_EX (A typesetting system)

Document: D. E. Knuth, *The $T_EX$book* [Ref. 45]

Use: T_EX allows you produce a typographic copy of your document. You prepare the document using your favorite editor program, putting in the coding for T_EX output as you proceed.

Availability: T_EX is available at LLNL on the LCC J Vax computer, the Computer Research Group (CRG) Vax, and some PCs.

User-Documentation Bibliography

Anonymous, *Nuclear Software Systems Software Engineering Guidelines*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-19228 (1981).

M. Gray and K. London, *Documentation Standards* (Brandon/Systems Press, New York, 1969).

G. J. Meyers, *Software Reliability* (John Wiley and Sons, New York, 1976).

GLOSSARY

black box testing see **functional testing.**

bubble chart see **data-flow diagram.**

by-example document A user document containing one or more examples of typical usage of a software product. The input and corresponding output of each example and the dialogue between the user and the software product are shown.

cohesion The relationship between the internal tasks within a module.

command language *Another name for* job-control language.

conceptual model 1. The mind's eye view of the product. 2. The logical (as opposed to the physical) concepts with which the system deals.

coupling A method of communication among the modules of a program.

data dictionary 1. A list of the types of data elements that appear as labels on the directed lines in a data-flow diagram containing definitions. 2. A structured description of a database. The data dictionary contains descriptions of the contents of the database, as distinct from the raw data held in the database itself.

data-flow diagram A graphical network-representation of a program made up of circles representing processes and directed lines representing data flow. A data-flow diagram is also known as a bubble chart.

data-structure diagram A graphical representation of a complex data item in a data dictionary.

decision-to-decision (DD) path A path of logical code-sequence that begins at an entry or decision statement and ends at an exit or decision statement.

dialogue The interaction between the user and the product.

exhaustive testing Executing the program with all possible combinations of values for program variables.

function key A specific keyboard key that, when pressed, causes a function to be performed rather than a character to be sent.

Glossary

functional testing Applying those test data derived from specific functional requirements without regard to the final program structure.

icon A graphical or pictorial shape that identifies a command or function.

instrumentation Inserting additional code into a program to collect information about how the program behaves while it runs.

interactive graphics The use of a tv screen and a pointing device to communicate (see **dialogue**, above).

invalid input Test data that lie outside the function domain that the program represents.

job-control language A language used to write the sequence of commands that will control the running (execution) of a set of programs (a job). The computer input for a job often consists of verbs.

menu A displayed list of options from which a choice can be made.

metadialogue The set of instructions that describes how a **dialogue** works. For example, the metadialogue for the command tty instructs the product to interact using the terminal; and the metadialogue for the command tv129 instructs the product to interact using the Television Monitor Display System (TMDS) device number 129.

model A physical or abstract representation of an entity or a phenomenon. A model helps a user understand how the product works, how to predict the effects of varying one or more input parameters, and how to predict the rules governing program structure (syntax form).

natural language 1. The user's conventional speaking or writing language rather than a computer-programming language with formal or prescribed rules.

product A utility routine (such as those listed in *Summary Sheets* [Ref. 22]), library, or operating system resulting from computer programming.

project definition One or more documents containing the project plan and project description for a software product. The project plan contains the purpose, milestones, resources, etc. and the project description the requirements, specifications, etc. The project definition, however, is a description of what is wanted <u>not</u> how it is provided.

Glossary

prologue A sequence of comments at the beginning of each program module specifying the module name, purpose, calling form, input, output, global and local variables, author, and date.

pseudocode A program-like, but informal, notation containing natural-language text used to describe how a procedure or program functions. Usually, the control flow is expressed in programming terms, while the actions are expressed as narrative prose. Pseudocode is mainly used as a design aid.

reference document A complete description of the software product explaining all commands, options, alternatives, errors, methods, etc.

regression testing **Testing** a previously verified program. This is required following program modification for extension or correction.

structural English A simple every-day English used to describe processes in a data-flow diagram.

structural testing **Testing** a program with data derived solely from the program structure.

structure charts Charts used to show the overall structure or hierarchy of a program in terms of the program modules and their interfaces.

stub A substitute component used temporarily in a program so that progress can be made. If a program must be tested before a procedure has been fully developed, the procedure could be replaced by a stub that is known to work. A stub could be used under a variety of circumstances--for example, it could be used to always return the same result, return values from a table, return an approximate result, consult a file, etc.

summary sheets Notebook-sized condensations of reference documents for software products.

template A generalized skeletal form that can be expanded to provide information about a specific routine or software product. A template allows a programmer to fill in information for building his own prologue or documentation.

testing 1. The process of executing a program with the intent of finding errors. 2. Examination of the behaviour of a program by executing the program on sample data sets.

Glossary

tutorial document Information that a teacher would use in explaining material to a group of students.

user interface That portion of a software product specifying how a person running the software communicates with the software.

walkthrough A product review by a small group of people, not all of whom were involved in the creation process.

REFERENCES

1. R. L. Glass, "A Minimum Standard Software Toolset," *ACM-SIGSOFT Software Engineering Notes* 7(4) (1982).

2. H. Moll, *The TRIX Report Editor*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-818 (1984).

3. J. C. Beatty, *REDPP--A Postprocessor for the TRIX/RED Report Editor*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-3012, Rev. 1 (1977).

4. K. O'Hair, *Computer Graphics by Example, Part 3--REDPP: A Post Processor for TRIX/RED*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 (1978).

5. D. Lai, *SPELLING--A Program to Check Spelling of Words in a File*, Lawrence Livermore National Laboratory, Livermore, CA, Unpublished (1983).

6. J. C. Beatty, *PICTURE--A Picture-Drawing Language for the TRIX Report Editor*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30156, Rev. 1 (1979).

7. K. O'Hair, *Computer Graphics by Example, Part 4--PICTURE: A Picture-Drawing Language for TRIX Report Editor*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 (1978).

8. C. Streeter, *SCI (Structure Chart Interface) Users Manual*, Lawrence Livermore National Laboratory, Livermore, CA (1982).

9. J. S. Chin, *MCHARTSC: A Program that Creates and Modifies Milestone Charts*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30165 (1978).

10. K. O'Hair, "Help with REDPP," *Tentacle*, 2(3), 16 (1982).

11. P. Keller, *FTE: A Resource-Allocation Program for Managers*, Lawrence Livermore National Laboratory, Livermore, CA, UCRL-52244 (1977).

12. B. Kelly, *Programming Environment Project Definition*, Lawrence Livermore National Laboratory, Livermore, CA, Unpublished (1983).

References

13. J. White, *Editing by Design* (Bowker, New York, 1982).

14. J. Minton, D. Schnabel, J. Huskamp, G. Whitten, and K. Dusenbury, *URLIB--A Subroutine Library for Writing Utility Routines*, Lawrence Livermore National Laboratory, Livermore, CA, M-048-Part 2 (1979).

15. K. O'Hair, *LR System User Manual*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-313, Draft (1985).

16. M. Page-Jones, *The Practical Guide to Structured Design* (Yourdon Press, New York, 1980).

17. A. Cecil, H. Moll, and J. Rinde, *TRIX AC--A Set of General-Purpose Text-Editing Commands*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-808, Draft (1985).

18. E. Yourdon and L. L. Constantine, *Structured Design* (Yourdon Press, New York, 1978).

19. W. S. Derby, J. T. Engle, and J. T. Martin, *LRLTRAN Language Used with the CHAT and CIVIC Compilers*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-302 (1981).

20. E. Yourdon, *Structured Walkthroughs* (Yourdon Press, New York, 1978).

21. K. Fong, T. Jefferson, and T. Suyehiro, *SLATEC Common Mathematical Library Source File Format*, Lawrence Livermore National Laboratory, Livermore, CA, UCRL-53313 (1982).

22. K. Dusenbury, *Summary Sheets*, Lawrence Livermore National Laboratory, Livermore CA, LCSD-50, Rev. 2 (1984).

23. R. E. Cooper, *FOCAL*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-1630 (1982).

24. J. Rowe, *KLEAN: Clean Up Messy Fortran*, Lawrence Livermore National Laboratory, Livermore, CA, OC-956 (1975).

25. R. Johnson, *KWICLIST*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-1647 (1981).

26. L. Chase, *MCE*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-5241, Rev. 2 (1982).

27. D. E. Johnson, *PRECOMP*, Lawrence Livermore National Laboratory, Livermore, CA, UR-920 (1979).

References

28. D. W. Thompson, *RELABEL*, Lawrence Livermore Laboratory, Livermore, CA, UR-214 (1979).

29. L. Chase, *SML*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-18887, Rev. 1 (1982).

30. G. J. Myers, *The Art of Software Testing*, (John Wiley & Sons, New York, 1979).

31. F. Gruenberger, "Program Testing and Validating," *Datamation* 14(7), 39-47 (1968).

32. Computer Documentation Group C-2, *Computing Division Plan for User Documentation*, Los Alamos National Laboratory, Los Alamos, NM, LA-9807-MS (1983).

33. K. Dusenbury, *Glossarium*, Lawrence Livermore National Laboratory, Livermore, CA, UCIR-929 (1975).

34. D. Walsh, *A Guide for Software Documentation*, Advanced Computer Corporation, New York (1969).

35. D. Seberger, *DDT*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-1620 (1981).

36. K. O'Hair, *Computer Graphics by Example, Part 1, FTE: Produce Resource Allocation Charts*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 Part 1 (1978).

37. K. O'Hair, *Computer Graphics by Example, Part 2, PLOTPK: Plot and Analyze Data*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 Part 2 (1978).

38. K. O'Hair, *Computer Graphics by Example, Part 3, REDPP: Post Processor for TRIX RED*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 Part 3 (1978).

39. K. O'Hair, *Computer Graphics by Example, Part 4, Picture*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 Part 4 (1978).

40. K. O'Hair, *Computer Graphics by Example, Part 5, CHARTIT: Color Bar Charts*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 Part 5 (1978).

References

41. K. O'Hair, *Computer Graphics by Example, Part 6, SHADIT: Shaded-Area Graphs in Color*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 Part 6 (1978).

42. K. O'Hair, *Computer Graphics by Example, Part 7, TV80LIB*, Lawrence Livermore National Laboratory, Livermore, CA, UCID-30166 Part 7 (1979).

43. A. Dittli, *Introduction to FRAMIS*, Lawrence Livermore National Laboratory, Livermore, CA, LCSD-555 (1981).

44. S. Sparks, N. Smith, and G. Ledbetter, *LTERM Tutorial, Version 2.0*, Lawrence Livermore National Laboratory, Livermore, CA, LSCD-3 (1985).

45. D. E. Knuth, *The $T_EX$book* (Addison-Wesley, Reading, MA, 1984).

LA

REVISION HISTORY

Rev.	Date	Description of Changes
--	1Apr86	New document

AVAILABILITY

Printed copies of this document are available in the Computer Documentation Library, T2106, Room 1001, Ext. 2-0592.

This document is also available online at LCC. To get a hardcopy sent to your output box, log onto a CDC 7600 and type

```
trix ac
print! nip guidelines boxann identification
end
```

where ! is the linefeed and *ann identification* is your box number and identification.

This document cannot be viewed at a TMDS monitor, because it contains graphics.

Comment Sheet

The Software Project Standards Committee welcomes comments from USD members and others. If you find errors or omissions, please let us know--especially if you are aware of useful tools that should be included in the next version of this document.

From:

Name_____

Address_____

_____

_____

Telephone number_____

My comments are:

Staple Here

Fold Here

Software Project Standards Committee,
Jeanne Martin, Chair

Lawrence Livermore National Laboratory

P. O. Box 808 L-300

Livermore, CA 94550

U.S.A.

Fold Here First