South Carolina Department of Transportation

Electronic Toll Collection System & Related Services

For the
Cross Island Parkway Toll Facility
Hilton Head, South Carolina
Contract P.O.# 231709

PROGRAMMER'S PROCEDURES MANUAL

Rev. 0.0

June 1998

LOCKHEED MARTIN

1. Methodology

1.1 Purpose

The purpose of this document is to outline the required practices for software development including standards and procedures. All development will adhere to the standards outlined in this document. The processes will be tailored and applied to the various projects and subsystems in a manner that best benefits the project. This chapter gives an overview of the development methodology. Chapter 2 describes the relevant software standards and guidelines, and chapter 3 describes the various procedures.

3.4.3 Design Review Follow-Up

A flow diagram for design review follow-up is shown in Figure 3-3.

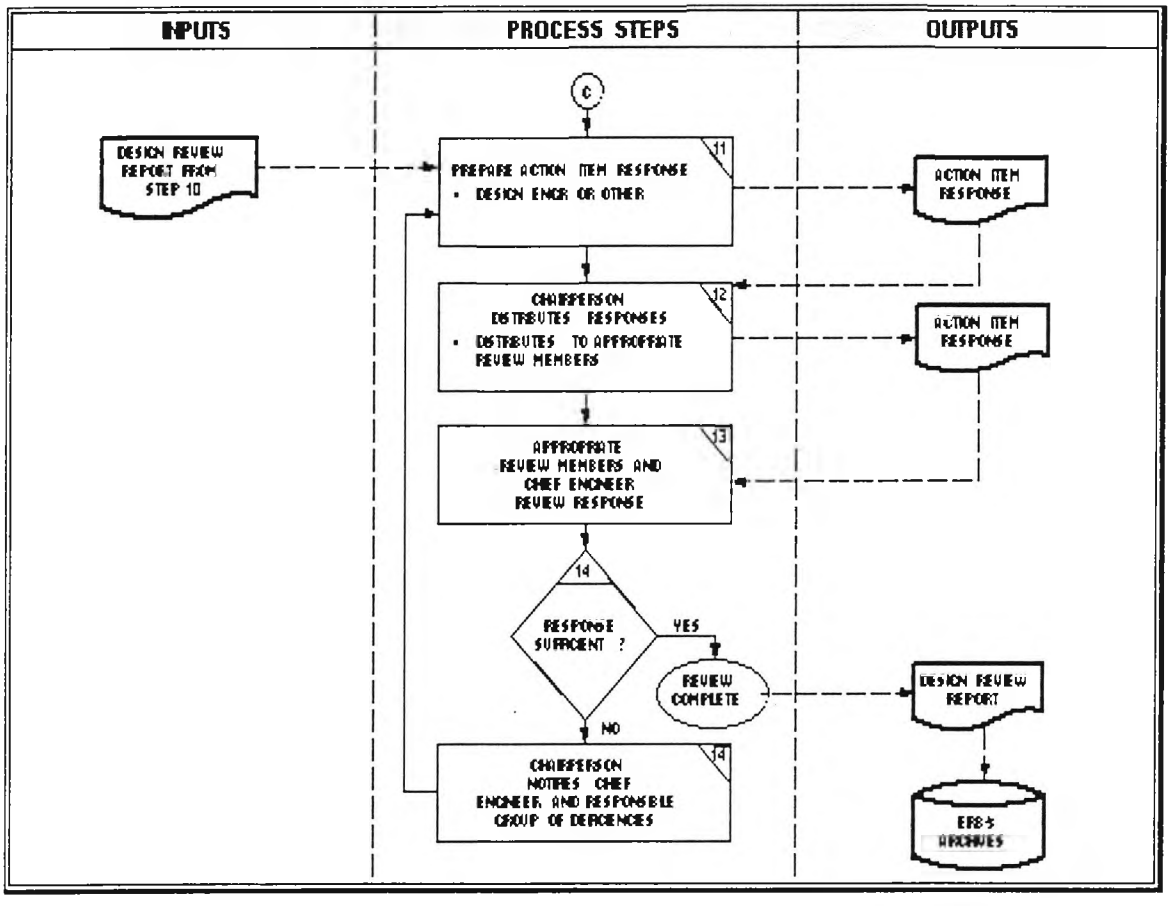

Figure 3-3 Design Review Follow-Up

Action Item responses shall be prepared by designated actionees, approved by the responsible manager, and addressed to the Design Review Chairperson.

1. Upon receipt of an action item response, the Chairperson shall distribute the response to the Design Review Board and to other appropriate reviewers.
2. The Chairperson and other action item response recipients shall review the response for completeness and correctness, then notify the ERB of any objection to closure of the action item within 2 weeks.

   It shall be the responsibility of the Chairperson to ensure that all action items are tracked for adequate closure according to schedule. It shall be the responsibility of the Chief Technical Officer to ensure that all action items are closed according to the established schedule.

Lockheed Martin IMS

PROGRAMMER'S PROCEDURES MANUAL
Rev. 0.0

Prepared by: Mary Thomas, Documentation Manager

Approved by: Natesh Manikoth, Chief Technical Officer

Approved by: Greg Saville, Technical Lead

Approved by: James J. Eden, Project Manager

Contents

3. The Chairperson's design review report will describe the conclusions, problems, and Action Items. It shall indicate the group or individual design review responsible for closing the Action Items with due dates.

1.2 Methodology

The overall software development methodology is briefly outlined in this section. The development process will be in the context of an object-oriented view of the system. The figure below is an overview of the processes involved.

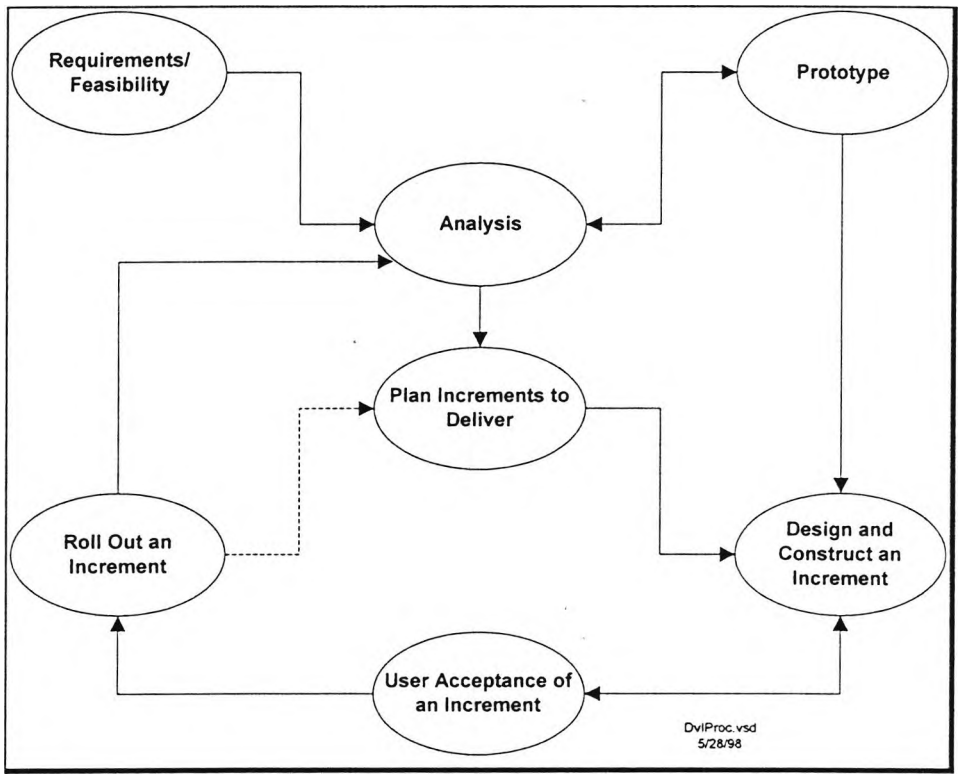

Figure 1-1 Development Processes

1.3 Requirements

The requirements process should:

1. Identify the scope of the development
2. Identify the details of all interfaces
3. Decide which use cases the system will support
4. Develop a first cut at time and cost estimate

Use case modeling should be adopted to document the outputs of this stage.

1.4 Analysis

The purpose of this phase is to fully define the problem in the proposed business solution. This phase should also provide a sound basis for later incremental development and for the refining of estimates made during the requirements phase.

The models developed are:

1. Business object model
2. Interface object model
3. Dynamic model
4. Interaction model

To document the outputs from this phase, use OMT with UML notation.

1.5 Prototype

The purpose of the prototyping phase is to agree on UI design with the users and to clarify requirements. Prototyping should be used primarily for UI-related development.

The models developed are:

1. GUIs
2. Interface object model

1.6 Plan Increments to Deliver

The purpose of this phase is to develop a plan that indicates the increments to deliver and to identify the mapping of the different use cases to the increments.

The models developed are:

Development schedule

1.7 Design and Build an Increment

The purpose of this phase is to develop usable functionality that is consistent with the requirements as documented in a use case. Also, this phase should be used to refine the understanding of the remaining increments.

The models developed/enhanced are:

1. Business object model
2. Interface object model
3. Storage object model
4. Dynamic model
5. Interaction model

To document the outputs from this phase, use OMT with UML notation. The applicable coding standards are detailed in Chapter 2.

1.8 User Acceptance of an Increment

The purpose of this phase is to demonstrate the functionality and to achieve formal acceptance for the increment. During this phase, a formal test plan should be created for the increment. Also, the relevant user training in the features being implemented in the increment should be developed.

1.9 Roll-Out of an Increment

During this phase, the completed system is rolled out into the user environment via established CM procedures.

2. Standards

2.1 PowerBuilder

2.1.1 PowerBuilder Coding Standards

2.1.1.1 General Standards

Many of the standards entitled 'C Standards' in this section are applicable to GUI coding. In particular, the established guidelines concerning the in-line commenting of source code should be applied by the GUI programmer as well.

2.1.1.2 Naming Conventions

Variables in PowerBuilder scripts should be named according to the following convention:

<scope initial><data type initial>_<variable_name>

The 'variable_name' component should be meaningful: for example: gs_VariableName for a global string, 1d_VariableName for an instance real, and si_VariableName for a shared integer.

2.1.1.3 PowerBuilder Objects

PowerBuilder objects should be named according to the following convention:

<PB_object_type><variable_name>

The 'PB_object_type' component is taken from the following list:

window	w_
data window	d_
drop down data window	dddw_
data window control	dw_

function	f_
menu	m_
structure	s_
query	q_
user object	u_

2.1.1.4 PowerBuilder Functions

PowerBuilder functions should be named according to the following convention:

<PB_function_type><variable_name>

The 'PB_function_type' component is taken from the following list:

window function	wf_
menu function	mf_
user object function	uf_
global function	f_

2.1.1.5 PowerBuilder Structures

PowerBuilder structures will be named according to the following convention:

<PB_structure_type><variable_name>

The 'PB_structure_type' component is taken from the following list:

window structure	ws_
menu structure	ms_
user object structure	us_
global structure	s_

Instances of PowerBuilder structures should be named according to the following convention:

<scope initial><object initial>str_<variable_name>

2.1.1.6 PowerBuilder Scripting Standards

Programmers should follow certain standards that are specific to coding in the PowerBuilder scripting language:

- Always use dot notation.
- Whenever possible, avoid referring to windows by name. Instead, use constructs such as the parent pronoun or the 'this' pronoun to refer to objects.
- Capitalize the first letter of all words for functions and for flow-of-control structures — e.g., If, Else, DWGetUpdateStatus.
- Declare all variables at the top of the PowerBuilder script.

2.1.1.7 PowerBuilder Portability Issues and Standards

To ensure that the PowerBuilder code developed for the Windows GUI will work correctly not only in the Windows implementation but also in the UNIX / Motif implementation of PowerBuilder, programmers should follow several additional guidelines formulated by Powersoft:

- Do not use DDE. Although DDE will be supported under UNIX, an application that uses it will be unable to communicate with any non-PowerBuilder applications.
- Whenever possible, use PowerBuilder functions and avoid the using external function calls. For example, a call to a Windows API function will not work on the UNIX platform.
- Do not use the file WIN.INI to store session options for the application. Instead, use the application's own .INI file.
- Do not use VBX controls, because they will not be supported under UNIX.
- Do not Windows message ids, because they have no meaning on the UNIX platform.
- Avoid the use of fully specified pathnames in PowerBuilder scripts, because name conventions are platform specific. If the use of a pathname is unavoidable, encapsulate the portability problem: build the name dynamically — by (writing and) using functions such as GetDiskName() and GetDirectorySeparator(). Only the module that contains these functions is platform specific.
- Do not use the second mouse button, because it will probably not be supported under UNIX.

2.1.1.8 Window Controls within PowerBuilder

Window controls within PowerBuilder will be able to retain the prefix that PowerBuilder places on them by default. The following is a list for reference:

CheckBox	cbx_
CommandButton	cb_
DataWindow	dw_
DropDownListBox	ddlb_
EditMask	em_

Graph	gr_
GroupBox	gb_
HorizontalScrollBar	hsb_
Line	ln_
ListBox	lb_
MultiLineEdit	mle_
Oval	oval_
Picture	p_
PictureButton	pb_
RadioButton	rb_
Rectangle	r_
RoundRectangle	rr_
SingleLineEdit	sle_
StaticText	st_
UserObject	uo_
VerticalScrollBar	vsb_

2.1.1.9 Configuration Control and Code Management

The LMIMS development environment includes the DEC (Digital Equipment Corporation) Code Management System as well as several software repository and change-control tools developed in-house.

2.1.2 PowerBuilder Windows Standards

2.1.2.1 Window Design Standards

2.1.2.1.1 Window Types

The application frame will be an MDI frame with MicroHelp. At initial application startup, this window will be maximized. If possible, the application should save the size and positioning of this window before closing and should restore it upon restarting.

Documents (sheets) will use Main as a window type. The application will ordinarily position newly opened documents by cascading. In any case, the application must open all documents in the same manner. If possible, before closing, the application should save the size and positioning of all documents, restoring them upon restarting.

Dialogs will use the Response window type and will be centered on the screen.

The application will present information to the user through windows of MessageBox type centered on the screen.

The application will provide Help through popup windows, filling (initially) the right-hand half of the screen.

2.1.2.1.2 Colors

To comply with the GUI mandate to enable the user to control the application, many of the colors used will come from those chosen by the user through the Windows Control Panel. To ensure that certain application features are visually clear, however, there are some important exceptions to this principle.

The MDI Frame background color will be the (user-selectable) Windows "application color."

Documents will use the Windows "windows color" for their background. The "text" of documents will be black, with Static Text (e.g., labels, headings) using the "window text" color of Windows. However, documents that are reports provide an important exception to these rules — all reports will use black text on a white background.

Window controls will use "window text" colored text on a "window color" background. An exception is the Single Line Edit control (SLE). All SLEs will use black text on a white background.

2.1.2.1.3 Fonts

The application will use the Arial TrueType font for all text displayed, specifically:

labels	10-point **bold** Arial
report titles	16-point **bold** Arial
headers	10-point **bold** Arial
static text	10-point **bold** Arial
data	10-point **bold** Arial
controls	10-point **bold** Arial

2.1.2.1.4 Borders

The application will use the following conventions for borders:

labels	none
headers	none
static text	none
editable data	3D Lowered
protected data	none
controls	3D Lowered
data windows	3D Lowered

2.1.2.1.5 Navigation Methods

Along with a menu system, the application will typically provide toolbars. However, any action that can be invoked from a toolbar **must** also be accessible through a pull-down menu. Also, every action will be accessible from the keyboard; the application will not require the use of a mouse.

Field traversal shall be from the top left of the window to its bottom right, and any command buttons will be located at the window's bottom right.

To ensure that the product is consistent with common Windows applications, the following names shall be used only to describe the actions indicated below:

Action	Meaning in Edit Context	Meaning in Database
Insert	inserts after the cursor	adds a record to the DB
Delete	removes highlighted line(after confirming)	removes a record from the DB
OK	saves and closes the window	updates database and closes the windows
Save / Apply	saves only	updates the DB only
Cancel	exits (and / or cancels action)	exits (and / or cancels action)

2.1.2.1.6 Field Formats

- Dates will be presented in the form mm/dd/yy.
- Times will be displayed in the form hh:mm A.M. or hh:mm P.M.
- All monetary amounts will use commas — for example, $52,500 and not $52500 — and will have the "$" sign.

2.1.2.1.7 Menus

The MDI Frame window will display a menu that contains at least one of the following:

File Window ΛΛΛΛ Help

Other items will appear in the menu as needed. In any menu, items currently unavailable to the user should be grayed. Do not include items that will never be available.
Menus will use accelerator keys and may use shortcut keys.
Cascaded menus will go no more than two layers deep.

2.1.2.1.8 Controls

Limit the number of controls to no more than 20 per window. Whenever possible, use standard user objects — rather than customized ones — for controls. Controls that are currently unavailable to the user should be grayed. Do not display controls that will never be available.

2.1.2.1.9 Specific Controls and their Uses

- **Static Text** displays text and should not be followed by a colon.
- **Command Buttons** initiate an action. Command Buttons should be 298-by-109 PowerBuilder units in size. Text within Command Buttons should be bold. Every screen should contain a default Command Button.
- **Picture Buttons** are Command Buttons that contain bitmaps instead of text. For performance reasons, the size of the displayed bitmap should match the size of the Picture Button
- **Checkboxes** turn options on and off. Use right-aligned text in Checkboxes. Text in Checkboxes should be **bold**.
- **Radio Buttons** enable the selection of mutually exclusive options. Use three or less at a time. Align Radio Buttons vertically, and right-align the text within them. Text in Radio Buttons should be **bold**.
- A **List Box** shows a predefined set of choices, enabling multiple selection and providing scrolling through a Vertical Scroll Bar. By default, the number of items displayed before scrolling is seven.
- A **Drop Down List Box** shows a predefined set of choices, enabling a single selection and providing scrolling through a Vertical Scroll Bar. A Drop Down List Box should always show the arrow and may allow editing. By default, the number of items displayed before scrolling is seven.
- In a **Single Line Edit**, data should be left-justified.
- A **Picture** should, for performance reasons, always be the same size as the bitmap it displays.
- A **Data Window** is used to display or manipulate data from the database. Report data should always be displayed in a Data Window with any "text" left-aligned and any "numerics" right-aligned at the decimal point. Labels in Data Windows should not be followed by colons.

2.1.2.2 Starting the Application

All applications will use the same startup screen(s), customized with suitable text. The code to perform these operations is available as a library object.

2.1.2.3 Providing Application Feedback to the User

- All reporting and corrective handling of errors will use a standard Error Screen., available as a library object.
- The application will provide Micro-Help whenever possible.
- The application may provide audible feedback, but it must not assume that audible feedback ever reaches the user.

2.2 Forté

This section presents a set of guidelines and standards for coding in the Forté application development environment. The goal is to provide a foundation for efficient and consistent coding across applications developed in Forté. Adherence to standards and conventions promotes the use of a consistent "language" between developers — enabling easier and more-cooperative development and maintenance of software applications. By adhering to a well-understood set of coding and naming standards, code is more readable and can help in the quality assurance and testing of developed applications.

Section 2.2.1 addresses the naming of different elements within the Forté environment. Section 2.2.2 lists simple coding conventions that focus on the efficiency and readability of TOOL code. The goals of these two sections are:

- For all development constructs, to provide names that are both descriptive and concise. In order to promote readability and maintainability across developers, excessive abbreviations and symbolic names should generally be avoided.
- To define rules for TOOL grammar, spacing, and punctuation to increase the "flow" and readability of TOOL methods. Capitalization and naming standards should be designed so that individual elements of the Forté TOOL language — for example, classes, attributes, method invocations, and variables — can be easily identified when reading and maintaining TOOL methods.
- To provide names and conventions that eliminate the need for renaming of Forté elements during application maintenance and enhancement
- To prevent the improper or inefficient use of Forté TOOL constructs that could contribute to poor performance and functionality within an application
- To provide a defined coding (TOOL) style to be used throughout the project for handling a host of development issues and architectures

2.2.1 Naming Standards

2.2.1.1 General Conventions

2.2.1.1.1 Capitalization

The Forté TOOL language elements listed below should be in mixed case with the first letter of each unique word capitalized:

Element	Examples
Workspaces	JohnDoe, JohnDoeTest
Projects	DomainServices, DomainWindows, DomainClasses, DomainFrontCounter Classes
Plans	
Forté Express Business Models	CustomerBM, OrderBM
Forté Express Application Models	PurchasingAM, CustomerServiceAM
Classes	Employee, Product
Methods	GetOrders(), ComputeTotal()
Attributes	aName, aStreetAddress, aQuantity
Events	eShipmentChanged, eEmployeeFired

Variables declared in TOOL code should be in mixed case with the first letter of the variable name in uppercase. All variables should include a lowercase "l" prefix.

Examples: lNewEmployee, lItemList, lAccountBalance

Declare and refer to constants in uppercase — use '_' (underscore) to separate compound word names.

Examples: FS_UPDATE, C_MAGENTA, MT_INFO

TOOL keywords — for example, begin, end, for, post — that are used in methods should be in lowercase.

When declaring variables in method code, reference to object datatypes — TextData, IntegerNullable, DateTimeData — should be in mixed case with initial capitalization. Scalar datatypes — string, integer, float — should be in lowercase.

2.2.1.1.2 Use of Abbreviations

Avoid abbreviations where possible — use only in cases where long names are unavoidable. Prefixes and suffixes clearly defined in naming standards are exceptions to this rule.

All abbreviations should be defined in project documentation.

2.2.1.2 Repository Workshop Elements

2.2.1.2.1 Project

Begin project names with a word that describes the application and that is uniquely identifiable within the first eight characters. This will enable easy identification of projects in the Repository Workshop that are related to the same application. Also, because Forté uses the names of projects to generate partition, directory structure, and runtime repository names, keeping the leading characters of a project name unique and descriptive promotes easier management of deployed applications.

Project names should be suffixed with an indication of the classes contained within them — for example, windows, services, business classes, utilities. Forté Express uses the following default suffixes when generating code from Business and Application Models:

Project Contents	Default Suffix
Business domain and related classes	Classes
Service classes and objects	Services
Windows and client-based objects	Windows

2.2.1.2.2 Business and Application Models (Forté Express)

Business Model names should be noun constructs and describe the primary class or classes included in the model — for example, Order, Employee, Customer. Because Business Models are primarily data oriented and can provide the foundation for multiple applications, phrases that describe a function or application within the business should not be included in Business Model names.

Application Model names should include the name of the Business Model from which they are generated. Also, each Application Model should include a description of the function or application area represented by the Application Model.

Projects, Business Models, and Application Models can normally be distinguished from each other either in the Repository Workshop by the icon associated with the name or by restricting the view of elements using the filter droplist in the workshop. If developers wish to distinguish between projects and models using naming conventions, suffixes rather than prefixes should be appended to project and model names.

Examples: OrderBM, OrderEntryAM, OrderFulfillmentAM
EmployeeBM, EmployeeRecruitingAM, EmployeeBenefitsAM

2.2.1.3 Project Workshop Elements

2.2.1.3.1 Classes

All class names should use nouns or noun phrases.

Business domain classes should describe the "real-world" objects represented by the class.

Examples: Customer, Order, Employee

Service class definitions should be suffixed with "Mgr"; service objects should be suffixed with "SO".

Service classes that manage the persistence of business domain objects — those that handle the storage and retrieval of data — should be named according to the primary business domain object managed by the service.

Examples: CustomerMgr, ProductMgr

Service classes that enforce business policy should be named according to the functions within the business impacted by the policy. Policy manager classes and service objects should include the word "Policy" in their names.

Examples: BillingPolicyMgr, HiringPolicyMgr

Window class definitions should be suffixed with "Window" and named according to the use case or function supported by the view.

Examples: SecurityTradingWindow, OrderEntryWindow

Classes defined for display purposes that map to compound widgets or display nodes should include "Display" in their names.

Examples: DepartmentDisplay, LocationDisplayNode

Exception classes should be suffixed with "Exception" and include the name of the class for which they are defined.

Examples: AddEmployeeException, CreditLimitExceededException

2.2.1.3.2 Cursors

Cursors should be declared with names that describe the data result sets that are retrieved by the cursor. Also, cursors should be suffixed with the word "Cursor".

Examples: FetchAllEmployeesCursor, FetchOrdersByNumberCursor

Constants should be prefaced with a defined character code designating the group or family of constants to which they belong. Examples of this naming convention are evident in Forté-defined constant names — FS_DRAGGABLE, MT_INFO. To distinguish user-defined constants from those

provided in the Forté projects, developers may want to create constant prefixes that are longer than two characters.

2.2.1.3.3 Domains

Domain names should describe the value set included in the domain and should be suffixed with the word "Domain".

Examples: AnimalDomain, PlantDomain, PrimeNumberDomain

2.2.1.4 Class Workshop Elements

2.2.1.4.1 Attributes

Business domain attributes should be descriptive of their real-world counterparts — for example, firstName, color, weight.

Pointer attributes (attributes that implement an association between two objects) should be named according to the role played by the attribute with respect to the object. For example, if the definition of a class called "Airplane" includes an attribute describing the Person (another class) who flies the airplane, an appropriate name for the attribute would be "Pilot".

Attributes that are arrays should be suffixed with the word "List" or "Sets".

Examples: ComponentList, AddressList, AttendeeSets

Window class attributes (widgets) are described in Section 2.2.1.5, Window and Menu Workshop Elements.

2.2.1.4.2 Methods

Method names should be created using verb-noun constructs. The "nouns" within a method name should refer to Forté elements — objects, attributes, events — that are operated upon by the method.

Examples: RefreshWidgets, GetCustomer

Methods should succinctly describe the task to be performed. To avoid confusion about its use, the behavior of a method should not include functionality beyond what is described by its name.

2.2.1.4.3 Events

Event names should use a noun-verb construct with the verb in past tense.

Examples: UDComplaintFiledEvent, FeePaymentCompletedEvent

Event names should include the object or service name for which the event is posted. Return Events have the suffix "RE", Completion Event have suffix "CE", and Exception Events have suffix "EE".

2.2.1.4.4 Constants

See Section 2.3.1.5 for a description of naming conventions for constants.

2.2.1.4.5 Event Handlers

Event Handler names should use a noun-verb construct with the verb in the past tense.

Examples: UDComplaintFiledEventHandler, FeePaymentCompletedEventHandler

Event handler names should include the object or service name for which the event is posted.

2.2.1.5 Window and Menu Workshop Elements

2.2.1.5.1 Simple and Compound Widgets

Simple widgets within a compound field that map directly to a business domain object *must* be named identically to the object's attributes. For example, widgets within an array field that maps to the Order class must have identical names to the attributes of the Order class (e.g., OrderID, CustomerName, TotalPrice).

If a compound field maps directly to an object, then the object name should be included in the compound field name. If the compound field is used solely to control display characteristics (e.g., alignment, visibility or invisibility of child widgets, window nesting), then the compound field should describe the area of the window in which it is displayed (e.g., ButtonBarGrid, StatusLinePanel).

To provide easy maintenance of windows, widget names should be closely associated with their displayed labels. Simply put, a pushbutton with a label of "Close" should include the word "Close" somewhere in its name.

Because many widgets can be converted to different types (e.g., a DropList can be converted to a FillInField), it is best to avoid including a widget's type in its name. If developers choose to include types in the names of widgets that are reasonably static, then it is best to include them as a suffix rather than a prefix. Common examples of widget-type suffixes include "Btn" for pushbuttons and "Grid" for gridfields.

Widget Type	Prefix	Examples

2.2.1.5.2 Menu Widgets

Menu widget names include the displayed label as part of the name — Copy, Cut, Open, Help.

To distinguish menu widgets from window widgets in the Class Elements list, menu widgets can be suffixed with an indicator of their type (this convention is acceptable for menu widgets because they cannot be converted between types). A suggested list of suffixes is:

Menu Widget Type	Suffix	Examples
Menu	Menu	EditMenu, FileMenu
Menu command	MC	PasteMC, OpenMC
List Item	MLI	CategoryMLI
Toggle	MT	ShowRulerMT
Separator	SP	EditSP

2.2.1.6 Method Workshop and TOOL Code Elements

2.2.1.6.1 Method Parameters

Preface parameter names with "p" to differentiate them from attribute or class names in TOOL code.

When passing objects as parameters, the name should provide an indication of the object being passed. If applicable, the parameter name should also describe the role of the object (e.g., pNewEmployee, pChangedOrder).

2.2.1.6.2 Variable Declarations

Variables that represent instantiated objects should describe either the name of the class or the role performed by the variable.

Examples: newEmployee : Employee = new;

purchasedGoods : Array of OrderItem = new;

empLookupWin : EmployeeWindow = new;

Variables for instantiated windows should be suffixed with "Win" to distinguish them from nonwindow classes.

2.2.1.6.3 Return and Exception Events

Return events defined on methods should follow the standard of "*methodname*_return".

Examples: FetchEmployees_return, RunReport_return

Exception events defined on methods should follow the standard of "*methodname*_exception".

Examples: ValidateOrders_exception, CollectProcessData_exception

2.2.2 TOOL Coding Conventions

2.2.2.1 General Conventions

To promote portability and maintainability, avoid hard-coding names external to the Forté environment into TOOL code statements, service object configurations, or environment configurations. Examples include database names or connect strings (use environment variables, where possible), operating system–specific commands (conditionally check the operating system environment).

Use Forté constants rather than hard-coded constant values in TOOL code. Place constants at the appropriate scope — Project, Class, or Method—when defining.

2.2.2.2 Line Length

Limit TOOL code lines to 80 characters or less in the Method Workshop. This prevents excessive line wrapping when printing method code.

2.2.2.3 Indenting

Indent four spaces for statement blocks; indent two spaces when a line must be broken.

2.2.2.4 Comments

Use /* ... */ for block comments (> 2 lines).

Use // for line comments — this enables user comments to be differentiated from Forté-generated comments in export files.

Comments should precede code or be included on the same line as the code that they describe.

To document an entire class, create a method with a name of "_Documentation" and include a block comment that describes the entire class. The "_" (i.e., the underscore) in the comment method name causes it to sort close to, or at the top of, the list of class elements.

2.2.2.5 Method Invocation (Same for events, cursors)

When invoking methods, always include parentheses after the method name, regardless of the number of parameters passed.

Use named parameters.

Always give the object name with the method — e.g., self.method(), newEmployee.GetSalary().

2.2.2.6 Overloaded Methods

Eliminate the use of default values for overloaded method parameters, because this may lead to ambiguous invocations of the method.

2.2.2.7 Parameter Passing

Parameters should be referenced by name — rather than by position — when invoking a method.

2.2.2.8 Grammar

Use qualifiers on "end" statements (e.g., end event, end for, end if).

Use spaces around colons and equal signs to improve code readability.

2.2.2.9 Statement Blocks

Indent nested statement blocks and comment them to indicate why they exist.

2.2.2.10 Transaction Handling

Be aware of dialog duration when coding transaction blocks; assume and strive for message dialog duration.

2.2.2.11 SQL and Database Conventions

Name database columns in SQL statements explicitly. Avoid the use of "*" in SELECT statements, to prevent ambiguous or improper mappings of columns to object attributes.

2.3 C Programming Language Coding Standards

This section formally specifies the C coding standards and practices that are to be used in the development of code for the LMIMS projects. These coding standards have been developed to ensure a unified appearance of the code and associated comments as well as to enhance the overall maintainability of the code throughout its life cycle. Within the context of this section, "shall" is used to express mandatory provisions, and "should" and "may" are used to express suggested and / or preferred provisions.

2.3.1 Lexical Elements

The following sections describe the coding standards, practices, and formats for lexical elements.

2.3.1.1 Character Sets

Alphanumeric characters and underscores shall be sufficient to represent any routine name, variable, or defined constant.

2.3.1.2 Routine Names

The routine name shall be in the following format: <system ID>_<function name>. The <system ID> prefix should generally be a five-letter prefix that identifies the particular software system. A single underscore separates the <system ID> prefix from the <function name>.

Example: plfoo_function

If an object-oriented approach is taken to naming routines, the <function name> should be composed of an <object> prefix and the <function name> separated by two underscores as follows: <system ID>_<object>_<function name>.

Example: plfoo_object__function

Distinct words in the <function name> shall be separated by single underscores:

Examples: plfoo_a_function_name, plfoo_object__a_function_name

Routine names shall be limited to a total of 28 characters.

2.3.1.3 Source Code File Names

The name of a file containing one or more C routines shall be the same as the primary routine it contains with an extension of C.

Each C file shall contain only one routine that is called from routines outside that file. This is the routine for which the file is named. If additional routines are included in the same file, they shall be used only within that file and shall not be called by any routine outside that file.

Unless specifically authorized, there shall be one IFDL screen per file, and the file shall have the same name as the screen with "_FORM" and the extension of .IFDL.

SQLMOD files shall be named in the format TCS_nnn, where nnn is the three-character identifier assigned to the SQL table to which the SQLMODs apply.

2.3.1.4 Variables

Variable names shall be descriptive. Distinct words in the variable name shall be separated by single underscores.

Example: foo_indicator

To maintain portability, variable names shall be limited to a total of 31 characters.

Global variable names shall be constructed in a manner similar to routine names as follows: <system ID>_<variable name> or <system ID>_<object>__<variable name>. The <system ID> prefix will generally be a 5-letter prefix that identifies the software system. A single underscore separates the <system ID> prefix from the <variable name>.

Examples: plfoo_foo_indicator, plfoo_object__foo_indicator

WARNING: Be certain that every variable referenced as "extern" is actually declared globally (outside of any functions) somewhere in a module that will be linked with the "extern" reference module. Under VAX / VMS, failure to do so will not yield and error. Instead, the linker will automatically create a global of the same type as the one referenced as "extern." This situation can lead to side effects that are very difficult to trace.

2.3.1.5 Defined Constants

Literal constants — defined with the #define precompiler directive — shall be composed of only uppercase letters, numbers, and underscores. These constants shall be named descriptively.

To avoid confusion, non-unique constants should be qualified within the file in which they are defined.

Examples: FOOVALUE, PLFOO_FOOVALUE, PLFOO_OBJECT__FOOVALUE

It is not necessary to use unique constant names among modules, but it will aid in the maintainability of the code by indicating the probable location of definition.

Constants that must be used among modules should always be qualified with the module name in which they are defined. This is necessary in order to avoid redefining constants during the precompilation phase. Constants that are defined in header files for utility libraries are good examples of possibly non-unique constants.

Commonly defined constants include TRUE, FALSE, ONE, TWO, THREE,..., DEBUG. Qualifying each of these with the module name will assure you that your constant will not be redefined. The example below illustrates the problem with multiple definitions of the same constants within the same compilation unit.

Example: #define ONE (short)1

#define ONE (double)1.0

The constant ONE is now redefined as a double-precision constant with a value of 1.0.

The constants "VAX" and "UNIX" shall be used with the #ifdef precompiler option to separate platform-dependent code. These constants are defined by the compiler itself. No attempt shall be made to redefine them.

Example: #ifdef VAX

statements

#endif

#ifdef UNIX

other statement

#endif

The #else precompiler option shall not be used in conjunction with the platform-dependent compilation sections. In the future, other platforms may be used that may not default to the #else case.

2.3.1.6 Macros

Macros are a type of defined constant in which argument substitution takes place when the macro is expanded during precompilation phase.

Macros offer the advantage of making code more readable and maintainable. The disadvantage is that they expand as code in the module in which they occur, thereby having the effect of repetitious code (they may increase the size of the object module).

Example: #define CHANGE_TO_UPPER_CASE(arg1) toupper(arg1)

Macros should be used only when necessary. In most cases, a routine can be written that is easier to understand than a macro. The following is an example of macro use that is necessary. The ANSI standard function for file deletion is "remove." UNIX implemented this function as "unlink." Similarly, DEC implemented this function as "delete." In these cases, the ANSI standard "remove" should be used and should be a defined macro that expands to its platform-specific implementation.

Example:

```
#ifdef VAX
#define remove(a) delete(a)
#endif

#ifdef UNIX
#define remove(a) unlink(a)
#endif
```

2.3.1.7 Banner Comments

A header comment block shall be included for every routine. The comment block is used to identify the routine, its use, any interfaces, and its change history. Each required section of a banner is described in the paragraphs following the example below.

Example:

```
/*************************************************************
* Title:          plfoo_foo
*                     Do the things that a foo function should do.
*
* Date:           19-May-1989
*
* Author:         M.R. Doe
*
* Summary:        This function will convert two single foos
*                 into a double foo. A completion status is also returned.
*
* Special Considerations:
*                 This functions should not be used for nonfoo conversions.
*
* Inputs:         foo1
*                     used as a prefix.
*                 foo2
*                     used as a suffix.
*
* Input/Output:   in_out_foo
*                     foo that is altered and returned. Input values
*                     must be positive integers less than 500.
*
```

```
* Outputs:        foo3
*                      foo composed of foo1 and foo2.
*
* Returns:        A short int indicating completion status.
*
* Calls:          plfoo_foo_adder
*                 plfoo_foo_formatter
*
* Called By:  plfoo_requester
*
* Globals:        plfoo_foo_indicator
*                 plfoo_foo_saver
*
* History:        19-May-1989  M.R. Colligan
*                      Initial release
*
*                 20-May-1989  M.R. Colligan
*                      SDR #123456  Fixed missing EOF problem.
*
*************************************************************/
```

2.3.1.7.1 Title

The routine name as it appears in the code. It shall be followed by a one- or two-line explanation of the routine that can be easily understood at a glance.

2.3.1.7.2 Date

The date of creation of the file in dd-month-yyyy format. This date may not be the date of its initial release because the routine may undergo several changes before it is actually released.

2.3.1.7.3 Author

Name of the software engineer who coded the routine

2.3.1.7.4 Summary

A short summary of the routine from a user's point of view. If it is important for the user to understand the algorithm that is used in the routine, then that information too shall be included. Usually, a brief explanation of how and when to use the routine will be enough.

2.3.1.7.5 Special Considerations

Any specific constraints, conditions, error handling procedures, etc., that should be conveyed to the user and / or the maintainer should be provided in this section.

2.3.1.7.6 Inputs

A list of formal input parameters as they appear in the code. A description of each parameter — containing the type and use of the parameter — is required. Also, limits shall be expressed here if they are critical.

2.3.1.7.7 Input / Output

Another option for parameters whose initial values are used and altered by the routine. A description of each parameter — containing the type and use of the parameter — is required. Also, limits may be expressed here if they are critical.

2.3.1.7.8 Outputs

A list of formal output parameters as they appear in the code. A description of each parameter — containing the type and use of the parameter — is required. Also, limits may be expressed here if they are critical.

2.3.1.7.9 Returns

A description of the type and possible values returned by the routine

2.3.1.7.10 Calls

A list of routines called by the routine. Only routines and system calls need to be listed here. Do not include C library functions such as "strcpy".

2.3.1.7.11 Called By

A list of routines that call this particular routine. "Various Routines" is used for common routines such as date / time conversions. "TBD" is used when the calling routine is not known.

2.3.1.7.12 Globals

A list of global variables that are referenced by the routine

2.3.1.7.13 History

The history consists of four pieces of information. The first entry is the release date. This is the date on which the code is actually released to CM. Again, the format is dd-month-yyyy. Next is the name of the engineer who is responsible for the change as coded.

IMPORTANT: The next line contains the SDR or SCR number for which the code change was made, followed by a brief description of the change(s) made for the SCR. Do not explain why the change was undertaken or what the problem was before the change.

2.3.1.8 In-Line Comments

In-line comments shall appear at the right-hand side of the page. These comments should be delimited by a single set of comment delimiters per lineComments shall not extend past column 80.

Example: Right

```
/*  This is a comment that is          */
/*  continued on the next line.        */
```

Do not continue in-line comments from one line to another as in the following "Wrong" example.

Example: Wrong

```
/*  This is a comment that is
    continued on the next line         */
```

To avoid the accidental commenting-out of intermixed code, in-line comments are not continued between lines. This frequently occurs when a line of code and its comment are deleted. Also, all left and right comment delimiters shall be aligned vertically as illustrated above.

Comments may not be necessary for every line of code. However, the function shall contain enough in-line comments to enable the reader to trace the flow of execution from the comments alone.

An in-line comment should accompany each SDR / SCR change and should give the SDR / SCR number along with a description of the change.

Example:

```
foo = foo_value + fudge_factor;
      /*  SDR #123456 - fudge          */
      /*  factor for precision         */
```

2.3.1.9 Comment Blocks

Avoid using comment blocks as a substitute for in-line comments. A series of good in-line comments is generally easier to follow. Successive in-line comments can be used to create a narrative of the code and can be placed at points that may be unclear without the comments.

Functions that require comment blocks to explain separate sections can usually be broken down into several smaller functions, each having its own banner comment.

2.3.2 Declarations and Types

2.3.2.1 Declarations

Each variable or function declaration shall begin a new line and shall be indented to the same level. Variables of the same type shall be listed on separate lines following the statement of the data type and must be delimited with commas.

```
Example:    short    foo_var1,    /*  The value of an uncomputed foo.    */
                     foo_var2;    /*  The value of a computed foo.       */
            char     foo_char;
```

Variable declarations may contain in-line comments to explain their use or typical values.

Variables that are initialized only once should be initialized in the declaration. Variables that are continually reset, such as loop counters and flags, should be set with assignment statements.

Constants shall be used in place of variables whose values do not change.

2.3.2.2 Typedefs

Typedefs should be used whenever the data they represent is logically distinct and should not be mixed in computations. It may also be useful to use typedefs when data must be constrained to a specific set of values.

Type definitions shall be named with the following format: <system ID>_<type name>_type. This will enable the reader to more easily recognize the source of a type when a variable of that type is declared.

To avoid confusion, never name a variable the same as its type. If the previous type naming format is used, this should not be a problem.

```
Example:    Right    foo_color_type    foo_color;
            Wrong    foo_color         foo_color;
```

2.3.2.3 Enumerated Types

Enumeration-type definitions shall contain a single enumerated value per line, and the enumeration-type values shall be in capitals.

Example:

```
enum
{
        BLUE,
        YELLOW,
        RED
} foo_primary_color_type;
```

2.3.2.4 Char Types

Single-character variables shall be assigned only single characters that are specified with single quotes.

Example:

```
char    foo;

Right   foo = 'A';
Wrong   foo = "A";
```

The case marked "Wrong" above will cause unexpected results, depending on the compiler that is used.

2.3.2.5 Boolean Types

Avoid using the predefined unsigned char values "TRUE" and "FALSE." It is safer to declare your own constants or enumerated values than to rely on your interpretation of these values.

Example:

```
#define   FOO_TRUE    1
#define   FOO_FALSE   0
```

2.3.2.6 Integer Types

Short integer types should be used whenever possible as applicable to the specific application. Do not use integer declarations. Use "long" or "short" types, because integers are not portable.

2.3.2.7 Float Types

Single-precision floating types should be used whenever possible. Double-precision types should be used whenever accuracy over several calculations is required.

All modules compiled under VAX / VMS shall be compiled with the /G_FLOAT option. G_FLOAT is an alternate internal representation of the floating type. This is necessary to avoid mixing floating types, because some third-party software modules used the G_FLOAT representation. When the G_FLOAT compiler option is used, all "float" variables in the module will be represented in this format.

2.3.2.8 Arrays

Because C array subscripting begins with "0", all arrays will be treated as such. Do not begin subscripting arrays with "1" — this is confusing from a maintenance point of view and may even cause some compilers to optimize the first element away.

2.3.2.9 Char Arrays

When working with character arrays, it is important to remember the null byte.

Example: Right

```
char    foo_array[11];
strcpy(foo_array,"0123456789");
```

Wrong

```
char    foo_array[10];
strcpy(foo_array,"0123456789");
```

2.3.2.10 Structures

Typedefs shall be used for all structures that are instantiated more than once. Structure-type definition shall contain a single field definition per line.

Example:

```
typedef struct foo_record_type
{
    short    field1;
    short    field2;
    short    field3;
};
```

2.3.3 Names and Expressions

2.3.3.1 Names

Do not attempt to reuse names that are already used by any of the C libraries — that is, do not attempt to name your own function with a standard library name such as "strcpy".

Do not overload variable names. This involves naming a variable the same as its type. Although syntactically correct in some cases, doing so may lead to confusion for the compiler and the maintenance engineer.

2.3.3.2 Expressions

In general, parentheses should be used to clarify the precise order of evaluation in any expression where the order may be unclear to a reader even if C does not require the parentheses to achieve the desired order.

2.3.3.3 Operators and Expressions

Parentheses shall be used to clarify the precise order of evaluation in any expression where two or more operators — logical or mathematical — are used, unless they are all the same operator or have the same level of precedence.

Example: A + B * C shall be coded as A + (B * C)

A + B – C is ok, because both operators are at the same precedence level

2.3.3.4 Type Conversions

Explicit type conversions shall be used wherever the result of an operation is unclear and wherever the operands are of different types. Do not assume the correctness of any automatic type conversion where operands are of different types.

Example:

```
short int   A;
float       B;
float       C;

Right       C = (float)A * B;
            C = (float)A * (float)2;      or C = (float)(A * 2)
```

Wrong C = A * B;

C + A * 2;

2.3.4 Memory

2.3.4.1 Dynamic Storage

All dynamically allocated storage shall be freed when it is no longer used. Do not leave memory deallocation to handled by process completion.

2.3.4.2 Static Storage

Static variables should be used only when the value of a variable must be maintained throughout execution of a program and the variable is not referenced by any other routine, function, or block of code.

2.3.5 Statements

Statements shall not extend beyond column 80.

2.3.5.1 If Statements

An "if" statement with multiple "else if" conditions should be used only where a "switch" statement would not be applicable. "If, else if, else" is an acceptable construct. If further cases are necessary, use a "switch" statement.

2.3.5.2 Compound If / Else Statements

Nested "if" and "else" statements shall always be enclosed in braces to prevent confusion, even if they are syntactically correct without them. The following example demonstrates the confusion that can occur.

Example: Right

```
if (condition1)
{
    if (condition2)
    {
                statement1;
    }
    else
    {
                statement2;
    }
}
else
{
    statement3;
}
```

Wrong

```
if (condition1)
    if (condition2)
                statement1;
    else
                statement2;
else
    statement3;
```

Notice that in the second case — the "Wrong" case — we rely on information of the first "else" to show that it belongs with the second "if". The first case is clear even without indentation.

2.3.5.3 Structured Statements and the Use of Braces

Braces shall appear on a line by themselves and shall be aligned vertically with the structure that they enclose. Do not indent braces from the current indentation level.

Example: Right

```
while (condition)
{
    statement1;
    statement2;
    .
    .
    .
    if (condition)
    {
        statement1;
        statement2;
    }
}
```

Wrong

```
while (condition){
    statement1;
    statement2;
    .
    .
    .
    if (condition)
        {
        statement1;
        statement2;
        }
    }
```

2.3.5.4 Case Statements

"Case / break" options that appear in a "switch" construct shall be vertically aligned and shall use braces to enclose all statements within each 'case / break" construct.

Use "fall / through" case options only when necessary. If used, be sure to comment them clearly, describing why they are being used (see example after next paragraph.).

Every switch shall have a default case, even if it contains no statements. The default case should be documented to explain the action taken. Generally, the default case will be used for the error condition or the general condition.

Example:

```
switch (expression)
{
    case 0:
    {
            statement1;
    }
    break;

    case 1:
    {
            statement1;
            statement2;
    }
```

```
case 2:     /*  case 1 falls into case 2                    */
            /*  for further processing                      */
        statement1;
        statement2;
}
break;
default:  /*  no action taken if                            */
break;    /*  expression is not recognized                  */
```

2.3.5.5 Loop Statements

Looping constructs — e.g., while, for — shall have a clear test condition for termination of the loop. Whenever possible, do not use abnormal conditions such as "exit", "return", or "break" statements to terminate a loop.

If the logical expression for loop termination contains several individual expressions, use parentheses and separate lines to enhance readability of the many individual conditions.

Example:

```
while (((condition1 && condition2) ||
    (condition3 && condition4)) &&
            (condition5 && condition6))
{
    statement1;
    statement2;
}
```

It is a good idea to not use long, complex test conditions because the net value of the expression may be difficult to determine for someone attempting to debug the code. If intermediary expressions can be computed and stored in test variables, the readability is enhanced and the net expression value can be readily examined.

Example:

```
loop_test1 = 1;
loop_test2 = 1;
loop_test3 = 0;

while ((loop_test1 || loop_test2) && loop_test3)
{
```

```
        .
        .
        .
        loop_test1 = condition1 && condition2;
        loop_test2 = condition3 && condition4;
        loop_test3 = condition5 && condition6;
}
```

2.3.5.6 Return Statements

There shall be only one normal exit point in a function. This point shall be the last line of the function and shall contain a "return" statement. Return values shall be used for all types of functions, except those that are declared as "void".

2.3.6 Functions

2.3.6.1 Modules

There shall be no more than one logical function per file. A file may contain more than one routine if all routines are logically related and only one routine is referenced externally. Therefore, there should be no more than one logical function per object module. There are some rare cases when multiple functions are required in a single object module. An example is the database access module, which is a collection of all database access functions. In this case, the individual source files may be concatenated prior to compilation.

2.3.6.2 Return Values

All functions that are not declared as "void" shall return a value for which they are declared.

2.3.6.3 Formal Parameters

Only scalar values shall be passed by value. All structures and arrays shall be passed by reference.

Remember, the name of an array is equivalent to its address. Therefore, it is rarely necessary to pass the address of the array name.

Example: char foo_array[100];

status = plfoo_foo_func(foo_array);

The exception occurs when the array is allocated by the function and returned as an output parameter.

Example: char *foo_array;

status = plfoo_foo_func(&foo_array);

Structures are always passed by reference. This is especially important when the structure contains a large amount of data, in which case passing it by value may overflow the call stack.

Example:		foo_struc_type foo_struct;
	Right	status = plfoo_foo_func(&foo_struct);
	Wrong	status = plfoo_foo_func(foo_struc);

In the case marked "Wrong" above, the contents of foo_struc will be copied onto the call stack. If foo_struc were a large data structure — for example, a 1000-element array — the call stack would overflow and kill the process.

Never return the address of a variable that is declared locally. You may not assume that the local memory is static.

Avoid using a pointer to store nonaddress data. Consider this example:

Example:

```
short       plfoo_foo_func(foo_char)
char        *foo_char;
{
    foo_char = "foo";
    return(0);
}
```

The variable "foo_char" will point to a copy of the string in the program's static area when the function returns. Although this causes no problems, it demonstrates fuzzy thinking and could easily be implemented as in this example:

Example:

```
short       plfoo_foo_func(foo_char)
char        *foo_char;
{
    strcpy(foo_char,"foo");
    return(0);
}
```

This time, the character string "foo" is copied into the memory location specified by foo_char rather than into foo_char itself. Alternately, memory could have been allocated for the string as in this example:

Example:

```
short       plfoo_foo_func(foo_char)
char        *foo_char;
{
    foo_char = malloc(4);
    strcpy(foo_char,"foo");
    return(0);
}
```

This is a viable solution, providing that the calling function remembers to free the memory when it is finished with it.

2.3.6.4 Formal Parameter Names

Overloading of parameter names is no longer allowed under VMS 5.1. Therefore, do not name parameters the same as their type names. This will not be a problem if all other coding standards are followed.

2.3.6.5 Function Length

Functions should contain no more than 50 executable statements (function calls, loops, assignment statements, and logical constructs). Of course, this is a rough limit, but it reflects the goal of limiting the scope and functionality of functions. There are obvious exceptions to this rule — dispatch routines and I/O-intensive routines often require a large number of statements to complete a single task. A dispatch routine, for instance, may contain a "switch" statement that contains hundreds of cases. This would be an allowable exception.

2.3.6.6 Function Calls

Each argument to a function in a function call should be placed on a separate line, if the clarity of the program is improved.

2.3.6.7 Function Arguments

Avoid using long expressions as arguments in function calls. It is generally more clear to assign the value of an expression to a variable and pass the value of the variable instead. This also makes it easier to examine the values of all arguments to a function in the debugger. For the same reason, avoid using functions as arguments to other functions. Again, if possible, assign the value of the function to a variable that can be passed by a value to the called function.

Example:

Right

```
status = plfoo_foo_func( foo_input1,
                         foo_input2,
                         &foo_output1,
                         &foo_output2 );
```

Wrong

```
status = plfoo_foo_func( ( (2.0/FOO_CONSTANT*foo_value) | (foo_value2) ),
                         plfoo_foo_func2(2.0, foo_value1),
                         &foo_struc + foo_offset,
                         &(foo_array[16 + foo_value1]) ) ;
```

Avoid passing global variables by reference. A function should not expect a parameter that references a particular global variable every time that it is called. Instead, eliminate the parameter and reference the global as an extern within the function.

2.3.6.8 Overloading Functions

Function shall not be named in such a way as to cause overloading of other user functions or standard library functions. An example would be a user function called "atoi" (ascii to integer) that would overload the standard library function, "atoi".

2.3.6.9 Exit Function

"Exit" functions shall be used only when an exception occurs and the program wishes to abort with a specific error condition set.

2.3.7 System Calls and Library Use

2.3.7.1 Sleep Statements and Polling

Whenever possible, avoid using "sleep" calls to create time delays. However, sleep cycles are a preference to busy waits and polling for event completion. Polling for an event in a tight loop is extremely expensive in terms of resources and will generally affect all other active processes. The use of ASTs (Asynchronous System Traps) is discouraged, because the same functionality is not necessarily available on all systems.

2.3.7.2 Use of System Services

System services should be used only by utility library functions. When system services are necessary, try to use UNIX or UNIX-like system calls. For instance, directory creation accomplished under UNIX with the "mkdir" function may also be done under VMS with the same "shell" command. Always attempt to use these UNIX-like system calls wherever they are available. See "Programming in VAX C" for a list of UNIX-like supported functions.

2.3.7.3 Use of "system"

Avoid using the "system" command to accomplish tasks for which there are alternatives. The "system" call will spawn a process to execute the command specified as its parameter(s). This is very expensive in terms of resources and time. In many cases, there is a standard library function available that will do exactly the same thing. Consult any ANSI C programming guide for a list of such functions.

2.3.7.4 Use of non-ANSI C Functions

Avoid using C library functions that are not supported as an ANSI standard. As an example, take the ANSI standard function for file deletion, "remove". VAX / VMS offers a function called "delete" that accomplishes the same thing. Likewise, APOLLO offers a function called "unlink". Neither of these functions is ANSI standard. The correct solution to this challenge was described earlier in section 2.3.1.5, "Defined Constants."

2.3.7.5 Shared User Library Functions

Functions that are included in shared user libraries should not contain callbacks. This means that any function contained within a library should not call any user functions (or reference any user globals) that are not contained within the same library.

2.3.8 Input-Output

2.3.8.1 Include Files

Include files shall not be nested beyond one level.

Contents of System ID / library include files shall be divided so that there are separate include files for public and private inclusion. Declarations that are used exclusively by the System ID / library (or by an object) should be separate from any declarations that must be included in any other System ID / library (or object).

The format for naming public include files is:

<system ID>_<object>__p<func | type | spec.h

or

<system ID>__p<func | type | spec.h

For private include files, use:

<system ID>_<object>__<func | type | spec.h

or

<system ID>_<func | type | spec.h

Example: plfoo_func.h
plfoo_type.h
plfoo_spec.h
plfoo_pfunc.h
plfoo_ptype.h
plfoo_spec.h

2.3.8.1.1 pfunc.h

The "func.h" file shall contain public function declarations that are "ifdef-ed" in such a way as to declare each function externally in every module except the one in which it is actually declared.

Example:
```
#ifndef RTN_plfoo_foo_function
extern short plfoo_foo_function();
#endif
```

By defining the constant RTN_plfoo_foo_function at the beginning of the plfoo_foo_function file, and undefining it at the end of the file, the external declaration of the function will not be made in the

function's module when it is compiled. All other files that include this "func.h" file will contain the external function definition when they are compiled.

2.3.8.1.2 func.h

The same can be said for the "pfunc.h" file, except that it shall contain all of the private function declarations in the same format as in the example above. The "pfunc.h" file should not be included by functions that do not have access to private functions — that is, the functions listed in the pfunc.h file are not exported.

2.3.8.1.3 ptype.h

All type definitions and defined constants that are public (exported) shall appear in this file.

Globals shall be defined and shall be externally referenced in this file. This can be done with the use of #ifdefs similar to the declaration of functions in "func.h". First, select a function that will always be included in an executable. If no such function exists, create one that will act as a dummy. Then, inside "type.h", arrange the declarations as they appear in this example:

Example:

```
#ifdef RTN_plfoo_foo_function
short plfoo_fooindicator = 0;
long plfoo_foo_var = 0;
#else
extern short plfoo_fooindicator;
extern long plfoo_foo_var;
#endif
```

This will cause the global variables to be defined globally once within the plfoo_foo_function's module — not defined within the plfoo_foo_function itself because the include statement appears before the function declaration — and to be externally referenced for all other modules that include the type.h file.

2.3.8.1.4 type.h

Private type definitions and defined constants should appear in this file.

2.3.8.1.5 pspec.h

This file is used to include the "func.h" and "type.h" files, as well as any other public files that may exist for the specified object / system ID / library.

2.3.8.1.6 spec.h

This file is used to include the "pfunc.h" and "ptype.h" files, the "spec.h" file, and any other private files that may exist for the specified object / system ID / library. It is necessary to include the "spec.h" file in order to pick up all definables related to the tool.

2.3.8.2 Common Files and Functions

2.3.8.2.1 C Include Files

The system will use the following C include files in the TCS$REF directory:

- DB.H — contains database definitions
- FORM.H — contains form definitions
- FORM_CNTRL.H — contains form definitions
- FUNC.H — contains public function prototypes
- GLOBAL.H — contains global variable definitions
- MENU.H — contains form definitions
- MSG.H — contains message definitions
- DMQ.H — contains all DECmessageQ definitions and types
- PUB_TYPE.H — contains CDD / Plus and general type definitions
- SCTR_TYPE.H — service center data types
- SADM_TYPE.H — system administration data types
- HOST_TYPE.H — host data types
- VIOL_TYPE.H — violation data types
- SUPV_TYPE.H — supervisor data types

Also, each C form processing module will use an include file for DECforms record descriptors in the format 'module name'_BUF.H.

2.3.8.2.2 Common C Functions

The system will use the following common C functions:

- TCS_ACCESS_CNTRL.C — Provides menu access control
- TCS_DAYS_DIFF.C — Calculates the difference, in days, between two VMS dates
- TCS_DB_FETCH_MEA.C — Retrieves many menu actioning rows
- TCS_DB_GET_EMA.C — Retrieves one employee access row
- TCS_DB_GET_LOGON.C — Retrieves all login-specific rows, terminal definitions, system parameters, and employee IDs

• TCS_DB_GET_SCR.C	Retrieves one screen titles row
• TCS_DB_GET_TAG.C	Retrieves one tag from the database
• TCS_ERR_LOGGER.C	Appends an error message to the end of the system error log file
• TCS_ERR_SIGNAL.C	Signals a function return error, and stops program execution
• TCS_FRM_AST_DATE_TIME.C	Displays the form dynamic date / time in AST mode
• TCS_FRM_DATE_TIME.C	Dispatched dynamic date / time routines
• TCS_FRM_CHK.C	Checks the status of each internal 'TCS$FRM...' form call. A failure will broadcast a message to the terminal through SYS$BRKTHRU
• TCS_FRM_CLOSE.C	Closes a DECforms form window
• TCS_FRM_HEAD_FOOT.C	Retrieves and displays all form header
• TCS_FRM_INIT.C	Opens (enables) a main form with the following: —Enables an AST and non-AST session —Starts the dynamic date / time AST —Retrieves a screen title row from the database —Gets menu action rows for a menu screen —Displays the facility name from the user global area —Retrieves and displays up to two footer messages from the message file
• TCS_FRM_IO.C	Performs DECforms form input / output
• TCS_FRM_INIT	TCS_FRM_OPEN must precede this
• TCS_FRM_MSG.C	Displays a DECforms message. —Uses the from send response to solicit user acknowledgment
• TCS_FRM_OPEN.C	Opens a DECforms form window.
• TCS_FRM_OUT.C	Performs DECforms form output only
• TCS_FRM_SET_TIMER.C	Sets a timer for dynamic update of forms
• TCS_FRM_TERM.C	Closes a DECforms main form
• TCS_GET_MSG.C	Retrieves message text based on the message code from MSG.H
• TCS_GET_PROCESS.C	Retrieves VMS user name, terminal ID, port, and node info

- TCS_PAD_STRING.C — Copies a source string to a target string padding with spaces to the specified target size
- TCS_RPT_DATE.C — Returns the current date / time for reports in the format 'DD-MMM-YYYY HH:MM:SS'.
- TCS_VERIFY_LOGON.C — Dispatches logon verifications

2.3.8.2.3 Common IFDL Functions

- TCS_DUAL_MENU.IFDL — Double-width menu template with a dynamic date and time
- TCS_MENU.IFDL — Single-width menu template with a dynamic date and time

2.3.8.2.4 Common SQLMOD Functions

- TCS_EMA.SQLMOD — Contains all employee access SQLMODS
- TCS_EMP.SQLMOD — Contains all employee ID access SQLMODS
- TCS_MEA.SQLMOD — Contains all menuing action SQLMODS
- TCS_SYS.SQLMOD — Contains all system parameter SQLMODS
- TCS_TER.SQLMOD — Contains all terminal definition SQLMODS
- TCS_SCR.SQLMOD — Contains all screen title SQLMODS

NOTE: Additional common functions will be identified and implemented as necessary.

The naming conventions for all other types of files should be agreed upon by the LMIMS Software Manager.

2.3.9 Coding Example

The following example illustrates some of the C coding standards and practices discussed in the preceding paragraphs. Note that this example illustrates a VAX environment.

```
    #define RTN_plfoo_foo_function
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*    Title:              plfoo_foo_function
*                        Checks input values for critical values.
*
*    Date:               19-May-1989
*
*    Author:             M.R. Doe
*
*    Summary:            This function tests three inputs for critical
*                        values.  A new value is returned through the in / out
*                        parameter while a combination of the inputs is
*                        returned through the output parameter.
*
*    Special Considerations:
*                        This program was not designed to work on the XYZ Project.
*
*        Inputs:             foo1
*                                test condition for positive status
*
*                            foo2
*                                test condition for negative status
*
*        Input/Output:       in_out_foo
*                                foo that is altered and returned.  Input values
*                                must be positive integers less than 500.
*
*        Outputs:            foo3
*                                composed of foo1 and foo2
*
*        Returns:            A short int indicating completion status
*
*        Calls:              plfoo_foo_adder
*
*        Called By:          plfoo_foo_requester          *NOTE 1
*
*        Globals:            plfoo_foo_indicator
*                            plfoo_foo_saver
*
*        History:            19-May-1989  M.R. Doe
*                                Initial release
*
```

```
*                         20-May-1989  M.R. Doe
*                              SDR #123456
*                         Return bad status if either input does
*                         not fall within critical bounds.
***************************************************/

#include        <stdio.h>
#include        "plexus_ui.h"
#include        "plfoo_pspec.h"
#include        "grfoo_spec.h"

short           plfoo_foo_function( short        foo1,
                                   short        foo2,
                                   short        *in_out_foo,
                                   long         *foo3 )

{
     extern    short int        plfoo_foo_indicator;
     short     local_status;         /*   a status of 1               */
     short     local_foo;                /*      indicates success              */

     if   ( (foo1 > 0) && (foo2 < 0) )
     {                                   /*      if both input flags fall       */
                                         /*      within critical boundaries     */
          if ( (*in_out_foo > 0) && (*in_out_foo < 500) )
          {                              /*      if in_out_foo falls within     */
               local_status = 1 ;        /*      boundaries...return max        */
               *in_out_foo = 500 ;       /*      value and good status          */
          }
          else                      /*   else return min value and   */
          {                              /*      bad status                     */
               local_status = 0;
               *in_out_foo = 0;
          }                              /*      output flag is sum of inputs   */
          *foo3 = plfoo_foo_adder(foo1, foo2);

          plfoo_foo_indicator = *in_out_foo;
                                         /*      save setting for               */
               }                         /*      future use else output is      */
     else                                /*      difference of inputs           */
     {
          local_foo = -foo2;
          *foo3 = plfoo_foo_adder( foo1,
                                   local_foo);
          local_status = 0;              /*      SDR#123456 - return bad        */
     }                                   /*      status                         */
     return( local_status );             /*      return status                  */
```

```
}
#undef RTN_plfoo_foo_function
```

NOTE 1: "Called by" shall be "TBD" when the auxiliary routine has not been written, and it shall be "Various Routines" for common modules that are called by many routines.

3. Procedures

3.1 Design Review Objectives

The objectives of the Design Review are to:

1. Ensure that the specified design is sound and is traceable to system requirements
2. Ensure that the design will meet the specified design and reliability requirements
3. Ensure that the risks are identified and that the attained status of the design justifies commitment of program resources to the next phase
4. Ensure that engineering documentation is complete, accurate, and technically adequate
5. Ensure that the design is producible, testable, and maintainable
6. Define action items necessary to resolve any design deficiencies found during review
7. Report on the productivity of the engineering processes used on the program via metrics
8. Ensure that design artifacts are available, presentable, and in detail sufficient to assess the design

3.2 Review Responsibilities

3.2.1 The Chief Technical Officer

The Chief Technical Officer may delegate whole categories of internal reviews — for example, code walkthroughs — to engineering staff, as appropriate. For external, internal, and Design Adequacy Assessments (DAA) reviews, the Chief Technical Officer will:

1. Identify the designs that shall be reviewed
2. Together with the Engineering Manager and the ERB, schedule the reviews, consistent with program schedule and technical requirements
3. Select the Chairperson for internal reviews and DAAs
4. Identify each of the supporting groups that will need to contribute to the review — and communicate this to the Chairperson
5. Identify and define any special problem areas, consultants, and specialists — and communicate this information to the Chairperson — so that a satisfactory Design Review Agenda, Data Package, and Checklists may be issued
6. Participate in the review and assessment of the topics being evaluated
7. Assign action items for internal reviews and DAAs
8. Ensure that each action item commitment consists of a description of the committed action, the name and organization of the person responsible for its completion, and the required completion date
9. Transmit Action Item updates to the Chairperson for inclusion into the Design Review Status Report
10. Be the final authority for the closure of Action Items
11. Ensure that the productivity of the engineering process is assessed via metrics
12. Ensure that the design specifics that have been presented prove that the design under review is adequate

3.2.2 The Program Management

For external reviews, the Program Manager will:

1. Together with the Chief Technical Officer, the Engineering Manager, the design review Chairperson, and the ERB, schedule the reviews consistent with program schedule and technical requirements; and determine the review cost budgets
2. Ensure that provisions are incorporated into the Proposal, PDP, Master Milestone Schedule, and Program Directives for the timely performance of the design reviews
3. Determine the customer level of participation in the Review

3.2.3 Review Coordinator

The Design Review Chairperson will:

1. At the direction of the Chief Technical Officer, select — or develop, as necessary — a checklist that defines the topics to be discussed at the design review
2. With the concurrence of the Chief Technical Officer, select members of the review board. Participation must be sufficient to thoroughly address all checklist topics and must address the Chief Technical Officer's requests for consultants and specialists.
3. Prepare and issue the agenda and data package
4. Schedule the meeting room and provide support equipment
5. Make arrangements for chairing and recording the Design Review Meeting; and prepare and distribute the Design Review Report
6. Establish an Action Item plan, including responsibility, need dates, reporting, and closure mechanisms
7. Track all action items and report status to the ERB for its monthly report on action item status
8. Archive all design review materials
9. Prepare appropriate design review summary reports

3.2.4 The Design Engineer

The design engineer will:

1. Provide the data package to the Chairperson at least 3 (nominal) working days prior to review date
2. Present the requirements, design approach, appropriate design artifacts/evidence, and development plans and solutions at the Design Review Meeting
3. Select and utilize other presenters, as necessary, to ensure a complete and understandable design disclosure
4. Present appropriate additional data and information not called for in the data package
5. Accept and coordinate responses for action items and recommendations in a timely fashion, nominally 5 working days
6. Request that ad hoc internal design reviews be scheduled by the Chief Technical Officer when necessary

3.2.5 Engineering Review Board (ERB)

The ERB Members will:

1. Ensure that an adequately rigorous program of internal reviews and DAAs is planned and implemented
2. Attend scheduled meetings that they have been invited to
3. Prepare for the Design Review by studying the Data Package and other pertinent material
4. Make recommendations for action; and provide consultation prior to, during, and after the Design Review meeting
5. Assess the adequacy of the design
6. Grade the design for overall thoroughness and for selected checklist topics

3.3 Review Process

3.3.1 Design Review Preparation

A flow diagram of the review preparation activities is shown in Figure 3-1.

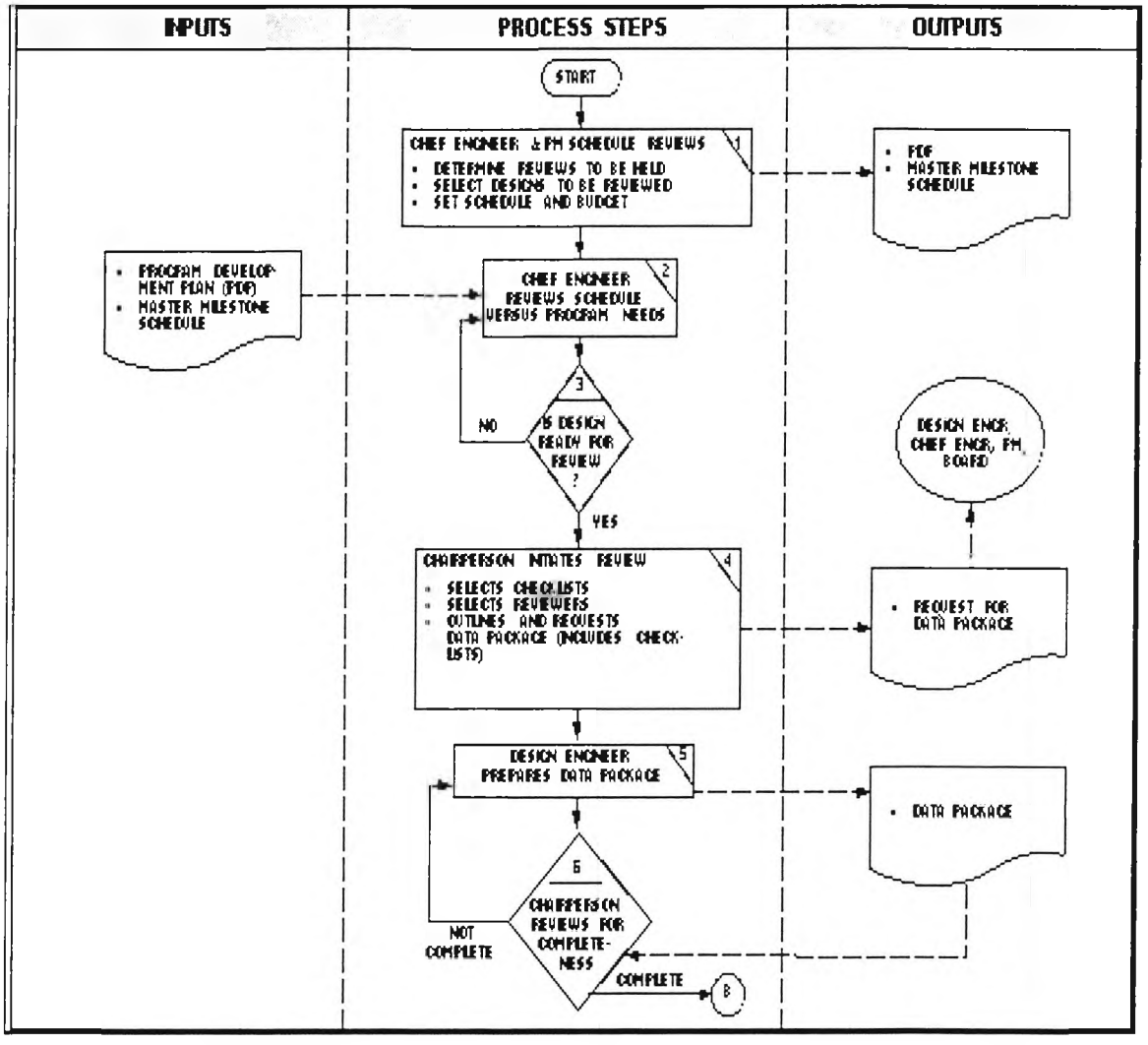

Figure 3-1 Review Preparation Activities

1. During the planning phase of each new development program, the Chief Technical Officer shall develop, define, and integrate a comprehensive plan for design reviews as required by the Program and based on this Procedure.

2. After program start, the master schedule of design reviews for the program shall be maintained by the Engineering Review Board (ERB). The ERB shall schedule and conduct the design reviews in accordance with the program plan and the requirements of this Procedure. The ERB may delegate this authority for reviews of limited scope and context, e.g., code walkthroughs. Requests to schedule additional design reviews shall be made to the ERB at least 2 weeks in advance of the review.

3. The ERB shall have the authority to schedule unplanned but necessary reviews, dispense with planned but unnecessary reviews, or rearrange review schedules, within constraints of the engineering budget.

4. The Chief Technical Officer shall designate a Chairperson for each design review and may recommend independent internal reviewers who are not actively involved with the design review.

5. The Design Review Chairperson shall determine the facilities, materials, and participants needed and shall issue notification of the review including a preliminary agenda.

6. The data package contents shall be specified by the Chairperson and shall be concurred with by the Chief Technical Officer. It shall include a checklist of review topics as appropriate to the equipment or system under review. Other topics may be added to satisfy contractual requirements if internal and contractual design reviews are combined.

7. The responsible design engineers shall prepare the design review data package containing supporting data for the topics specified. Five working days (nominal) prior to the review, the design engineers will provide to the chairperson a complete Design Review Data Package.

8. When the Chairperson determines that it meets requirements, the Package and the design review agenda shall be distributed — at least 3 working days prior to the review — to the ERB and other reviewers invited by the ERB.

9. The Package shall provide the level of detail that enables each participant to become familiar with the related design in advance of the review. This will ensure a constructive and comprehensive review and will maximize each participant's contributions.

10. The Chairperson shall defer the scheduled Design Review when, in his/her opinion after review of the Data Package, the existing information does not meet the objectives of the review. The design engineers and the Chief Technical Officer will be notified of such action. After deficiencies in the Data Packages have been corrected, the review will be rescheduled.

3.3.2 Design Review Content

A design review will be a coordinated review of an engineering product to ensure that it meets a previously established set of product requirements — including those associated with functionality; performance; cost; producibility; testability; operability; maintainability; and environment, health, and safety. Generally, design reviews are of two types:

3.3.2.1 External (Program) Design Reviews

These reviews are contractually required by the procuring agency or customer. They generally focus on the total product design, directed by the Program Manager, and they represent major program milestones. The reviewers are a broad audience of customers, co-contractors, and users. They generally do not permit a detailed review of design compliance with requirements. Program Design Reviews are not the subject of this Procedure.

3.3.2.2 Internal Design Reviews

These reviews focus on whether an engineering product satisfies a previously established set of requirements and on the productivity of the engineering process via metrics. The reviews generally focus on an element of the product design — compliant with this Procedure, directed by the Chief Technical Officer — with timing flexible enough to respond to shifts in design status. Engineering performance against cost and schedule commitments will also be a topic of these reviews. The reviewers are a small group of engineers who have knowledge of the item under review. These design reviews should complement and support the Program Design Reviews.

3.3.2.3 Design Adequacy Assessment (DAA)

The DAA process consists of these seven steps:

1. Validation of design process — Review the methods and procedures used during the design phase and ensure that they were followed. Deviations from procedures should be reviewed for risk to the program if procedures were eliminated and/or modified.
2. Validation of requirements — Review the requirements baseline to ensure that it is stable enough to proceed without major risk to the program.
3. Validation of design

- Review documentation of mission scenarios, design threads, modes of operation, allocated requirements, COTS integration, and component-to-component interface definitions.
- Review design documentation and results from analysis, traceability, models, simulation, and prototypes to evaluate the optimization, correlation, completeness, and risks associated with the allocated technical requirements and the design.
- Review design information for completeness, validation techniques, and adequacy (will it work?).

4. Validation of production environment — Review the production environment to ensure that the environment supports the required level of production efficiency.
5. Validation of test and integration plans — Review the test and integration plans to ensure that requirements will be verified and that a plan to integrate the components exists prior to system test.
6. Assessment of risk — Quantify the risk of moving to the next phase with open requirement and/or design issues.
7. Action plan — Develop an action plan to correct requirements, design, test, and integration deficiencies that must be mitigated to an acceptable level of risk to proceed.

The design review agenda will contain a formal announcement distributed by the Design Review Chairperson to the ERB members at least 5 working days prior to the review. The announcement identifies:

1. Purpose, definition, objective, and scope of the review
2. Schedule and place of the design review meeting
3. Participants — presenters, reviewers, subcontractors, others
4. Any meeting agenda details — items and activities to be evaluated

A design review data package will be a set of review materials that have been requested by the Chairperson and assembled by the design engineers. Ten working days prior to the scheduled Review, the design engineer will distribute the data package to the Chairperson, who addresses all topics and unique checklist items. The data package shall be reviewed by the Chairperson and, if sufficient, will be distributed to the Board along with the Design Review Agenda at least 5 working days prior to the review.

3.3.3 Conducting The Design Review

A flow diagram for conducting the design review is shown in Figure 3-2.

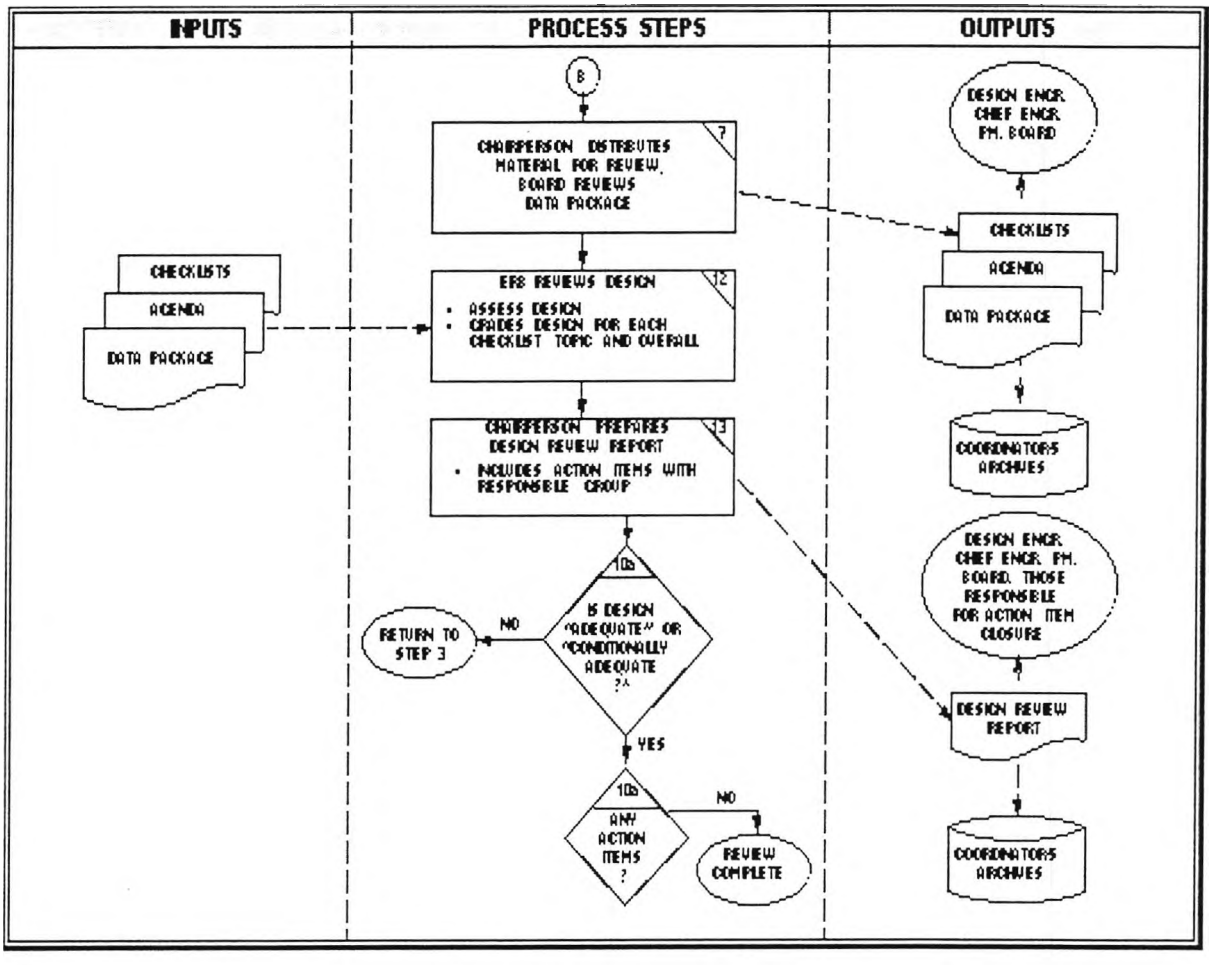

Figure 3-2 Conducting the Design Review

1. Members of the ERB and other reviewers shall thoroughly study all materials included in the data package prior to the design review meeting. Questions may be addressed directly to the Engineering group responsible for the preparation of the material prior to the date of presentation. Such questions and pre-review discussions are encouraged to enable clarifications, additions, or corrections to be made prior to the review presentation.
2. The Chairperson for each review shall have overall authority to conduct the review. With the support of the design reviewers, the Chairperson shall conduct the review so that the prescribed topics are adequately addressed and so that any required follow-up is identified by a formal Action Item that assigns responsibility and establishes a planned completion date. The design engineers will present and discuss the

material — consisting primarily of the Data Package — that is to be reviewed

When appropriate, a demonstration of the actual performance of the design will be conducted. The ERB members will review the design using the design review checklists to ensure that the design will perform its intended functions reliably and economically.

The Chairperson shall maintain a record of the significant concerns discussed and shall request written statements of concern and requests for action from participants, using the Action Item form. At the end of the design review meeting, the Chairperson shall read all the forms and shall determine appropriate disposition as formal Action Items.

Action Item due dates shall be assigned by the Chairperson to support program schedule requirements. The Program Manager or designee shall be present during disposition of action items to ensure that additional work requested is within the scope of the program. Action Items require formal written responses and are maintained by the Chairperson.

The Chairperson shall ensure that, prior to adjournment, all immediate concerns about the design adequacy raised during the review have been resolved or identified as action items.

3. The design under review shall be graded by the ERB for overall completeness and selected checklist topics as follows:

   Adequate — The design is adequate when the review team believes, based on the information presented, that the design meets its performance- and nonperformance-related requirements and, therefore, will perform as intended.

   Conditionally adequate — The design or evidence of design is presently inadequate, but the review team believes that it can be made adequate by completion of specified action items.

   Inadequate — The design or evidence of design is inadequate when the review team believes, from the information presented, that the design does not meet its requirements and, therefore, will require substantial redesign, or that an assessment cannot be made because of missing or incomplete design evidence.

The Chairperson shall prepare the Design Review Report within 10 working days of the review and shall issue it to the Program Manager, Chief Technical Officer, all the review participants, and other personnel involved in the design review, as appropriate. If issues are identified whose resolution exceeds the charter of the Board (i.e., issues affecting contract scope), the Chairperson is responsible for bringing these to the attention of the Chief Technical Officer and/or the Engineering Manager.

A copy of the report is to be retained in the ERB's archives and in the Design Engineer's design record file.

3.4 Design Reviews

3.4.1 Purpose

This procedure establishes the requirements for planning and conducting design reviews for systems and software. This section addresses internal reviews — an integral part of the design process held without the customers present. The primary goals of the internal reviews are to discover, understand, and mitigate technical issues and risks and/or to communicate the results of thought and analysis to other personnel to achieve a common understanding of the overall system under review.

3.4.2 Scope

This procedure applies to all programs conducted by Engineering. Design reviews shall be planned for, and conducted on, new or modified engineering designs of systems, subsystems, components, etc., to ensure that each design meets the requirements of the applicable program(s) and that each adheres to the established design standards.

3.4.2.1 Design Review Checklist

Design Review Checklist		
1.	Are the design objectives clearly stated?	
2.	Has attention been paid to the usability of the user interface? Have the users been involved in that decision?	
3.	Are there models for all critical parts of the system?	
4.	Have interfaces to other portions of the system been considered, and has the impact of the current feature on other parts of the system been addressed in the design?	
5.	Is there a high-level functional model of the system?	
6.	Is there an operational description?	
7.	Is there an explanation of the test procedure, test cases, test results, and test analysis to ensure that the model is correct.?	
8.	Is there a discussion of the alternatives and why they were rejected?	
9.	Are the major implementation alternatives and their evaluations presented in the design documents? Cost Time Resources for the alternates?	
10.	Is there an evaluation of the model that ensures that the requirements will be satisfied.? Consider:	

Design Review Checklist		
	Performance Storage requirements Quality of results Ease of use Maintainability Adaptability Generality Technical excellence Simplicity Flexibility Readability Portability Modularity	
11.	Is there any hardware dependency?	
12.	Afterthoughts—Examine the last three things crammed into the design.	
13.	What has been crammed in?	
14.	Could you tell that they are crammed in?	
15.	What was not considered when this change was made?	
16.	What would happen if you left this change out of the design?	
17.	Are there any items in the design for which the justification is "we have always done it this way"?	
18.	What happens for the error cases? Have any exceptions to the flow been indicated? If not, are there any that were missed?	
19.	Have we forgotten the user?	

3.4.2.2 Design Review Report

Design Review Summary Report

Review _____ *Starting Time* _____

Date _____ *Ending Time* _____

Feature Id _____

Produced By _____

Brief Description

_____

_____

_____

_____

Materials Used In Review	*Description*
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Participants	*Signature*
1. Leader _____	_____
2. Recorder _____	_____
3. _____	_____
4. _____	_____
5. _____	_____
6. _____	_____
7. _____	_____

Accepted

_____ *As Is*

_____ *Minor Revisions*

Not Accepted

_____ *Major Revisions*

_____ *Rebuild*

_____ *Review Not Completed*

Supplementary Materials

_____ *Issues List*

_____ *Related Issues List*

_____ *Other*

Description

_____

_____

_____

Glossary of Terms

ANSI	American National Standards Institute
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
AST	Asynchronous System Trap
CM	Configuration Management
DB	Database
DDE	Dynamic Data Exchange
DEC	Digital Equipment Corporation
e.g.	exempli gratia ("for example")
GUI	Graphical User Interface

i.e.	id est ("that is" — used to restate, define, or clarify)
I / O	Input / Output
LMIMS	Lockheed Martin Information Management Systems
MDI	Multiple Document Interface
SLE	Single Line Edit
SQL	Structured Query Language
TBD	To Be Determined
VBX	Visual Basic Extension
VMS	Virtual Memory System