AFHRL-TP-89-84

AIR FORCE HUMAN RESOURCES LABORATORY

AD-A221 806

ADVANCED ON-THE-JOB TRAINING SYSTEM: COMPUTER PROGRAMMING STANDARDS DOCUMENT

Douglas Aircraft Company
A Division of McDonnell Douglas Corporation
2450 South Peoria
Aurora, Colorado 80014

TRAINING SYSTEMS DIVISION
Brooks Air Force Base, Texas 78235-5601

April 1990

Interim Technical Paper for Period August 1985 - December 1989

AIR FORCE SYSTEMS COMMAND
BROOKS AIR FORCE BASE, TEXAS 78235-5601

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has reviewed this paper, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This paper has been reviewed and is approved for publication.

HENDRICK W. RUCK, Technical Advisor
Training Systems Division

RODGER D. BALLENTINE, Colonel, USAF
Chief, Training Systems Division

This technical paper is printed as received and has not been edited by the AFHRL Technical Editing staff. The opinions expressed herein represent those of the author and not necessarily those of the United States Air Force.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1990	Interim - August 1985 to December 1989

4. TITLE AND SUBTITLE

Advanced On-the-job Training System:
Computer Programming Standards Document

5. FUNDING NUMBERS

C - F33615-84-C-0059
PE - 63227F
PR - 2557
TA - 00
WU - 02

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Douglas Aircraft Company
A Division of McDonnell Douglas Corporation
2450 South Peoria
Aurora, Colorado 80014

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Training Systems Division
Air Force Human Resources Laboratory
Brooks Air Force Base, Texas 78235-5601

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFHRL-TP-89-84

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Advanced On-the-job Training System Computer Programming Standards Document was prepared to be used mainly by members of the AOTS Development Contractor, McDonnell Douglas (MDAC), Software Development Team as a training and reference document. The Air Force monitoring and review personnel used the document to gain insight into the operation of the MDAC AOTS Software Development Team. Some of the material contained in the document has been adapted from the McDonnell Douglas Aircraft Company Software Engineering Practices Manual. The document presents a set of standards that governed the production of code for the AOTS. These standards apply to the coding and detailed documentation of the AOTS Systems Design.

14. SUBJECT TERMS

advanced on-the-job training system
software design training document

15. NUMBER OF PAGES

44

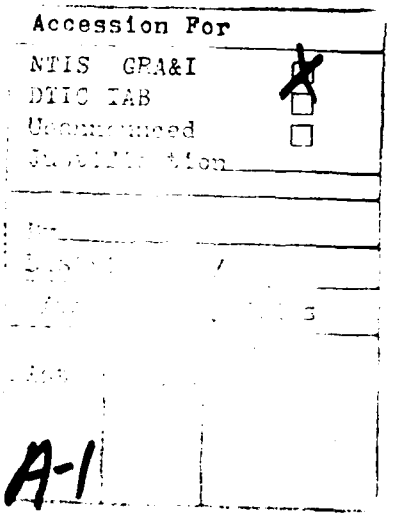
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

AFHRL Technical Paper 89-84 April 1990

ADVANCED ON-THE-JOB TRAINING SYSTEM: COMPUTER PROGRAMMING STANDARDS DOCUMENT

Douglas Aircraft Company
A Division of McDonnell Douglas Corporation
2450 South Peoria
Aurora, Colorado 80014



TRAINING SYSTEMS DIVISION
Brooks Air Force Base, Texas 78235-5601

Reviewed and submitted for publication by

Jack L. Blackhurst, Major, USAF
Chief, Advanced On-the-job Training System Program

This publication is primarily a working paper. It is published solely to document work performed.

SUMMARY

The Advanced On-the-job Training System (AOTS) was an Air Staff directed, AFHRL developed, prototype system which designed, developed, and tested a proof-of-concept prototype AOTS within the operational environment of selected work centers at Bergstrom AFB, Texas and Ellington ANGB, Texas from August 1985 through 31 July 1989. The AOTS Computer Programming Standards Document was prepared to be used mainly by members of the AOTS development contractor, McDonnell Douglas (MDAC), Software Development Team, as a training and reference document. The Air Force monitoring and review personnel used the document to gain insight into the operation of the MDAC AOTS Software Development Team. Some of the material contained in the document has been adapted from the McDonnell Douglas Aircraft Company-STL Software Engineering Practices Manual. The document presents a set of standards that governed the production of code for the AOTS. These standards apply to the coding and detailed documentation of AOTS system designs, and this is a supplement to the AOTS Software Development Plan which is published as a separate AFHRL Technical Paper. The Ada language documentation (MIL-STD-1815) and a preliminary McDonnell Douglas document (ADA Design Lanaguage Manual) were used as references in the development of these standards.

PREFACE

This paper was prepared for the AOTS Software Development Team, in accordance with Contract Number F33615-84-C-0059, CDRL 20. The AFHRL Work Unit number for the project is 2557-00-02. The primary office of responsibility for management of the work unit is the Air Force Human Resources Laboratory, Training Systems Division, and the Air Force AOTS manager is Major Jack Blackhurst. The procedures described in this document will be updated as necessary.

This paper is organized according to the provisions of Data Item Description (DID) DI-M-30409 which also references two other DIDS, DI-E-30111 and DI-E-30112. Ada, a higher-order structured programming language, will be used for development of the AOTS software. As a result, several of the sections required by this first DID are not applicable. This will be noted in the appropriate sections. Several sections have been retitled to conform with modern techniques. The intent, however, is the same as specified in the DID. Section 2 is specified in the DID as Flowcharts. Section 2 in this document is titled Program Design Language. Section 4 is specified in the DID as Subroutines. Section 4 in this paper is titled Subprograms.

Some of the material contained in this document has been adapted from the MDAC-STL Software Engineering Practices Manual for use by the AOTS Software Development Team.

Table of Contents

Table of Contents

1. INTRODUCTION

1.1 PURPOSE OF DOCUMENT

This document presents a set of standards that govern the production of code for the Advanced On-the-job Training System (AOTS). These standards apply to the coding and detailed documentation of AOTS system designs and thus is a supplement to the Software Development Plan previously submitted.

The Ada language documentation (MIL-STD-1815) and a preliminary McDonnell Douglas document (Ada Design Language Manual) were used as references in the development of these standards. The appropriate Appendix F providing specifics of the compiler implementations for the Alsys Ada compiler on the IBM PC/AT and compatibles (Zenith 248) and the Digital Equipment Corporation Ada compiler for the VAX 8600 are supplements of MIL-STD-1815.

1.1.1 Coding standards

Coding standards encourage simplicity of expression, consistent use of style, and uniformity in the use of the Ada language. By following these standards, programmers will be producing a superior product that will be easy to read, understand, and maintain.

1.1.2 Documentation standards

Documentation standards establish guidelines for inclusion of critical and non-critical information in source files. Critical information is required to understand the purpose and operation of a unit of code; non-critical information enhances that understanding. Documentation standards also encourage simplicity of expression in textual material.

1.2 INTENDED AUDIENCE

1.2.1 Primary audience

This document is intended is to be used mainly by members of the MDAC AOTS software development team as a training and reference document. In addition, MDAC technical non-development personnel will need to be aware of the content in order to ensure compliance with the provisions herein.

1.2.2 Secondary audience

The USAF monitoring and review personnel will use this document to gain insight into the operation of the MDAC AOTS development team.

1.3 EXPLANATION OF TERMS

Two classes of instructions are used in this manual: rules and guidelines. These are defined below followed by a glossary of other terms used in this manual.

1.3.1 Rules

Rules are to be followed under all circumstances. Only a software team leader can grant exceptions. All such exceptions shall be accompanied by an explanatory comment in the source code. An explanation of any such exception will be required at reviews.

1.3.2 Guidelines

Unlike rules, guidelines are practices that are expected to be followed. Exceptions to guidelines can be made at the programmer's own discretion. An explanation of any such exception may be required at any review.

1.3.3 Glossary

Ada	A higher-order programming language developed for the Department of Defense
ADL	Ada Design Language
AOTS	Advanced On-th-job Training System
CDRL	Contract Data Requirements List
DID	Data Item Description
HIPO	Hierarchy plus Input-Processing-Output
MDAC	McDonnell Douglas Astronautics Company
PDL	Program Design Language
UDF	Unit Development Folder(also referred to as Software Design Notebook(SDN)

1.4 CHANGE PROCEDURES

The procedures in this document are subject to change as experience with their use is gained. The following procedure should be used when any changes are made.

1. Proposed changes to rules and guidelines are to be made in writing to a team leader.

2. The team leader will forward them to the Software Manager.

3. Periodically, the Software Manager will call a meeting of team leaders to evaluate all proposed changes. At the meeting, each change will be discussed and either approved or rejected for inclusion in this document.

4. Following the meeting, the Software Manager [illegible]

the submitter of the decision made at the meeting.

5. If the proposal is accepted, an update to this document will be issued containing the changed rules and guidelines following the procedures for documentation control specified in the Configuration Management Plan.

2. PROGRAM DESIGN LANGUAGE

The DID calls for this section to have information regarding the use of flowcharts and, as stated in the preface to this document, has been changed to Program Design Language.

The use of a Program Design Language(PDL) is excellent in providing the details of the design but is lacking in giving an overall view of the software structure. A graphical hierarchy will be generated showing the Ada package dependencies to provide, at a glance, where a particular subset of PDL fits into the overall design. This graphical hierarchy will be portrayed by continuing the use of HIPO (Hierarchial Input-Processing-Output) Visual Table of Contents discussed in the Software Development Plan and used in the top level design performed during Phase I.

The remainder of this section contains information concerning the use of Ada as a Program Design Language (PDL).

2.1 REASONS FOR USE

The Ada language, to be used for implementation of AOTS, lends itself readily as a PDL. Several aspects of Ada make it useful as a PDL. First, Ada provides excellent mechanisms which support information-hiding and object-oriented design. Second, Ada encourages consistency of design because the design itself may be compiled by the Ada compiler. This also allows automated checking of various aspects of the design, such as package interfaces and procedure calls. Third, use of Ada allows a natural expansion of the design into fully-implemented Ada code, thus reducing the likelihood of errors in translating from the design to the implementation.

Since the Ada language is used as the PDL, the terminology Ada Design Language (ADL) is appropriate.

2.2 METHOD OF USE

The MDAC-STL Ada Design Language Reference Manual shall be a partial guide for using ADL. This manual describes how to design software using ADL and suggests a subset of Ada to be used for design. ADL will be used to specify the complete package specification and the types, objects, and subprograms in the body of the package.

The specification of a package is expressed in ADL by using actual Ada type and object declarations. The subprogram interfaces are also expressed using actual Ada code to define the subprograms and their calling sequence.

The body portion of a package is expressed in similar fashion. Again the internal type, object, and subprogram declarations are given using Ada. Subprogram design is expressed using Ada to

define the subprogram with comments to show the processing performed.

This process allows a compilable detailed design. The compilation process insures that valid concepts are used in the design and verifies the interfaces are properly constructed and used.

An example of the basic template for the detailed design using ADL is shown in Appendix A.

3. PROGRAM STRUCTURE

This section describes the method of design structuring to be used by the program designers. Programs designed using Ada consist of subprograms, packages, and tasks. The following describe these constructs and how they will be used in the design process.

3.1 SUBPROGRAMS

Subprograms, comprised of procedures and functions, are the basic units for Ada programs. They are similar to their namesakes in other programming languages. Each subprogram should be designed to perform one function. Rules and guidelines for subprograms are in Section 4.

3.2 PACKAGES

The package is one of the most powerful conceptual tools available to the Ada designer and programmer. A package consists of two major parts: the package specification and the package body. The specification portion contains declarations of the data types and procedures that are accessible outside of the package. The body portion contains the actual code.

The separation of the package specification from the package body allows a great deal of information hiding to take place. Information hiding is a technique used by designers to keep the design modular and simple. This separation also makes it possible to make changes to the actual implementation of the package at a later date, without affecting any of the other parts of the system.

A package should contain a group of logically related data types and procedures. Not all the data types and procedures need to be included in the package specification. Procedures and data that are local to the package are declared inside the package body. This prevents accidental misuse of data and procedures that are internal to a package.

The package tends to be the lowest level of a program structure in that all references to the elements of a package are through the package specification. Configuration control is thus performed at the package level as the specification must be treated as a single compilable entity and defines the external interface to the package. Because of this, the Ada package can be thought of as synonymous with a Computer Program Component (CPC). Due to the recompilation rules of Ada the package specification and body will be identified as separate configuration items to minimize the recompilation needed.

3.3 TASKS

An Ada task is an independent process that can operate concurrently with other processes. Tasks can also communicate and share resources with other tasks.

Like a package, a task consists of a specification and a body. The specification describes the purpose of the task and contains the declarations of entries. Task entries are declared in a similar fashion to procedures, and make the functions of the task accessable to other tasks. The body contains the subprograms that do the work of the task.

3.4 Structure Design

Actual Ada program identification is via procedure subprograms that can be linked to allow program execution. The CPCIs will be decomposed into logical functions to be built into these programs.

Further decomposition will identify objects of these logical functions. These objects will be defined using Ada packages. The implementation of an object will be hidden to the extent practical using Ada private types or by type definitions visible only in the package body.

The actions that must be performed on the objects will be identified next. These actions will be visible external to the package for those actions requiring external initiation. The actions internal to the object will have restricted access by defining them only in the package body. The minimum practical visibility of the actions will be used to reduce the impact of any future changes in the implementation of an action. The actions for an object will be implemented using subprograms and tasks.

4. SUBPROGRAMS

In Ada, subprograms may take the form of functions, procedures, and task entries. This section presents rules and guidelines for the style of subprograms in the program composition. See Section 1.4 for an explanation of rule and guideline.

Note that the DID requires that several other items be included in this section. These items are handled by the compiler and operating system or defined in the high-order language definition. The items in this category are the linkage requirements and conventions, standard form of a calling sequence, data or address transfers, use of registers, restrictions on register or memory usage, restrictions on entry points, placing in memory of constants, temporary storage, and the storing of results of computations. Naming conventions are discussed as part of Appendix B.

4.1 CONSTRUCTION OF SUBPROGRAMS

This subsection contains rules and guidelines concerning the creation and organization of subprograms.

4.1.1 A subprogram should not exceed 100 lines of executable code. (Guideline)

This guideline serves to restrict the subprogram to a reasonable size for purposes of code review, inspection, testing, and maintenance. In addition, limiting the size of a subprogram helps to ensure compliance with the concept of allocating only one system function to a subprogram.

4.1.2 The name of a task, task entry, package, procedure, or function shall be included in the 'END' statement. (Rule)

This rule improves the readability and comprehensibility of the source code.

4.1.3 Common software units should be used rather than repeating a segment of code (i.e., use of generics). (Guideline)

This guideline reduces the likelihood of duplication of existing code. In addition, use of generics improves system maintainability by making more extensive use of previously tested software.

4.1.4 Functions shall contain only one RETURN statement. It shall be located at the end of the function or immediately before an exception handler located at the end of the function. (Rule)

The presence of multiple RETURN statements in a function raises the possibility of creating unreachable code and also violates the structured programming principle of "single entry, single exit" subprograms. Therefore, multiple RETURN statements will not be used.

4.1.5 Procedures shall not contain a RETURN statement. (Rule)

Transfer of control from the procedure back to the calling entity will occur at either a RETURN statement or an END statement. Since the END statement is required, use of a RETURN statement would create multiple exits from the procedure. This is to be avoided.

4.2 SUBPROGRAM INTERFACES

This section contains rules and guidelines concerning the type of interfaces to be used between subprograms, and restrictions on the use of parameter lists.

4.2.1 Data interfaces between subprograms shall be via parameters rather than global variables. (Rule)

This rule minimizes the possibility of unexpected or inadvertent changes to widely used data items. Wherever extensive data transfer is required, use of interface records is recommended.

4.2.2 Default passing of arguments to a referenced procedure or function should be avoided. (Guideline)

This guideline minimizes possible confusion in identifying actual arguments passed during a procedure or function call.

4.2.3 Use of subprograms with varying numbers of parameters should be avoided. (Guideline)

Subprograms are most easily used and understood when they have a fixed number of parameters. However, there are cases where it would be desirable to have the same subprogram perform slightly differently based on the presence or absence of a particular parameter.

5. INTERRUPTS

The Ada language is designed to handle interrupts, so that it may be used in embedded software systems. However, AOTS is not an embedded system, nor will it need to handle interrupts from hardware as part of the application software. Hardware interrupts will be handled by the operating system.

6. TIMING CONSIDERATIONS

All timing considerations as given in the DID are handled by the Ada language or by the operating system thus requiring no further elaboration.

7. MEMORY/REGISTER USAGE

The usage of registers and memory mapping are controlled by the Ada compilers and the operating system thus requiring no further elaboration here.

8. CHECKOUT AND TEST FEATURES

This section describes checkout and test features that must be designed and coded into the program. The features used for the AOTS software development effort consist of an overall approach using operating system capabilities for testing and debugging and special case unit testing involving special debugging code imbedded into the unit.

8.1 OVERALL APPROACH

Given the presence of on-line debugging software as part of the Dec Ada programming environment, there are no requirements for designing software test instrumentation code into the initial version of any unit. Instead, test cases will be manually selected to check the unit's ability to meet the specified requirements. All test results will be recorded and stored in Unit Development Folders (UDFs), described in paragraph 10.1. Should these test cases detect any problems, the on-line debugger provides features to aid in isolating those problems.

8.2 SPECIAL CASE UNIT TESTING

Depending upon software complexity measures and the judgement of the responsible team leader, special case unit testing may be specified for particular units. Special case testing would involve strategic placement of debug statements which aid the user in analyzing data and execution location. Debug statements consist of control to optionally execute the debug statements, debug data variables, and terminal input/output statements. Results from special case unit testing will be recorded in the UDFs.

9. CODING TECHNIQUES AND RESTRICTIONS

This section describes the particular coding techniques used in the production of code for AOTS. These techniques are expressed as rules and guidelines. See Section 1.4 for an explanation of the terms rule and guideline.

9.1 SOURCE FILES

9.1.1 Source File Formatting

9.1.1.1 Each logical block shall be indented three (3) columns from the enclosing block. (Rule)

This rule ensures that the structure of the code will be reflected by its physical layout on the screen or page.

Example:

```
begin
   m := 10;
   n := 5;

   for k in m+2 .. m+n loop
      m := k + 1;
      n := m + 2;
      Target_Array(k) := m + n;
   end loop;
end;
```

9.1.1.2 Continuation lines shall be indented at least three (3) columns from the starting line. (Rule)

This rule ensures that continuation lines will not be mistaken as new statements.

Example:

```
Partial_Denominator := 1.0 -
                       (Current_Altitude / Earth_Radius) -
                           Flat * Temp_Scalar;
```

9.1.1.3 Separate lines should be used for declarations of objects and types. (Guideline)

Using separate lines for declarations allows easier location and identification within the source code.

9.1.1.4 Blank lines should be used to provide vertical spacing for clarity within the source code. (Guideline)

Use of blank lines increases the readability of the source file's contents. Blank lines should be used to separate logically related sections of source code.

9.1.2 Statement Grouping

9.1.2.1 Declarations should be grouped in a consistent manner. (Guideline)

Grouping of declarations makes it easier to locate an individual declaration. One method is to group declarations with similar characteristics (constants, simple types, record types, etc.). A second method is to group declarations based on some common application.

Examples:

Grouping by declaration characteristics:

```
type Cartesian_Coordinates is (X, Y, Z);
type North_Pointing_Coordinates is (East, North, Up);

type Cartesian_Vectors is
      array (Cartesian_Coordinates)      of INTEGER;
type North_Pointing_Vectors is
      array (North_Pointing_Coordinates) of INTEGER;

Cart_Index   : Cartesian_Coordinates;
Cart_Vector  : Cartesian_Vectors;
NP_Index     : North_Pointing_Coordinates;
NP_Vector    : North_Pointing_Vectors;
```

Grouping by declaration usage:

```
type Cartesian_Coordinates is (X, Y, Z);
type Cartesian_Vectors is
      array (Cartesian_Coordinates)      of INTEGER;

Cart_Index   : Cartesian_Coordinates;
Cart_Vector  : Cartesian_Vectors;

type North_Pointing_Coordinates is (East, North, Up);
type North_Pointing_Vectors is
      array (North_Pointing_Coordinates) of INTEGER;

NP_Index     : North_Pointing_Coordinates;
NP_Vector    : North_Pointing_Vectors;
```

9.1.2.2 Statements with a related purpose should be grouped together wherever reasonably possible. (Guideline)

Grouping by statement purpose increases readability of the source code by discouraging random placement of unrelated source statements.

Examples:

Good:

```
Student_Record.Name := Input_Record.Name;
Student_Record.SSN  := Input_Record.SSN;
Student_Record.GPA  := Input_Record.GPA;

Display('Input record read');
```

Bad:

```
Student_Record.Name := Input_Record.Name;
Student_Record.SSN  := Input_Record.SSN;

Display('Input record read');

Student_Record.GPA  := Input_Record.GPA;
```

9.1.2.3 Vertical alignment of ':' in a record declaration should be employed. (Guideline)

Where this can be accomplished without expending an inordinate amount of time or effort, the vertical alignment provides improved readability in the source file.

9.2 COMMENTS

Comments are the programmer's means for providing documentation of the design and actions of the software in a place where it can never get lost or separated from the code to which it applies. For this reason, the comments in the code are perhaps the most important type of documentation.

Comments take the form of block and side-bar comments. Block comments are used to describe the processing of a block of code. Side-bar comments are used to describe the action of a single statement.

9.2.1 Use of comments

All comments should add value to the software unit. (Guideline)

Comments should provide meaningful summary commentary to the processing; they should not merely repeat what is already obvious in the code. They should be written in plain English and should use as little "jargon" as possible. In general, comments should be one level of abstraction higher than the source code implementation. For example, "Move to next column" would be preferable to "add 1 to icol".

9.2.2 Header Blocks

Every software component that is maintained as a separate entity shall have a header block at the beginning of the software component. (Rule)

For detailed information on header block contents, refer to the appropriate header block convention in Appendix A.

9.2.3 Block Comments

9.2.3.1 All block comments shall begin with '--' in columns 1 and 2. (Rule)

The presence of the '--' token at the left margin provides a simple, clear indication of a comment.

9.2.3.2 Block comments should be clearly distinguished from the rest of the code. (Guideline)

Improves readability of the source file. Examples of clearly distinguishing block comments from source code might include a line of asterisks, dashes, etc.

9.2.4 Side-bar Comments

9.2.4.1 Side-bar comments should begin at least 5 spaces after the end of the statement to which they pertain. (Guideline)

Use of spaces increases the readability of the commenting by providing a clearer distinction between the source statement and the associated comment. If insufficient space exists on the line, the comment should be placed on the line after the source statement.

Examples:

Comment on same line:

```
Total := Total + Hours(i);      -- sum of hours worked
```

Comment on next line:

```
Number_Registered := Number_Passed + Number_Failed;
                  -- compute number of students registered
```

9.2.4.2 Side-bar comments which need to be continued on succeeding lines shall be indented to the level of the first part of the comment and shall not be continued on lines containing other statements. (Rule)

The indentation makes the comments more readable. Keeping the continuation lines separate from other statements helps avoid confusion as to the source of the comment.

Example:

```
A := A + B;     --comment for
                --a := a + b
B := B + C;
C := C + D;
```

9.3 CODE CONSTRUCTION

This subsection contains rules and guidelines pertaining to the construction of the source code.

9.3.1 Language Extensions

Use of vendor-specific language extensions shall be avoided. If language extensions are used, however, they shall be clearly and completely documented. (Rule)

Documentation of this type improves the maintainability of the source code by identifying the exact areas which will require modification if the source code is to be transported to another machine.

The main area of language extensions in Ada is in the PRAGMA statement. Since the implementation of the PRAGMA set varies from one compiler to another, using any vendor-specific PRAGMAs makes the source code non-portable.

9.3.2 Control Structures

9.3.2.1 The permissible control structures are as follows:

1) sequence
2) selection
   a) If-Then
   b) If-Then-Else
   c) If-Then-ElseIf
   d) Case
   e) Select
3) iteration
   a) Do-While
   b) Do-Until
   c) Loop-End Loop
   d) For-End For

e) Exits from inside Loop-End Loop
and For-End For

(Rule)

These control structures are the ones widely accepted for use in structured programming.

9.3.2.2 Use of GoTo statements should be avoided. (Guideline)

Absence of GoTo statements reduces the likelihood of disorganized transfer of flow of control. GoTo statements can be used only when the circumstances allow no other option.

9.3.2.3 Multi-loop exits shall not be employed. (Rule)

This rule minimizes the potential for unnecessarily complex flow of control.

Examples:

In the following example, loops Alpha and Beta both have single-level exits:

```
Alpha:
   loop
      exit Alpha when ...
      ...
      Beta:
         loop
            ...
            exit Beta when ...
         end loop Beta;
      ...
      ...
   end loop Alpha;
```

Multi-loop exits increase the potential for disorganized transfer of the flow of control. This loop contains a multiple-loop exit:

```
Alpha:
   loop
      ...
      Beta:
         loop
            ...
            exit Alpha when ...
         end loop Beta;
      ...
   end loop Alpha;
```

In this case, the "exit Alpha" transfers the flow of control out of Beta implicitly and out of Alpha

explicitly. The implicit exit from Beta is a "blind spot" which could raise difficulties later during maintenance.

9.3.2.4 Coding techniques shall reflect simple, concise, and clearly organized flow-of-control logic and expression processing. (Rule)

Source code that contains "trick" functions or processing, or is unnecessarily complex, is more difficult to understand and maintain than simpler code. In addition, such code is more subject to software defects. Accordingly, such code is to be avoided.

9.3.3 WITH and USE clauses

The WITH clause should be used without the USE clause, whenever reasonable. (Guideline)

This forces an identification of the package from which the reference element came, which provides more insight into the code during later maintenance. Common exceptions will be use of widely used utility packages and specifiying a USE clause of very limited scope to improve the overall readability of the code.

9.3.4 Enumeration Types

9.3.4.1 An enumeration type should be used for named lists of items. (Guideline)

Use of enumeration types enhances the simplicity, readability, and maintainability of the source code.

9.3.4.2 A two-value enumeration type should be used instead of boolean values if an appropriate word pair exists. (Guideline)

Two-value enumeration types improve clarity of the source code by unambiguously identifying the desired values. Maintainability is also improved. For example, in the event that the two-value set should have to be expanded to three values, the modifications made to the source code are minimal.

Examples:

```
type Statuses is (invalid, valid);
type Parities is (odd, even);
type Power_Levels is (low, high);
```

9.3.5 Error Detection

Errors resulting from incorrect data or operator action shall not cause program aborts (including aborts resulting in execution termination). (Rule)

The AOTS software must be robust enough to defend itself against incorrect data and/or naive software users. The Ada exception mechanism allows handling of all but extreme system errors providing a mechanism for orderly program termination.

9.4 STATEMENT FORMATTING

This section contains rules and guidelines for formatting of source statements.

9.4.1 Only one (1) source statement shall be coded per line. (Rule)

Coding only one statement per line enhances code readability and maintainability.

9.4.2 The following naming conventions should be observed in coding source statements:

1) Reserved Words: all lower case letters
2) Predefined Identifiers: all upper case letters
3) Identifiers, Variables, and Constants: a logical mixture of upper and lower case

(Guideline)

The use of naming conventions provides greater readability and maintainability of the source code.

Example:

```
Qsize : constant INTEGER := 10;
subtype Qindex is INTEGER range 0 .. Qsize - 1;
Index : Qindex;
          ....

if Index > Qsize then
   raise overflow
end if;
```

Appendix B contains further data on naming conventions.

9.4.3 Blank spaces should be used to delimit all keywords within a statement line. (Rule)

Use of blank spaces enhances readability of the source code.

9.4.4 Parentheses shall be used to:

1) change default order of expression evaluation
2) ensure expected evaluation of both logical and arithmetic expressions (i.e., to clarify arithemetic expressions involving two (2) or more operators of different precedence levels)

(Rule)

Parentheses provide an unambiguous means of determining the exact processing of an expression. They also contribute to source file readability. Parentheses should not, however, be used excessively in coding expressions since overuse can actually obscure the expression. In addition, excessive use of parentheses can lead to mismatched parenthesis pairs.

Examples:

```
x := (a + b) / (c + d);

x := a + (b / c) + d;
```

9.5 STRONG DATA TYPING

The strong data typing feature of Ada should be exploited to the fullest extent practical for a given application. (Guideline)

The strong data typing feature of Ada enables many errors to be detected during compilation that would normally go undetected until later in the development process. This includes use of derived types to preclude inadvertent mixing of variables with the same base type but logically dissimilar concepts, the use of subtypes to restrict the values of an object to those allowable, and the use of private types to restrict the operations that can be performed external to the defining package.

9.6 IDENTIFIERS, VARIABLES, AND CONSTANTS

9.6.1 The naming conventions appearing in Appendix B should be followed throughout the project. (Guideline)

Naming conventions help to ensure consistency in the source code, and make it easier to maintain.

9.6.2 Named association rather than positional association shall be used for aggregates, except in initialization statements. (Rule)

Use of named association improves the readability of the code and provides exact identification of the data item field being assigned.

Examples:

```
valve := (Name      => "Water     ",
          Location  => "Warehouse ",
          Open      => TRUE,
          Flow_Rate=> 37.65);

valve : Valve_type  := ("  ","  ",FALSE,0.0);

matrix : array(1..10) of INTEGER := (1..10 => 0);
```

9.6.3 Symbolic constants shall be used instead of literals. (Rule)

Use of symbolic constants improves the general readability of the source code by decreasing the abstraction level of the expression(s) involved.

Examples:

```
First_Month : constant Month_Name := January;
Pi          : constant            := 3.141_592_65;
```

9.6.4 All variables used within a program unit shall be initialized. (Rule)

Initializing all variables ensures that data of unknown value will not be allowed to enter the processing sequence. It also compensates for compilers that do not perform automatic initialization.

9.6.5 Package and variable names should be made as descriptive as the language and naming conventions allow. (Guideline)

Use of descriptive names makes the source code more readable and the flow of processing more comprehensible. However, names that are too lengthy make the source code extremely verbose. See Appendix B for naming conventions.

9.7 NUMERIC CONVENTIONS

9.7.1 Rounding of arithmetic results should be delayed until the final computational step. (Guideline)

This prevents the introduction of inaccurate significant digits into the expression processing. This guideline should be followed unless numerical analysis of the situation dictates otherwise.

10. ADVANCED TECHNIQUES

This section contains a description of the Unit Development Folder (UDF) which will be used in developing AOTS software.

10.1 UNIT DEVELOPMENT FOLDERS

A Unit Development Folder functions as the history of an Ada package throughout its life cycle. From initial design through testing to formal customer acceptance, specific information regarding the unit's intended function, actual performance, and test results are entered into the UDF for permanent reference. Following formal acceptance, the UDF contains all records pertaining to software defects discovered in the package. The UDF also contains listings of any modifications made to the source code to repair the identified defects.

UDFs are assembled in an incremental manner, with new information being entered into the file, once that information is available. Upon completion, a UDF should contain:

1. Cover sheet and log of inclusions or updates to the UDF.
2. Specific requirements and functional capabilities which were flowed to the unit. This may be references to the CPCI Development and Product Specifications.
3. Current source listing. This listing will contain the detailed design as ADL with comments around the non-compilable portions. The design will be carried forward into the actual coding phase as part of the source. The source listing will thus provide the current state of the development effort.
4. All design walkthrough reports, code review minutes and comments, and corresponance pertaining to the unit will be maintained in chronological order with the most recent first.
5. Software trouble reports, if any defects are uncovered.
6. Software change records will be included to provide a log of changes to the unit.
7. Specific test procedures and predicted results for unit performance. Integration and system level testing for a unit will be included as part of the procedures for the higher level units providing the access to perform the test.
8. Actual results from performance of the specific test procedures.
9. Remarks and notes (general observations and/or rationale for direction of development).

The programmer responsible for a unit maintains the related UDF. The UDF is the central place for maintaining all the necessary information about a particular unit and will be heavily relied on when generating the "as-built" product documentation.

Appendix A

Computer Software Component Header

The Component Header shall be comprised of a combination of actual Ada code and specifically formed comments to describe the information needed to effectively use and maintain a component. In Ada the package can be considered a component. The package specification provides the interface description for use by other components. The package body provides the actual implementation of this interface. The implementation information should be hidden in the package body to be consistent with the Ada concepts. As such the component header for a package specification should include information for use of the component and the package body should include the information (along with the specification) for the maintenance of the package.

The Component Headers for a package specification and package body are similar in layout and will contain the following items:

1. Name
2. Identification Number
3. Purpose
4. Package Interfaces
5. History
6. Types/Objects/Exceptions
7. Procedure/Function Interface
8. Procedure/Function ADL (package body only)

Each of these sections are described in greater detail below with a specific example of a header for reference.

1. Name

The name shall be descriptive of the processing performed by the software component.

2. Identification Number

This is the design identification number used to identify the component for software configuration management purposes. The Ada recompilation rules suggest separate identifications for the specification and body. These are shown as comments at lines 9 and 80.

3. Purpose

This is a concise description of the design purpose of the software component. The two portions tend to be the same. They are shown at lines 11-12 and 82-83.

4. Package Interfaces

Contains the packages referenced by this component and the reason for the reference in a brief descriptive form. References to commonly used utility packages do not require specific references. Examples of this are given at lines 14 and 85-86.

5. History
This section contains a list of entries showing the initial developer of the component and records of the changes performed on the document. The change history includes the person making the change, the date and version number of the change, and a description of the change. The initial format of the history section is shown at lines 16-20 and 88-92.

6. Types/Objects/Exceptions
This is the set of all types, objects, and exceptions defined by this component. The descriptive nature of Ada provides the bulk of the information necessary for this section. Further tabular descriptions, used for documentation purposes in other languages, would be repetitive. A comment should precede any type, object, or exception to provide further insight into its usage. Examples are shown at lines 24-33 and 96-112.

7. Procedure/Function Interface
The procedure or function interface is specified only in the specification for those procedures or functions visible outside of the package. The procedures or functions visible only in the package body will have forward declarations with documentation comparable to the specification. This allows a simple change in the visibility of a routine if later needs so dictate. The interface consists of a comment block around the formal declaration that contains five subsections:

Purpose - Concise description of the purpose of the routine;
Interface - Description of the input/output parameters;
Assumptions - Any assumed conditions of value limitations;
Side Effects - Non-obvious impacts (e.g. global variable settings);
Error Conditions - Error code description or exceptions raised.

Examples of the header for a procedure/function are shown at lines 34-67 and 114-134.

8. Procedure/Function ADL
Each routine in the package specification or local to the package body will have ADL provided as comments. This ADL should take the design to the level that defines the basic components of the routine and the additional routine or package needs. A stub showing the placement of the ADL comments is shown at lines 142 and 147.

Computer Software Component Header examples:

```
 1  with a;
 2  use a;
 3
 4  package Menus is
 5
 6
 7  ------------------------------------------------------------
 8  --
 9  -- ID:           TBD
10  --
11  -- Purpose:      Provide a common interface for all menu
12  --               processing for consistency throughtout AOTS.
13  --
14  -- Interfaces:   A            Basic AOTS types
15  --
16  -- History:
17  --     Prepared by:     G. McBride
18  --     Baseline date:   TBD
19  --
20  --     Revised by       Date     Number    Description
21  --
22  ------------------------------------------------------------
23
24  -- Definition of menu options that are usable thruout AOTS
25      type Menu_Options_Type is (Exit_Menu, Display, Add,
26                   Change, Delete, Print, Copy, Select);
27
28  -- Enable/disable switch to define options available to user
29     type Option_Switch_Type is (Disabled, Enabled);
30
31  -- Array to specify the total options enabled for this menu
32     type Option_List_Type is array(Menu_Options_Type'range)
33          of Option_Switch;
```

```
34  ------------------------------------------------------------
35  -- Purpose:    Displays the enabled options allowing the
36  --             user to select from the displayed list and
37  --             returns the selected option.
38
39     function Select_Option (Screen_ID : STRING;
40                             Screen_Version : FLT;
41                             Primary_Description : STRING;
42                             Secondary_Description : STRING;
43                             Allowed_Options : Option_List_Type)
44              returns Menu_Options_Type;
45
46  ------------------------------------------------------------
47  --
48  -- Interface:
49  --    Screen_ID             Formal screen identifier
50  --    Screen_Version        Version number of calling package
51  --    Primary_Description   First line description of this
52  --                          menu's purpose
53  --    Secondary_Description Second line description of this
54  --                          menu's purpose
55  --    Allowed_Options       Array containing all possible
56  --                          options allowed set enabled and
57  --                          all others set to disabled.
58  --    Returns               The option selected
59  --
60  -- Assumptions:             Screen_ID has valid format,
61  --                          descriptions < 64 characters each
62  --
63  -- Side Effects:            None
64  --
65  -- Error Conditions:        None
66  --
67  ------------------------------------------------------------
68
69  end Menus;
70
```

```
 71  with A, Help;
 72  use  A;
 73
 74  package body Menus is
 75
 76  ------------------------------------------------------------
 77  --
 78  -- Name:            Menus
 79  --
 80  -- ID:              TBD
 81  --
 82  -- Purpose:         Provide a common interface for all menu
 83  --                  processing for consistency thruout AOTS
 84  --
 85  -- Interfaces:      A         Basic AOTS types
 86  --                  Help      Standardized help displays
 87  --
 88  -- History:
 89  --     Prepared by:      G. McBride
 90  --     Baseline date:    TBD
 91  --
 92  --     Revised by        Date     Number    Descrition
 93  --
 94  ------------------------------------------------------------
 95
 96  -- Display strings for the available options
 97     Option_Display : constant array (Menu_Options_Type'range)
 98                      of STRING(50) :=
 99 ("Exit from this menu                                    ",
100  "Display                                               ",
101  "Add                                                   ",
102  "Change                                                ",
103  "Delete                                                ",
104  "Print                                                 ",
105  "Copy                                                  ",
106  "Select                                                ");
107
108  -- Maximum options that can be specified for a single menu
109     Max_Options : constant INT := 9;
110
111  -- Correspondence table for enabled option to display number
112     Options_List : array(1..9) of Menu_Options_type;
113
```

```
114  ------------------------------------------------------------
115  -- Purpose:      Sets the Options_List table with only the
116  --               options enabled to allow more convenient
117  --               display and selection access.
118  --
119     procedure Set_Option_List(Allowed_Options :
120                               Option_List_Type);
121  ------------------------------------------------------------
122  --
123  -- Interface:
124  --    Allowed_Options    Array containing all possible
125  --                       options allowed are set enabled
126  --                       and all others set to disabled.
127  --
128  -- Assumptions:          None
129  --
130  -- Side Effects:         Updates the package global
131  --                       Options_List.
132  -- Error Conditions:     None
133  --
134  ------------------------------------------------------------
```

```
135
136
137     function Select_Option (Screen_ID : STRING;
138                             Screen_Version : FLT;
139                             Primary_Description : STRING;
140                             Secondary_Description : STRING;
141                             Allowed_Options : Option_List_Type)
142                 returns Menu_options_Type is
143
144     begin
142  --     Pseudo-code for Select_Option
143     end Select_Option;
144
145     procedure Set_Option_List(Allowed_Options :
146                                   Option_List_Type) is
147     begin
148  --     Pseudo-code for Set_Option_List
149     end Set_Option_List;
150
151  end Menus;
```

Appendix B

Naming Conventions

1. General

The fact that Ada permits long identifier names should be exploited to create identifiers which enhance readability in the code. Care should be taken, however, to avoid identifiers which are unnecessarily long.

Lead cap all identifiers. Underscoring characters ('_') should be used to separate full English words.

2. Types

Names of types should be nouns that indicate the class of objects to which they belong. Names of types should end with _Type to distinguish between types and objects. Plural names may simplify the selection of object names and provide a close correlation between the types and the objects. For example:

```
type Coordinates_Type is (X, Y, Z);
type Feet_Type        is digits 11;
type Vectors_Type     is array (Coordinates_Type)
                         of Feet_Type;


type Nodes_Type;
type Pointers_Type is access Nodes_Type;
type Nodes_Type is
   record
      Data : Feet_Type;
      Next : Pointers_Type;
   end record;

type Turning_Directions_Type is (Right_Turn, Left_Turn);
```

In naming type and object identifiers, a logical mixture of upper and lower case letters should be used.

3. Objects

Names of objects generally should be singular nouns or noun phrases. For example:

```
Vector : Vectors_Type;
Unit_Normal_Vector : Vectors_Type;
Turn_Direction : Turning_Directions_Type;
```

In naming type and object identifiers, a logical mixture of upper and lower case letters should be used.

4. Tasks and Packages

Tasks and packages generally should be named with noun phrases. For example:

```
package North_Pointing_Navigation is ...
package Basic_Data_Types is ...
package Waypoint_Steering is ...
package Slow_Loop is ...
package Medium_Loop is ...
package Fast_Loop is ...
```

In naming packages, tasks, and task entries, a logical mixture of upper and lower case letters should be used.

5. Procedures and Functions

Procedure and function names generally should be verb phrases. For example:

```
procedure Goto_Transmit_Mode is ...
procedure Compute_Earth_Relative_Horizontal_Velocities is ...
procedure Matrix_Scalar_Divide is ...
function Select_Option is ...
```

In naming procedures, a logical mixture of upper and lower case letters should be used.