NUREG/CR-6463, Rev. 1

Review Guidelines for Software Languages for Use in Nuclear Power Plant Safety Systems

Final Report

Manuscript Completed: August 1997
Date Published: October 1997

Prepared by
M. Hecht, D. Decker, S. Graff, W. Green,
D. Lin, G. Dinsmore, S. Koch

SoHaR Incorporated
8421 Wilshire Boulevard
Beverly Hills, CA 90211

R. Brill, NRC Project Manager

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

Prepared for
Division of Systems Technology
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001
NRC Job Code W6674



DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible electronic image products. Images are produced from the best available original document.

Abstract

Guidelines for the programming and auditing of software written in high level languages for safety systems are presented. The guidelines are derived from a framework of issues significant to software safety which was gathered from relevant standards and research literature. Language-specific adaptations of these guidelines are provided for the following high level languages: Ada83 and Ada95; C and C++; International Electrotechnical Commission (IEC) Standard 1131-3 Ladder Logic, Sequential Function Charts, Structured Text, and Function Block Diagrams; Pascal; and PL/M, . Appendices to the report include a tabular summary of the guidelines and additional information on selected languages.

Table of Contents

List of Figures xii

List of Tables xiii

Executive Summary xv

Acknowledgments xviii

List of Acronyms xix

1 Introduction 1-1
1.1 Scope 1-1
1.2 Methodology 1-2
1.2.1 Task 1 Methodology 1-4
1.2.2 Task 2 Methodology 1-6
1.2.3 Task 3 Methodology 1-7
1.2.4 Tasks 4 and 5 Methodology 1-7
1.2.5 Task 6 Methodology 1-7
1.2.6 Task 7 Methodology 1-7
1.2.7 Task 8 Methodology 1-8
1.2.8 Task 9 Methodology 1-8
1.3 Technical Basis 1-10
1.4 Contents Overview 1-11
References 1-13

2 Generic Safe Programming Attributes 2-1
2.1 Reliability 2-3
2.1.1 Predictability of Memory Utilization 2-4
2.1.2 Predictability of Control Flow 2-8
2.1.3 Predictability of Timing 2-13
2.1.4 Predictability of Mathematical or Logical Result 2-15
2.2 Robustness 2-15
2.2.1 Controlling Use of Diversity 2-16
2.2.2 Controlling Use of Exception Handling 2-18
2.2.3 Checking Input and Output 2-19
2.3 Traceability 2-20
2.3.1 Controlling Use of Built-In Functions 2-20
2.3.2 Controlling Use of Compiled Libraries 2-21
2.4 Maintainability 2-21
2.4.1 Readability 2-22

2.4.2 Data Abstraction 2-26
2.4.3 Functional Cohesiveness 2-27
2.4.4 Malleability 2-28
2.4.5 Portability 2-28
References 2-30

3 Ada 83 3-1
3.1 Reliability 3-1
3.1.1 Predictability of Memory Utilization 3-1
3.1.2 Predictability of Control Flow 3-6
3.1.3 Predictability of Timing 3-22
3.2 Robustness 3-26
3.2.1 Controlled Use of Software Diversity 3-26
3.2.2 Controlled Use of Exception Handling 3-26
3.3 Traceability 3-30
3.3.1 Use of Built-In Functions 3-31
3.3.2 Use of Compiled Libraries 3-31
3.3.3 Ada Run-time Environment 3-32
3.3.4 Maintaining Traceability Between Source Code and Compiled Code 3-32
3.3.5 Minimizing Use of Generic Units 3-33
3.4 Maintainability 3-33
3.4.1 Readability 3-33
3.4.2 Data Abstraction 3-39
3.4.3 Functional Cohesiveness 3-40
3.4.4 Malleability 3-41
3.4.5 Portability 3-41
References 3-43

4 C and C++ 4-1
4.1 Reliability 4-1
4.1.1 Predictability of Memory Utilization 4-1
4.1.2 Predictability of Control Flow 4-10
4.1.3 Predictability of Timing 4-35
4.2 Robustness 4-39
4.2.1 Controlled Use of Software Diversity 4-39
4.2.2 Controlled Use of Exception Handling 4-39
4.2.3 Input and Output Checking 4-42
4.3 Traceability 4-43
4.3.1 Minimizing the Use of Built-In Functions 4-43
4.3.2 Minimizing the Use of Compiled Libraries 4-44
4.3.3 Utilizing Version Control Tools 4-45
4.4 Maintainability 4-45

4.4.1 Readability 4-45
4.4.2 Data Abstraction 4-53
4.4.3 Functional Cohesiveness 4-55
4.4.4 Malleability 4-55
4.4.5 Portability 4-55
References 4-59

5 PLC Ladder Logic 5-1
5.1 Reliability 5-1
5.1.1 Predictability of Memory Utilization 5-1
5.1.2 Predictability of Control Flow 5-2
5.1.3 Predictability of Timing 5-13
5.2 Robustness 5-16
5.2.1 Transparency of Functional Diversity 5-17
5.2.2 Exception Handling 5-17
5.2.3 Error Containment 5-24
5.3 Traceability 5-24
5.3.1 Use of Built-in Functions 5-24
5.3.2 Use of Compiled Libraries 5-25
5.4 Maintainability 5-26
5.4.1 Readability 5-26
5.4.2 Data Abstraction 5-30
5.4.3 Functional Cohesiveness 5-32
5.4.4 Malleability 5-32
5.4.5 Portability 5-32
5.5 Security 5-33
References 5-35

6 Sequential Function Charts 6-1
6.1 Reliability 6-1
6.1.1 Predictability of Memory Utilization 6-1
6.1.2 Predictability of Control Flow 6-2
6.1.3 Predictability of Timing 6-5
6.2 Robustness 6-7
6.2.1 Transparency of Functional Diversity 6-8
6.2.2 Exception Handling 6-8
6.2.3 Input and Output Checking 6-9
6.3 Traceability 6-10
6.3.1 Use of Built-In Functions 6-10
6.3.2 Use of Compiled Libraries 6-10
6.4 Maintainability 6-11
6.4.1 Readability 6-11
6.4.2 Data Abstraction 6-15

6.4.3 Functional Cohesiveness 6-15
6.4.4 Malleability 6-16
6.4.5 Portability 6-16
References 6-17

7 Pascal 7-1
7.1 Reliability 7-1
7.1.1 Predictability of Memory Utilization 7-1
7.1.2 Predictability of Control Flow 7-5
7.1.3 Predictability of Timing 7-15
7.2 Robustness 7-16
7.3 Traceability 7-18
7.3.1 Controlling Use of Built-in Functions 7-18
7.3.2 Use of Compiled Libraries 7-18
7.4 Maintainability 7-20
7.4.1 Readability 7-20
7.4.2 Data Abstraction 7-23
7.4.3 Malleability 7-24
7.4.5 Portability 7-24
References 7-25

8 PL/M 8-1
8.1 Reliability 8-1
8.1.1 Predictability of Memory Utilization 8-1
8.1.2 Predictability of Control Flow 8-3
8.1.3 Predictability of Timing 8-19
8.2 Robustness 8-21
8.2.1 Controlled Use of Software Diversity 8-21
8.2.2 Controlled Use of Exception Handling 8-21
8.2.3 Input and Output Checking 8-22
8.3 Traceability 8-24
8.3.1 Use of Built-in Functions 8-24
8.3.2 Use of Compiled Libraries 8-25
8.4 Maintainability 8-25
8.4.1 Readability 8-25
8.4.2 Data Abstraction 8-34
8.4.3 Functional Cohesiveness 8-39
8.4.4 Malleability 8-40
8.4.5 Portability 8-41
References 8-42

9 Ada 95 9-1
9.1 Reliability 9-1

9.1.1 Predictability of Memory Utilization 9-2
9.1.2 Predictability of Control Flow 9-15
9.1.3 Predictability of Timing 9-21
9.1.4 Predictability of Mathematical or Logical Result 9-25
9.2 Robustness 9-46
9.2.1 Controlled Use of Software Diversity 9-46
9.2.2 Controlled Use of Exception Handling 9-46
9.3 Traceability 9-55
9.3.1 Use of Built-In Functions 9-55
9.3.2 Use of Compiled Libraries 9-55
9.3.3 Ada Run-time Environment 9-56
9.3.4 Maintaining Traceability Between Source Code and Compiled Code 9-56
9.3.5 Minimizing Use of Generic Units 9-57
9.4 Maintainability 9-59
9.4.1 Readability 9-59
9.4.2 Data Abstraction 9-65
9.4.3 Functional Cohesiveness 9-66
9.4.4 Malleability 9-66
9.4.5 Portability 9-66
References 9-69

10 PLC Structured Text 10-1
10.1 Reliability 10-1
10.1.1 Predictability of Memory Utilization 10-1
10.1.2 Predictability of Control Flow 10-1
10.1.3 Predictability of Timing 10-9
10.2 Robustness 10-12
10.2.1 Exception Handling 10-13
10.3 Traceability 10-14
10.3.1 Use of Built-in Functions 10-14
10.3.2 Use of Compiled Libraries 10-14
10.4 Maintainability 10-15
10.4.1 Readability 10-15
10.4.2 Abstraction 10-19
10.4.3 Functional cohesiveness 10-20
10.4.4 Malleability 10-21
10.4.5 Portability 10-21
References 10-22

11 Function Block Diagrams 11-1
11.1 Reliability 11-1
11.1.1 Predictability of Memory Utilization 11-1

11.1.2 Predictability of Control Flow 11-1
11.1.3 Predictability of Timing 11-8
11.2 Robustness 11-11
11.2.1 Exception Handling 11-11
11.3 Traceability 11-12
11.3.1 Use of Built-in Functions 11-12
11.3.2 Use of Compiled Libraries 11-12
11.4 Maintainability 11-13
11.4.1 Readability 11-13
11.4.2 Abstraction 11-18
11.4.3 Functional cohesiveness 11-19
11.4.4 Malleability 11-19
11.4.5 Portability 11-20
References 11-21

APPENDIX A. Language Descriptions A-1
A.1 Overview of PLCs A-2
A.1.1 Software Development for PLCs A-2
A.1.2 Runtime Environment A-3
A.1.3 IEC 1131-3 Programming Languages A-5
A.2 PLC Ladder Logic A-7
A.2.2 PLC Ladder Logic Example A-11
A.2.3 General Description - Ladder Logic Programming Shell A-13
A.2.4 Ladder Logic Modularization A-14
A.3 Sequential Function Charts A-15
A.3.1 SFC Steps A-18
A.3.2 SFC Transitions A-18
A.3.3 SFC Actions A-18
A.3.4 SFC Control Structures A-19
A.4 Structured Text Language A-21
A.5 Function Block Diagram Language A-25
A.6 PL/M A-27
A.6.1 Language History A-27
A.6.2 Generation of Executable PL/M Programs A-28
A.6.3 Language Overview A-29
A.6.4 General Guidelines for Using PL/M A-30
A.6.5 New Project Guidelines and Recommendations A-32
References A-34

Appendix B. Summary of Language Guidelines B-1
Generic (Language Independent) Attributes B-2
Ada83 ..... B-9
C and C++ B-29

PLC Ladder Logic B-44
IEC 1131 Sequential Function Charts B-51
Pascal B-59
PL/M B-66
Ada95 ..... B-73
PLC Structured Text B-96
Function Block Diagrams B-101

Appendix C Glossary C-1

Appendix D Relationship of Generic Attributes to Other Work D-1
D.1 IEEE Standard 603 ..... D-1
D.2 IEC Publication 880 ..... D-3
D.3 IEEE Std 7-4.3.2 1993, Appendix F D-5
D.4. Rome Laboratory Software Quality Framework D-6
D.5 Other Published Research D-8
References D-10

Appendix E Backgrounds of Subject Matter Experts and Reviewers E-1

List of Figures

Figure 1-1 Overview of Guideline Development Process 1-3
Figure 1-2 Decision Diagram for Defining Attributes from Existing Literature 1-6
Figure 2-1 Top Level Attributes 2-1
Figure 2-2 Reliability and Lower Level Attributes (procedural languages) 2-5
Figure 2-3 Reliability and Lower Level Attributes (object-oriented languages) 2-6
Figure 2-4 Robustness and Lower Level Attributes 2-15
Figure 2-5 Traceability and Lower Level Attributes 2-20
Figure 2-6 Maintainability and Lower Level Attributes 2-22
Figure 5-1 Use of *goto* 5-3
Figure 5-2 Sample of "Complex" Control Structure 5-4
Figure 5-3 Use of an Initialization Subroutine 5-7
Figure 5-4 Ladder Logic Multiple RETURN 5-8
Figure 5-5 Health Monitoring Routine Sample Program 5-19
Figure 5-5 Health Monitoring Routine Sample Program (continued) 5-20
Figure 5-6 Fault Routine That Alarms and Halts Sample Program 5-22
Figure 5-7 Fault Routine That Restarts Operation (Sample Program) 5-23
Figure 11-1 Examples of Unconditional and Conditional Jumps 11-2
Figure 11-2 Inappropriate Comparison with Floating Point Zero 11-5
Figure 11-3 Safe Method for Comparison with Floating Point Zero 11-5
Figure 11-4 Example of Inputs Obscured by Surrounding Blocks 11-14
Figure 11-5 Example of Inappropriate FBD Layout 11-15
Figure A-1 General description of a PLC software environment. A-3
Figure A-2 Real time execution of PLC program. A-4
Figure A-3 Ladder logic "rung" with IF/THEN configuration. A-11
Figure A-4 Example of Ladder Logic. A-12
Figure A-5 Subroutine calling in Ladder Logic. A-14
Figure A-6 Subroutine interface (parameter passing). A-15
Figure A-7 Subroutine call interface (parameter passing). A-15
Figure A-8 Example of Sequential Function Chart A-17
Figure A-9 Sequential Chart for Traffic Light A-20
Figure A-10 Examples of Function Blocks in IEC 1131-3 ASCII Notation A-26
Figure A-11 Example of FBD Data Types A-26

List of Tables

Table 1-1 Sources Used for the Identification of Software Safety Attributes 1-5
Table 1-2 Subject Matter Experts 1-9
Table 1-3 Error Data Sources for Validation of Attributes 1-9
Table 1-4 Technical Basis Criteria and How They Were Addressed in this Document 1-10
Table 1-5 Language Cross Reference 1-12
Table 4-1 Examples of Problems Caused by Increment and Decrement Operators 4-22
Table 4-2 Problems in Mixing Signed and Unsigned Variables 4-26
Table 8-1 Optimization and Hardware Flags. 8-18
Table A-1 Contacts A-9
Table A-2 Coils A-10
Table A-3 List of ST Operators (in descending order of precedence) A-23
Table A-4 PL/M Compilers A-28
Table D-1 Comparison of Generic Attributes with IEEE Std-603-1991 Criteria D-2
Table D-2 Relationship between Top Level Generic Attributes and IEC 880 Recommendations D-4
Table D-3 Support Provided by Attributes of Chapter 2 to Items of Concern in ACES Analysis of IEEE 7-4.3.2 ..... D-5
Table D-4 Chapter 2 Attributes and Factors in the USAF Rome Laboratory Framework ... D-7
Table D-5 Relationship between Generic Attributes and Safety Concerns or Criteria Identified by Other Researchers D-9

Executive Summary

This report provides guidance to the NRC on auditing of programs for safety systems written in the following ten high level languages: Ada83 and Ada95, C and C++ (discussed together in one chapter), International Electrotechnical Commission (IEC) Standard 1131-3 Ladder Logic, Sequential Function Charts, Structured Text, and Function Block Diagrams, Pascal, and PL/M. It could also be used by those developing safety significant software as a basis for project-specific programming guidelines. The focus of the report is on programming, not design, requirements development, or testing. However, it is not intended as a general programming style guide; excellent sources for such guidance already exist.

A uniform framework for the formulation and discussion of language-specific programming guidelines was the basis for developing the guidelines. The framework is a 3-level hierarchy. At the top of the hierarchy are *top level attributes,* i.e., attributes which largely define a general quality of software related to safety. Four top level attributes were defined. These are:

- *Reliability.* The predictable and consistent performance of the software under conditions specified in the design basis. This top level attribute is important to safety because it decreases the likelihood that faults causing unsuccessful operation will be introduced into the source code during implementation.

- *Robustness.* Robustness is the capability of the safety system software to operate in an acceptable manner under abnormal conditions or events. This top level attribute is important to safety because it enhances the capability of the software to handle exception conditions, recover from internal failures, and prevent propagation of errors arising from unusual circumstances.

•*Traceability.* Traceability relates to the feasibility of reviewing and identifying the source code and library component origin and development processes, i.e., that the delivered code can be shown to be the product of a disciplined implementation process. Traceability also includes being able to associate source code with higher level design documents. This top level attribute is important to safety because it facilitates verification and validation, and other aspects of software quality assurance.

•*Maintainability.* The means by which the source code reduces the likelihood that faults will be introduced during changes made after delivery. This top level attribute is important to safety because it decreases the likelihood of unsuccessful operation resulting from faults during adaptive, corrective, or perfective software maintenance.

Immediately below these top level attributes are *intermediate attributes,* i.e., related to the top level attribute but not sufficiently specific to define guidelines. An example of an intermediate level attribute is predictable memory utilization. At the lowest level are *base attributes,* i.e., attributes

sufficiently specific to define guidelines. An example of a base attribute is to avoid dynamic memory allocation. The guideline which can be derived from this base attribute for C programs is to avoid the use of `malloc` in safety system software.

Guidelines for Ada were developed for both the original 1983("Ada 83") and the 1995 standards ("Ada 95"). For Ada 83, the discussion encourages use of strong typing and exception handling features in Ada 83, but strongly discourages the use of tasking. Certain pragmas such as unchecked deallocation or suppression of run-time constraint checking are also strongly discouraged. The Ada guidelines were based on the those for the earlier language with additional consideration of object oriented features. The discussion encourages use of strong typing and exception handling features in Ada95, but strongly discourages the use of tasking. Certain pragmas (compiler and system directives) such as unchecked deallocation or suppression of run-time constraint checking are also strongly discouraged.

Guidelines for C and C++ were combined into a single chapter because of the close relationship between the two languages and because programs written in C++ are also likely to contain C code as well. Although C programs can interact extensively with operating systems or real time kernels, a discussion of these issues is not included because it is related to specific operating system characteristics and is beyond the scope of this study. The discussion emphasized the problems in memory allocation and deallocation, pointers, control flow, and software interfaces.

Guidelines for Ladder Logic were discussed for the language as defined by the IEC 1131-3 standard, but emphasized that implementations vary significantly among vendors. Ladder Logic is fundamentally different from other high level languages in that it is more symbolic, has a limited number of data types, and has a more limited syntax. Another difference is that Ladder Logic is closely associated with PLCs, computers specialized for real time industrial control. This specialization results in unique I/O capabilities but limited information processing features. The graphical syntax of Ladder Logic requires that safety system programs be well organized in both their control flow and the structure of their internal data storage.

Guidelines for IEC 1131-3 Sequential Function Charts (SFCs) also recognized the differences among vendor implementation as well as fundamental difference between the programming paradigm for that language and those of other languages. SFCs are intended as a way to organize the control flow of lower level software modules written in other languages defined by the IEC 1131-3 standard (including Ladder Logic). The guidelines emphasized the proper use of SFCs given their intended purpose and orientation. The guidelines also identified potential pitfalls in the application of SFCs to safety systems.

Guidelines for the IEC 1131-3 Structured Text (ST) and Function Block Diagram (FBD) languages assumed strict conformance to the standard because there is less variation among vendor offerings. ST is a text-based language similar to Pascal whereas FBD uses a graphical representation. Both languages are closely associated with PLCs, computers specialized for real-time industrial control. This specialization results in unique I/O capabilities but limited information processing features.

The guidelines emphasize the proper use of these languages given their intended purpose and orientation.

The discussion of Pascal addressed not only the ANSI standard, which is fairly limited, but also the most popular extensions. Addressing the extensions is important because they are more widely used in real time and near-real time systems than is the standard language. The focus of the discussion was similar to C, dealing with memory allocation and deallocation, pointers, and software interfaces.

PL/M is a language that has been used extensively in microprocessor control applications, but which is now no longer being supported by its corporate progenitor. The guidelines that were developed were similar to those of C and Pascal. However, a specific concern for the use of PL/M in safety systems is the preservation of the technical base including people, software tools, and support environments.

Appendices to the document include (a) additional descriptive material on the specialized real time control languages discussed in this report (PLC Ladder Logic, SFCs, ST, FBDs and PL/M), (b) tabular summaries of the guidelines in the main body of the report, a glossary together with an assessment of their importance, (c) a glossary, (d) additional material on the origin of the generic attributes, and (e) a brief description of the background of the report contributors.

This report was prepared as an account of work sponsored by the Nuclear Regulatory Commission, an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any employees, makes any warranty, expressed or implied, or assumes legal liability or responsibility for any information, apparatus, product, or process disclosed in this report, or represents that its use by such a third party would not infringe privately owned rights. The opinions, findings, conclusions, and recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NRC. Use of these guidelines will assist auditors in identifying problems in the implementation of safety system programs, but it does not guarantee that such problems will not occur. The emphasis of these guidelines was on common attributes and related problems; it was not possible for the subject matter experts to exhaustively consider all legal constructs in each of the languages.

Acknowledgments

We acknowledge the support and interest of the NRC Office of Research and in particular, that of Mr. Robert Brill, the project manager. The additional review and comments from the National Institute of Standards and Technology and from Dr. David Binkley are also appreciated. We also wish to thank Mario Gareri, Michael Waterman, John Gallagher, and all the other individuals from the NRC who contributed their views and comments to enhance this document. We also wish to thank members of the ISO Ada95 Annex H Rapporteur's Group whose work enhanced this document..

List of Acronyms

ANSI	American National Standards Institute
BSO	Boston System Organization
CPU	Central Processing Unit
DPMI	DOS Protected Mode Interface
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
HMI	Human Machine Interface
ICE	In Circuit Emulator
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
IL	Instruction List
ISO	International Standards Organization
LRM	(Ada) Language Reference Manual
NIST	National Institute of Standards and Technology
NRC	Nuclear Regulatory Commission
PID	Proportional+Integral+Derivative
PLC	Programmable Logic Controller
RTE	(Ada) Run-time Environment
SCADA	Supervisory Control and Data Acquisition
SFC	Sequential Function Chart
SME	Subject Matter Expert
SPC	Software Productivity Consortium
ST	Structured Text
TVA	Tennessee Valley Authority

1 Introduction

This is the revised final report prepared in accordance with the requirements of Nuclear Regulatory Commission (NRC) Contract RES 04-94-046.[1] This document describes characteristics and programming guidelines for the following high level languages.

- Ada83
- Ada95
- C and C++
- IEC 1131-3 Ladder Logic
- IEC 1131-3 Sequential Function Charts
- IEC 1131-3 Structured Text
- IEC 1131-3 Function Block Diagrams
- Pascal
- PL/M

The report provides guidance to the NRC for reviewing high-integrity software in nuclear power plants. Thus, the focus of the document is on *implementation* (i.e., programming). Issues related to *design, requirements, verification and validation,* and the *development process* are covered in other industry standards and NRC reports (e.g., IEEE 7-4.3.2-1993, IEC 880, NUREG/CR 5930, NUREG/CR 6263, and NUREG/CR 6293). In this document, these topics are covered only to the extent that they affect implementation.

This report was prepared as an account of work sponsored by the Nuclear Regulatory Commission, an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any employees, makes any warranty, expressed or implied, or assumes legal liability or responsibility for any information, apparatus, product, or process disclosed in this report, or represents that its use by such a third party would not infringe privately owned rights. The opinions, findings, conclusions, and recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NRC.

1.1 Scope

Certain programming practices can affect the safety of digital systems, and hence, guidelines can be developed to enhance their dependability. This document identifies such guidelines for safety related software written in the high level languages identified above. This report is not intended as a general programming style guide; excellent sources already exist for these languages. However,

[1]The original version of NUREG/CR 6463 covered only the following 6 languages: Ada 83, C/C++, PLC Ladder Logic, IEC 1131-3 Sequential Function Charts, Pascal, and PL/M

this document could be used to review the development of safety-critical systems to supplement guidance in existing coding standards or as part of the basis for reviewing non-safety grade software incorporated in safety grade systems.

Because of the focus of this work, many programming topics were excluded unless they directly affected safety. Such topics include object-oriented analysis and design, code reuse, and efficiency (e.g. minimizing resource requirements or optimizing for response time).

The applicability of the generic attributes and language specific guidelines is affected by many characteristics of a safety-related system. Where possible, these have been noted in the document. However, not all such factors can be anticipated by the subject matter experts who contributed to the language specific sections . Moreover, the general subject of coding practices and styles can be controversial. Users of this document should take both the guidance contained in this document, the specific project characteristics and the existing practices of the development organization into account as they consider the application of these guidelines.

1.2 Methodology

Figure 1-1 shows the process by which the language guidelines were developed. The work is divided into 9 tasks, five of which were performed under a base contract; the remainder were performed under a contract modification which extended the scope and period of performance. The following is a listing of the tasks:

- Task 1, Generic Characteristics: Define language independent software attributes affecting safety
- Task 2, Language Assessment: Relate language independent software attributes to language specific programming guidelines
- Task 3, Peer Review: Revise results of Tasks 1 and 2 based on review by independent Subject Matter Experts (SMEs) acting as reviewers.
- Task 4, Seminar: Present results (base contract)
- Task 5, Final Report (base contract)
- Task 6, Language Assessment (additional languages: Ada95, IEC 1131 Function Block Diagrams, IEC 1131 Structured Text)
- Task 7, Validate guidelines using problem reports from actual software development projects

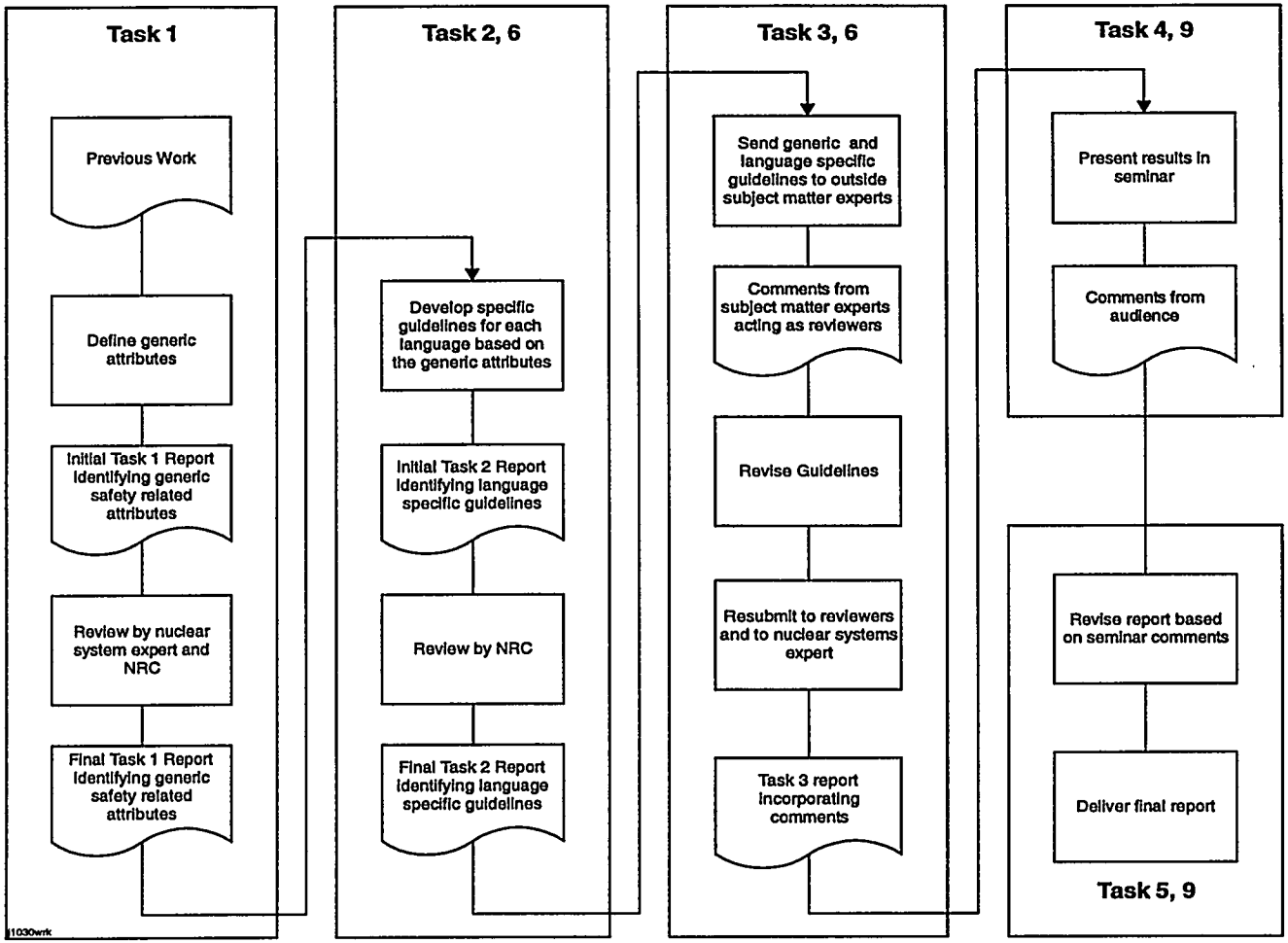


Figure 1-1 Overview of Guideline Development Process

- Task 8, Hypertext Version of the Final Report

- Task 9, Seminar for additional languages

The following subsections discuss the methodology in greater detail.

1.2.1 Task 1 Methodology

In Task 1, generic attributes of computer languages were defined through the following iterative 3-step process:

1. Identify safety related software attributes from review of existing work.
2. Classify and group attributes.
3. Validate classification.

In the first step, attributes related to safety identified in relevant standards and the current literature were identified. Table 1-1 identifies the sources from which the majority attributes were extracted. The attributes from Step 1 were aggregated and regrouped into a three level hierarchy as follows:

- *Top level attributes:* attributes which largely define a general quality of software related to safety. An example of a top level attribute is reliability.

- *Intermediate attributes:* attributes related to the top level attribute but which are not sufficient specific to define guidelines. An example of an intermediate level attribute is predictable memory utilization.

- *Base attributes:* Attributes related to intermediate attributes and sufficiently specific to define guidelines. An example of a base attribute is to avoid dynamic memory allocation. The guideline which can be derived from this base attribute for C programs is to avoid the use of `malloc` in safety systems.

The process was iterative. An initial framework was established, and the grouping and classification was modified as additional references were consulted and attributes added. The decision diagram for defining and classifying the attributes is shown in Figure 1-2.

Table 1-1 Sources Used for the Identification of Software Safety Attributes

Andersen, O. and P.G. Petersen, *Standards and regulations for software approval and certification,* Elektronik Centralen Report ECR 154 (Denmark), 1984.
Bowen, T.P. and G.B. Wigle and J.T. Tsai, "Specification of Software Quality Attributes" Report, 3 Vols. RADC-TR-85-37, available from NTIS, 1985.
Gottfried, R.and D. Naiditch, *Using Ada in Trusted Systems, Proc. of COMPASS 93,* May, 1993, National Institute of Standards and Technology, Washington, DC, 1993.
Institute of Electrical and Electronic Engineers, Nuclear Power Engineering Committee, IEEE Std-603-1991, *IEEE Standard for Nuclear Power Generating Stations.*
Institute of Electrical and Electronic Engineers, IEEE-Std-7 -4.3.2-1993, *IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Station.*
International Electrotechnical Commission (IEC), "Software for Computers in the Safety Systems of Nuclear Power Stations," Standard 880.
McDermid, J.D., ed., *Software Engineer's Reference Book,* CRC Press, Inc., Cleveland, Ohio, 1993.
Leveson, N.G. and C.S. Turner, *An Investigation of the Therac-25 Accidents,* University of California, Irvine Technical Report 92-108, Irvine, California, 1992.
McGarry, F., "The Impacts of Software Engineering," briefing presented to the NRC Advisory Committee on Reactor Safeguards (ACRS), August 21, 1992.
Murine, G.E., "Rome Laboratory Framework Implementation Guidebook", RL-TR-94-149, USAF Rome Laboratory, March 1994.
Parnas, D.L., A.J. van Schouwen and S.P. Kwan, "Evaluation of Safety Critical Software," *Communications of the ACM,* Vol. 33, No. 6, p. 636, June, 1990.
Proceedings of the Digital Systems Reliability and Nuclear Safety Workshop, NUREG/CP-0136, NIST SP 500-216, 1993.
Smith, D.J. and K.B. Wood, *Engineering Quality Software: A review of Current Practices, Standards, and Guidelines Including New Methods and Development Tools.* New York: Elsevier Applied Sciences, 1989.
U.S. Department of Defense, DoD-Std-2167A, *Software Development Standard.*
Witt, B.I. and F.T. Baker and W.W. Merritt, *Software Architecture and Design.* Van Nostrand Reinhold, New York, 1994.

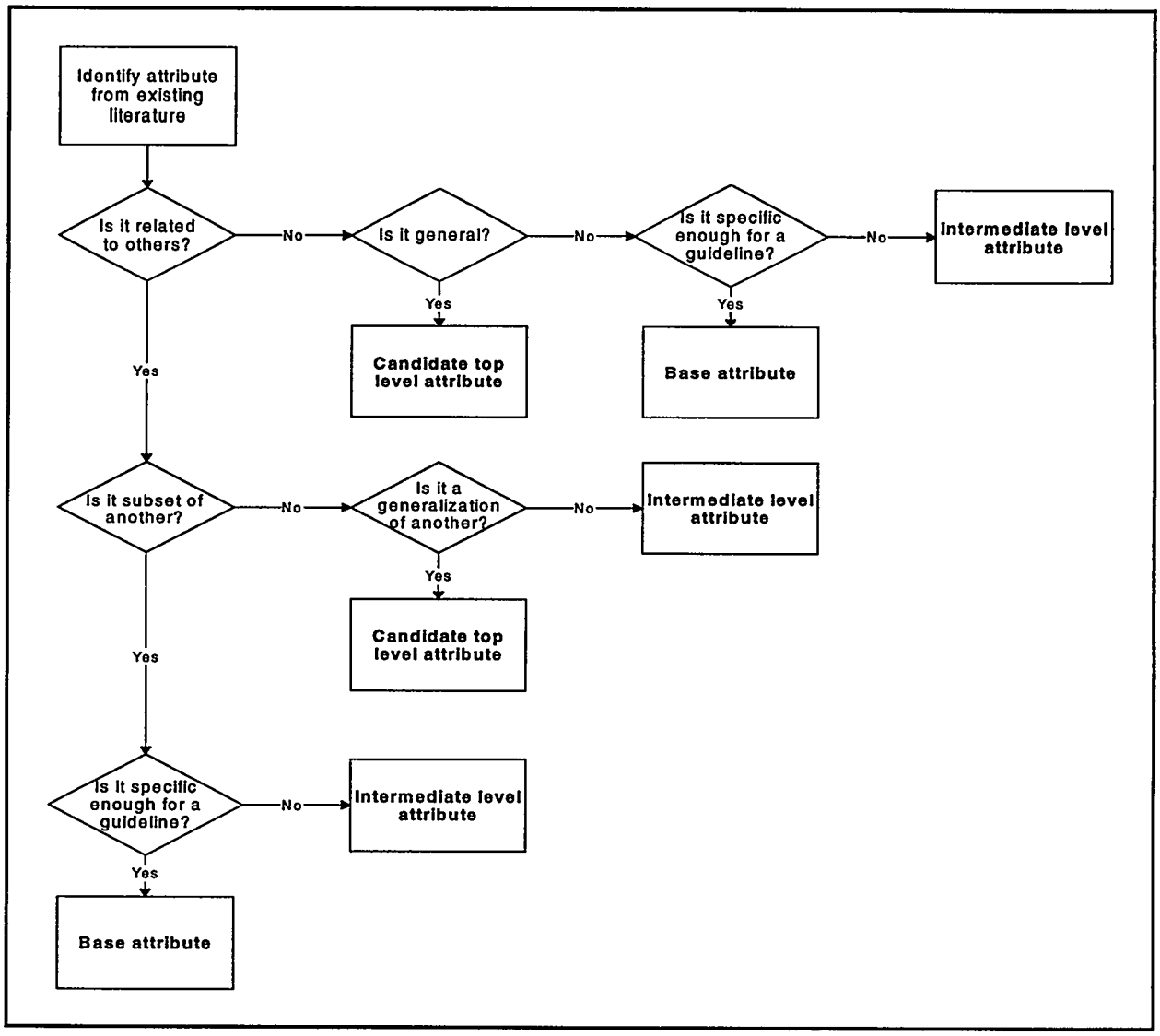


Figure 1-2 Decision Diagram for Defining Attributes from Existing Literature

1.2.2 Task 2 Methodology

In Task 2, these attributes were provided to an initial set of Subject Matter Experts (SMEs) who developed language-specific guidelines. These experts developed language-specific guidelines as stand-alone documents in conjunction with the authors of the Task 1 report, who also served as reviewers. The SMEs were briefed on the specific nature of this work, that is, concentrating on safety and language-specific issues. The SMEs were also instructed to provide published literature citations as references for any points that they felt would be controversial. Each SME report prepared for a Task 2 report was reviewed and revised. This process allowed for the resolution of technical disagreements and uncertainties. The results of the SMEs' work were then edited for

uniformity and integrated into a single document. The results of the Task 2 report were then sent to a panel of expert reviewers for their comments. Preliminary and final copies of this report were prepared.

1.2.3 Task 3 Methodology

In task 3, the generic attributes and language specific guidelines were submitted to an independent set of SMEs who served as reviewers. These reviewers provided an initial round of comments, after which the guidelines were revised. The guidelines were then resubmitted to the reviewers for a final round of evaluations.

Each SME has one or more graduate degrees and a substantial background in software development in both safety-critical systems and in the particular language for which the criteria were developed. Appendix E provides additional information on the software development background of these individuals.

1.2.4 Tasks 4 and 5 Methodology

The Task 3 report was circulated for comment within the NRC as well as to selected individuals outside of the NRC. As part of Task 4, a seminar was conducted at which time additional comments and feedback on the specific guidelines and the general conclusions of the report were gathered. These comments resulted in additional changes which were then incorporated into the final document

1.2.5 Task 6 Methodology

The methodology used in task 6 was similar to that used in the previous tasks. However, there were two differences: (a) additional guidelines were developed to handle the object-oriented design and coding issues introduced by Ada95, and (b) as of this writing of this introduction, the languages covered in this report have not had the same level of operational use and experience in high integrity systems as most of the languages in the main report. The first issue was addressed by modifying the guidelines contained in the main body of this report with a revised Chapter 2. For the second issue, the work on Ada95 was enhanced by participation in the ISO Annex H Rapporteurs Group.

Table 1-2 lists the SMEs who served as authors and reviewers for tasks 2, 3, 5, and 6.

1.2.6 Task 7 Methodology

The classification was validated by comparing the attributes with the causes and descriptions of

failures in two major air traffic control projects (the Federal Aviation Administration Advanced Automation System and Voice Control Switching System) as well as incident reports from the Eagle 21 reactor protection system upgrades at the Tennessee Valley Authority (TVA) Sequoyah Nuclear Plant. For the C and Ada languages, a total of 150 specific failure reports were associated with specific guidelines. A search for failure reports on IEC 1131 was undertaken by Additional validation came from other published large scale studies of software failures. These are identified in Table 1-3.

1.2.7 Task 8 Methodology

This report was translated into the Hypertext Markup Language (HTML), and links were provided between generic guidelines, language-specific guidelines, and, where applicable, synopses of problem reports.

1.2.8 Task 9 Methodology

A seminar based on the guidelines prepared for Ada95, IEC 1131 Function Block Diagrams, IEC 1131 Structured Text was presented at the NRC.

Table 1-2 Subject Matter Experts

Language	Guideline Author SMEs	Reviewer SMEs
Ada83	S. Graff W. Green	B. Sanden, Ph.D K.S. Tso, Ph.D E. Shokri, Ph.D
Ada95	G. Dinsmore, Ph. D E. Shokri, Ph.D	K.S. Tso, Ph.D
C	D. Lin, Ph.D A. Tai, Ph.D	A. Sorkin, Ph.D E. Shokri, Ph.D K. Ossia, Ph.D
IEC 1131-3 Sequential Function Charts	S. Koch, Ph.D H. Hecht, Ph.D	D. Decker J. Pollard
IEC 1131-3 Structured Text IEC 1131-3 Function Block Diagrams	D. Decker K. Ossia, Ph.D	J. Pollard
IEC 1131-3 Ladder Logic	S. Koch, Ph.D H. Hecht, Ph.D	D. Decker J. Pollard
Pascal	S. Graff M. Hecht	A. Sorkin, Ph.D
PL/M	D. Wendelboe	A. Sorkin, Ph.D M. Justice
Nuclear Systems	J. Leivo	

Table 1-3 Error Data Sources for Validation of Attributes

Thayer, R., *"Software Reliability Study,"* Rome Air Development Center report RADC TR 76-238, March, 1976.
Chillarege, R., "Orthogonal Defect Classification," *IEEE Trans. SW Engineering*, November, 1991.
TVA Letter to NRC Dated May 10, 1990, *Sequoyah Nuclear Plant (SQN) — Eagle 21 Functional Upgrade Commitments*, NRC Public Document Room, Accession #910715001.
Advanced Automation System Program Trouble Report data (IBM/Loral) January, 1993 to July, 1994, U.S. Federal Aviation Administration Contract DTFA01-88-C-00042.
Voice Switching and Communication System Change Request (SCR) data (Harris Corp.), January, 1991 to July, 1994, Federal Aviation Administration Contract DTFA01-87-C-00002.

1.3 Technical Basis

Five criteria for a technical basis on which the use of digital systems could be justified were defined in NUREG/CP-0136 (Beltracchi, 1994, p. 39). Table 1-4 shows how these criteria have been addressed in this document.

Table 1-4 Technical Basis Criteria and How They Were Addressed in this Document

Technical basis criterion	How addressed
1. The topic has been clearly coupled to safe operations.	The rationale for each guideline has been stated in this document
2. The scope of the topic is clearly defined.	Section 1.1 describes the scope of language specific safety concerns.
3. A substantial body of knowledge exists, and the preponderance of the evidence supports a technical conclusion.	Language-specific guidelines were based on generic attributes of safety critical software using the methodology defined in Section 1.2. References associated with the guidelines are provided at the end of each chapter Language-specific guidelines for each language were prepared by SMEs with an average of 20 years' overall programming experience. Language specific guidelines were reviewed by independent SMEs
4. A repeatable method to correlate relevant characteristics with performance exists.	Not addressed in this document. Due to the paucity of failure data on digital nuclear safety systems and the (fortunate) rarity of events resulting in challenges to such systems, a repeatable method for correlating the identified attributes with safe operation is not possible at this time. However, data collection to permit assessment of the guidelines using actual failure experience is planned for a later enhancement of this document.
5. A threshold for acceptance can be established.	Not directly addressed in this study. The guidelines identify qualitative attributes rather than quantitatively measurable parameters. Substantial progress in research on the quantitative failure behavior of high integrity software is necessary to formulate a threshold.

The guidelines developed in this work provide a basis for the auditing and development of dependable software in safety systems, but can not be considered exhaustive because they are written without knowledge of the specific systems, language variants, and software development environments to which they may be applied. Certain guidelines proposed by SMEs were rejected based on the judgment of the editors or Task 3 SMEs that they were obscure or overly prescriptive, that is, limiting the use of a language or advocating a certain style where the safety benefit was unclear. On the other hand, not all guidelines included in this document may be applicable to a specific project because of the presence or absence of certain requirements and design constraints, the characteristics of a particular development environment, the testing program, or other factors.

Use of these guidelines will assist auditors in identifying problems in the implementation of safety system programs, but it does not guarantee that such problems will not occur. The emphasis of these guidelines was on common attributes and related problems; it was not possible for the subject matter experts to exhaustively consider all legal constructs in each of the languages.

1.4 Contents Overview

This report is organized as follows: the second chapter of the report describes the generic attributes for software safety and the resultant guidelines. Chapters 3 through 11 describe language-specific guidelines for Ada-83, C and C++, PLC Ladder Logic, IEC 1131 Sequential Function Charts, Pascal, and PL/M, Ada95, IEC 1131-3 Structured Text, and IEC 1131-3 Function Block Diagrams. References are provided for the languages at the end of each chapter. Appendix A includes an introductory discussion of PLCs, Ladder Logic, Sequential Function Charts, and PL/M. Appendix B includes tables summarizing the language specific guidelines for the 6 languages discussed in the main body of the report. These tables are intended to provide a brief overview of the guidelines and to satisfy the requirement for a language matrix in the Statement of Work. Appendix C is a glossary, Appendix D provides additional technical basis for the report, and Appendix E summarizes the qualifications of the subject matter experts participating in the report.

Table 1-5 is a cross reference by language. It provides recommended selections of the report to readers interested in a specific language.

Table 1-5 Language Cross Reference

Language	Relevant Chapters	Relevant Appendices
Ada83	Chapter 2 (generic guidelines) Chapter 3 (Ada specific guidelines)	Appendix B.1 (guideline summary and weighting factors) Appendix C (Glossary)
Ada95	Chapter 2 (generic guidelines) Chapter 9 (Ada95 specific guidelines)	Appendix B.7 (guideline summary and weighting factors) Appendix C (Glossary - Main Report)
C and C++	Chapter 2 (generic guidelines) Chapter 4 (C and C++ specific guidelines)	Appendix B.2 (guideline summary and weighting factors) Appendix C (Glossary)
IEC 1131-3 PLC Ladder Logic	Chapter 2 (generic guidelines) Chapter 5 (PLC Ladder Logic Specific Guidelines)	Appendix A.1 (PLC description) Appendix A.2 (Ladder Logic description) Appendix B.3 (guideline summary and weighting factors) Appendix C (Glossary)
IEC 1131-3 Sequential Function Charts	Chapter 2 (generic guidelines) Chapter 6 (SFC Specific Guidelines)	Appendix A.1 (PLC description) Appendix A.3 (SFC description) Appendix B.4 (guideline summary and weighting factors) Appendix C (Glossary)
IEC 1131-3 Structured Text	Chapter 2 (generic guidelines) Chapter 10 (ST Specific Guidelines)	Appendix A.1 (PLC description) Appendix A.4 (ST description) Appendix B.8 (guideline summary and weighting factors) Appendix C (Glossary - Main Report)
IEC 1131-3 Function Block Diagrams	Chapter 2 (generic guidelines) Chapter 11 (FBD Specific Guidelines)	Appendix A.1 (PLC description) Appendix A.5 (FBD description) Appendix B.9 (guideline summary and weighting factors) Appendix C (Glossary - Main Report)
Pascal	Chapter 2 (generic guidelines) Chapter 7 (Pascal specific guidelines)	Appendix B.5 (guideline summary and weighting factors) Appendix C (Glossary)
PL/M	Chapter 2 (generic guidelines) Chapter 8 (PL/M specific guidelines)	Appendix A.4 (PL/M description) Appendix B.2 (guideline summary and weighting factors) Appendix C (Glossary)

References

Beltracchi, L., "NRC Research Activities", *Proceedings of the Digital Systems Reliability and Nuclear Safety Workshop*, NUREG/CP-0136, conducted by the NRC in conjunction with NIST, March, 1994.

Hecht, H. et al., *Verification and Validation Guidelines for High Integrity Systems*, NUREG/CR-6293, Vols. 1 and 2, March, 1995.

Institute of Electrical and Electronic Engineers, *Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations,* ANSI/IEEE Std 7-4.3.2-1993.

International Electrotechnical Commission, *Software for Computers in the Safety Systems of Nuclear Power Stations,* IEC Standard 880, 1986.

National Institute of Standards and Technology, *High Integrity Software Standards and Guidelines,* NUREG/CR-5930, NIST SP 500-204, September, 1992.

Saaltink, M., and S.. Michell, *Ada95 Trustworthiness Study: Guidelines on the Use of Ada95 in the Development of High Integrity Systems, Version 1.0*, Document TR 96-5499-94, ORA Canada, Ottawa, Ontario, September, 1996.

Seth, S., et. al., *High Integrity Software for Nuclear Power Plants: Candidate Guidelines, Technical Basis, and Research Needs*, NUREG/CR-6263, MTR 94W0000114, Vols. 1 and 2, June, 1995.

2 Generic Safe Programming Attributes

This chapter describes generic, or language-independent, attributes of safe programming. These attributes are used as a basis for deriving the language-specific guidelines described in the following chapters. As noted in the previous chapter, the attributes have been defined in a hierarchical, three-level framework. The top-level attributes, shown (bold) in Figure 2-1, are:

- *Reliability.* Reliability is the predictable and consistent performance of the software under conditions specified in the design basis. This top level attribute is important to safety because it decreases the likelihood that faults causing unsuccessful operation will be introduced into the source code during implementation.

- *Robustness.* Robustness is the capability of the safety system software to operate in an acceptable manner under abnormal conditions or events. This top level attribute is important to safety because it enhances the capability of the software to handle exception conditions, recover from internal failures, and prevent propagation of errors arising from unusual circumstances (not all of which may have been fully defined in the design basis).

- *Traceability.* Traceability relates to the feasibility of reviewing and identifying the source code and library component origin and development processes i.e., that the delivered code can be shown to be the product of a disciplined implementation process. Traceability also includes being able to associate source code with higher level design documents. This top

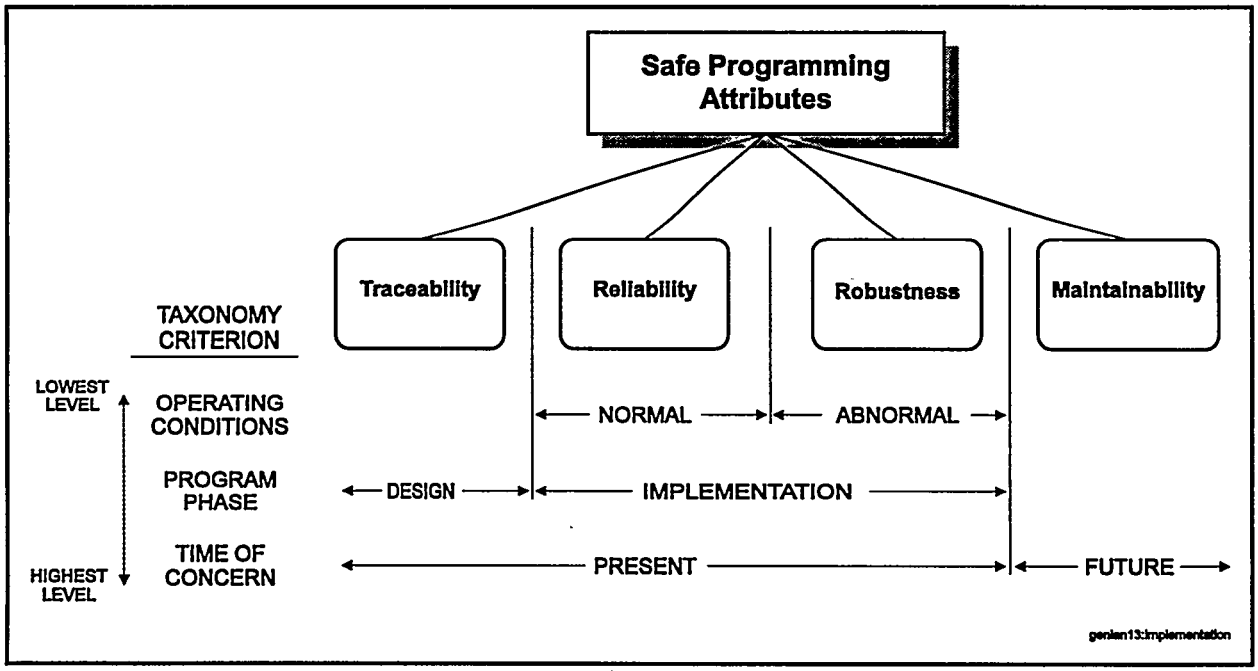


Figure 2-1 Top Level Attributes

level attribute is important to safety because it facilitates verification and validation, and other aspects of software quality assurance.

- *Maintainability.* Maintainability is the means by which the source code reduces the likelihood that faults will be introduced during changes made after delivery. This top level attribute is important to safety because it decreases the likelihood of unsuccessful operation resulting from faults during adaptive, corrective, or perfective software maintenance.

Sections 2.1 through 2.4 discuss each of these attributes in greater detail. Appendix B lists and summarizes the associated lower level attributes, their relative priorities, and mitigation approaches (where applicable). Appendix D shows their relationship to applicable Institute of Electrical and Electronic Engineers (IEEE), International Electrotechnical Commission (IEC), and Department of Defense (DoD) standards and frameworks. It also contains a discussion of how these attributes compare with other work in software safety.

The guidelines set forth below assume the existence of "project guidelines" which have been adopted by the development contractor and approved by the responsible agency. It is also assumed that the project guidelines were reviewed by the auditor before beginning any review of the actual code. In some of the guidelines of this document, reference is made, usually implicitly, to the project guidelines. All references in this document of a comparative nature, such as a reference that some code characteristic should not be "excessive" or "too small", for example, should always be interpreted with respect to the project guidelines and good safety engineering practice.

After the first publication of this document, several enhancements to it were begun. In particular, a large number of trouble reports relating to Ada83 and C were analyzed and correlated with the the specific guidelines defined for these two languages for the purpose of providing World Wide Web support to the chapters discussing these languages. This analysis suggested a need for a slight revision of the lower-level attributes used to characterize the guidelines. The analysis also suggested a firmer theoretical basis for the taxonomy of faults underlying the guidelines. A presentation of this revised taxonomy and its theoretical basis follows.

First, it is easy to see that the purpose of this document (to aid in the review of software used in nuclear power plant safety systems) matches the top-level attributes well and that a clear taxonomy exists which can be shown as basis for these attributes. In particular, the scope of the review envisioned by this document extends from a review of software related design documents to line-by-line source code examination. The need encompasses both the software as it exists currently and as it will be modified in the future. The present software state is reviewed in accordance with the first three attributes (reliability, robustness, and traceability); the future software state is examined in terms of the fourth attribute, maintainability. The present software state is further examined in terms of its implementation (reliability and robustness) and its design (traceability -- i.e., does the implementation match the system requirements, the mathematical model of the system physics?). Finally, the distinction between the reliability attribute and the robustness attribute is the former's

emphasis on nominal operating conditions and the latter's emphasis on off-nominal operating conditions. The resulting top-level taxonomy is shown (in normal face type) in Figure 2-1.

2.1 Reliability

In the software context, reliability is either (1) the probability of successful execution over a defined interval of time and under defined conditions, or (2) the probability of successful operation upon demand (IEEE, 1977). That the software executes to completion is a result of its proper behavior with respect to system memory and program logic. That the software produces timely output is a function of the programmer's understanding of the language constructs and run-time environment characteristics. Thus, the intermediate attributes for reliability set forth in the original version of this document were :

•*Predictability of memory utilization.* There is a high likelihood that the software will not cause the processor to access unintended or unallowed memory locations.

•*Predictability of control flow.* There is a high probability that the processor will execute instructions in sequences intended by the programmer.

•*Predictability of timing.* There is a high probability that the software executing within the defined run-time environment will meet its response time and capacity constraints.

The same analysis that produced the top level taxonomy described above also suggested, for the object-oriented Ada95 and possibly C++, a fourth intermediate attribute as well as clarification of the meaning of these three attributes. The fourth attribute is:

- *Predictability of mathematical or logical result.* There is a high probability that the software executing within the defined run-time environment will yield the programmer-intended mathematical or logical result.

The further clarification of these intermediate attributes is they all refer to the immediate or proximate result of the specific line(s) of code under review at the point the guideline is applied. Further justification for the addition of this fourth intermediate-level attribute comes from (Saaltink, 96, page 5). In discussing Ada95, the authors identify "functional predictability" as a high integrity software needs category, where this category is defined as "the predictability of the values of outputs or, for concurrent systems, the sequence of interactions or outputs". This definition appears to include both the predictability of control flow as well as the predictability of mathematical or logical result.

As shown in Figures 2-2 and 2-3, for the non-object oriented languages and for the object-oriented languages, respectively, each of these intermediate attributes has multiple base attributes. As may be seen by comparing these two figures, interpreting the intermediate attributes in accordance with the preceding paragraph results in a change in the assignment of base attributes to intermediate

attributes for the object-oriented Ada95. Although not shown on Figure 2-3, some characteristics of base attributes otherwise assignable to the higher-level attributes of Readability and Maintainability apply to Reliability and could be assigned to the new fourth intermediate attribute. The figures also show that base attributes related to object-oriented programming (control over polymorphism, minimization of dynamic binding, and control over overloading) assigned in Figure 2-2 to both memory utilization and control flow are assigned to the new fourth intermediate attribute in Figure 2-3. These attributes are discussed further in the following sections.

The effect of adding this fourth intermediate attribute is that the Ada95 chapter is organized somewhat differently from the other language chapters.

This revised taxonomy has not been applied to C++ even though this is oviously an object-oriented language. The reasons for this is that the authors' experience with C++ suggests that such benefit would at this time be small. Most C++ programmer are still primarily using the features of C++ which do not exploit the object-oriented characteristics of the language. Programmers are largely using C++ classes and C++ compilers for programs which are really C programs.

Future analysis may show the object-oriented language attribute structure for Reliability may apply to procedural languages as well. However, since this has not yet been shown, the attribute structure for procedure-oriented languages will continue to be used for all the languages except Ada95. To simplify the presentation in this chapter, the older, procedure-oriented attribute structure will be used throughout this chapter, with parenthetic remarks as appropriate for object-oriented languages.

2.1.1 Predictability of Memory Utilization

This section discusses the following base attributes that facilitate the predictability of memory utilization:

•Minimizing dynamic memory allocation
•Minimizing memory paging and swapping.

2.1.1.1 Minimizing Dynamic Memory Allocation

Dynamic memory allocation is used in programs to temporarily claim (allocate) memory when necessary during run time and to free the memory (also during run time) for other uses when no longer needed. The safety concern is that when memory is dynamically allocated in a real-time system, the software may not subsequently release all or some of it. This can happen either because:

•The application program allocates memory to itself but does not free it as part of normal execution paths, or

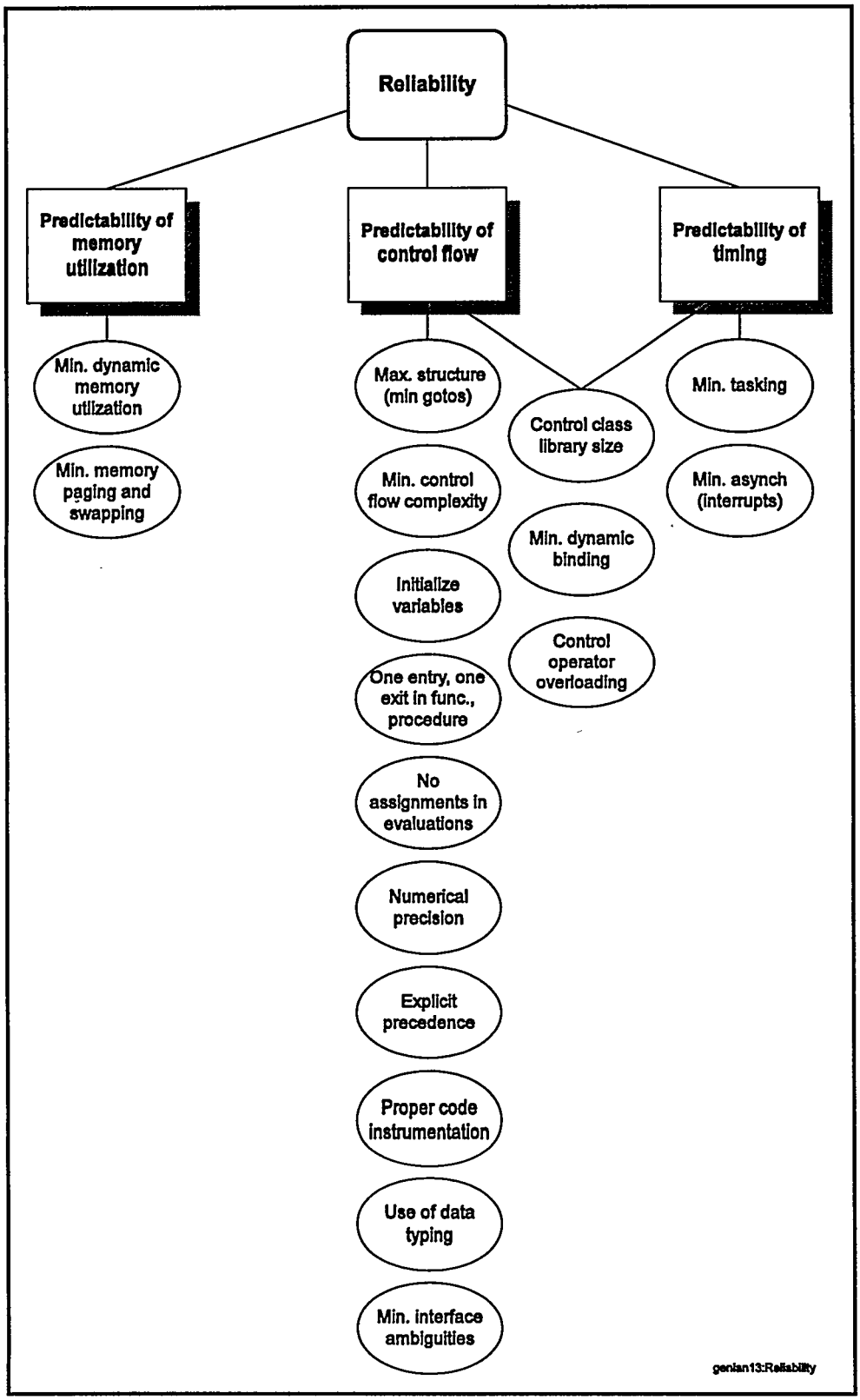


Figure 2-2 Reliability and Lower Level Attributes (procedural languages)

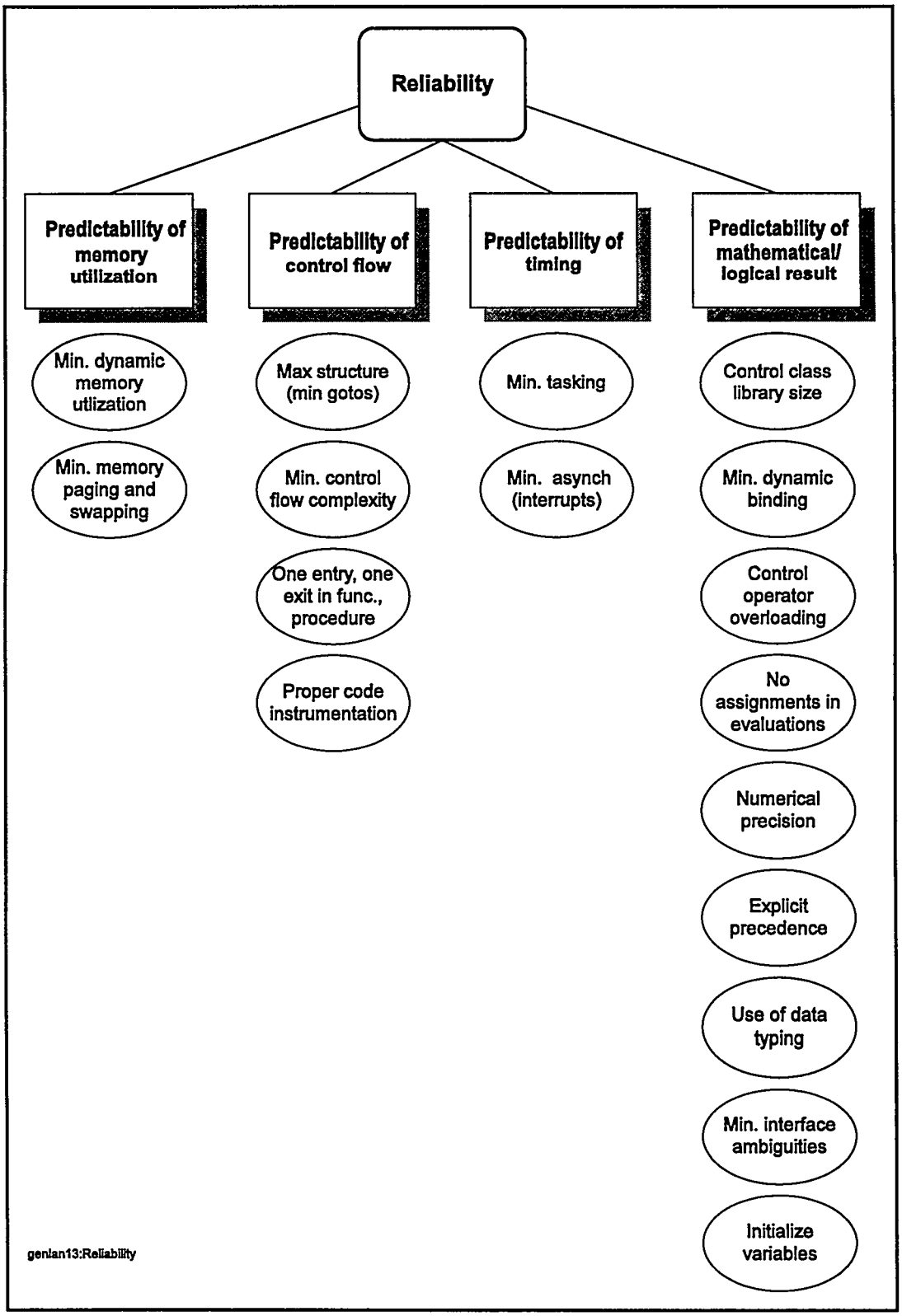


Figure 2-3 Reliability and Lower Level Attributes (object-oriented languages)

•A program which has temporarily allocated memory to itself is interrupted in its execution prior to executing the statement which releases the memory.

Either of these situations will cause the eventual loss of all usable memory and a loss of all safety system functions. Dynamic memory allocation in digital safety systems should therefore be minimized.

If dynamic memory allocation is unavoidable, the source code should include provisions to ensure that:

•All dynamically allocated memory during a specific execution cycle is released at the end of that cycle, and

•The possibility of interruption of execution between the point where memory is dynamically allocated and when it is released is minimized (if not totally eliminated); there should also be provisions in the application code that will detect any situation where dynamically allocated memory has not been released and release such memory .

2.1.1.2 Minimizing Memory Paging and Swapping

Memory paging is the use of a part of a disk (or other form of secondary or bulk memory) to store infrequently used primary memory areas. When these memory areas are needed by a running program, the operating system causes them to be read from the disk and loaded back into the primary memory. Process swapping is the use of part of a disk (or other form of bulk memory) to store the memory image of an entire inactive process (including its data areas such as a stack space and heap space). When it is time for the process to be executed, the image is loaded from the disk back into the primary memory for use by the CPU. In any event, the specific usage of memory and the portion of storage used for swapping is indeterminate.

Both capabilities were developed for interactive and batch timesharing systems, where the demand for memory was greater than the amount installed in the computer system. However, they are inappropriate for safety systems because these indeterminacies in memory and storage utilization can, in turn, cause significant delays in response time and use complex interrupt-driven functions to handle the memory transfers. In addition, these capabilities depend on electromechanical components (if a disk is used as the secondary storage device) which are subject to failure.

If an operating system and hardware that support memory paging or process swapping are used in a safety system, this feature should be disabled at the operating system level. There should be enough primary memory for all data and programs. If there is any question that these features were not disabled, there should be provisions in the safety applications software ensuring that all critical functions and their data areas are in primary memory during the entire period of execution. Such

provisions in the source code include operating system calls ("pinning"), compiler directives, and operating system scripts.

2.1.2 Predictability of Control Flow

Control flow defines the order in which statements in a program are executed (i.e., sequential, branching, looping, or procedural) (Meek, 1993). A predictable control flow allows an unambiguous assessment of how the program will execute under specified conditions.

Related base attributes are:

- Maximizing structure
- Minimizing control flow complexity
- Initializing variables before use*
- Single entry and exit points for subprograms
- Minimizing interface ambiguities*
- Use of data typing*
- Accounting for precision and accuracy*
- Order of precedence of arithmetic, logical, and functional operators*
- Avoiding functions or procedures with side effects*
- Separating assignment from evaluation*
- Proper handling of program instrumentation
- Controlling class library size*
- Minimizing use of dynamic binding*
- Controlling operator overloading.*

These attributes and their relevance to safety are discussed in the following subsections. The attributes marked with an asterisk are treated as lower level attributes associated with the intermediate-level attribute, predictability of mathematical/logical result, in Chapter 4, the chapter dealing with the Ada95 language.

2.1.2.1 Maximizing Structure

"Spaghetti code" is a common derogatory reference to code with `GOTO` or equivalent execution control statements that cause an unstructured shift of execution from one branch of a program to another. The safety concern is that the execution time behavior is difficult to trace and understand. `GOTO` statements can cause undesirable side effects because they interrupt execution of a particular code segment without assurance that subsequent execution will satisfy all conditions that caused entry into that segment. Standards discouraging or prohibiting such coding practices have been in place for more than two decades (e.g., MIL-Std-1679). Structure is maximized by the elimination

of `GOTO` statements and use of appropriate block structured code. The `case`, `if..then..else`, `do until`, and `do while` constructs permit branching with a defined return and without introducing the uncertainty of control flow associated with `GOTO` or equivalent statements (Dijkstra, 1972; DoD-Std-2167A, Appendix C).

2.1.2.2 Minimizing Control Flow Complexity

An indication of control flow complexity is the number of nesting levels for branching or looping. Excessive complexity makes it difficult to predict the flow of a program and impedes review and maintenance. A specific safety concern is that the control flow may be unpredictable when unanticipated combinations of parameters are encountered. Excessive nesting can usually be avoided by the use of functions or subroutines in place of in-line branches. A specific limit on nesting, as part of project or organizational coding guidelines, can mitigate safety concerns.

2.1.2.3 Initializing Variables Before Use

When variables are not initialized to a known value at the beginning of an execution cycle, safety is impaired because they may contain "garbage" values (residue from the previous use of that memory area). Run-time predictability requires that memory storage areas set aside for process data be set to known values before being accessed (set and used). The specific result of using unknown initial values of a variable depends upon how that variable is used in the software. A compiler cannot be depended on to automatically reset memory areas set aside for variables (Gottfried, 1993; Naiditch, 1993).

2.1.2.4 Single Entry and Exit Points for Subprograms

Multiple entry and exit points in subprograms introduce control flow uncertainties similar to those caused by `GOTO` statements (DoD-Std-2167A, Appendix C). Run-time execution flow predictability is enhanced by having only a single entry to and exit from each program. Because predictability of execution flow is a characteristic important to safety, multiple entry and exit points in subroutines or functions are undesirable even if the language supports them.

2.1.2.5 Minimizing Interface Ambiguities

Interface faults include mismatches in argument lists when calling other subprograms, communicating with other tasks, passing messages among objects, or using operating system services. An example of such a fault is reversing the order of arguments when calling a subroutine. Previous research on software failures has shown that this category of faults is quite significant (Chillarege, 1992; Thayer, 1976). Recent failures in actual systems have also served to underscore

the need for reasonableness checks on input and output (Baber, 1997). Coding practices that can reduce or eliminate the probability of interface faults include:

◦Ordering arguments to alternate different data types (reducing the chance that two adjacent arguments will be placed in an incorrect order).

◦Using named notation rather than ordering or position notation for languages that support such notation, e.g., `display(value=>TC5, units=>EU)` rather than `display(TC5, EU)`.

•Testing for the validity of input arguments at the beginning of the subprogram.

•Routines should have adequate pre- and post-conditions specified. A pre-condition is an assurance that all local variables are initialized and all input variables meet appropriate reasonableness checks. A post-condition is an assurance that all output variables meet appropriate reasonableness checks. See also Section 2.1.2.3.

2.1.2.6 Use of Data Typing

Acceptance of data that differ from those intended to be used by a program can cause failures, and such failures that occur during an exception condition may have particularly adverse effects on safety (IEEE, 1993). This concern can be addressed by declaration of data types. Originally, the primary advantage of declaring data types was to allow compilers to reserve the correct amount of memory. However, data typing is useful for improved definition of interfaces (see above), increased legibility (for reviews), and compile time and run time checking. These originally ancillary uses have now become the primary motivators for data typing and have prompted the use of *strong typing* in which additional declarations, at least that of a valid range, are required. The safety issues associated with data typing include (IEEE, 1993; DoD-Std-2167A, Appendix C):

•Limiting the use of anonymous types (e.g., general integer or floating point without upper and lower limits) in strongly typed languages.

•Ensuring that the limits on data types are not excessively constrained so that spurious exceptions or error messages are not generated (this is an issue in strongly typed languages).

•Minimizing type conversions, and eliminating implicit or automated type conversions (e.g., in assignments and pointer operations).

◦Avoiding mixed-mode operations. If such operations are necessary, they should be clearly identified and described using prominent comments in the source code.

•Ensuring that expressions involving arithmetic evaluations or relational operations have a single data type—or the proper set of data types for which conversion difficulties are minimized.

•Limiting the use of indirection such as array indices, pointers (in Pascal or C), or access objects (in Ada) to situations where there are no other reasonable alternatives, and performing validation on indirectly addressed data prior to setting or use to ensure the correctness of the accessed locations. Strongly typed pointers, array indices, and access types reduce the possibility of referencing invalid locations.

2.1.2.7 Accounting for Precision and Accuracy

The software implementation must provide adequate precision and accuracy for the intended safety application (IEEE, 1993). Safety concerns are raised when the declared precision of floating point variables is not supported by analysis—particularly when small differences between large values are used (e.g., when computing rate of change from the difference between current and previous values, calculating variances, or performing filtering operations such as moving averages).

2.1.2.8 Order of Precedence of Arithmetic, Logical, and Functional Operators

The default order of precedence of arithmetic, logical, and other operations varies among languages. Developers or reviewers may make incorrect precedence assumptions when explicit parentheses are not used—particularly in complex expressions (DoD-Std-2167A, Appendix C). Therefore the use of parentheses or other mechanisms for ensuring a clear statement of the order of evaluation of operations should be used.

2.1.2.9 Avoiding Functions or Procedures with Side Effects

A side effect is a change to any variable not returned by that function that persists after the completion of the function. This includes changes to files, hardware registers, etc. An example of such a side effect would be a change in a global variable not in the function parameter list. Side effects can lead to problems with unplanned dependencies and can cause bugs that are very hard to find.

2.1.2.10 Separating Assignment from Evaluation

Assignment statements (e.g., `extern_var := 100`) should be separated from evaluation expressions (e.g., `if sensor_val < temp_limit`). The separation can be violated when subprograms are used as part of the evaluation. For example, a filtering function may be used as part of an evaluation rather than simply the sensor value:

```
if(func(a) < templimit).
```

Execution of **func(a)** may also set a global or external variable, using an assignment statement. For example:

```
func(t);
     /* data declarations */
begin
     . . .
          /* initialization, execution, or evaluation code */
     extern_var:=0;
          /*an external variable declared at a higher scope
            and used by this routine */
     . . .
end.
```

As a result, when the subprogram **func** is called, it will set an external variable to a value of 0. The value of this variable may be used by other programs in calculations, logical decisions, or output. Although this change may have been explicitly intended by the programmer, it is very difficult for others to follow. It is acceptable for the subprogram **func** to assign values to variables *providing that these variables are visible only within the subprogram,* i.e., they are local variables rather than global or external variables. A related attribute is minimization of the use of global variables discussed below.

2.1.2.11 Proper Handling of Program Instrumentation

Program instrumentation collects and outputs certain internal state values of a program during execution and allows the developer to check if particular aspects of the specification have been correctly implemented (Liao, 1991). Specific safety related issues are:

•*Minimizing Run-time Perturbations*: Instrumentation that interferes with the normal execution flow is undesirable in safety applications. For example, extensive "write" or other output statements can result in the execution of a significant amount of library code associated with outputting values; a less intrusive means may be to write such values to external memory locations where they can be processed later. It may also mean writing data in binary format for off-line format processing (i.e., conversion to human-readable text and numeric values). To minimize differences in behavior between test and normal operation, it may be desirable to keep certain instrumentation code in place in the actual environment.

•*Maintaining Visibility of Instrumentation in Runtime Source Code*: Some software tools alter compiler generated object (or executable) files in order to insert instrumentation (Campbell, 1994; Castellano, 1994). This is generally not acceptable in a safety system because the impact of such changes is not visible in the source code and its effect on execution cannot be reviewed.

•*Conforming to Software Instrumentation Guidelines*: Review is facilitated (and therefore safety is enhanced) if the instrumentation practices are described in project specific engineering notebooks. Guidelines are needed to identify what types of output mechanisms are to be used, and under which conditions they should not be used. For example, a measure mentioned above for minimizing runtime interference is at odds with the data abstraction and error containment attributes described later in this section.

2.1.2.12 Controlling Class Library Size

Control of class library size is important to avoid a system that becomes unmanageable or has large performance penalties because it has too many classes and objects (Cuthill, 1993). Safety is enhanced if project-specific guidelines limit the number of classes and objects... and the actual software conforms to these guidelines.

2.1.2.13 Minimizing Use of Dynamic Binding

Binding denotes the association of a name with a class. Dynamic binding permits the name/class association to be deferred until the object designated by the name is created at execution time. The unpredictability of the name/class association creates safety concerns. It also reduces the predictability of the runtime behavior of an object-oriented program and it complicates debugging, understanding, and tracing (Royce, 1993). Restrictions on, or elimination of, dynamic binding is desirable for safety-critical applications.

2.1.2.14 Controlling Operator Overloading

Polymorphism (operator overloading) can improve readability and reduce complexity by allowing a single subprogram or operator (in Ada) or object behavior (in C++ and Ada95) to be used for different data types. However, it can also be problematic from the perspective of predictability because it is unclear how a compiler will bind code for different polymorphisms (e.g., how would a multiply operation on a multidimensional array be bound to scalars or one-dimensional arrays) (Royce, 1993). Guidance on use of operator overloading in a project-specific or organizational coding standards manual is therefore desirable for safety-related applications, together with verification that the code complies with this standard.

2.1.3 Predictability of Timing

Predictability of timing is crucial in a safety system used in real time control (Kopetz, 1993; Leveson, 1992; Turner, 1992). For example, a reactor shutdown system must generate a trip signal

within a specified interval of operating parameters falling outside of allowable ranges. Also, diesel engine startup sequences require events to happen within a defined time interval. Base attributes related to object oriented programming that have relevance to this intermediate attribute were discussed under previous headings:

- Controlling class library size
- Minimizing use of dynamic binding, and
- Controlling operator overloading.

Two additional base attributes related to timing discussed in the following subsections are:

- Minimizing the use of tasking, and
- Minimizing the use of interrupt driven processing.

2.1.3.1 Minimizing the Use of Tasking

Although tasking (in languages such as Ada) provides an attractive model for concurrent processing, its use is undesirable in safety-critical applications for the following reasons:

•There are timing uncertainties associated with differing implementations by compiler vendors, interactions with underlying operating systems (or real time kernels), and the design of the hardware platform.

•The sequence of execution is uncertain when several calling alternatives are waiting to be executed because it is not always clear which call will be selected (Gottfried, 1993; Naiditch, 1993).

•Tasking allows time critical errors, such as race conditions and deadlocks, to develop. Such differences are difficult to debug (Royce, 1993).

Therefore, tasking is to be avoided in safety systems unless there is a compelling justification.

2.1.3.2 Minimizing the Use of Interrupt Driven Processing

Using interrupt driven processing to handle the acceptance and processing of plant and operator input can reduce average response time, but usually leads to non-deterministic maximum response times. Interrupt driven processing was implicated in at least one of the Therac-25 accidents (Leveson, 1992; Turner, 1992). Reference documents and standards related to digital system safety generally discourage or prohibit its use (IEC 880). Avoiding interrupt driven processing facilitates analysis of synchronization and run-time behavior, and avoids the non-determinism of response time inherent in interrupt driven processing.

2.1.4 Predictability of Mathematical or Logical Result

Predictable mathematical or logical result means that the results realized at the completion of execution of the low level of code (source line or construct) being examined are the results intended and expected by the programmer who wrote the code. The term "logical" is intended to extend the term "results" to the case where the code is manipulating Boolean data and will yield a Boolean result.

As mentioned previously in this chapter, this intermediate level attribute is being only applied to object-oriented Ada95 at this time. Accordingly, there is no need for separate generic guidelines for the lower level attributes assigned to this intermediate level-attribute other than the guidelines presented in this chapter for procedure-oriented languages.

2.2 Robustness

Robustness refers to the capability of the software to continue execution during off-nominal or other unanticipated conditions. A synonym for robustness is survivability (Bowen, 1985; Wigle, 1985). Robustness is an important attribute for a safety system because unanticipated events can happen during an accident or excursion, and the capability of the software to continue monitoring and controlling a system in such circumstances is vital.

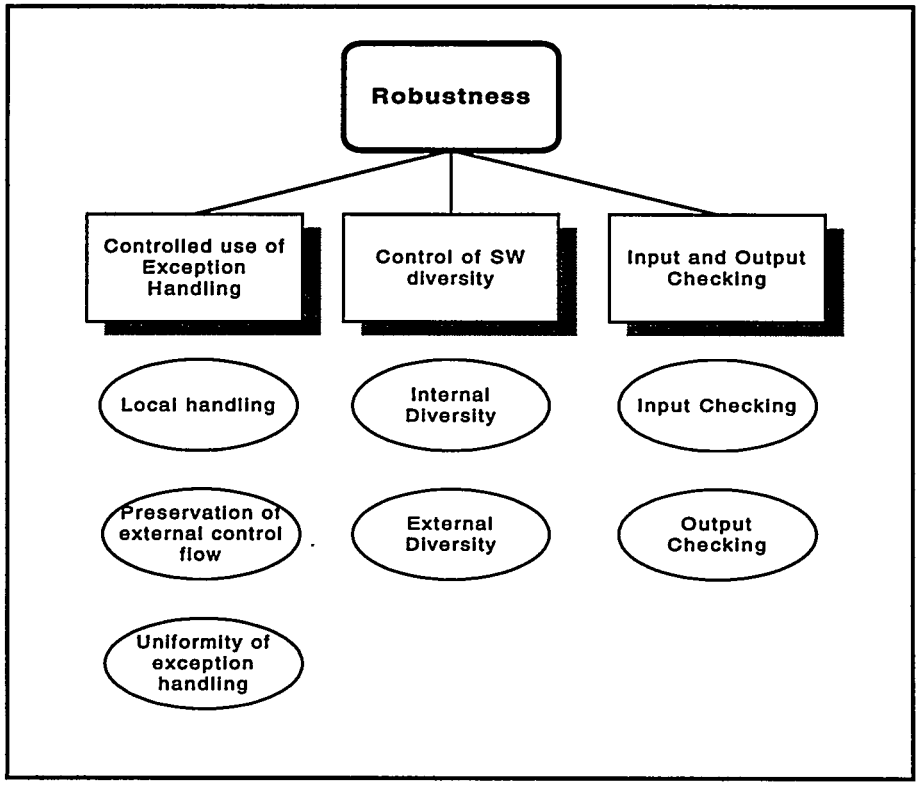


Figure 2-4 Robustness and Lower Level Attributes

As shown in Figure 2-4, the intermediate attributes for robustness are:

•Controlling use of diversity
•Controlling use of exception handling
•Checking input and output.

These attributes and their relevance to safety are discussed in the following subsections.

2.2.1 Controlling Use of Diversity

The decision to employ diverse software implementations is a design-level function and is therefore outside the scope of this document. However, if diversity is called for in the design or requirements, it should be controlled in its application. The principal issue with the use of diversity in software is the possibility that common-mode software failures may cause redundant safety systems to fail in such a way that there is a loss of safety function (Committee, 1995).

The possibility of common-mode failures between independently developed software routines is not easily eliminated. Any shared specification can lead to common-mode failures. The same problem exists in developing test data to check the software -- the testers may omit the same off-nominal or unusual cases that the developers overlooked. Further, in order to use the approach where the outputs of multiple versions of software can be compared in real time (or to be able to compare intermediate results), the designs from independent teams may be overspecified. Such detailed common specifications may result in little software design diversity.

In addition, multiple versions of the software written independently from the same requirements specification are effective only against coding errors (and sometimes only a limited set of these). On the other hand, empirical evidence suggests that most safety problems (and most errors found in operational software) stem from errors in the software requirements, especially misunderstandings about the required operation of the software. Software intended to provide redundancy may not achieve this goal but may simply duplicate the misunderstandings (Committee, 1995). In reviewing safety-critical software, analysis of software diversity to determine common-mode failures is important.

There are two base attributes:

•Controlling internal diversity
•Controlling external diversity.

2.2.1.1 Controlling Internal Diversity

When only internal diversity is used, the interfaces to all versions must be identical. In other words, any sensor data or parameters from calling procedures should be passed identically to all versions, and output data from any version should be accepted and used by other parts of the system. However, internal operations and storage of local data should occur diversely in the multiple module versions or instantiations. Internal diversity is facilitated by an object-oriented approach in which the same messages and methods are used, but the internal algorithms and data representations differ (Cuthill, 1993). Internal diversity should be implemented in accordance with the design and with project-specific guidelines. These should address:

•*Diverse algorithms.* Using different algorithms, unit conversions, and process parameters (when called for or allowed in the requirements or design) minimizes the possibility of a design or implementation-related failure.

•*Diverse data validation.* Using alternate schemes for sensor (or other input) data and output data validation minimizes the possibility of a design or implementation-related failure.

•*Diverse exception handling routines.* This measure reduces the probability that an error in the exception handling or processing will occur simultaneously on multiple versions.

•*Different data types, structures, and storage allocation.* This measure reduces the possibility that unanticipated interactions between the object code generated by the compiler and the operating system will cause data or code to be inadvertently overwritten simultaneously on multiple versions.

•*Diverse libraries and subroutines.* Avoiding the use of the same application software subroutines, compiler-supplied library routines, and operating system provided application programming interfaces. This measure reduces the possibility of a simultaneous failure due to a defect in such routines.

•*Diverse order of arithmetic operation.* Changing the order of arithmetic operations in conversions, arithmetic, and assignment statements by using commutative, associative, and distributive properties reduces the possibility of simultaneous failures due to unanticipated overflow conditions generated by intermediate results or problems in numerical precision.

•*Diverse order of input and output operation.* Performing I/O operations in different orders reduces the possibility of simultaneous timing-related failures (such as a deadlock) or data-driven failures (i.e., a program crash due to a particular data value).

2.2.1.2 Controlling External Diversity

Where external diversity is used, safety is enhanced if it is implemented in a disciplined manner in accordance with design documents. The design documents should reflect the diversity imposed by requirements, hazard analyses, and similar sources. External diversity is achieved by using different interfaces among the versions, and may be combined with internal diversity. External diversity is necessary when different languages are used for different versions, and may also be used to obtain sensor data through a different channel. Uncontrolled or unspecified external diversity can lead to a proliferation of interfaces which impact safety due to difficult maintenance, testing, verification, and validation.

2.2.2 Controlling Use of Exception Handling

Exception handling deals with abnormal system states and input data (IEEE, 1993). Exception handling provisions in some languages facilitate the establishment of an alternate execution path in the event of conditions which, although unexpected, result in states that can be defined in advance. Problems can arise in the use of exception raising and handling, however, because execution flow during exception conditions is often difficult to trace.

Base attributes with respect to exception handling include (DoD-Std-2167A, Appendix, D):

- Handling of exceptions locally
- Preserving external control flow
- Handling of exceptions uniformly.

2.2.2.1 Handling of Exceptions Locally

Propagation of exceptions through several levels of a program can cause the precise nature of the exception to be misinterpreted at the place where the exception handling is implemented. This cause of system failure (with potentially serious safety implications) is avoided if exceptions are handled locally.

2.2.2.1 Preserving External Control Flow

Interruption of control flow external to the routine in which the exception was raised creates uncertainty in the execution subsequent to the exception handling. Safety is enhanced by preservation of control flow external to the module responsible for the exception.

2.2.2.2 Handling of Exceptions Uniformly

Undisciplined use of exception handling can result in inconsistent processing of the same exception condition in different parts of the code. At worst, it can result in some exceptions being raised and

not handled. These problems can be avoided by guidance on the use of exceptions as part of the coding practices procedures of the organization or the specific project. Topics to be included in this guidance are:

•General and project specific exceptions which have been defined and are allowed
•Placement of exception handling code
•Enumerating all intended side effects and verifying that there are no other side effects
•Ensuring the integrity of critical state data during exception processing
•Criteria for distinguishing what conditions should be handled through control flow constructs as part of normal processing versus abnormal conditions where use of exception handling is appropriate.

2.2.3 Checking Input and Output

Data corruption due to a transient failure or an invalid result can have serious consequences on subsequent processing if allowed to propagate. The base attributes related to input and output checking mitigate such consequences by containing the error. The two base attributes discussed in the following subsections are:

•Input data checking and
•Output data checking.

2.2.3.1 Input Data Checking

Input data includes data from another routine, data from the external environment, and data stored in memory from a previous iteration. Input data should be checked for validity before processing. Such checks reduce the probability of incorrect results or corrupted data being propagated. At a minimum, the values of the inputs should be checked for data type and being within an acceptable range. If possible, reasonableness checks on the data should also be performed. Provisions should exist in the safety system software to detect invalid input and to bring the module to a known state (i,e., default or previously valid values) as defined in the higher-level design.

2.2.3.2 Output Data Checking

Output data—whether to the external environment, to another routine or stored for use in a subsequent iteration—should be checked for validity. At a minimum, this validity check should ensure that the values are of the appropriate data type and are within acceptable ranges. It is more desirable that the values also be checked for reasonableness. However, such reasonableness checks should not be so restrictive that they spuriously reject correct values. Provisions for handling rejected output values according to the design should also be present in the software.

2.3 Traceability

As defined earlier in this chapter, traceability refers to attributes of safety software which support verification of correctness and completeness compared with the software design. As shown in Figure 2-5, the intermediate attributes for traceability are:

- Readability
- Controlling use of built-in functions
- Controlling use of compiled libraries.

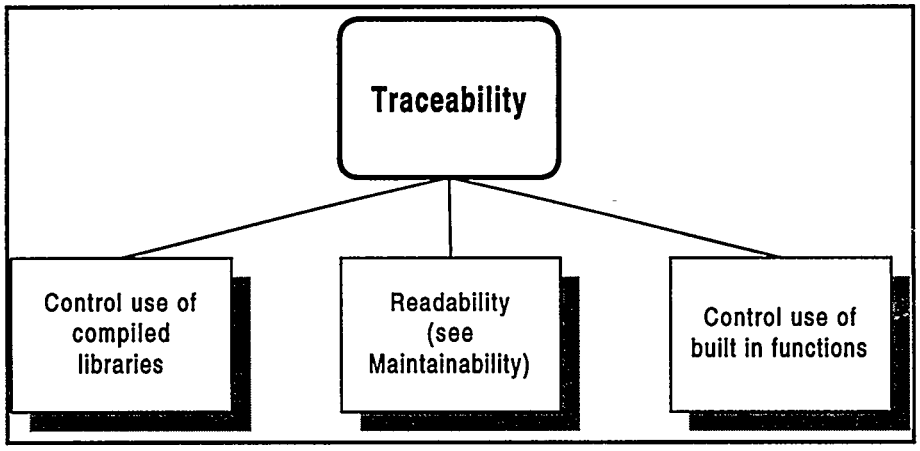


Figure 2-5 Traceability and Lower Level Attributes

Since readability is also an intermediate attribute of maintainability, it is discussed in Section 2.4. The latter two attributes and their relevance to safety are discussed in the following section.

2.3.1 Controlling Use of Built-In Functions

Nearly all languages include built-in functions for frequently used programming tasks to maximize programmer productivity. However, the limitations of these functions and the way in which they handle exceptions may not be as well known as those of the basic language constructs. Thus, the use of such functions raises safety concerns.

Concerns over the use of built-in functions can be addressed through organizational or project specific guidelines. Regression test cases make it possible to establish conformance with expected results of new releases of compilers and runtime libraries. Thus, test cases, procedures, and results of previous testing for allowable built-in functions should be retained. Testing should also assess behavior for out-of-bounds and marginal conditions (e.g., negative arguments on a square root routine; improperly terminated strings for a string copy routine, etc.) in the specific runtime environment.

2.3.2 Controlling Use of Compiled Libraries

Compiled libraries are routines written and compiled by an entity other than the development group. Applications of compiled libraries include input/output operations, device drivers, or mathematical operations that are not defined in the standard language. Such libraries can be supplied by compiler vendors, third parties, or other departments of the development organization. Concerns for such libraries are similar to those for built-in functions.

Concerns over the use of compiled libraries can be addressed by controlling the use of function calls to such libraries through organizational or project-specific guidelines. Like built-in functions, a set of test cases, procedures, and results should be maintained. The test cases should assess behavior for normal, out-of-bounds, and marginal conditions in the specific runtime environment. Regression testing should be performed for each new release of the compiled library.

2.4 Maintainability

Software maintainability reduces the likelihood that errors will be introduced while making changes. The intermediate attributes related to maintainability that affect safety include:

•*Readability*: those attributes of the software that facilitate the understanding of the software by project personnel

•*Data abstraction:* the extent to which the code is partitioned and modularized so that the collateral impact and probability of unintended side effects due to software changes are minimized

•*Functional cohesiveness:* the appropriate allocation of design level functions to software elements in the code (one procedure; one function)

•*Malleability:* the extent to which areas of potential change are isolated from the rest of the code

•*Portability:* the major safety impact of which is the avoidance of non-standard functions of a language.

Figure 2-6 shows these lower level and associated base attributes, which are discussed further in the following subsections.

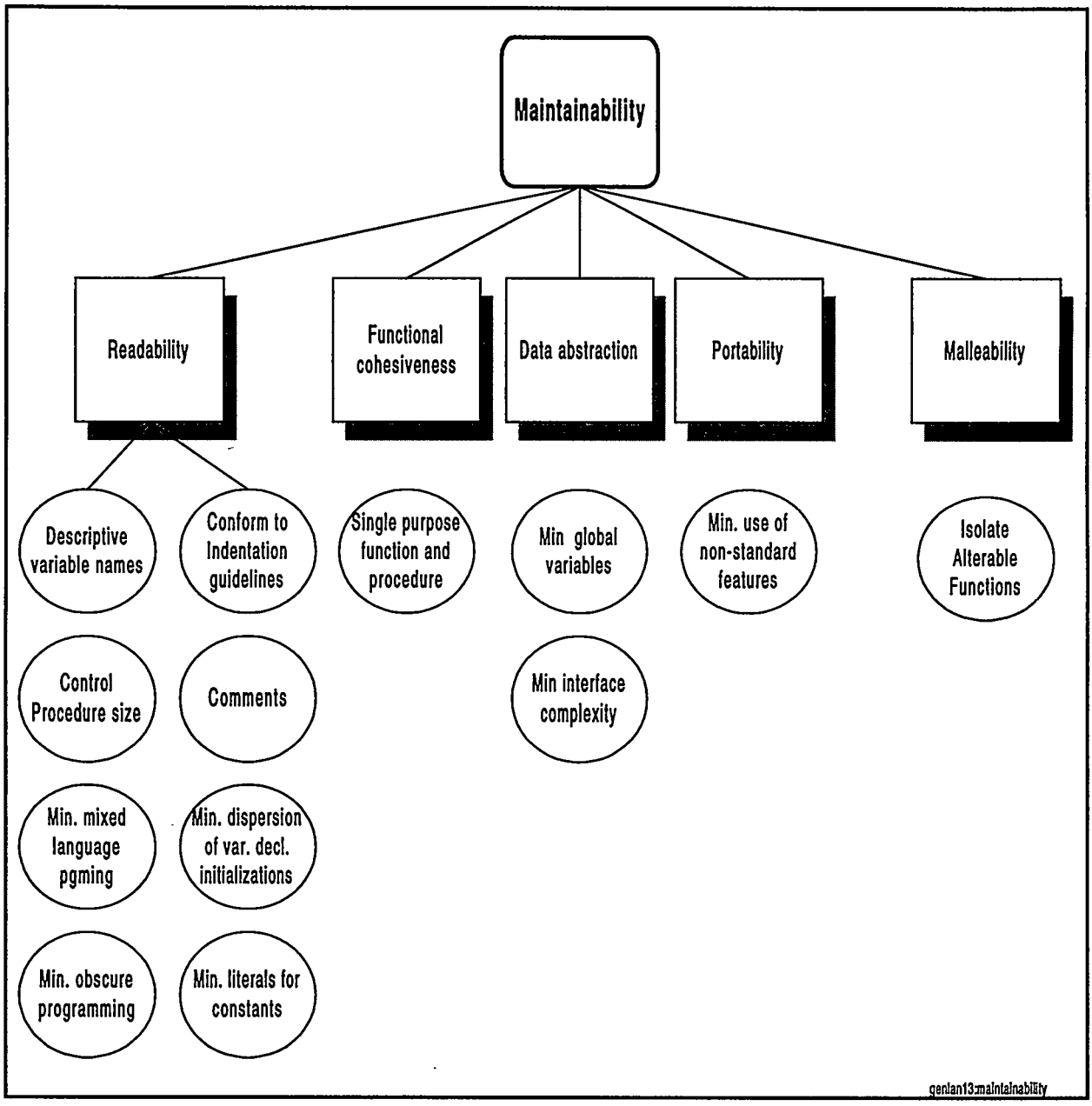


Figure 2-6 Maintainability and Lower Level Attributes

2.4.1 Readability

Readability allows software to be understood by qualified development personnel other than the writer. The importance of readability for maintainability can be seen by a study performed at the NASA Goddard Software Engineering Laboratory (McGarry, 1992), in which manual code reading (desk checking) was found to be more effective than structural or functional testing for finding coding faults. It is reasonable to extrapolate that readability would also enhance identifying code

to be changed during corrective or adaptive maintenance and would reduce the probability of introducing new faults during such maintenance.

There are no general standards for readability that can be mandated or even recommended. However, organizational or project-specific coding style and practices manuals (or related guidelines) are expected for safety-critical systems. The following base attributes are related to readability:

•Conforming to indentation guidelines
•Using descriptive identifier names
•Commenting and internal documentation
•Limiting subprogram size
•Minimizing mixed language programming
•Minimizing obscure or subtle programming constructs
•Minimizing dispersion of related elements
•Minimizing use of literals.

These attributes are discussed in the following subsections.

2.4.1.1 Conforming to Indentation Guidelines

Appropriate indentation facilitates the identification of declarations, control flows, non-executable comments, and other components of source code (DoD-Std-2167A, Appendix C). Indentation guidelines are generally part of a project specific or organizational programming style or standards. Significant issues to be addressed by indentation practices are the handling of:

•Programming blocks (sequences of statements bounded by **`begin`** and **`end`**)
•Comments
•Branching constructs (e.g., **`if...then...else`**, **`case`** statements, loops, etc.)
•Multiple levels of nesting (e.g., a **`do`** loop within a **`do`** loop)
•Variable and subroutine declarations
•Compiler directives
•Exception raising and handling.

2.4.1.2 Using Descriptive Identifier Names

Names for variables, procedures, functions, data types, constants, exceptions, objects, methods, labels, and other identifiers that are not easily understood can impede review and maintenance. Safety concerns arising from naming practices can be alleviated when names are required to be descriptive, consistent, and traceable to higher-level (i.e., software design) documents (DoD-Std-2167A, Appendix C). Naming conventions are an important part of the coding style and practices manual. Examples of issues to be addressed include:

•Identification of plant input data (e.g., should the variable refer to a sensor, or should it be called `loop1_hot_leg_TC1`)
•How looping variables should be named (e.g., `i,j,k` or longer titles)
•Local renaming of identifiers (e.g., `average_procedure.mean` renamed as `mean`)
•Distinguishing between different categories of identifiers (e.g., a suffix on all data types with an `_T` to distinguish them from variables)
•Lists of project-specific terminology and reserved words (e.g., restrictions on the use of the terms "alarm", "limit", etc.).

Use of the same name for a different purpose is to be avoided unless obviously advantageous and, when employed, should be accompanied by clear, consistent, and unambiguous notations. Multiple use of the same name can be confusing. A further problem can occur if the language supports precompiled units (such as Ada). A variable with the same name in two different packages, one of which is used by the other, may be interpreted by the compiler in a different manner than intended by the program writer. In some cases, the programmer may have omitted the declaration of a name in a package. Thus, another package can cause a different variable with the same name to be used in a totally unintended manner (Campbell, 1994; Castellano, 1994). If the particular branch or execution path is not encountered frequently, it is possible that such a fault would not be discovered until it causes a run-time failure.

Use of reserved words for user-selected identifiers (in languages where this feature is allowed) is undesirable (DoD-Std-2167A, Appendix C).

2.4.1.3 Commenting and Internal Documentation

Incomplete comments, inconsistent formats, and comments that are not updated to reflect the current code impede review and raise safety concerns. These problems can be minimized by guidance in the organizational or project coding standards that controls comments and internal (to the program) documentation. Examples of items, when incorporated, that should be located in the prologue section include the following (DoD-Std-2167A, Appendix C):

•The subprogram or unit purpose and how achieved
•Functions and performance requirements, and external interfaces that the subprogram or unit helps implement
•Other subprograms or units called and their dependencies
•Use of global and local variables and, if applicable, memory and register locations together with special maintenance instructions
•The responsible programming department or section
•Date of creation of the unit
•Date of latest revision, revision number, problem report number, and title associated with the revision
•Intended failure behavior and related information for all major segments of the code

•Inputs and outputs, including data files referenced during unit entry of execution
•Comments on the purpose, scope, and limitations on each argument (for subprograms with arguments).

Similar examples for documentation within the code include:

•Reference to higher level design documentation in comments associated with data type, variable, and constant declarations
•Purpose and expected results at the beginning of branches and programming blocks
•Detailed in-line comments explaining unusual constructs and deviations from programming practices.

2.4.1.4 Limiting Subprogram Size

Some documents recommend specific limits on the source code of each subprogram or unit. For example, an average of 100 non-expandable statements and a maximum of more than 200 such statements has been recommended (DoD-Std-2167A, Appendix C). Concern with the size of subprograms was one of the motivators for the adoption of structured programming. In Dijkstra's words, "Widespread under-estimation of the specific difficulties of size seems to be one of the major underlying causes of software failure" (Dahl, 1972; Dijkstra, 1972). Small subprograms (one or two pages) are easier to review than longer ones. However, the limits on allowable size must also take into account the nature of the program and the language. In nuclear safety and control systems, a given code must frequently handle a multitude of sensed quantities, and the data declarations (with required comments) for these can by themselves amount to more than a page. The criterion for this base attribute is therefore that guidance on size be provided, rather than a universal numerical threshold.

2.4.1.5 Minimizing Mixed Language Programming

Mixed language programming (e.g., assembly language for interrupt handling and high-level languages for other processing) presents difficulties for reviewers and maintainers and is therefore a safety concern. When this practice cannot be avoided, the difficulties can be minimized by placing the "foreign" language code adjacent to the dominant language routine with which it interfaces (e.g., an in-line assembly compiler directive in the input processing routine associated with an interrupt) so that readability is enhanced.

2.4.1.6 Minimizing Obscure or Subtle Programming Constructs

Obscure coding constructs can generally be characterized as the use of indirect techniques to decrease the amount of coding or CPU processing required to achieve a result. Such coding practices

present problems in review and maintenance and hence are a safety concern. For example shifting an integer to the left is equivalent to doubling its value. However, the former construct would be obscure if the design calls for doubling the value (i.e., it would be preferable to perform the multiplication); the latter construct would be obscure if the design calls for shifting the value to the left (i.e., it would be preferable to perform the shifting operation in the source code rather than multiplying by 2). Appropriate commenting can minimize the impact of obscure or marginally obscure coding changes (e.g., adding the value to itself as a means of doubling it).

2.4.1.7 Minimizing Dispersion of Related Elements

If related elements of the code are dispersed in a program, it is necessary to refer to multiple locations within a source listing during reviews and maintenance. However, the specific nature of the dispersion varies by language. For example, some languages allow for interface specifications separated from the body of the code; others allow for "prototyping" for a similar purpose. In languages with strong data typing, it may be desirable to centralize all type declarations in a single file (or set of files); in object-oriented languages, it may be desirable to segregate base classes from derived classes. Review is facilitated and safety is enhanced if project-specific guidance is provided on the placement of related elements in the code.

2.4.1.8 Minimizing Use of Literals

Literals (i.e., an actual number or string in the source code) are more difficult to identify than names to which a constant value is assigned at the beginning of the module (DoD-Std-2167A, Appendix C). Literals impact safety because they decrease readability and complicate maintainability — particularly if the literal is associated with a process parameter which may be tuned or a conversion factor which may be changed upon recalibration of an instrument. It is far easier to change one value set at the beginning of a file than it is to guarantee that all literals associated with such a parameter have been changed completely and correctly throughout all relevant files.

2.4.2 Data Abstraction

Data abstraction is the combination of data and allowable operations on that data into a single entity, and establishment of an interface which allows access, manipulation and storage of the data only through the allowable operations. It is an important contributor to safety by virtue of reducing or eliminating potential side effects of changing variables either during runtime or in software maintenance activities (Parnas, 1972). This principle is associated with the following specific base attributes:

- Minimizing the use of global variables
- Minimizing the complexity of the interface defining allowable operations.

These attributes are discussed further in the following subsections.

2.4.2.1 Minimizing the Use of Global Variables

It is desirable to limit the use of global variables in safety related programs (Parnas, 1990; van Schouwen, 1990; Kwan, 1990) because of the potential for unintended side effects. Readability is enhanced if variables are set and used in the same routine. These variables can be made available to other routines through established and controlled interfaces which minimize the possibility of unintended interactions. For the same reasons, dependencies among internal stored data of different routines need to be avoided or controlled.

To avoid potential safety concerns, local variables within different programs should not share the same storage locations (DoD-Std-2167A, Appendix C).

2.4.2.2 Minimizing the Complexity of Interfaces

Interfaces are a frequent cause of software failures (Thayer, 1976). Complex interfaces are difficult to review and maintain and are therefore not desirable in safety related programs. Characteristics that contribute to complexity include:

•Large numbers of arguments used in calling routines
•Use of terse expressions when different modes or options are used (e.g., `arraymult(a,b,2)` instead of `arraymult(a,b,crossproduct))`
•Lack of easily understood restrictions and limitations on the use of allowable operations.

2.4.3 Functional Cohesiveness

Functional cohesiveness refers to a clear correspondence between the functions of a program and the structure of its components. Functional cohesiveness has a single base attribute.

2.4.3.1 Single Purpose Function and Procedures

Review and maintenance are facilitated when every given procedure, subprogram, or function implements only one task or purpose specified in the software design. Subprograms, functions, or procedures that perform multiple tasks should be separated and written as separate functions. A simple way to test if a function is a single purpose function is to check to determine if the function can be summarized by a sentence in the following form (Parnas, 1990):

"verb + object(s)"

If multiple purposes or tasks specified in the design must be grouped into a single subprogram, function, or procedure, then justification of the grouping should be documented.

2.4.3.2 Single Purpose Variables

The principle of single purpose functions should be applied to variables. A variable should be used for a single purpose only (Plum, 1991).

2.4.4 Malleability

Malleability is the ability of a software system to accommodate changes in functional requirements (Parnas, 1990; van Schouwen, 1990; Kwan, 1990). Malleability extends data abstraction with the motivation toward isolating areas of potential change. To implement a malleable software system, it is necessary to identify what is expected to be constant and what is expected to be changed, and to isolate what is expected to be changed into easily identifiable areas that can be altered with a minimum of collateral changes. Malleability has a single base attribute.

2.4.4.1 Isolation of Alterable Functions

Review and maintenance are facilitated when functions that can be altered are isolated, so that changes in these do not affect other code or data. In many cases, such functions are hardware-related functions that need to be changed when the platform changes, the system changes, or when new devices are used to replace old devices.

For example, when a new display device is used to replace an old display device, graphics-display-related functions may need to be modified. Thus, the functions associated with the graphics controller should be grouped together in the same file, kept in close physical proximity, and organized in a manner which minimizes changes to other modules.

To a large extent, the isolation of alterable functions is a design issue related to data abstraction. As such, a detailed discussion is beyond the scope of this document.

2.4.5 Portability

From the perspective of safety, the benefits of portability are the adherence to standard programming constructs that yield predictable and consistent results across different operating platforms (Witt, 1994; Baker, 1994; Merrit, 1994). Thus, code which is reused or converted to run on a different platform will be easier to maintain. Attributes related to portability which have been discussed elsewhere include:

- Minimizing the use of built-in functions
- Minimizing the use of compiled libraries
- Minimizing dynamic binding

•Minimizing tasking
•Minimizing asynchronous constructs (interrupts).

The single base attribute related to portability is avoiding use of non-standard, or "enhanced" constructs specific to a particular compiler or a compiler in combination with the execution platform (Smith, 1989; Wood, 1989).

2.4.5.1 Isolation of Non-Standard Constructs

Where non-standard constructs are necessary, they should be clearly identified together with the rationale, limitations, and version dependencies.

References

Andersen, O. and P.G. Petersen, *Standards and regulations for software approval and certification,* Elektronik Centralen Report ECR 154 (Denmark), 1984.

Baber, R.L., "The Ariane 5 Explosion as Seen by a Software Engineer", World Wide Web page *www.cs.wits.ac.za/~bob/ariane5.htm,* February, 1997.

Bowen, T.P. and G.B. Wigle and J.T. Tsai, "Specification of Software Quality Attributes" Report, 3 Vols. RADC-TR-85-37, available from NTIS, 1985.

Bullock, J.B., briefing charts contained in Working Group Report on Software Reliability Verification and Validation, *IEEE/NRC Working Conference on Advanced Electrotechnology Applications to Nuclear Power Plants,* IEEE Cat. No.TH0073-7, January, 1980.

Campbell, D. and V. Castellano and O. Cole, et. al., *Ada/6000 Tool Set,* O.C. Systems, Fairfax, VA, 1994.

Chillarege, R., "Orthogonal Defect Classification," *IEEE Trans. SW Engineering*, November, 1991.

Committee on Application of Digital Instrumentation and Control Systems to Nuclear Power Plant Operations and Safety, "Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues" Report, National Academy Press, 1995.

Cuthill, B., "Applicability of Object Oriented Design Methods and C++ to Safety Critical Systems," *Proceedings of the Digital System Reliability and Nuclear Safety Workshop,* NUREG CP-0136, NIST SP 500-216, 1993.

Dahl, O.J. and E.W. Dijkstra and C.A.R. Hoare, *Structured Programming,* Academic Press, London and New York, 1972.

Gottfried, R. and D. Naiditch, *Using Ada in Trusted Systems, Proc. of COMPASS 93,* May, 1993, National Institute of Standards and Technology, Washington, DC, 1993.

Henderson, J., "Low level programming," in *Software Engineer's Reference Book,* J.D. McDermid, ed., CRC Press, Inc., Cleveland, OH, 1993.

Institute of Electrical and Electronic Engineering, IEEE Std 100-1977, *IEEE Standard Dictionary of Electrical and Electronic Terms.*

Institute of Electrical and Electronic Engineers, Nuclear Power Engineering Committee, IEEE Std. 603-1991, *IEEE Standard for Nuclear Power Generating Stations.*

Institute of Electrical and Electronic Engineers, IEEE-Std-7 -4.3.2-1993, *IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Station.*

International Electrotechnical Commission (IEC), "Software for Computers in the Safety Systems of Nuclear Power Stations," Standard 880, 1986.

Kopetz, H., "Real-time systems," in *Software Engineer's Reference Book,* J.D. McDermid, ed., CRC Press, Inc., Cleveland, OH, 1993.

Leveson, N.G. and C.S. Turner, *An Investigation of the Therac-25 Accidents,* University of California, Irvine Technical Report 92-108, Irvine, CA, 1992.

Liao, Y., "Requirements for Directed Automatic Instrumentation Generation for Program Monitoring and Measuring," in *IEEE Trans. SW Engineering*, 1991.

McGarry, F., "The Impacts of Software Engineering," briefing presented to the NRC Advisory Committee on Reactor Safeguards (ACRS), August 21, 1992.

Meek, B.L., "Early High-Level languages," in *Software Engineer's Reference Book,* J.D. McDermid, ed., CRC Press, Inc., 1993.

Murine, G.E., "Rome Laboratory Framework Implementation Guidebook", RL-TR-94-149, USAF Rome Laboratory, March 1994.

Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM,* Vol. 15, No. 12, 1972.

Parnas, D.L. and A.J. van Schouwen and S.P. Kwan, "Evaluation of Safety Critical Software," *Communications of the ACM,* Vol. 33, No. 6, p. 636, June, 1990.

Royce, W.W., written comments in *Proceedings of the Digital Systems Reliability and Nuclear Safety Workshop,* NUREG/CP-0136, NIST SP 500-216, 1993.

Smith, D.J. and K.B. Wood, *Engineering Quality Software: A review of Current Practices, Standards, and Guidelines Including New Methods and Development Tools.* New York: Elsevier Applied Sciences, 1989.

Thayer, R., "Software Reliability Study," Rome Air Development Center report RADC TR 76-238, March, 1976.

U.S. Department of Defense, "Weapon System Software Development," MIL-Std-1679 (Navy), 1978.

U.S. Department of Defense, DoD-Std-2167A, *Software Development Standard,* Appendix C, 1986.

U.S. Department of Defense, DoD Std 2167A, *Software Development Standard,* Appendix D, 1986.

Witt, B.I. and F.T. Baker and W.W. Merritt, *Software Architecture and Design.* Van Nostrand Reinhold, New York, 1994.

3 Ada 83

This chapter discusses Ada83 (DoD-Std-1815A)-specific guidelines. Ada 95 is discussed in a later chapter[2]. Section 3.1 identifies reliability-related attributes; Section 3.2 discusses robustness-related attributes; Section 3.3 discusses traceability-related attributes; and Section 3.4 describes maintainability-related attributes. A summary matrix is contained in Appendix B, together with language-specific weighting factors. These factors were influenced by Ada's strong typing and exception handling capabilities.

3.1 Reliability

The intermediate attributes of reliability related to Ada are as follows:

- Predictability of memory utilization
- Predictability of control flow
- Predictability of timing.

Ada-specific guidelines are described in the following subsections.

3.1.1 Predictability of Memory Utilization

Base-level attributes related to the predictability of memory utilization in Ada are as follows:

- Minimizing dynamic memory utilization
- Minimizing memory paging and swapping.

Specific guidelines for these attributes are discussed in the following subsections.

[2]Ada 95 differs with Ada 83 in several major areas, making Ada 95 potentially more suitable over the long term for developing safety-critical systems. The most important improvements are (a) providing object-oriented features, (b) new features for more responsive task communication such as protected types for emulating the monitor structure, and (c) hierarchical library structuring.

3.1.1.1 Avoiding Dynamic Memory Utilization

The generic[3] guidelines apply to Ada. Dynamic memory allocation should be avoided. Errors resulting from dynamic memory allocation can include (SPC, 1989, pp 76, 112 - 113):

1. Memory leaks that can cause the software to run out of memory. This problem is likely to occur in Ada since an access object (pointer) ceases to exist when its scope is exited, but the allocated memory it points to remains allocated.

2. Corruption of data due to multiple pointers to the same areas. Such corruption can be difficult to impossible to correct or even detect. This error condition can lead to the system crashing, frequently due to an exception being raised at a point distant from where the data were corrupted. This makes tracing the cause of the crash difficult.

The following are Ada-specific guidelines related to memory allocation. The final four guidelines are mitigation approaches and are relevant if dynamic memory allocation is determined to be unavoidable by the system designers.

- *Avoid explicit dynamic memory allocation.* The Ada primitive **new** causes memory to be allocated during execution. The following Ada code is an example of the use of dynamic memory for a linked list:

```
type Cell;
type Link is access Cell;
type Cell is
      record
            Value: Element;
            Next : Link;
      end record;

L: Link := null;                          -- initialization unnecessary
L:= new Cell;                             -- allocation of memory
```

- *Avoid dynamically created tasks.* Tasks should be elaborated only at system initialization. Dynamically created tasks also cause dynamic memory allocation in Ada. The dynamic memory utilization problem is aggravated in this case because the generic subprogram the programmer can utilize to deallocate objects in memory, Unchecked_Deallocation, does not apply to tasks or to objects that have tasks as components. This issue of dynamic tasks is discussed further in section 3.

---

[3]It should be noted that "generic guidelines" refers to the non-language specific guidelines of Chapter 2, not to the Ada construct.

- *Avoid recursion.* Recursion also uses dynamic memory space. Therefore, recursive procedures or functions should not be used. Recursion depth can be large, even infinite if the terminating condition does not occur. An unanticipated large number of recursive calls can use up available memory (SPC, 1989; Hutcheon, 1992). Recursion can frequently be recognized by having a subprogram call within a subprogram of the same name, as seen in the following example.

```
procedure RECURS_EXAMPLE(arg1: in type1 arg2: in type2) is
      arg1a:  type1;
      arg2a:  type2;
begin
      sequence of statements
      RECURS_EXAMPLE(arg1a=>arg1 arg2a => arg2);
      more statements
end RECURS_EXAMPLE;
```

Mutual recursion involving two or more subprograms can also occur. Depending on the arrangement and physical location of the source code for these subprograms, mutual recursion can be difficult to detect from source code. For example:

```
            procedure P(....)  is
            begin
                  .....
                  Q(....);
                  .....
             end P;

and

            procedure Q(....) is
            begin
                  .....
                  P(....);
                  .....
            end Q;
```

- *Do not instantiate generic units during runtime.* If generic units are used, they should be instantiated only during initialization (Jones, 1988). However, as will be described in the section on traceability (section 3.3.3), generic units are not desirable in safety significant software.

- *Minimize use of local large composite objects.* A memory allocation problem on the stack can occur if large composite objects are declared as local objects of a subprogram. Avoid the

use of dynamic arrays as in **`P(array(<>) of ...)`**.

- *Minimize use of unconstrained types.* Unconstrained types such as record types with unconstrained dynamic bound, and string types must be used with caution because of the impact on memory allocation.

- *Use **`length`** clauses if dynamic memory allocation is necessary.* If dynamic memory allocation is necessary in a safety application, a `length` clause reserves in advance a pool of specified size of dynamic memory for any allocated objects of a given datatype. To take full advantage of this feature, the programmer must keep track of the number of objects currently allocated from the pool and ensure that this number does not exceed the capacity of the pool.

- *Provide handlers for the predefined exception **`STORAGE_ERROR`** if dynamic memory allocation is necessary.* If dynamic memory allocation is necessary in a safety application, providing handlers for the **`STORAGE_ERROR`** exception allows for graceful recovery from the situation of running out of dynamic memory. Without such handlers, the exception is propagated to the run-time executive and will most likely result in a crash of the system. The handlers should be provided for all program unit bodies in which memory is dynamically allocated, as well as in recursive subprograms (SPC, 1989; pp 77-78).

- *Explicitly handle dynamic memory deallocation if dynamic memory allocation is necessary.* Any automatic garbage collection facility provided by a compiler should not be used because it may affect timing. The pragma **`CONTROLLED`** is provided so that the program can disable automatic garbage collection (reclamation of unused memory)[4]. If dynamic memory allocation is necessary in a safety application, the application program should take full control for dynamic memory allocation and deallocation. Avoid the use of dynamic arrays, as in **`Procedure P(A:array(<>) of ...)`**.

- *Do not assign values of dynamically allocated access objects to other access objects.* If dynamic memory allocation is necessary in a safety application, the application program should not use multiple variables pointing to the same memory location. The danger is that when the shared memory space is deallocated, another variable may still point to the released memory space unless each one is explicitly set to null by the application program. If an application (e.g. a linked list) necessitates such multiple accesses, it must be justified and documented.

---

[4] It should be noted that according the language definition, there is no mandatory garbage collection requirement. It is up to the compiler implementation to provide such a facility.

```
procedure update_X is
   type three_D_Type is
      record
         x_coord : array(1..100) of float;
         y_coord : array(1..100) of float;
         z_coord : array(1..100) of float;
      end record;
   type three_D_pointer_type is access three_D_Type;

   procedure Dispose is new Unchecked_Deallocation(object => three_D_type,
                                                   Name   => three_D_pointer_type);

   ...
   p,q                  : three_D_pointer_type;
   three_D_display  : other_3D_type;  -- a 3-D subtype defined elsewhere
   ...
   begin
      p:=new three_D_pointer_type;-- dynamically allocate access objects p and q
      ...                         -- p is assigned a value somewhere in the code
      q:=p;                       -- q has been set to the value of p
                                  -- this is the source of the problem
     ...
     Dispose(p);    -- p has been set to null - now q contains an illegal value
     ...

     three_D_display :=q.x_coord;
                 -- annunciator_display will have unintended contents.
                 -- program may continue execution with undetected error
     ...
     three_D_display := p.x_coord;
                 -- CONSTRAINT_ERROR exception will be generated by this statement

end update_X;
```

The above example instantiates a procedure called **Dispose** to handle integers from the generic procedure **Unchecked_deallocation** for deallocating dynamically allocated memory units. It then allocates two access objects (**p** and **q**) on the stack, sets the value of **p**, sets the value of **q** based on **p**, deallocates **p** but leaves **q** pointing to inaccessible memory. Somewhere later in the code, the value of **q** is used in an assignment statement. The result may be technically invalid, but if it is within the constraints of the type, it will be displayed with no external manifestation of an error condition. On the other hand, if the explicitly deallocated access object (**p**) is used in a different assignment statement, the error will be detected and an exception will be raised. While neither condition is desirable, an undetected incorrect data value is far worse than a detected incorrect data value which causes

an exception to be generated (and hopefully handled without causing an unacceptable system state). The above example demonstrates not only the potential dangers in dynamically allocated variables but also the need to understand the detailed behavior of the `Unchecked_deallocation` procedure and how its use can lead to subtle errors. Important points of its behavior include:

(a) After completion of its execution, the value of the given parameter is **null**.
(b) If the given parameter is **null**, the call has no effect.
(c) If the given parameter is not **null**, the memory pointed by it is returned to the heap.

This last point is of the greatest significance to the above example. Because Ada has no runtime support such as a reference counter, it is possible to define two or more access objects (pointers) to a given location and free the space using only one of those access objects . The other access object(s) would still have an illegal access value(s) and might cause a hazard if used in subsequent processing.

3.1.1.2 Minimizing Memory Paging and Swapping

The generic guidelines are applicable on the system level. Ada itself contains no features for memory paging and swapping.

3.1.2 Predictability of Control Flow

Base level attributes related to the predictability of control flow in Ada are as follows:

•Maximizing structure
•Minimizing control flow complexity
•Initializing variables before use
•Single entry and exit points for subprograms
•Minimizing interface ambiguities
•Use of data typing
•Accounting for precision and accuracy
•Order of precedence of arithmetic, logical, and functional operators
•Avoiding functions or procedures with side effects
•Separating assignment from evaluation
•Proper handling of program instrumentation
•Controlling class library size
•Minimizing use of dynamic binding
•Controlling operator overloading.

These attributes and their relevance to safety are discussed in the following subsections.

3.1.2.1 Maximizing Structure

Maximizing structure means minimizing the explicit transfer-of-control statements that change the control flow from the basic set of sequential, conditional, and loop constructs. Most such statements can result in unreachable code. The following guidelines are applicable.

- *Do not use* **goto** statements. The generic guideline on maximizing structure by avoiding **goto** statements applies to Ada. The use of **goto**s can obscure program flow logic. This statement should be used only when there is no alternative. In Ada, where certain types of transfer of control have been incorporated into the language under other names such as exit, there is no real reason to use a **goto** in an Ada program (Sanden, 1994). Consider the following example.

```
<<B_Label>> statement_1;
            goto A_Label;
            statement_2;          -- unreachable code
            statement_3;          -- unreachable code
            statement_4;          -- unreachable code
<<A_Label>> statement_5;
            statement_6;
            statement_7;
            goto B_Label;
            statement_8;          -- unreachable code
```

- *Use only one* **exit** *statement per loop.* At least one **exit** statement is needed in loops without iteration schemes (LRM, 1995). Thus, only one **exit** statement should generally be used for the loop within the loop or for any nested loops.

- *Use only one* **return** *statement per function.* Multiple **return** statements can make the meaning of a subprogram confusing. Thus, function subprograms should have only one **return** statement and procedure subprograms should either use the normal exit at the end of the body or have only one **return** statement if the end of the body is inaccessible, for example, an infinite loop just before the end of the body.

While maximizing structure is desirable for normal program flow, different rules apply to exception handling as discussed in Section 3.2.2. When exceptions are raised, other considerations (e.g., timing, intermediate operations, etc.) dominate. The guidelines on exception handling discuss **raise** statements in more detail.

3.1.2.2 *Minimizing Control Flow Complexity*

The generic guideline applies to Ada. The language-specific guidelines for minimizing control flow complexity are as follows:

- *Limit nesting levels.* As noted in the generic report, there should be explicit organizational or project-specific limits on nesting. These limits may be determined in part with respect to a particular language and execution platform. The style guidelines for Ada published by the Software Productivity Consortium recommend a maximum nesting level of three to five (SPC, 1989; pp 83 - 84).

- *Use `if..elsif` instead of nested `if..else`.* Use of an **`if..elsif`** in place of nested **`if..else`** statements helps avoid program structural and logical errors (Barnes, 1984; p 62), as shown in the following example:

```
-- Use
if condition_1 then
  statement_1;
elsif condition_2 then
  statement_2;
end if;

-- instead of
if condition_1 then
  statement_1;
else
    if condition_2 then
      statement_2;
    end if;
end if;
```

  Always provide an `else` branch to `if` statements if there is a remote chance that the conditions specified by the other `if` statements are exhaustive.

- *Use case statements for multiple branches.* The **`case`** statement serves as a switch for multiple branches and allows one evaluation for them. It is a powerful alternative to the **`if`** statement when the branch to be taken depends upon the value of a discrete expression, and it is preferred if more than two conditions or branches are called for in the software design. To avoid a syntax error, the **`when others`** construct must be included if there are any possible values not given in other alternatives, as seen in the following example (SPC, 1989; p 85):

```
-- Use
case thermal_alarm is
      when core    => core_thermal_alarm(sensor_value);
      when inlet   => inlet_thermal_alarm(sensor_value);
      when outlet  => outlet_thermal_alarm(sensor_value);
      when others  => do_something;
end case;

-- instead of
if thermal_alarm = core then
  core_thermal_alarm(sensor_value);
elsif thermal_alarm = inlet then
  inlet_thermal_alarm(sensor_value);
elsif thermal_alarm = outlet then
  outlet_thermal_alarm(sensor_value);
else
  do_something;
end if;
```

It should be noted that the **case** statement is not an all purpose replacement for the **if.. then...else** construct. A **case** statement is only possible if the cases depend on the different values of one expression with a limited range of possible values. (In the example on this page, **thermal_alarm** is an enumerated type with a limited set of possible values.) In that situation, the case construct is always preferable over an **if.. then...else** unless the number of branches is small.

3.1.2.3 Initialization of Variables before Use

The generic guideline with respect to initialization of all variables applies to Ada. Variables should be initialized to some known value at the beginning of an execution cycle before using them. A compiler cannot be depended on to reset variables automatically (Gottfried, 1993; SPC, 1989, pp 103-104). However, even if the compiler could be relied on to initialize values, the safety concern would still exist because the compiler cannot be expected to initialize all objects with suitable values.

Ada provides a variety of syntaxes for data initialization upon elaboration of a variable as shown in the following example:

```
subtype Number_Of_Widgets is Natural range 0 .. 1_000;
Accumulator : Number_Of_Widgets := 0;

type Coefficients is array (1 .. 3, 1 .. 3) of Weights;
Example_Coefficients : Coefficients
  := (      ( 1.0,   0.5,   0.1),
            ( 0.5,   1.0,  -0.3),
            ( 0.1,  -0.3,   1.0)  );

type Complex_Numbers is record
  Real_Part             : Float := 0.0;
  Imaginary_Part: Float := 0.0;
end record;
Zero : Complex_Numbers; -- Automatically initialized to(0.0, 0.0)
                        -- when elaborated (unreliable)

Square_Root_Of_Minus_1 : Complex_Numbers
  :=  (Real_Part => 0.0, Imaginary_Part => 1.0);



type A is array (1 .. 100) of Character;
      AA :A := (others => 'x');
                        -- Aggregate initialization:
                        -- multiple elements
                        -- of an array can be given initial values
                        -- by means of the construct 'others ==>'

type B is array (years, months) of Integer;
      BB: B := (others => (others => 0));
                       --  Without this construct,
                       --  it would be  impractical to initialize
                       --  a large array.
```

The following are Ada-specific initialization guidelines.

- *Initialize in function body if initialization occurs via a function call.* If initialization occurs via a function call, initializations should be done in a program body rather than in the variable declaration since the function body may not have been elaborated when the variable declaration was encountered (SPC, 1989; pp 103-104).

- *Restrict use of aggregate assignments for initialization of large objects.* As shown in the above example, aggregates are a useful way of initializing large arrays. However, the initialization of large objects via aggregates should occur with caution. The reason for this guideline is that some compilers accomplish aggregate assignments by first building a temporary version of the object with the specified values in system memory and then copying the contents into the actual object. If the size of the temporary version exceeds available

memory, the result could be a system crash.[5] In such cases, testing should be done to ensure that the aggregate assignment can be performed acceptably under operational conditions. An alternative is to perform initialization in the program unit body rather than in the objects' declarations for large objects.

There are two cases in Ada where *explicit* initialization of a variable need not be done to comply with the guideline. First, all objects of access type (i.e., pointers) are automatically initialized to **null** by the compiler. Second, type definitions for records may contain default initialization values for all components; whenever objects of those record types are elaborated, their components are set to the defaults in the absence of an explicit initialization (DoD-STD-1815A; Section 3.7).

3.1.2.4 Single Entry and Exit Points for Subprograms

Although the generic guideline is applicable with respect to one *normal* entry and exit point per subprogram[6], the guideline has limited applicability due to Ada's exception handling and tasking features. Ada-specific guidelines are:

- *One normal entry and exit per subprogram.* Subprograms (procedures and functions) should have one normal (as opposed to exception) entry and one normal exit. The word `return` should appear exactly once in each function and not be used in a procedure. In exceptional cases, however, multiple exits can be used if they increase readability.

- *Limit the number of exception entry/exit points.* The number of these points should be kept as low as possible. Each of these exception propagation exit/entry points should be documented clearly. The propagation of an exception raised in a subprogram to the caller of the subprogram should be limited or not used at all because such propagation creates an additional exit point for the first subprocedure and an additional entry point for the caller's exception handler. More points on propagation of exceptions are discussed in Section 2.2.2.

- *Avoid multiple task entry points.* Each active program unit (i.e., task) may have multiple interaction points with other active program units. The number of these interaction point should be designed to minimize program complexity both within the task and the entire program. Additional points on tasking are described in Section 2.2.

---

[5]Such a situation actually occurred in the experience of one of the writers of this section. In an image processing application, a 1024 x 1024 array of pixels was initialized by an aggregate of the form `((others => 0), others => 0)`. This caused the entire system, including the operating system and the other jobs being executed concurrently, to crash without any error messages. Determining the cause was complicated by the fact that the Ada code was syntactically and semantically correct.

[6]It is more appropriate to refer to entry and exit points in program unit bodies rather than in subprograms in the case of the Ada language.

3.1.2.5 Minimizing Interface Ambiguities

The generic guideline to minimize interface ambiguity applies to Ada. Ada automatically provides features that eliminate many interface errors. For example, constraint checking is performed on values of actual input parameters to ensure they are not out of range. Another example is that the indices of the first and last elements in an array or array slice-parameter are automatically passed in with the actual array parameter. Nevertheless, the language does not eliminate interface ambiguities.

The following are specific guidelines:

- *Specify argument modes.* Arguments with procedures and entries should have their modes specified in their declarations rather than relying on the default mode (SPC, 1989; p 68). Specifically:

```
procedure Quadratic(a, b, c: in  Float;    root1, root2 : out Float);
```

rather than:

```
procedure Quadratic(a, b, c : Float; root1, root2 : out Float);
```

While the latter declaration is acceptable syntax (and in that sense, is unambiguous to the compiler), explicit use of modes avoids confusion to programmers and reviewers.

- *Restrict use of the* **in out** *mode.* The **in out** mode should be used only for parameters whose value will be changed by the procedure. It should not be specified for parameters used exclusively as either **in** or **out** parameters. When used in place of an **in** mode, it is possible to modify a value that should be constant unintentionally. Using **in out** for an **out** mode causes fewer problems, but it does obscure the intent of the parameter. This mode is frequently used in the case of an output parameter whose value is read inside a subprogram; when this situation leads to a compilation error, many programmers will change the mode from **out** to **in out** rather than taking the trouble to declare and use a local variable.[7] For example, programmers will code as follows:

[7]In Ada 95 reading an **out** mode parameter is allowed. According to the Ada 95 rationale, too many programmers were forgetting to copy the value of the local variable into the output parameter at the end of procedures.

```
procedure Find_Max (In_The_List : in Some_Array_Type;
                    Maximum     : in out Element_Type) is

begin
    Maximum := Element_Type'first;
    for List_Index in In_The_List'range loop
        if In_The_List(List_Index) > Maximum then -- value read here
            Maximum := In_The_List(List_Index);
          end if;
      end loop;

end Find_Max;
```

instead of coding:

```
procedure Find_Max       (In_The_List : in     Some_Array_Type;
                           Maximum    :    out Element_Type) is

    Local_Max : Element_Type := Element_Type'first;

begin
    for List_Index in In_The_List'range loop
        if In_The_List(List_Index) > Local_Max then
            Local_Max := In_The_List(List_Index);
        end if;
    end loop;
    Maximum := Local_Max;

end Find_Max;
```

- *Use named parameter associations.* Named parameter associations should be used by the calling routine for functions, procedures, and task entries whenever there are two or more parameters of the same type in the parameter list. Using named parameter associations improves readability and reliability (Booch, 1983; p 106). The following example shows the use of named parameter associations for a quadratic equation evaluation procedure.

```
Quadratic(a      => second_order_coefficient,
          b      => first_order_coefficient,
          c      => constant_term,
          root_1 => first_root,
          root_2 => second_root;
          OK     => status) ;
```

- *Refer to the target data type rather than the pointer's type when referencing data.* When data referenced by a pointer are to be read or modified in a subprogram and the value of the

pointer itself is not to be used, the declaration and call of the subprogram should refer to the target data type rather than the pointer's data type as shown below.

```
type Target_Type is array (1 .. 100) of Component_Type;
type Pointer_Type is access Target_Type;

The_Data : Pointer_Type := new Target_Type'(others => 0);

-- Better subprogram declaration
procedure Print(The_Data : in     Target_Type);

-- Better subprogram call
Print(The_Data.all);

-- Worse subprogram declaration
procedure Print(The_Data : in     Pointer_Type);

-- Worse subprogram call
Print(The_Data);
```

This practice removes ambiguity about which data are to be processed in a subprogram, that is, the data being pointed to or the pointer. For **in** mode parameters, this practice removes the possibility of modifying data meant to remain unchanged, since it is possible to modify data pointed to by an **in** mode access type parameter. The practice also allows checking for out-of-range data values. However, care must be taken when passing a large object by value to avoid memory overflows.

- *Avoid aliased parameters.* Aliased parameters should be avoided. They can arise from using the same actual parameter for more than one formal parameter (and calling both by reference), using overlapping array slices, referencing global variables, and using pointers referencing the same data for different actual parameters. Results can be dependent on compiler-specific implementations such as the order of evaluation of actual parameters. Even when called by value, passing the same actual to two formal parameters or passing a global variable to a procedure is discouraged.

3.1.2.6 Use of Data Typing

The generic guidelines for data typing apply to Ada. Ada was made a strongly typed language in order to provide the potential for increased safety. Code should take advantage of this feature to the maximum extent possible. The following specific guidelines are related to the full use of data typing:

•Use of built-in functions
•Use of compiled libraries
•Use of generics.

Because readability is also an intermediate attribute of maintainability, it is discussed in Section 3.4. Ada-specific guidelines for the other attributes are discussed in the following sections.

3.3.1 Use of Built-In Functions

The generic guidelines have limited capability. The only built-in functions in Ada are those that are Ada operations. These operations may be overloaded. Because Ada does not provide an extensive number of built-in functions, each project builds or acquires (either through reusing or purchasing) additional functions. It should be noted that a separate guideline on the use of compiled libraries recommends that externally developed libraries be acquired as source code.

Externally developed software should be subjected to at least the same degree of developmental control and verification as the project-specific code. This would include assessment of the accuracy, limitations, robustness, and exception handling of the functions. Test cases, procedures, and results of previous testing should also be maintained for these libraries. The test cases should assess behavior for out of bounds and marginal conditions (e.g., negative arguments on a square root routine, improperly terminated strings for a string copy routine, and similar conditions) in the specific run-time environment.

3.3.2 Use of Compiled Libraries

The generic guidelines related to controlled use of compiled libraries are applicable to Ada. The reasons for limiting or avoiding the use of compiled libraries in safety systems are as follows:

•*Lack of visibility.* Libraries can be used to shield the programmer from the details of the lower level implementation; however, it is exactly that feature that prevents the programmer from knowing the accuracy, limitations, robustness, and exception handling of the built-in functions. Programmers and designers must consider how to handle error conditions such as invalid parameters, numerical instability of the calculation, non-convergence of a result, arithmetic overflow, and underflow. These different forms of failure may well require handling in different ways according to the severity of the impact of the error on the calculation. In compiled libraries, the error handling mechanisms may not provide the needed visibility to allow programmers to handle these situations (Tafvelin, 1987).

•*Inconsistency in error handling.* A basic consistency data and control flow for error handling is necessary for developing and maintaining reliable systems. However, there is no guarantee that libraries will have consistent methods of handling exceptions.

•*Difficulties during maintenance and upgrades.* As software is maintained and new versions of compilers are acquired, libraries may become outdated.

If compiled libraries are used, then thorough testing and error tracking are necessary as described in the generic guidelines.

3.3.3 Ada Run-time Environment

The Ada RTE plays a critical role in ensuring the timing and correct execution of the compiled Ada code. However, it is not directly accessible by the programmer and falls into the category of built in functions or compiled libraries from that perspective. The concerns related to testing, error tracking, documentation, and development control described in the previous two sections also hold true for the Ada RTE.

3.3.4 Maintaining Traceability Between Source Code and Compiled Code

For a safety application, it is vital to ensure that the source code in a project baseline corresponds to the compiled object code. Traceability between source and object code is needed to avoid the uncertainty of what versions of separately compiled units are included. However, the support of the Ada language for separate compilation can pose a challenge to this traceability. When possible, the entire source (with the exception of compiled libraries, see Section 3.3.2) should be compiled on one occasion. This is the most authoritative way to establish complete traceability between source and executable.

However, it may not be possible to perform a single compilation because:

1. The source code is too large.

2. To support portability, implementation dependent source code is being placed in separate compilation units from other Ada source code.

3. It may be desirable or necessary to incorporate externally developed components in compiled rather than source form.

If separate compilation is needed the following guidelines apply:

- *Partitioning of compilation.* Only those compilation units required for execution of a compilation undergoing compilation unit should be made visible (using a **with** clause) to each unit, i.e. the `with` clauses should not include superfluous compilation units (Jones, 1988).

- *Use of tools.* Tools should be acquired that maintain the libraries in a sufficiently transparent manner to allow such traceability without the need to compile all the source code be at one time.

3.3.5 Minimizing Use of Generic Units

The Ada language includes generic units (packages or subprograms) to enhance reusability. However, their use in safety systems is problematic because they obscure the traceability between source code and executable. They are templates, not packages or subprograms, and it is not immediately clear from reading the source exactly what is running in the executable code. Use of generic units should therefore be minimized (Sanden, 1994).

However, generics may be necessary in Ada—particularly predefined generic units. If generics are used, they are subject to the following guidelines.

- *Instantiation only during initialization.* This guideline was discussed in section 3.1.1 on predictability of memory management.

- *Use only the parameter list for transferring data.* No global variables should be used to supplement the parameter list and used in the bodies of other subprograms. The parameter list should be comprehensive for all intended uses.

- *Document restrictions on parameters.* The use of and restrictions on generic parameters should be identified and documented (Jones, 1988).

3.4 Maintainability

This section discusses the Ada-specific attributes of the following intermediate attributes related to maintainability:

- Readability
- Data abstraction
- Functional cohesiveness
- Malleability
- Portability.

Base-level attributes and Ada-specific related guidelines are discussed in the following sections.

3.4.1 Readability

The following base attributes are related to readability:

- Conformance to indentation guidelines
- Descriptive identifier names
- Comments and internal documentation
- Limitations on subprogram size
- Minimizing mixed language programming
- Minimizing obscure or subtle programming constructs
- Minimizing dispersion of related elements
- Minimizing use of literals
- Controlled use of renaming.

The Ada-specific guidelines associated with these attributes are discussed in the following subsections. It should be noted that the controlled use of renaming is an Ada-specific attribute that was not included in the generic guidelines.

3.4.1.1 Conformance to Indentation Guidelines

The generic indentation guidelines are applicable. The following additional guidelines apply:

- *Data structures.* Indent and align beginnings and endings of data structures.

- *Line Continuation* use different levels of indentation to distinguish between indentations for statements and for line continuation (SPC, 1989, pp. 9-11; DoD-STD-2167A, App. F).

3.4.1.2 Descriptive Identifier Names

The guidelines developed for the generic descriptive identifier names attribute are applicable to Ada. The following additional guidelines apply:

- *Follow project-specific guidelines on naming.* Project specific guidelines on the use of names for variables, type definitions, procedures, functions, records, arrays, slices, exceptions, constants, generic instantiations, access objects, and other identifiers should be developed and followed in each program. The guidelines should also address naming of items in different packages (if applicable), how names change based on scope, and other project-specific considerations.

- *Separate words.* Words in compound names should be separated with underscores as indicated in the following example (SPC, 1989; p. 17)

```
Rads_Per_Second
Core_Temperature
```

- *Use underscores with larger numbers.* Underscores should be used with large numbers to promote readability on numbers (SPC, 1989; p. 20). This is shown in the following example:

```
type Populations is range 0 .. 10_000_000_000;

type Social_Security_Numbers is range 000_00_0000 .. 999_99_9999;
```

- *Use care in abbreviations.* Abbreviations should not be used if they can be misunderstood. For example, **`Time_of_Day`** should be used instead of **`TOD`** (SPC, 1989; p 20).

3.4.1.3 Comments and Internal Documentation

The guidelines associated with the generic attributes are applicable. In addition, the following Ada-specific guidelines apply:

- *Relate the code to higher level design considerations.* Explanatory comments should not duplicate the Ada syntax or semantics, but should clarify the coded data structures or process algorithms at a more descriptive level than the code. "Comments should be technically correct and should address a reader who is an Ada programmer" (DoD-STD-2167A).

- *Use blank lines.* Related code such as declarations, loops, blocks, cases, and exception handlers should be grouped, separated with blank lines, and described with Ada comments (DoD-STD-2167A).

- *Identify "escapes" from language restrictions:* Escapes from Ada language restrictions (suppression of type checking, unchecked conversions, use of other languages, etc.) are discouraged in other portions of this chapter. However, if they are used, they should be clearly indicated in the comments together with rationale and impact.

- *Use comments when renaming.* The scope of renaming should be indicated in comments physically adjacent to the renaming statements.

- *Comment exception raising and handling.* Comments should be used to facilitate the tracing between exception raising and handling, and to provide traceability back to design documents where the exceptions and handlers were designed.

- *Identify dynamic memory allocation with comments.* As noted earlier, dynamic memory allocation is not desirable in a safety system. If used, however, there should be comments to identify when memory is allocated and released.

- *Identify tasking with comments.* As noted previously, tasking and intertasking communication poses many safety challenges. Comments should provide traceability to a design, and the design itself should clarify issues associated with timing, intertask communication, and avoidance of the risks associated with tasking.

3.4.1.4 Limitations on Subprogram Size

The guidelines associated with this generic attribute are applicable. There are no additional specific guidelines.

3.4.1.5 Minimizing Mixed Language Programming

The guidelines associated with this generic attribute are applicable. The use of machine-level[12] language or a non-Ada higher-level language should be avoided in Ada program units. The reasons for avoiding other languages are listed below.

1. There is no uniform way to implement machine-level code in an Ada source program, There will be differences in lower-level details, such as register conventions, that would hinder implementation and portability.

2. The problems with employing pragma `INTERFACE` are complex[13]. These problems include pragma syntax differences, conventions for linking/binding Ada to other languages, and mapping Ada variables to foreign language variables, among others.

3. Other languages do not provide a means of expressing low-level machine features in a high-level fashion as well as Ada does (Booch, 1983; p 264).

If use of other languages cannot be avoided, it should be minimized and controlled. The following are Ada-specific guidelines:

- *Isolate and clearly document machine language inserts.* If machine-level code inserts must be used to meet a project requirement, isolate the platform-specific implementations in a separate package. Include the commentary that a machine-level code insert is being used and state what function the insert provides and (especially) why the insert is necessary. Document the necessity of using machine-level code inserts by delineating what went wrong with the attempts to use other higher level constructs (SPC, 1989; p 146).

---

[12]In Ada the term "machine-level" language is equivalent to "assembly" language.

[13]A subprogram written in another language can be called if all data transfer is via parameters and function results. The Interface pragma is the mechanism for achieving this.

- *Isolate Higher-level language inserts, document the* **INTERFACE** *pragma, and account for interface limitations*: Subprograms employing the pragma **INTERFACE** should be isolated to an implementation-dependent (interface) package. The requirements and limitations of the interface and pragma **INTERFACE** usage should be clearly documented (SPC, 1989; p 146). As noted above, the conventions used by other compilers are not specified by Ada. Thus, validating the interface and ensuring that it is free from potential interface problems can be a complex undertaking. However, a thorough examination is required for safety significant systems.

3.4.1.6 Minimizing Obscure or Subtle Programming Constructs

The guidelines associated with this generic attribute are applicable. There are no additional language-specific guidelines.

3.4.1.7 Minimizing Dispersion of Related Elements

The guidelines associated with this generic attribute are applicable. There are no additional Ada-specific guidelines. In Ada, appropriately designed packages can minimize dispersion of related elements. This is so since a data structure and any subprograms operating on it can be collected in an information-hiding package in such a way as to give other parts of the software controlled access to the data exclusively via a well-defined interface.

3.4.1.8 Minimizing Use of Literals

The guidelines associated with this generic attribute are applicable. The following are additional Ada-specific guidelines:

- *Use constants instead of literals.* The use of constants supports maintainability by assuring that all values referencing a constant are automatically changed by a single change to the constant declaration. The exception to this guideline is that numeric literals may be used in well-established formulae or conversions where such values will not change and where readability will be enhanced by the use of such literals (e.g., in the quadratic equation).

- *Use attributes.* An additional Ada-specific guideline is that Ada attributes should be used wherever possible in place of literals, as indicated in the following example. This practice facilitates the propagation of consistent changes when objects related to the constant are changed.

```
MAX_LINE_LENGTH : constant := 132;

type Lines is array (1 .. MAX_LINE_LENGTH) of Character;
Line  : Lines;

-- Use
for Column in Line'range loop
    if Column = Line'first then
      ...
    elsif Column = Line'last then
      ...
    end if;

-- instead of
for Column in 1 .. 132 loop
    if Column = 1 then
      ...
    elsif Column = 132 then
      ...
    end if;
```

3.4.1.9 Controlled Use of Renaming

Renaming is frequently used to reduce the length of unwieldy, fully qualified names and to make clear ambiguous or inappropriate names. The renamed identifier can also be an aid to understanding the use of a routine. However, renaming also complicates and obscures the traceability from the procedure or function call to the source code. This makes debugging and maintenance harder. Renaming of subprograms can cause unintended overloading that the designers, programmers, and maintainers may not realize or fully understand.

The following example (from Mil-Std-1815A) illustrates the problem:

```
function ROUGE return COLOR renames RED ;
function ROT   return COLOR renames RED ;
function ROSSO return COLOR renames ROUGE ;
```

The function **RED** has been renamed as **ROUGE** in the first line and **ROT** in the second. In the third line, the renaming on the first line (**ROUGE**) has itself been renamed to **ROSSO**. This renaming makes it difficult to understand where a problem occurs if the function **RED** needs to be debugged.

The following guidelines can mitigate these problems while preserving the benefits of renaming:

- *There should be only one level of renaming.* A renamed identifier should not be renamed a second time.

- *All renaming should be done in accordance with project-specific conventions.* Project-specific conventions should be developed for variable naming and renaming.

- *Maintain a centralized list of names.* A "registry" of renaming should be maintained for each project. The scope of each renaming should also be clearly indicated in the registry.

3.4.1.10 Use representation clauses for bit mapping.

In many safety systems, there is an interface to a set of hardware discrete switches that affect the state of the system. Such bit maps are typically stored internally as integers. However, representation clauses and enumeration types can be used to effectively represent this status information in a meaningful way, which facilitates review and also reduces the possibility of coding errors as the following example demonstrates.

```
Type Line_Status_Type IS
      (Valve_1A_Open, Valve_2A_Open, Valve_3A_Open,
       Valve_1B_Open, Valve_2B_Open, Valve_3B_Open)


FOR Line_Status_Type USE
      (Valve_1A_Open     => 2#0000_0001#,
       Valve_2A_Open     => 2#0000_0010#,
       Valve_3A_Open     => 2#0000_0100#,
       Valve_1B_Open     => 2#0001_0000#,
       Valve_2B_Open     => 2#0010_0000#,
       Valve_3B_Open     => 2#0100_0000#);
```

The array must be sorted in strict ascending order. It is better to use a name than a positional association (Cohen, 1986, p. 780).

3.4.2 Data Abstraction

This section discusses Ada-specific data abstraction guidelines for the following attributes:

- Minimization of global variables
- Minimization of the complexity of interfaces
- Use of the Ada package for encapsulating programs and data.

3.4.2.1 Minimization of Global Variables

A global variable in Ada can be declared in the main procedure or in a package specification. Unless the entire program is small, neither should be used. A variable that must remain in existence and retain its value longer than the execution of a single subprogram should be declared in a package body. The package specification should include those procedures and functions that operate on the variable in the package. Such information hiding ensures that the variables are not updated in unintended ways.

3.4.2.2 Minimization of Complexity of Interfaces

The generic guidelines apply to this attribute. There are no additional Ada-specific guidelines.

3.4.2.3 Use of the Ada Package for Encapsulating Data and Related Programs

The Ada *package* feature was developed to control visibility of names and access to data. As such, it is a useful mechanism to prevent inadvertent alteration of data or execution by other programs. Some examples of appropriate use of the package construct in safety systems are contained in guidelines elsewhere in this chapter. A full discussion of this topic, however, is a design issue and beyond the scope of this document. It is covered extensively in other publications on the Ada 83 language (Shumate, 1989; Cohen, 1986, SPC, 1989).

The only implementation-specific guideline is that the project programming guidelines and the system design itself should identify standards and conventions for :

- Defining interfaces, type definitions, and data structures (including records, arrays and strings) in packages
- Organization of compilation units
- Use of predefined compilation units (e.g., `SYSTEM` and `STANDARD`).

3.4.3 Functional Cohesiveness

Functional cohesion measures the degree to which a subprogram performs a single, problem-related, well-understood function. The generic attributes relating to (1) a single design level function per subprogram element and (2) each identifier having a single purpose both apply to Ada . There is no additional language-specific guidance.

3.4.4 Malleability

The generic attribute applies to Ada. There is no additional language-specific guidance.

3.4.5 Portability

The generic attribute applies to Ada. From the perspective of safety, the benefits of portability are the adherence to standard programming constructs that yield predictable and consistent results across different operating platforms. Code that has been designed to be portable will be easier to maintain when it is reused or converted to run on a different platform. The general principle is avoiding use of nonstandard, or "enhanced", constructs specific to a particular compiler by itself or in combination with the target execution platform. Where nonstandard constructs are necessary, they should be clearly identified together with the rationale, limitations, and version dependencies (SPC, 1989; pp. 127-155).

Attributes related to portability, which have been discussed elsewhere, include the following:

- Minimizing the use of built-in functions
- Minimizing the use of machine code and foreign languages
- Minimizing the use of compiled libraries
- Minimizing dynamic binding
- Minimizing tasking
- Minimizing asynchronous constructs (interrupts).

The following are additional language-specific guidelines:

•*Do not use busy loop to suspend execution.* Aside from the fact that a busy loop wastes processor resources, the timing of a standard loop cannot be determined when the code is ported to a different compiler, different machine, or even different operating systems. For example:

```
-- Use
delay 3.74 ;
-- Do not use following because of timing differences
for I in 1 .. 6874 loop
  null ;
end loop ;
```

Also, any knowledge of the execution pattern of tasks should never be used to achieve timing requirements, because of the uncertainty during porting (SPC89, p. 141).

- *Validate assumptions about the implementation of language features when specific implementation is not guaranteed or specified.* For example, there may or may not be a

correlation between **SYSTEM.TICK** and package **CALENDAR** or type **DURATION.** Although such a correlation may exist, it is not required to exist (SPC, 1989; p 141).

- *Avoid the use of package* **SYSTEM** *constants except in attempting to generalize other machine dependent constructs.* Since the values in this package are implementation provided, unexpected effects can result from their use (SPC, 1989; p 146). The values of the constants in the **SYSTEM** package should not be changed.

- *Use only pragmas and attributes defined by the Ada Standard.* The Ada LRM (Mil-Std-1815A) defines the following pragmas: **controlled, elaborate, inline, interface, list, memory_size, optimize, pack, page, priority, shared, storage_unit, suppress, system_name** and the following attributes: **address, base, callable, constrained, count, first, first_bit, last, last_bit, pos, pred, range, size, small, storage_size, succ, terminated, val, value, width.** However, the Ada standard permits an implementor (compiler vendor) to add pragmas and attributes to exploit a particular hardware architecture or software environment. Although potentially attractive, non-standard pragmas and attributes are not only non-portable, their limitations may not be as well understood nor tested as are the predefined counterparts. It should be noted that predefined pragmas and attributes in and of themselves may not be totally portable because of the latitude allowed in their interpretation by compiler implementors.

- *Avoid the direct invocation of, or implementation dependence upon, an underlying host operating system or Ada run-time support system.* Features of an implementation not specified in the Ada LRM will usually differ between implementations. Specific implementation-dependent features are not likely to be provided in other implementations. Even if a majority of vendors eventually provide similar features, they are unlikely to have identical formulations. Indeed, different vendors may use the same formulation for (semantically) different features.

- *Minimize and isolate the use of the predefined package* **LOW_LEVEL_IO.** This package is intended to support direct interaction with physical devices that are usually unique to a given host or target environment. In addition, the data types provided to the procedures are implementation defined. This allows vendors to define different interfaces to an identical device (SPC, 1989; p 152).

- *Restrict and isolate variables of type* **SYSTEM.ADDRESS** or with the attribute **ADDRESS.** These are hardware-specific variables that should be kept in a "maintenance location" in the code.

References

International Standard ANSI/ISO/IEC-8652, *Ada 95 Reference Manual*, Intermetrics, Inc., Cambridge, MA, 1995.

Ada 95 Rational, Intermetrics, Inc., Cambridge, MA, 1995.

American National Standards Institute/U.S. Department of Defense, *Reference Manual for the Ada Language*, ANSI/DoD-STD-1815A, 1983.

Barnes, J. G., *Programming In Ada*, Second Edition, Addison-Wesley Publishing Company, Menlo Park, CA, 1984.

Booch, G., *Software Engineering with Ada*, California, The Benjamin Cummings Publishing Company, Menlo Park, CA, 1983.

Cohen, N., *Ada as a Second Language*, Prentice Hall, Englewood Cliffs, NJ, 1986

Gall, J., *Systematics: How Systems Work and Especially How they Fail*, The New York Times Book Company, New York, NY, 1975.

Gottfried, R. and D. Naiditch, *Using Ada in Trusted Systems, Proceedings of COMPASS 93*, May, 1993, National Institute of Standards and Technology, Washington, DC, 1993.

Hutcheon, A., et al., *A Study of High Integrity Ada*, (UK) Ministry of Defense contract: SLS31c/73 Language Review, Document Reference SLS31c/73-1-D, Version 2, 9 July 1992.

Jones, S, K. Mitchell, M. J. Mardesich, et. al., *BCAG Digital Avionics Ada Standard,* Boeing Company, Document No. D6-53339, November, 1988

Kernighan, B. and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill, New York, 1974.

Page-Jones, M., *The Practical Guide to Structured System Design*, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1980.

Pyle, I., *Developing Safety System : A Guide Using Ada,* Prentice Hall, Englewood Cliffs, NJ, 1991.

Sanden, B. I., *Software Systems Construction with Examples in Ada.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

Software Productivity Consortium (SPC), *Ada Quality and Style Guidelines for Professional Programmers*, Van Nostrand Reinhold, New York, NY, 1989.

Tafvelin, S., ed, *Ada Components: Libraries and Tools*, Cambridge University Press, Cambridge, MA, 1987.

U.S. Department of Defense, *Defense Systems Software Development*, DoD-STD-2167A, Appendix D, 1 August 1986.

4 C and C++

This section discusses the safety issues of C and C++ languages in safety systems. The languages are discussed together because of the C heritage in C++ and because they may be used together in a safety application. However, the applicability of the discussion to one or both languages is clearly indicated in the text[14]. The discussion is primarily independent of the underlying execution platforms, that is, hardware, kernel, and/or operating system. Exceptions to this generalization are noted in the text.

This chapter is organized in accordance with the framework of Chapter 2. Section 4.1 discusses reliability-related attributes; Section 4.2 discusses robustness-related attributes; Section 4.3 discusses traceability-related attributes; and Section 4.4 describes maintainability-related attributes. A summary matrix showing the relationship between generic and language-specific guidelines, together with weighting factors, is included in Appendix B.

4.1 Reliability

In the software context, reliability is either (1) the probability of successful execution over a defined interval of time and under defined conditions, or (2) the probability of successful operation upon demand (IEEE, 1977). The reliability of software means the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE, 1990). The reliability depends on the run-time predictability of the following:

- Memory utilization
- Control flow
- Timing.

C-specific guidelines derived from these generic attributes are described in the following sections.

4.1.1 Predictability of Memory Utilization

Unpredictable memory utilization can cause the loss of programs, instructions, and data which, in turn, can cause system failures. Unpredictable memory utilization can be categorized into two main categories: (a) violation of available memory restrictions and (b) unauthorized use of memory blocks. The first four base attributes refer to the first category and the remainder to the second.

---

[14]It should be noted that what is applicable to C is generally applicable to C++; however the reverse is not true.

- Minimizing dynamic memory allocation
- Minimizing memory paging and swapping
- Minimizing memory usage caused by inefficient parameter passing mechanisms
- Minimizing recursive function calls
- Utilizing boundary checking for memory-related functions
- Utilizing functions with well-defined behavior
- Using wrappers for memory-related functions
- Proper array indexing.

4.1.1.1 Minimizing Dynamic Memory Allocation

Following guidelines are applicable to both C and C++

Although dynamic memory allocation increases memory utilization efficiency, it can cause unpredictable memory utilization which, in turn, could result in system failure (Hatton, 1994, p149). The potential problems caused by dynamic memory allocation include:

1. Allocating memory without subsequently freeing it.
2. Attempting to access memory that has not been allocated.
3. Utilizing memory that has already been freed.
4. Insufficient available memory for the dynamic memory requirements.

Thus, dynamic memory allocation should be avoided. If dynamic memory must be used, the related functions should be used defensively, and the allocated memory should be explicitly released as soon as possible.

Following discussion applies to C

In C the dynamic memory allocation and deallocation functions are **`calloc`**, **`malloc`**, **`realloc`**, **`strdup`**, and **`free`**. In addition to the above problems, other dynamic memory allocation potential problems arise in C because of two reasons: (1) dynamic memory allocation functions provide different services depending on the values of input parameter (Maguire, 1993) and (2) dynamic memory management functions are not sufficiently protected against potentially incorrect input.

The following function serves as an example:

```
void *realloc(void *pv , size_t size).
```

The function will perform one of the following actions depending on the input (Maguire, 1993):

(a) If the new size of the memory block is smaller than the old size, **`realloc`** releases

the unwanted memory at the end of the block and pv is returned unchanged;

(b) If the new size is larger than the old size, the expanded block may be allocated at a new address and the contents of the original block copied to the new location. A pointer to the expanded block is returned, and the extended part of the block is left uninitialized.

(c) If one attempts to expand a block and **realloc** cannot satisfy the request, NULL is returned.

(d) If **pv** is **NULL**, then **realloc** behaves as **malloc** (size) and returns a pointer to a newly allocated block, or **NULL** if the request cannot be satisfied.

(e) If the new size is 0 and pv is not **NULL**, then **realloc** behaves as **free** (pv) and **NULL** is returned.

(f) If pv is **NULL** and size is 0, the result is unknown.

- *Use library copy and move functions with specific lengths.* As will be discussed below, library copy and move functions with specific lengths (e.g., **strncopy** rather than **strcpy**) should be used.

The following discussion applies to C++ only

In C++, the functions to dynamically allocate and free memory are **new** and **delete**. The following guideline applies.

- *Ensure that all classes include a destructor.* To avoid memory leaks, all classes must include a destructor that releases any memory allocated by the class. Constructors must themselves be defined in a way to avoid possible memory leaks in case of failures. Ensure that for all derived classes there are virtual destructors.

4.1.1.2 Minimizing Memory Paging and Swapping

Following guidelines are applicable to both C and C++

The generic guidelines apply. There are no additional language-specific guidelines.

4.1.1.3 Controlling Parameter Passing to Routines

Following discussion applies to C

The generic guidelines apply. Of particular concern in the use of C or C++ with small microcontrollers is the limited stack size. Passing of many arguments or large structures may cause a stack overflow (particularly in microcontrollers where stack memory may be limited) that, in turn, would cause a system failure. The following are language-specific guidelines:

- *Limit the number and size of parameters.* The ANSI/ISO C standard only guarantees 31 parameters in one function call (section 5.2.4.1 of ANSI/ISO 9899-1990), and this establishes an upper limit on the number of arguments that can be passed in a call. If this number of parameters is limiting for the application, alternate means of passing data should be considered. These alternatives include the use of arrays, structures, or global variables. Arrays are always passed by reference (i,.e., using a pointer) and therefore, the limitation becomes a function of the heap space. Structures can be passed on the stack or using pointers. As is described in the following guideline, use of pointers is preferred for larger structures to minimize the possibility of a stack overflow. Global variables are also a less desirable means of passing data because of the undesirability of passing data by means of side effects. However, use of global variables may prove to be a more desirable alternative than using a structure or array if the variables have no well defined interrelationship. Section 4.4 contains additional guidelines on using global variables as a means of data interchange.

- *Use pointers to conserve stack space for larger variables.* In C and C++, parameters are put on stack when calling a subroutine. As noted above, stack memory is a limited resource, and overflowing the stack has unpredictable (and nearly always undesirable) results. ANSI C requires converting an array to a pointer when it is passed to a subroutine (Section 6.7.1, ANSI/ISO 9989-1990). However, C structures can also require a large amount of memory. Because automatic conversion to pointers is not automatically in ANSI C done for unions and structures, this conversion must be perform by the programmer as shown in the following example:

```
#define SSN_LEN          (12)
#define DAYS_PER_MONTH   (31)

typedef struct employee_struct
{
char         ssn[SSN_LEN];
short        dept_id;
short        working_hours[DAYS_PER_MONTH];
short        vacation_hours;
double       vacation_ratio;
...
}

void update_vacation_hours(employee_struct *worker)
{
short i;
short total_hours=0;

for (i=0; i<DAYS_PER_MONTH; i++)
total_hours += worker->working_hours[i];

worker->vacation_hours = total_hours+worker>vacation_ratio;
}

int main(int argc, char *argv[])

employee_struct employee;

update_vacation_hours(&employee); /* passing the pointer */

...
}
```

Dereferencing should be done inside the receiving function to manipulate the structure. When a pointer to a variable is passed to a function, any modifications to the variable inside the function are reflected in the original variable itself.

4.1.1.4 Minimizing Recursive Function Calls

Following guidelines are applicable to both C and C++

Recursion is a process in which a software module calls itself (IEEE, 1990).

Although they normally generate efficient code, recursive function calls can cause unpredictable stack memory utilization and are sources of stack overflow. Unbounded recursive function calls should be avoided in safety systems. If a recursive function has to be utilized, the stack usage should be minimized by minimizing both the number of parameters to the function and the automatic variables in the functions.

If recursion must be used, a compiler option to check for stack overflows during runtime should be invoked. This option generates code with stack checking to avoid overwriting memory when stack overflow occurs. An explicit exception handling routine should also be written to handle the stack overflow condition. If the compiler does not have stack overflow checks, an upper bound on the number of recursive function calls should be established (e.g., a limit on the length of an array being sorted), which is an appropriate fraction of the space.

4.1.1.5 Utilizing Memory-Related Functions with Boundary Checking

Following discussion applies to C

Utilizing functions with boundary checking can reduce unpredictable memory usage. Functions with a boundary limit should be used in place of functions without such a limit. Functions with a boundary limit are **strncat**, **strncmp**, and **memmove**.

Although the functions **strncpy** and **memcpy** also have boundary limit checks, they should not be used in safety systems for the reasons described in sections 4.1.1.6 and 4.1.1.7. Functions *without* a boundary limit are **strcat**, **strcmp**, and **strcpy**. Using these functions can overwrite memory outside the intended range of addresses.

In the following example, **str2** is longer than **str1**; therefore, the execution of the function can overwrite 10 bytes of memory outside **str1**.

```
char str1[20], str2[30];
. . .
strcpy(str1, str2);
. . .
```

Variables in those locations can be unintentionally changed. The function **memmove** can be used to correct this problem, as seen below.

```
#define STR1_LEN  (20)
#define STR2_LEN  (30)
```

```
. . .
char str1[STR1_LEN], str2[STR2_LEN];
. . .
memmove(str1, str2, STR1_LEN);
. . .
```

The function call here limits the bytes copied to str1 to be **STR1_LEN**, which is the size of **str1**. No matter what the contents of **str2** are, it cannot write outside **str1**.

This does not mean that the use of functions with boundary checking completely eliminates safety problems. Most memory management functions in C are confusing and could pose a safety risk if not carefully understood and protected against. As an example, consider the following function call (Spuler, 1994):

```
. . .
strncpy(s1, s2, 20);
. . .
```

This function call has a hidden danger in that **s1** will not have the **NULL** character (indicating the end of string) if **s2** contains more that 19 characters. One possible solution is that the programmer can assign the NULL character to the end of s1 immediately after the function call. The best possible solution for avoiding this type of unsafe behavior is for the programmer to create a safe and specific function for each needed memory-related action. The following example depicts such a version of the **strncpy** function (Spuler, 1994).

```
void safe_strncpy (char *s1, char *s2, int n)
{
    int i;
     for (i=0;(i<n-1) && (s2[i] != '\0'); i++) {
         s1[i] = s2[i];
     }
      s1[i] = '\0';
}
. . .
```

This will provide the programmer with a function that can be tested in advance. Where non-overlapping objects are guaranteed, the bounded forms of string library functions are safe.

A similar fault avoidance technique can be used for input functions such as **gets** as shown in the following example (Spuler, 1994) :

```
char s[5];
char *result;
 ...
result = gets(s);
if (result == NULL) {
   ...
   ...
}
```

If the user enters more than 4 characters, **gets** will overwrite the memory which does not belong to string **s**. The solution is to use a function that has a specific limit on the number of characters to be read. For this example function, **fgets** provides a more desirable alternative. The programmer can safely use `fgets(s,5,stdin)`. However, with **fgets** the newline (i.e., **\n**) will be included at the end of the string parameter, which should be replaced with a null character after the function calls.

Following discussion applies to C++ only

In C++, bounds checking may be integrated into the class definition so that the low-level functions need not carry the overhead. This is especially true for numerical analysis routines where functions like the inner product are called many times. For example, if the lengths of vector arguments are already checked against the bound before being passed to an inner product function, there is no need to add bounds checking to the function.

4.1.1.6 Use of **memmove** *for Moving Blocks of Memory*

Following guidelines are applicable to both C and C++

The memory move function **memmove**, should be used instead of the memory copy function **memcpy** (Plum, 1991). The reason is that the **memmove** function first copies the source to a temporary area, then copies the temporary area to the destination area. Thus, even if part of the source and destination overlap, the result will not be affected, and the required contents of the source will be copied to the destination. Where non-overlapping objects are guaranteed, the bounded forms of string library functions are safe.

4.1.1.7 Examining Memory at Power Up

Following guidelines are applicable to both C and C++

For C and C++ embedded system programs, volatile memory should be examined at power up. This reduces the possibility of a system running on unreliable data. The program of an embedded system should also be checked by some type of checksum code to prevent program corruption after the system is delivered.

4.1.1.8 Wrapping of Built-in Functions for Memory-Related Operations

Following guidelines are applicable to both C and C++

In order to prevent problems, built-in functions should be contained within a programmer-defined "wrapper" function which checks for input and other exception conditions (Hatton, 1994; p. 200). Another solution is for the programmer to create application-specific functions for memory related actions such as copying memory blocks.

Following discussion applies to C

The following discussion provides an example for the string copy and get string functions. Although it was noted that use of bounded functions such as **`strncpy`** are preferable to unbounded functions such as **`strcpy`**, it is not a sufficient condition in all circumstances. In the following call:

```
. . .
strncpy(s1, s2, 20);
. . .
```

there is a potential problem when **`s2`** does not have a **`NULL`** character (indicating the end of the string) if it contains more than 19 characters. The "wrapper" function created by the programmer should ensure that there is a **NULL** character to the end of **`s1`** immediately after the function call and should check for other exception conditions. Wrapping should be used for other built in functions such as **`fgetpos`**, **`ftell`**, **`bsearch`**, **`qsort`**, and **`time`** (Hatton, 1994; pp. 48 and 200).

The most fundamental solution for avoiding uncertainty from potentially undefined behaviors is that the programmer accepts a more conservative option and creates his/her own safer and possibly application-specific functions for memory-related actions such as copying memory blocks.

A example of a programmer-defined string copy function was given in section 4.1.1.5.

4.1.1.9 Proper Array Indexing

Following guidelines are applicable to both C and C++

Automatic boundary checking in C and C++ is not as strong as in some other languages. For example, there is no boundary checking for an array index during runtime. If the index of an array is outside the array boundary, it will not be detected during runtime. In C and C++, the array index starts from 0 rather than 1. In an array of 100 members, the valid indices for the array are from 0 to 99.

The following is an example of incorrect array indexing. The two last assignment statements for the **data_array** will insert values in an area of memory which are not part of the intended array.

```
#define BUF_LEN  (100)
int data_array[BUF_LEN], i;
/* initialize buffer */
for (i=1; i<=BUF_LEN; i++)
      data_array[i] = 0;       /* wrong */
data_array[BUF_LEN] = i;       /* wrong, BUF_LEN is outside of the array */
```

If the intent was to assign the final value of the array with a value of 0, then the following is the corrected code

```
#define BUF_LEN  (100)
int data_array[BUF_LEN], i;

/* initialize buffer */
for (i=0; i<BUF_LEN; i++)      /* start from 0, end at BUF_LEN -1 ( < not <= ) */
      data_array[i] = 0;

data_array[BUF_LEN-1] = i;
```

4.1.2 Predictability of Control Flow

The order in which statements in a program are executed is determined by the flow of control (Meek, 1993). Predictability of control flow is the capability to determine easily and unambiguously which path the program will execute under specified conditions.

The guidelines in this section are as follows:

- Maximizing structure
- Minimizing control flow complexity
- Initializing variables before use

•Single entry and exit points for subprograms
•Minimizing interface ambiguities
•Use of data typing
•Accounting for precision and accuracy
•Order of precedence of arithmetic, logical, and functional operators
•Avoiding functions or procedures with side effects
•Separating assignment from evaluation
•Proper handling of program instrumentation
•Controlling class library size
•Minimizing use of dynamic binding
•Controlling operator overloading.
•Protecting macros to reduce side effects
•Eliminating mixing signed and unsigned variables
•Enabling and heeding compiler warnings.

The final three guidelines do not appear as generic attributes and are specific to C and C++.

4.1.2.1 Maximizing Structure

Following guidelines are applicable to both C and C++

The generic guidelines apply. The instruction *goto* should be eliminated in safety systems. In addition, functions such as **setjmp** and **longjmp**, should also be eliminated, unless it can be guaranteed that the function that invoked setjmp has not terminated when **longjmp** is called. Since these two functions can jump from one subroutine location to another subroutine, they can cause more serious problems than the **goto** instruction (e.g. leaving variables unpopped in the stack). If a **goto** must be used, its use should be documented and justified.

The use of **goto** should be avoided except when used to jump to code processing a common error condition (usually at function exit).

4.1.2.2 Minimizing Control Flow Complexity

Following guidelines are applicable to both C and C++

The generic guidelines apply. Complicated control flow makes the program difficult to understand and maintain and is the source of unpredictable control. The following are specific guidelines.

- *Use the* **switch** *construct.* In safety systems, the **switch ... case** construct should be used to replace multiple **if ... else if ... else if ...** statements if possible (Porter, 1993). In the example

below, **test_value** is the only term used for evaluation.

```
if (test_value == 0)
{
      . . .
}
else if (test_value == 1)
{
      . . .
}
else if (test_value == 2)
{
      . . .
}
else
{
      . . .
}
```

Thus, the code could be replaced by the following:

```
switch (test_value)
{
      case 0:
            . . .
      break;

      case 1:
            . . .
      break;

      case 2:
            . . .
      break;

      default :
            . . .
      break;
}
```

- *Use brackets.* When utilizing **`if ... else`** statements, the code block should be bounded by brackets to avoid mismatches between **`if`** and **`else`**. A mismatch example is shown below.

```
if ( . . . )
      if ( . . . )
else
```

The programmer may have intended to match the **else** with the second **if,** which is quite different from the above code. By utilizing brackets, this problem could have been avoided.

In safety systems, brackets should be utilized to bound all code blocks in **if ... else** statements, as shown below.

```
if ( . . . )
{
      if ( . . . )
      {
      }
}
else
{
}
```

- *Define defaults.* When utilizing the **switch .. case** construct, a **default** case should be explicitly defined as shown in the following example.

```
#define DRAW_CIRCLE     (1)
#define DRAW_RECTANGLE  (2)
#define DRAW_TRIANGLE   (3)
#define DRAW_LINE       (4)

switch (condition)
{
      case DRAW_CIRCLE :
            /* draw circle */
            . . .
      break;

      case DRAW_RECTANGLE :
            /* draw rectangle */
            . . .
      break;

      case DRAW_TRIANGLE :
            /* draw triangle */
            . . .
      break;

      case DRAW_LINE :
            /* draw line */
            . . .
      break;

      default :
            /* display wrong condition */
            . . .
      break;
}
```

To avoid forgetting a break when another case statement is added, the **`default`** should have a **`break`** statement to terminate it (Porter, 1993).

- *Check for dead code.* Code that is inside the **`switch`** construct but does not belong to any of specified branch is unreachable or "dead" code. This type of code is usually located between the beginning of the switch and its first case branch. The programmer using **`switch`** should check the possibility of unreachable code inside **`switch`**.

4.1.2.3 Initialization of Variables and Pointers Before Use

Following guidelines are applicable to both C and C++

The generic guidelines apply. All variables and pointers should be initialized before use (Porter, 1993; Kernighan, 1978). There are three basic types of variables in C and C++: global variables, static variables, and automatic variables. Although the compiler will initialize all static variables to zero, variables with an automatic scope will contain "garbage" before the program explicitly initializes them. Global variables may or may not be initialized by the compiler. The following are specific guidelines:

- *Reinitialize automatic variables.* In the C and C++ languages, automatic variables lose their locations and their values after each function return; therefore, they should be re-initialized before they are used again. Variables should be initialized as soon as practical after their declaration.

- *Initialize global variables in separate initialization routines.* Initialization of global variables and static variables should occur in initialization routines rather than in variable declarations in real-time safety systems for the following reasons:

  1. Such routines ensure that the variables are properly set during a warm reboot. Such rebooting is a common practice and is included in a design to prevent overflows of counters and timers and to ensure that systems will not get into an infinite loop. Warm reboots are also triggered by watchdog timers and are part of recovery from infinite loops and deadlocks.

  2. To ensure deterministic reinitialization times. The timing for initialization during declarations is unspecified in the ANSI C standard.

- *Initialize global variables only once.* Global variables should be initialized once. Multiple initialization of global variables in different modules should not be done—even if allowed by the compiler and linker.

- *Do not use pointers to automatic variables outside of their scope.* Pointers to automatic variables should not be used outside of their declared scope. The value stored in a pointer to an automatic variable will contain garbage outside the function scope.

- *Initialize pointers.* Initialization problems can also occur in pointers. In safety systems, all pointer variables in C should be initialized to `NULL`, and all pointer variables in C++ language should be initialized to 0 (Plum, 1991). The pointer should then be tested for a valid value before being used. In C and C++, when a pointer is defined, it does not have a memory location associated with it. Using an uninitialized pointer will overwrite an unintended portion of memory. Incorrectly overwriting memory can cause serious problems, including system crashes.

An example of using an uninitialized pointer is shown below:

```
long *buf_ptr;

*buf_ptr = some_value;
...
```

Because **buf_ptr** is not initialized, it will contain an undetermined value based on the previous use of that memory location. This undetermined value will determine where the value **some_value** will be placed.

The correct code is as follows:

```
#define some_value (13L)
long *buf_ptr;
long value;

buf_ptr = &value;
/* initialize the pointer */
*buf_ptr = some_value;
/* assign a value */
...
```

Because **buf_ptr** is initialized to point to the value, the number will be written to the memory location of the variable rather than to an unspecified memory location.

The above example should be rewritten as follows:

```
long *buf_ptr=NULL;
long value;

buf_ptr = &value;
/* initialize the pointer */
...
if (buf_ptr != NULL)
/* test initialization */
*buf_ptr = 13;
...
```

- *Ensure that the indirection operator is present for each pointer declaration.* Each pointer should have an indirect operator (*) when it is declared (Porter, 1993). The following example shows how the C syntax facilitates omitting the indirection operator:

```
long *member_ptr, group_ptr;  /* wrong, group_ptr doesn't have
                                 indirect operator (*) */
```

The correct declaration is as follows[15]:

```
long *member_ptr;
long *group_ptr;  /* correct */
```

- *Use the ~ operator when initializing to all 1's.* When initializing all bits of an integer type to all 1's, use bitwise **not 0.** That is, use the following:

```
all_1_variable = ~0;
```

If the variable type size changes from 16 to 32, it will initialize all 32 bits to 1.

Following discussion applies to C

C assists programmers in initialization by providing the facility of specifying initial values along with declarations. However, It does not require that all objects[16] be initialized (Eckel, 1995). Moreover, in some cases, the initialization of an object is not only to assign a specific bit-pattern value to the object location, but it might need taking special actions to facilitate smooth initialization of the object's life (e.g., allocating corresponding resources to the objects).

The following discussion applies to C++ only

In C++ it is possible to consider any correlated data set as an object and provide facilities for constructing an instance of the data set and destroying the current instance of the data set in a systematic way.

[15]To reduce the possibility of forgetting the indirect mark (*), it is recommended that each pointer declaration be written in a separate line.

[16]That is, variable, structures, or arrays

4.1.2.4 Single Entry and Exit Points in Subprograms

Following guidelines are applicable to both C and C++

The generic guidelines apply. Use of single entry and exit points in functions can facilitate their validation checks. The programmer can easily use these two points to check the validity of input data entering the function and also the validity of the actions taken by the function. Multiple entry and exit points in subprograms introduce control flow uncertainties similar to those caused by the *goto* instruction (Plum, 1991; Kernighan, 1978). The following are specific guidelines.

- *Avoid multiple return statements.* Single exit points for functions is especially important in C, since C does not provide return consistency checks for functions. Some compilers will accept a function that has one branch of the code reaching the end of the function code (i.e., the last bracket) without executing any return statement (Spuler, 1994). For example, in the following routine, the returned value is undefined if the argument is negative.

```
int positive(int x)
{
   if(x>0) return TRUE
   else
   {
  /* a set of statement without any return */ )
   }
}
```

  Although acceptable within the function definition of C, this routine is unacceptable from the perspective of safety. Having only a single exit point, which is reached by all branches, eliminates the possibility of mistakenly omitting one of many **`return`** statements. If there is a compelling need for multiple entry and exit points, say to avoid **`goto`** or convoluted control flows, all such points should be clearly documented, and a rationale provided. Multiple **`return`** statements must be clearly tagged with comments. Implicit **`return`** statements should be avoided.

- *Avoiding* **`setjmp`** *and* **`longjmp`**. The ANSI C functions, **`setjmp`** and **`longjmp`** should not be used in place of a normal **`return`** statement, since they can jump outside a function and deviate from the normal control flow.[17] An addition problem in using *goto*, **`setjmp`**, or **`longjmp`** is that the initialization of the automatic variables is not performed (ANSI/ISO 9989-1990, section 6.1.2.4). The **`longjmp`** and **`setjmp`** should be used only for exception handling—and with care.

[17]It may be acceptable to use these ANSI C functions for exception handling as discussed later in this report.

- *Avoid function pointers.* Although C does not allow multiple entry points, it does allow a pointer value to be used as the address of a function to be called. Thus C allows any address to be called by assigning an integer to the function pointer.[18] Function pointers should be avoided.

The following discussion applies to C++ only

- *Restricting use of* **`throw`** *and* **`catch`**. The C++ **`catch`** and **`throw`** exception handling mechanism should be used with caution and tested thoroughly to verify the maturity and reliability of the compiler implementation.

4.1.2.5 Minimizing Interface Ambiguities

Following guidelines are applicable to both C and C++

The generic guidelines apply as indicated below. Interface errors account for a large portion of coding errors (Chillarege, 1992; Thayer, 1976). An example of such errors is reversing the order of arguments when calling a subroutine. The coding style that can reduce or eliminate the probability of misusing an interface enhances safety. The following guidelines can reduce interface ambiguities:

- *Use function prototyping* (Porter, 1993; Kernighan, 1978; Hatton, 1994). The ANSI C standard requires function prototypes with parameter definitions which make it possible to perform data type checking on parameters (ANSI/ISO 9989-1990, section 6.5.4.3). If there are no parameters, the parameter list should be declared as **`void`** to ensure proper data type checking. Also when a function has no return value, its type should be declared as **`void`**.

  The following example shows a function prototype for a function with a return type of integer and three parameters.

[18] However, this can be considered an unconstrained call rather than multiple entry points.

```
/* function prototype */
int Function1(int first_param,
              long second_param,
              int third_param);
...

/* function definition */
int Function1(int first_param,
              long_second_param,
              int third_param)
{
      int return_value;
      ...
      return return_value;
}
```

A function without a return type and parameters is shown below.

```
void Function2(void);   /* function prototype */
...
void Function2(void)          /* function definition */
{
      ...
}
```

- *Do not use functions that accept an indefinite number of arguments.* A function with a variable number of arguments is difficult to verify. Moreover, the behavior of a function that accepts a variable number of arguments and is called without a function prototype that ends with an ellipsis is also undefined (Hatton, 1994; p. 50).

- *Order parameters so that different data types are alternated.* This practice reduces the chance that two adjacent parameters will be placed in an incorrect order. Judicious use of structures or classes may reduce the number of function arguments by grouping together several items of similar kind, e.g., height/ width/ length or row/ column.

- *Ensure that arguments are of a compatible type with the function prototype.* The behavior of a function called with a function prototype when the function is not defined with a compatible prototype is not defined in C (Hatton, 1994; p. 50).

- *Avoid use of variable length argument lists.* It is preferable to use default values for function arguments than to use a variable number of arguments. Exceptions can be made in the case

of **printf**, **scanf**, and other similar library functions.[19]

- *Test the validity of input arguments at the beginning of a routine and test the validity of the results before returning from the routine.* Such testing is important for avoiding errors that can compromise the integrity of the system (Kernighan, 1978). An example is shown below.

```
double value, result;
...

/* check for valid input range */
if ((value > -1.0) && (value < 1.0))
      result = acos(value);
else
{
      /* report input range error */
      ...
}
```

Range checking inside a function is preferred. The checking in the example above is outside the function **acos** because the function is an ANSI C library function and is provided by compiler manufacturers.

- *Using byte alignment of compilers.*[20] Most C and C++ compilers allow programmers to determine how a variable is aligned in structures and unions. These structures and unions can be parameters, passed by their pointers, or can be written to files to interface with other programs. A consistency-of-alignment method should be included in the project software development guidelines. Byte alignment, which saves resources such as memory and disk space, should be utilized in small-scale safety systems with limited resources. Using word alignment or double-word alignment when required by the CPU is acceptable.

- *Eliminate expressions in parameter passing to subroutines or macros.* Since the order of evaluating parameters is unspecified in the C language (Annex G of ANSI/ISO 9989-1990), using expressions as parameters raises safety concerns. For example:

```
short param1, param2;
...
function1(param1++, param2 = param1 + 1); /* wrong */
```

[19]However, see the earlier guideline on the use of wrapper functions

[20]the storage of the adjacent data in the following byte (as opposed to the following word or double word).

The following section of code corrects the problem in the above example.

```
short param1, param2;
...
param1++;
param2 = param1 + 1;
function1(param1, param2);
```

- *Eliminate Increment (++) and decrement (--) operators from macro and function calls.* Removing the increment and decrement operators from macros and functions eliminates the possibility of undefined expressions. Although they provide a more efficient way of adding 1 or subtracting 1 to a variable, their use in argument lists raises safety concerns. They should only be used in isolated expressions for incrementing loop counts. Table 4-1 illustrates problems caused by increment and decrement operators in function calls.

Table 4-1 Examples of Problems Caused by Increment and Decrement Operators

Problem	Problem Syntax and Corrected Syntax	Comment on Problem Syntax

Unspecified behavior	Problem Syntax:
	`function_call (i++);`
	
	Corrected Syntax:
	`i++;`
	`function_call(i);`
		Whether the variable `i` is increased before the function call or after is unspecified (Spuler, 1994).

Unspecified behavior	Problem Syntax
	`function_call ((i++));`
	
	Corrected Syntax:
	`i++;`
	`function_call(i);`
		The extra parentheses do not guarantee when the variable `i` is increased. The variable still may be increased before starting the function_call, or after the function is executed (Spuler, 1994).

Unintended change	Problem Syntax:
	`#define MAX(x, y)(x>y)? x:y`
	`up_limit = MAX(++i, j);`
	
	Corrected Syntax
	`++i;`
	`up_limit = MAX(i, j);`
		This expression will be expanded by the preprocessor as:
		
		`up_limit = (++i > j) ? ++i : j;`
		
		Variable `i` could be increased by 2. The first increment happens at (++i > j); the second one happens when the comparison is true, and ++i is assigned to `up_limit`. Depending upon the values of `i` and `j`, `i` can be increased by 1 or 2, which is unlikely to be the intent of the programmer.

- *Use bit masks, not bit fields.* Bit fields and masks are used for reading setting status registers in hardware and for reporting status to other portions of the system. Bit field assignment is implementation defined (Section 6.5.2.1 ANSI/ISO 9989-1990). When a bit field is defined in a program, a compiler can assign any bit(s) to it, either higher bit(s) in a memory or lower bit(s). This may create interface problems when bit field variables are written to a file and the file is accessed by another program written in another language or compiled by another compiler (Porter, 1993; Hatton, 1994). Problems may also be created when the variable is communicated to another system. Bit field variables should not be utilized in safety systems, a bit mask should be instead. The following is an example of the use of bit field variables in which short integers are used to store the value of a send and receive flag.

```
#define BUFSIZE   (1024)
typedef struct comm_struct
{
      short send_flag          : 1;
      short receive_flag       : 1;
      unsigned char      buf[BUFSIZE];
};
comm_struct comm_var;
...
if (comm_var.send_flag)
{
...
}
if (comm_var.receive_flag)
{
...
}
...
```

The problem with this code is that should there be a need to port it to another system or compiler, it us unclear whether the placement of the bits will be properly interpreted by the CPU during runtime. A better practice is to explicitly place and check bits using a bit mask as shown below:

```
#define BUFSIZE   (1024)              /* buffer size */
#define SEND_FLAG (0x01)              /* bit 0 */
#define RECEIVE_FLAG    (0x02)        /* bit 1 */
typedef struct comm_struct
{
      int   flag;       /* bit 0: SEND_FLAG, bit 1: RECEIVE_FLAG*/
      unsigned char buf[BUFSIZE];
};
comm_struct comm_var;
...
if (comm_var.flag & SEND_FLAG)
{
```

```
...
}
...
if (comm_var.flag & RECEIVE_FLAG)
{
...
}
...
```

4.1.2.6 Controlled Use of Data Typing

Following guidelines are applicable to both C and C++

Acceptance of data that differ from those intended for use by a program can cause system failures. The following measures should be taken to reduce data typing errors.

- *Limit the use of implementation-dependent types.* Data types whose sizes are machine- or compiler-dependent types should be used with caution. For C, these types are **`float`**, **`char`**, and **`int`**. Unrestricted use of these data types could cause interface and portability problems. The utilization of these data types as Input/Output variables or as structure and union fields should be avoided in safety systems. Data type **`float`** should be replaced by **`double`** and data type **`char`** should be replaced by either **`signed char`** or **`unsigned char`**. In many cases, data type **`int`** should be replaced by **`short int`** or **`long int`** if the actual size of these types are known. This data type is used in many built-in function and procedure calls, as well as in externally developed libraries. Thus, it is not possible to eliminate **`int`** from safety-critical code. However, **`int`** should be used with care, and all occurrences should be clearly documented. When possible, variables should be declared as **`short`** or **`long`** (which are of known size for all machines with a given word length), and then cast to the required **`int`** type for interfacing. Though popular, the data type **`int`** is not machine- or compiler-independent. If the lengths of implementation-dependent (integer or floating point) types have an impact on the operation of the software, this must be documented.

- *Minimize the use of type conversions and eliminate implicit or automated type conversions.* In addition to the general guideline to limit the number of explicit conversions, a tighter restriction should be placed on conversions of pointers. Use of one pointer should not cast a different type of pointer (Plum, 1991).

- *Avoid the use of mixed-mode operations.* Operations using multiple data types should be avoided. If such operations are necessary, they should be clearly identified and described using prominent comments in the source code. Explicit casts should be used if practical in

order to make the designer's intentions clear.

The following example demonstrates the potential problems:[21]

```
#define BUF_SIZE    (32)
signed char count, in_buf[BUF_SIZE];
int  scale, result;

...
count = in_buf[0];
scale = 2;
result = 2 * count * scale;
```

Since the range of a **signed char** type is from -128 to 127, the expression can generate unexpected results. For example, when **count** is 127, 2 * **count** is 254 which is -2 as a **signed char** variable. The result is -4 after -2 * **scale**, which is different from the expected 2 * 127 * 2 or 508.

The following are two possible corrections:

Correction 1: Changing the variable type

```
#define BUF_SIZE  (32)
signed char in_buf[BUF_SIZE];
int count, scale, result;     /* count is int now */

...
count = (int) in_buf[0];
result = 2 * count * scale;
```

Correction 2: Casting the variable type

```
#define BUF_SIZE  (32)
signed char count, in_buf[BUF_SIZE];
int scale, result;

...
count = in_buf[0];
result = 2 * (int)count * scale;
```

The first correction approach (changing the variable type) is preferred since it reduces the type conversion when the variable *count* is used in multiple places.

[21]The reader should note the recommended restrictions on the use of **int** in the previous paragraph

- *Use a single data type in evaluations and relational operations.* Expressions involving arithmetic evaluations or relational operations should have either a single data type or the proper set of data types for which conversion difficulties are minimized (Porter, 1993). This guideline is related to the above discussion on minimization of mixed-mode operations.

- *Avoid the use of* **`typedefs`** *for unsized arrays.* Although legal, such constructs are obscure badly supported, and error-prone (Hatton, 1994, p. 75).

- *Avoid multiple declarations of one identifier with several types.* Even if multiple declarations result in no compiler errors, they may be a source of confusion or even of undefined behavior.

•*Avoid mixing signed and unsigned variables.* Mixing signed and unsigned variables in arithmetic and logical operations raises safety concerns and should be avoided in safety systems. Explicit casts should be used if practical in order to make the designer's intentions clear. Mixing signed and unsigned variables in arithmetic and logical operations can create unexpected results (Porter, 1993). A hexadecimal number FFFF is −1 in a signed 16-bit integer and is 65535 in an unsigned 16-bit integer. This difference can change the outcome of a comparison and the result of an arithmetic operation. Mixing signed and unsigned variables in arithmetic operations can also create overflow problems. Table 4-2 illustrates two problems with mixing signed and unsigned variables.

Table 4-2 Problems in Mixing Signed and Unsigned Variables

Problem	Problem Syntax	Comment on Problem Syntax

Comparison problem	`int i;`
	`unsigned int ui;`
	`i = -1;`
	`ui = 2;`
	`if (i > ui )`
	`  {`
	`  /* do A */`
	`  }`
	`else`
	`  {`
	`  /* do B */`
	`  }`
		When comparing a signed variable with an unsigned variable, the compiler will automatically convert the signed value to an unsigned value. The result is just the opposite of what the programmer intended to do. In this example, variable **`ui`** needs to be cast as a signed integer. In some other cases, the signed variables need to be cast as unsigned variables. Sometimes, both variables need to be cast as a long integer. A signed 16-bit variable can be cast as an unsigned variable only when its value is greater than or equal to zero (nonnegative number), and an unsigned 16-bit variable can be cast as a signed variable only when its value is less than hexadecimal 7fff or decimal 32767.

Division problem	`int i, result;`
	`unsigned int ui;`
	
	`i = -1;`
	`ui = 2;`
	`result = i / ui;`
		When there is a signed and an unsigned variable in a division, the compiler will automatically convert the signed value into an unsigned value. The value −1 will be interpreted as 65535. The result is 32767, not the expected 0. To solve this problem, the unsigned variable **`ui`** needs to be cast as a signed variable. In some other cases, casting the **`unsigned int`** to signed may not be correct. The proper solution is to eliminate mixing signed and unsigned variables in division operations.

- *Limit use of indirect addressing.* Validation of indirectly addressed data should be performed prior to setting or using it to ensure the correctness of the accessed locations. Use of void pointers should be limited.

- *Do not declare the same identifier for multiple incompatible types.* The behavior of a program using a data type or a function with incompatible types is not defined (Hatton, 1994; p. 49).

4.1.2.7 Precision and Accuracy

Following guidelines are applicable to both C and C++

Safety related software must provide adequate precision and accuracy for the intended application (IEEE Std-7-4.3.2-1993). At the same time, the software must also tolerate the inconsistencies emerging from operations on floating point numbers. The following are specific guidelines for C and C++.

•*Use double precision.* Data type **`double`** should be used for floating point variables in safety systems. As noted earlier, the **`float`** data type should not be used because it may not provide adequate precision and accuracy and because it limits portability.

•*Account for floating point properties in relational operations.* The equality comparisons on floating-point numbers should be avoided in safety systems since the machine representation of floating-point numbers may lack precision and may have a small residual error. Inequality comparisons should be utilized and equality comparisons should be avoided on floating-point numbers (Porter, 1993; Kernighan, 1978).

The following example demonstrates the potential problems.

```
double value;   /* temporary variable for return value */
...
if (value == 0.0)
{
      /* calculate something */
}
```

The condition **`value == 0.0`** in the above example is likely to be false because of rounding errors, even if the value is expected to be zero. The condition should be modified as follows:

```
#define FLOATING_POINT_TOLERANCE    (0.00001)
if( (value < (0.0 + FLOATING_POINT_TOLERANCE)) &&
      (value > (0.0 - FLOATING_POINT_TOLERANCE)))
{
      /* calculate something */
}
```

- *Account for truncation in integer operations.* If a floating-point arithmetic operation can generate truncation and rounding errors, integer arithmetic may generate such errors more often. Integer truncation errors are generated by division. In C and C++ languages, the results of integer divisions are always truncated (e.g. 5/3 = 1). If a result is negative, even the method of truncation is implementation dependent. The result of -5/3 can be -2 or -1, depending upon the compiler. The truncation method that a compiler uses may not be the same as the truncation method that a developer or a reviewer assumes is being used. Truncation errors can cause safety concerns when the results with truncation are used in comparisons and conditions for control decisions. Therefore, a rounding-off technique should be utilized. A typical rounding-off method is to perform the division in double, add 0.5 to the result, and cast the result back to an integer, as seen in the following example.

```
long  int result;
long  int total_energy;
long  int stations;

result = (long int) ((double) total_energy / (double) stations + 0.5);
```

However, this rounding off method may apply to positive results only. Whether it applies to negative results will depend on the combination of how the compiler handles the division and how a developer wants the rounding off to be performed. The negative results may require subtracting 0.5 instead of adding 0.5 for rounding off.

- *Account for optimization.* Within the rules of precedence, order of evaluation of sub-expressions in C is implementation-defined. This may lead to unexpected results in the presence of optimized code being generated by the compiler. This is especially an issue with floating point computations. A compiler might replace ((1.0+x)-x) with 1.0 at compile time, when the floating point rounding error is what the program is trying to compute. Note that the above optimization is guaranteed to always be correct for integer types.

- *Ensure that arithmetic conversion produces a result that can be represented in the space provided.* When conversion or casting is necessary, care must be taken to ensure that enough memory space is available. For example, if an integer floating-point expression is cast down

or converted to a shorter data type, care must be taken to ensure that the value is representable in the shorter type (Hatton 1994, pp. 55 and 56).

4.1.2.8 Use of Parentheses Rather Than Default Order of Precedence

Following guidelines are applicable to both C and C++

Generic guidelines apply. The default order of precedence of arithmetic, logical, and other operations varies between languages. Developers and reviewers may make incorrect precedence assumptions when explicit precedence relations are not used, particularly in complex expressions (Kernighan, 1986). Also, an overloading operator in C++ may change the precedence. (Section 4.1.2.13 for a related discussion.). The following are specific guidelines.

- *Use parentheses in bitwise operators.* In the C and C++ languages, bitwise operators have lower precedence than logical operators. Parentheses must be utilized in comparisons and conditions that have bitwise operators. For example :

```
if ((I & 0x01) == (j | 0x02))
/* do something */
```

- *Use parentheses in comparisons and conditions.* Parentheses must also be utilized in comparisons and conditions that have assignment operators (Plum, 1991) because assignment operators have lower precedence than logical operators. This is shown in the following example.

```
/* read a key from keyboard */
if ((key = getch()) == FUNCTION_KEYS)
key = getch();
```

- *Use parentheses in macros.* Parentheses can be used to protect macros to reduce side effects. Using macros can make code more readable and can reduce repetitive code. However, without proper parentheses, macros can introduce side effects, as shown below.

```
#define square(x)       x * x

int delta;
int sqr;

...
sqr = square(3+delta);  /* problem */
```

The preprocessor will expand the above expression as:

```
sqr = 3 + delta * 3 + delta;
```

which is equivalent to:

```
sqr = 3 + (delta * 3) + delta;
```

This is completely different from the square of **`3 + delta`**. The problem shown in the example is that the macro **`square(x)`** is not protected. To ensure that a macro is fully protected, the expression should be parenthesized as follows:

```
#define square(x)       ((x) * (x))
```

In some cases, use of parentheses may result in lower readability. If parentheses are excessive, then macros should not be used and alternative forms should be employed to achieve readability.

- *Ensure that the values of expressions do not depend on the order of evaluation.* As noted above, within the rules of precedence, order of evaluation of sub-expressions in C/C++ is implementation-defined. Unlike some other languages, for example, FORTRAN, parentheses in C/C++ only override precedence, and have *no* other effect on order of evaluation. Where order of evaluation is critical, for example, in floating point computations, expressions should be broken up into multiple statements, since the end of a statement is a sequence point in C/C++, and the ordering of sequence points is guaranteed to be preserved.

Any expression potentially having side-effects, e.g., containing a function evaluation, should not depend upon order of evaluation. Generally speaking, integer expressions without side-effects are independent of order of evaluation. Both C and C++ use "short-circuiting" (Spuler, 1994) in the evaluation of logical expressions. That is, as soon as the final value of an expression is determined (for example, a zero value in an AND expression is encountered), the remaining sub-expressions are not evaluated. Other unevaluated parts of the expression are ignored. Although short-circuiting increases the efficiency of the evaluation procedure, it may have unexpected results if not used carefully as illustrated in the following example:

```
if (x < y && (ch=getchar()) != EOF)
{
 ....
 }
```

4.1.2.9 Avoiding Functions or Procedures with Side Effects

Generic guidelines are applicable.

4.1.2.10 Separating Assignment from Evaluation

Following guidelines are applicable to both C and C++

Generic guidelines apply to C and C++. The following are language-specific guidelines.

- *Separate relational and assignment operators.* The assignment operator is one equal sign, "="; the relational operator is a double equal sign "==". An assignment statement, such as **`assign_this = value`** should be separated from an evaluation expression such as **`if(value1 == value2)`** (Porter, 1993). The following two valid statements (in both C and C++) illustrate the potential problem:

```
/* Example 1 */
     while (evaluation = 1)
     {     value1 == value2;
           ...
     }

/* Example 2 */
     while (evaluation == 1)
     {     value1 = value2;
           ...
     }
```

Example 1 causes an infinite loop in the program because the evaluation occurring immediately after the `while` is always true.

If it is not possible to avoid separation of assignment and evaluation statements, the following mitigating measures should be used:

1. Parenthesize any embedded assignment in an evaluation expression.

2. Ensure that the order of evaluation does not affect the value of the assignment statement. This includes accounting for the "short circuit" evaluation mechanism used in C and C++.

4.1.2.11 Proper Handling of Program Instrumentation

Following guidelines are applicable to both C and C++

Generic guidelines apply. Program instrumentation collects and outputs certain internal state values of a program during execution and allows the developer to ascertain that particular aspects of the specification have been correctly implemented (Liao, 1991).

4.1.2.12 Control of Class Library Size

The following discussion applies to C++ only

Generic guidelines apply to C++. There are two specific guidelines.

•*Limitation of class library size.* Limiting the library size minimizes the chance of a system becoming unmanageable or having large performance penalties because it has too many classes and objects (Cuthill, 1993).

•*Avoiding multiple inheritance.* Multiple inheritance should not be used in safety systems (Porter, 1993) because of ambiguities (Cargill, 1992) and maintenance problems (Hatten, 1994). An example of ambiguity is shown below:

```
class file_base
{
      protected:
            void Init();
            ...
};

class io_port
```

```
{
      public:
            void Initialization
            {
                  Init();
            }

      ...

      private:
            void Init();
};

class file_io: public file_base, public io_port
{
      public:
            file_io()
            {
                  Init();     // ambiguous
            }

};
```

This ambiguity may be detected by some compilers, but it may not be detected by others.

4.1.2.13 Minimizing Use Of Dynamic Binding

The following discussion applies to C++ only

The generic guidelines apply. Binding denotes the association of a variable with a class. Dynamic binding allows the name/class association to be deferred until the object designated by the name is created at runtime. The unpredictability of the name/class association creates safety concerns, reduces the predictability of the runtime behavior of an object oriented program, and complicates debugging, understanding, and traceability.

4.1.2.14 Control of Operator Overloading

The following discussion applies to C++ only

Generic guidelines apply to C++. Operator overloading can improve readability and reduce complexity by allowing an object behavior to be used for different data types. However, overloading can also be problematic from the perspective of predictability because the precedence of one operator may not be consistent (as will be described below). When using operator overloading, the following guidelines should be followed (Porter, 1993):

- *The meaning of an overloaded operator should be natural, not clever* (Cargill, 1992. Binkley, 1995). It is generally recognized that there are advantages to localizing related elements in a single module. If any of the operators for a class are redefined, the operator's original meaning should be preserved. That is, if addition operator + is redefined for a class, the operator should still have the sense of adding something to the class instance. This is a case where operator overloading is useful for achieving uniformity across data types.

- *Operation order should be ensured by parentheses* (Porter, 1993; Kernighan, 1978). When performing floating-point arithmetic, bitwise exclusive OR operator ^ may be redefined as an exponentiation operator. However, a bitwise exclusive OR operator has different precedence than an exponentiation operator.[22] When a floating-point exponentiation operator is overloaded to a bitwise exclusive OR operator, it changes the precedence of such operators for exponentiation, as seen in the following example.

```
double base1, base2, sum_of_squares;

base1 = 3.0;
base2 = 4.0;
sum_of_squares = base1^2.0 + base2^2.0;
```

Since an addition operator has higher precedence than a bitwise exclusive OR operator, the compiler will evaluate the expression as:

```
sum_of_squares = (base1^(2.0+base2)^2.0);
```

which is different from the expected result of 25.0. To get the correct results, parentheses should be used to keep the precedence of the exponentiation operator, as indicated by the following:

```
base1 = 3.0;
base2 = 4.0;
sum_of_squares = (base1^2.0) + (base2^2.0);
```

•*Explicitly define class operators.* Since the default constructor, copy constructor, destructor, and the operators **operator=, operator&,** and **operator<comma>** all have default meanings, they should be explicitly defined in every class. To avoid unwanted implicit calls to these functions, declare them private (Binkley, 1995).

---

[22] A bitwise exclusive OR operator has lower precedence than an addition operator while an exponentiation operator has higher precedence than an addition operator.

•*Ensure consistency of pointer operators.* For a class that defines the operators **`operator->`**, **`operator*`**, and **`operator[]`**, ensure the equivalences between **`p->m`**, **`(*p).m`**, and **`p[0].m`**. Otherwise this will avoid unexpected errors when programmers assume the equivalence (Binkley, 1995).

•*Ensure consistency of increment operators.* For a class that defines the operators **`operator+`**, **`operator=+`**, **`operator++`**, and **`operator++(int)`**, ensure the equivalence of **`x=x+1`**, **`x+=1`**, and **`++x`** and their relationship to **`x++`**. Note that the use of ++ is generally discouraged (Binkley, 1995).

4.1.2.15 Enable and Heed Compiler Warnings

Following guidelines are applicable to both C and C++

Both C and C++ are complex enough that programmers should employ all available mechanisms to create a safe programs. Although relying on compilers alone is not a useful practice, warnings produced by compilers are a valuable source of information on abnormal and potentially dangerous parts of the program. All optional compiler warning should be enabled. Every warning messages should be analyzed carefully.

4.1.3 Predictability of Timing

Predictability of timing is crucial in a safety system used in real time control (Kopetz, 1993; Leveson, 1994). Some related guidelines were discussed in the previous subsections including:

- Control of class library size (section 4.1.2.12)
- Minimizing dynamic binding (section 4.1.2.13)
- Control of operator overloading (section 4.1.2.13).

Two additional guidelines are:

- Minimizing the use of tasking
- Minimizing the use of interrupt-driven processing.

These additional guidelines are discussed below.

4.1.3.1 Minimizing the Use of Tasking

Following guidelines are applicable to both C and C++

Although multitasking provides an attractive model for concurrent processing, its use is undesirable in safety systems for the following reasons:

1. Multitasking creates uncertainties in execution, timing, and resource utilization.

2. C and C++ do not support multitasking. Their standard library functions may not be reentrant functions (ANSI 9984-1990, section 5.2.3). Using those functions in multitasking environments may therefore cause unanticipated results.

Tasking requires compelling justification.

4.1.3.2 Minimizing the Use of Interrupt Driven Processing

Following guidelines are applicable to both C and C++

Using interrupt-driven processing to handle the acceptance and processing of plant and operator input can reduce average response time, but usually leads to nondeterministic maximum response times. If an interrupt-driven processing has to be used, the processing time within interrupt service routine should be minimized.

When interrupt driven processing must be used, the following guidelines mitigate the associated risk:

- *Limit interrupt processing*. The code and processing time within the interrupt service routine should be minimized. Any data checking and data processing should be done after the interrupt processing. Typically, a circular buffer can be used to store the incoming data (buffers should be large enough to avoid data overruns).

- *Limit function calls*. Function calls within interrupt service routines should be minimized, and only reentrant functions should be called by interrupt service routines. ANSI/ISO C standard does not guarantee any standard library functions to be reentrant (ANSI/ISO 9989-1990, section 5.2.3).

  For example :

```
/* data buffer size */
#define BUFSIZE   (2048)
```

```
/* Buffer index wrap around mask. This wraparound method works only when
the buffer size is a power of 2 */
#define BUF_INDEX_MASK  (BUFSIZE - 1)

/* COM port address */
#define COM_PORT_ADDR   (0x2f8)

/* COM port interrupt vector address */
#define COM_ISR_ADDR          (12)

/* time out in 2 second */
#define TIMEOUT_LIMIT         (2*CLOCK_PER_SECOND)


/* local variables */
static int data_in_index;
static unsigned char data_buf[BUFSIZE];


/* local function prototype(s) */
static void Init(void);
static interrupt new_com_isr(void);


/*-------------------------------------------------------------------
  Description:    This function initializes the COM port, interrupt
                  vector, and buffer index variables.
  input var:      none
  output_var:     none
  return:   none
  global var:
-------------------------------------------------------------------*/
static void Init(void)
{
      data_in_index = 0;

      /* other initialization */
}


/*-------------------------------------------------------------------
Description :     This function is called when there is an RS232 (COM
                  port) interrupt.  It reads a byte from the COM port and
                  saves it in the data buffer.
input var:  none
output var: none
return val: none
global var: data_buf -- new data is save int the buffer
            data_in_index -- used and modified.
-------------------------------------------------------------------*/
static interrupt new_com_isr(void)  {
```

```
data_buf[data_in_index++] = inp(COM_PORT_ADDR);
data_in_index &= BUF_INDEX_MASK;
}

main ()
{
      int return_code = 0;
      interrupt orig_com_isr;
      cloct_t last_time;

      /* save the original interrupt service routine address */
      orig_com_isr = get_vector(COM_ISR_ADDR);
      data_out_index = 0;
      Init();


      /* set new interrupt service routine */
      set_vector(new_com_isr);

      last_time = clock();
      while ((clock() - last_time) <= TIMEOUT_LIMIT)
      {
            if (data_in_index != data_out_index)
            {
                  /* process new data */
                  data = data_buf[data_out_index++];
                  data_out_index &= BUF_INDEX_MASK;
                  ...

                  /* update time out count */
                  last_time = clock();
            }
      }

      /* restore original interrupt service routine */
      set_vector(orig_com_isr);

      /* exit this program */
      return return_code;
}
```

Interrupt routines may be required to handle inputs from external devices, but such routines should be kept as short and simple as possible. Masking of interrupts, nested interrupts, and interrupt processing in general all cause non-deterministic behavior. Also, some form of locking or mutual exclusion may be required when using interrupts.

4.2 Robustness

Robustness refers to the capability of the software to survive off-normal or other unanticipated conditions, or the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions (IEEE, 1990). Since unanticipated events can happen during an accident or excursion, it is vital for a safety system to survive an accident and continue working. This section discusses the following topics related to robustness:

- Controlled use of software diversity
- Controlled use of exception handling
- Input and output checking.

4.2.1 Controlled Use of Software Diversity

Following guidelines are applicable to both C and C++

The generic guidelines apply to both internal and external diversity. There are no additional language-specific guidelines.

4.2.2 Controlled Use of Exception Handling

An exception is an event that causes suspension of normal program execution (IEEE, 1990). Exception handling deals with abnormal system states and input data (IEEE, 1993). This section discusses guidelines related to the following attributes:

- Local handling exceptions
- Preservation of external control flow
- Uniformity of exception handling.

4.2.2.1 Local Handling of Exceptions

Following guidelines are applicable to both C and C++

The generic guidelines apply. Exceptions should be handled locally.

Propagation of exceptions through several levels of a program can cause the precise nature of the exception to be misinterpreted at the place where the exception handling is implemented. This cause of system failure can be avoided if exceptions are handled locally. This section describes suggested

approaches to local handling of the following types of exceptions: addressing, data, input/output, overflow/underflow, operation, and protection.

- *Addressing exceptions.* Addressing exceptions can be caused by an uninitialized or improperly set pointer. For example, an uninitialized static pointer will have **NULL** as its value. Writing to the uninitialized pointer will overwrite system memory which can cause catastrophic system failure. There is no way to recover from such a condition. Hence, addressing exceptions must be prevented as described in section 4.1.1.

•*Data exceptions.* Data exceptions can be data-domain errors or data-range errors. Both categories can occur when calling a library function. After calls to any mathematics functions in the standard library, the variable **errno,** which is declared in the **error.h** file, should be checked for possible data exceptions.

•*Input/output exceptions.* Input/output exceptions can be related to files. After a function call to open a file **(fopen)** or to seek a location in a file **(fseek)**, the result should be checked to verify if the function call is successful. Function **fopen** can fail when the file does not exist or when the file open mode and the file attributes do not match (e.g., to open a file in write mode, but the file is read only). Function **fseek** will fail if the specified location does not occur in the file. If the function call fails, the program should not continue without handling the exception condition related to the failure.

Before closing a file, the program should verify whether the file is currently open to avoid accidentally closing another stream. If the file is not currently open, the file pointer is **NULL**, and a catastrophic failure may occur. For example, **NULL** can be interpreted as stream number 0 which is the keyboard in MS-DOS. Closing a **NULL** pointer can lock up the keyboard and disable the user interface. When the system requires a user input, it cannot receive it because the keyboard is locked. The system cannot do anything until it is reset.

An input/output exception handling example is shown below:

```
#define DATA_FILE "safety.dat"
#define OPEN_FILE_ERROR "ERROR==>cannot open file %s"

      FILE *fp;

      fp = fopen(DATA_FILE, "w+t");
      if ( fp == NULL)
      {
            /* report file open error */
            cprintf(OPEN_FILE_ERROR, DATA_FILE);

            /* exception handling */
            ...
```

```
}
else
{
}
...

/* if the file is opened, close it */
if (fp != NULL)
      fclose(fp);
```

•*Overflow and underflow exceptions.* Some overflow and underflow exceptions can also be checked by examining the variable error, especially after calling a mathematics library function. Without checking the variable error, the result cannot be assumed to be correct. One of the most common such exceptions is divide by zero. To avoid this condition, the denominator should be verified as being nonzero before a division is be performed.

•*Operation exceptions.* Operation exceptions can be race condition, data or address bus busy, device busy, device idle, or lack of memory. A timer with an expiration time (deadline) is a technique to handle operation exceptions. For example, there should be a deadline or "time out" when the system is waiting for a response from a remote station. The action after the time-out should be well defined.

•*Protection exceptions.* A protection exception is an abnormal event caused by system locks on shared resources such as files. An example is that an application is trying to open a file while the file is locked by another application. When such an exception happens, a retries should be performed up to a predefined limit. The likelihood of such an exception can be reduced by opening files only when they are needed, locking only required records rather than the entire file, or opening a file in the correct mode (i.e. do not open read-write mode when the operation only requires a file read).

If it is not possible to place exception handling locally, thorough testing and analysis is necessary to verify the proper behavior of the program in the exception state.

4.2.2.2 Preservation of External Control Flow

Following guidelines are applicable to both C and C++

Generic guidelines apply. Interruption of control flow external to the routine in which the exception was raised creates uncertainty in the execution subsequent to the exception handling. Safety is enhanced by preservation of control flow external to the module responsible for the exception. When an exception occurs, the external control flow should be preserved. This requires the module not only to handle the exception internally, but also to set flags. These flags are used for external

communication. If it is not possible to preserve external control flow, then thorough testing and analysis should be used to verify behavior.

Asynchronous exceptions can only be handled by catching signals. The effect of handling the exception in this way can be localized to the module containing the handler, and flags can be used to communicate the error to other modules. Additional related comments on the use of setjmp/longjmp in error handling are in section 4.1.2.1.

4.2.2.3 Uniformity of Exception Handling

Following guidelines are applicable to both C and C++

Generic guidelines apply. Exceptions should be handled uniformly. Section 4.2.2.1 described the likely types of exceptions to be encountered in C and C++ and how they can be handled locally. The following are additional language-specific guidelines on handling exceptions uniformly.

- *Rely on signals and traps rather than operating system features for handling of exceptions.* Some commercial real-time operating systems that may be incorporated into safety systems have additional support for exception handling. However, in order to ensure uniform and predictable handling of exceptions, these operating system features should be used only as a last resort in safety systems. It is preferable that signals and traps related to exceptions be intercepted and handled by the safety software unless the exception handling standard and methods of an operating system are well documented and understood.

- *Use `throw` and `catch` in favor of `setjmp` and `longjmp` in C++.* C uses **`setjmp`** and **`longjmp`** in the Standard C library for exception handling purposes. The problem with these functions is that it is virtually impossible to recover effectively from a complicated exception condition (Plauger, 1995). However, C++ provides a cleaner exception-handling mechanism using the **throw, catch** mechanism (Plauger, 1995). C++ programmers should make use of this uniform exception-handling mechanism, although compiler implementations may need to be validated.

4.2.3 Input and Output Checking

Following guidelines are applicable to both C and C++

Generic guidelines apply. A specific guideline relating to the use of pointers for input or output operations.

- *Check pointers before use.* Pointers should be checked before use to ensure that the location from which data are being read is valid. Such checking is shown in the following example:

```
FILE *fp;                          /* define a pointer */

fp = (FILE *) NULL;                /* initialize the pointer */

fp = fopen( ... );                 /* assign the pointer */

if (fp != (FILE *) NULL)           /* check the pointer */
{
      ...
}

if (fp != (FILE *) NULL)           /* check the pointer */
{
      fclose(fp);
      fp = (FILE *) NULL;          /* clear the pointer */
}
```

4.3 Traceability

Traceability refers to attributes of safety software that support verification of correctness and completeness as compared to the software design. The intermediate attributes for traceability are as follows:

•Readability
•Minimizing use of built-in functions
•Minimizing use of compiled libraries
•Utilizing version control tools
•Utilizing comments and internal documentation

Because readability is also an intermediate attribute of maintainability, it is discussed in Section 4.4. C and C++ specific guidelines for the latter two attributes are discussed below.

4.3.1 Minimizing the Use of Built-In Functions

Following guidelines are applicable to both C and C++

The generic guidelines apply. C and C++ include built-in functions, sometimes called intrinsic functions (Koeman, 1995) for frequently used programming tasks in order to maximize programmer productivity. The use of these functions raises safety concerns for the following reasons:

1. The requirements for developing those built-in functions may not be the same as those of the safety systems.

2. The input and output data validation and exception handling may not be the same as that needed in safety systems.

3. The number of built-in functions may vary from one compiler to another. A function supported by one compiler may not be supported by another compiler. For example, compilers for embedded systems generally do not support all ANSI C standard functions.

Because of these concerns, the use of built-in functions should be minimized. When built-in functions are used, their use should be supported with documented testing and tracking of anomalies. Although the built-in functions should be minimized in safety systems, it may not be possible to eliminate all built-in functions because a language is not complete without those functions and some task may not be able to be performed. When built-in functions are used, only functions in ANSI C Standard should be called. Wrapper functions should be used for potentially problematic standard functions (Hatton, 1994).

4.3.2 Minimizing the Use of Compiled Libraries

Following guidelines are applicable to both C and C++

The generic guidelines apply. Compiled libraries can be supplied by compiler vendors or third parties to support input/output operations or mathematical operations which are not defined constructs within the basic language. All concerns discussed in sections 4.3.1 and 4.4.1 also apply to compiled libraries. Like built-in functions, the use of compiled libraries should be minimized. In addition, libraries provided by commercially oriented vendors may not have been developed with the same safety standards as the project for which they are used. The following are additional language-specific guidelines.

- *Ensure that names in externally developed libraries are distinct from those in the compiler or those developed within the project.* Functions with the same names but different purposes—or even the same purpose and different characteristics—can cause unintended behavior.

- *Document all cases of dynamic binding to externally developed libraries.* As was noted in section 4.1, dynamic binding should generally be avoided in safety systems. However, if dynamic binding with an externally developed library is needed in a safety function, all should be justified and documented. Each use should be supported with documented testing and tracking of anomalies.

- *Ensure that development and runtime shared libraries are identical.* Shared libraries, i.e. those which exist on the target machine and are linked at run time, should be used only if they are guaranteed to be identical to libraries on the developer's machine.

4.3.3 Utilizing Version Control Tools

Following guidelines are applicable to both C and C++

All C and C++ software should be kept under configuration management utilizing version control tools. Version control tools ask the author to document the changes when he/she makes changes, thereby minimizing the possibility of interface errors due to incompatible versions. A good version control package also provides a comparison utility that allows a user to compare the changes between source files of any two versions.

4.4 Maintainability

This section discusses the C and C++ specific attributes of the following intermediate attributes related to maintainability:

- Readability
- Data abstraction
- Functional cohesiveness
- Malleability
- Portability.

Base-level attributes and specific C and C++ guidelines are discussed in the following sections.

4.4.1 Readability

Readability allows software to be understood by qualified development personnel other than the

author. Readability is an important characteristic of programs, as almost all programs are modified or debugged by someone other than the original author at some time during the life of the program. Although readability should in large measure be based on project-specific guidelines, there project-independent issues that should be addressed. These issues and related guidelines are discussed in the following subsections.

4.4.1.1 Conformance to Indentation Guidelines

Following guidelines are applicable to both C and C++

The generic guidelines apply. Appropriate indentation facilitates the identification of declarations, control flows, nonexecutable comments, and other components of source code. Spaces are preferred to tabs for indentation since tabs may have different spaces on different file editors or printers. Indentation guidelines are as follows:

- Programming blocks should be bounded with brackets.
- Comments should have the same indentation as the objects being described.
- Branching constructs (i.e., **`if ... else ... ;`** and **`switch ... case,`**) should be indented.
- Looping blocks (i.e., **`for, while`**, and **`do ... while`**) should be indented.
- Automatic variables should be indented.
- Compiler directives should be indented.

The following example shows a function with recommended indentation:

```
top level -->main()
            {
                    /* loop variable */
second level ---> int i;

                    /* sub-block */
                    for (i=0; i<MAX_LOOPS; i++)
                    {
third level ----------> if (...)
                        {
fourth level ---------------->while (...)
                              {
fifth level ---------------------> ...
                              }
                        }
                    }

second level ---> switch
                    {
third level ----------> ...
                    }
            }
```

4.4.1.2 Descriptive Identifier Names

Following guidelines are applicable to both C and C++

The generic guidelines apply. The names of variables, routines, macros, and labels should be descriptive and closely related to the entities that are represented. Short and cryptic names should be avoided. The single additional guideline relates to variable names. Differences between variables with related names should occur early within the name (e.g. **`level2_sensor`** rather than **`sensor_level2`**). Although the ANSI/ISO C standard only guarantees the number of significant characters for an internal identifier and macro names to be 32, the number of significant characters for an external identifier should be limited to 6 (ANSI/ISO 9989-1990, section 5.2.4.21).

4.4.1.3 Comments and Internal Documentation

Following guidelines are applicable to both C and C++

The generic guidelines apply. Inadequate comments impede review and maintenance (Kernighan, 1978). The commenting guidelines in Chapter 2 are relevant. The following are additional guidelines for internal documentation:

- A routine should have a header that describes the input and output variables, the return type of the routine, the meaning of the return value if there is a return value, referenced and modified global variables, and an explanation of any arithmetic equations and algorithms in the routine. It should also document the modules it accesses.

- Comments should be used where subtle programming tricks are used or where critical steps are executed.

- Nested comments should not be used. When a block of code is no longer used, it should be removed from the source code to avoid confusion to developers and reviewers. For instance `#if(0) ... #endif` should be used to temporarily comment-out a block of code (Porter 1993). Some compilers have an option that allows nested comments. This option should not be enabled in safety-system development.

- Use care in mixing comment delimiter styles. Some C compilers allow C++ style comment "//". When using it in C language, cautions should be taken. A code with ("/* // This is a comment */") may work with C compilers, but it may not work with C++ compilers.

- The end brackets of loops and **if** blocks should be tagged with comments.

4.4.1.4 Limitations on Subprogram Size

Following guidelines are applicable to both C and C++

The generic guidelines apply. Subroutines should be limited in size, depending largely on project guidelines. The ANSI/ISO C standard limits are 127 identifiers within the block scope declared in a block and 31 parameters in a function definition (ANSI/ISO 9989-1990, section 5.2.4.2.1). Subroutines in C must not exceed these limits.

4.4.1.5 Minimizing Mixed Language Programming

Following guidelines are applicable to both C and C++

The generic guidelines have limited applicability. It may be acceptable, necessary, or desirable to mix C and C++ programs. However, other types of mixed language programming are a safety concern because (1) they present difficulties for reviewers and maintainers and (2) they cause interface errors because of different calling conventions and different data representations.

When this practice cannot be avoided, risks can be mitigated by the following measures:

- *Physical proximity.* Placing the "foreign" language code adjacent to the dominant language routine with which it interfaces.

- *Use of the asm directive.* The `asm` directive should be used where possible to include assembly code in C. Where separate assembly code must be used, macros should be defined to hide calling convention details.

4.4.1.6 Minimizing Obscure or Subtle Programming Constructs

Following guidelines are applicable to both C and C++

The generic guidelines apply. Obscure or subtle programming can generally be characterized as the use of indirect techniques to decrease the amount of coding or processing time required to achieve a result. Such coding practices present problems in review and maintenance and hence are a safety concern.

The guidelines for minimizing obscure or subtle programming are (Kernighan, 1978):

a) Write clearly; do not be too clever,
b) Make it correct before making it faster,
c) Make it clear before making it faster, and
d) Do not sacrifice clarity for efficiency.

When obscure code cannot be avoided (e.g., due to timing or memory constraints), comments should minimize the impact. The following are specific guidelines for C and C++

Following discussion applies to C

- *Avoid use of the* `?:` *operator.* The `?:` operator is another form of the if-then-else statement. The `?:` operator makes the code more difficult to read should be avoided in favor of the more conventional if-then-else construct.

- *Use table-driven alternatives when appropriate.* The following is an example to determine the next state of a state-machine with the following state-transition: `0->1, 1->0, 2->3, 3->4,` and finally `4->2` (Maguire, 1993). The following three equivalent code fragments illustrate the effect of chosen language features in the safety and simplicity of the code:

```
/*    option 1 : use of ?:    */
```

```
     ((x<=1)?(x?0:1) : (x==4)?2:(x+1))

/*     option 2 : use of nested if   */
      if(x<=1)
      {
         if(x!=0)
            x=0;
         else
            x=1;
      }
      else
      {
          if(x==4)
             x=2;
          else
             x=x+1;
      }


/* option 3 : use of table-driven selection */

    static const nextvalue[]=(1,0,3,4,2)

    x = nextvalue[x];
```

The following discussion applies to C++ only

•*Avoid using default parameters to combine functions.* For example, the use of the single function **`lookup(char *name, int code=-1)`**— where the value of `code` determines whether **`lookup`** should fail if `name` is not found — may not be clear to the reviewer. The more appropriate way is to define a new function for this purpose. Note that use of default parameters is acceptable in general (Binkley, 1995).

•*Avoid complex expressions inside a condition.* For example, **`if(i&mask==0)`** is equivalent to **`if(i&(mask==0))`** and not to **`if((i&mask))==0)`**. In this case the reviewer is expected to remember the operator precedences to verify the intent of the programmer. Replace it with **`long masked_i=i&mask; if(masked_i==0)`** (Binkley, 1995).

•*Maximize the use of the scope resolution operator.* The scope resolution operator : : should be used to indicate explicitly which of a collection of functions or variables is being used. This includes globals accessed as **`::global_variable`** (Binkley, 1995).

•*Avoid pointers to members.* They unnecessarily complicate the code. Use virtual functions or redesign (Binkley, 1995).

•*Use the **virtual** keyword wherever necessary.* For a C++ member function declared in a base class the keyword virtual should be used explicitly in the declaration of the function and all declarations and definitions of the functions in each derived class (Binkley, 1995).

4.4.1.7 Minimizing Dispersion of Related Elements

Following guidelines are applicable to both C and C++

The generic guidelines apply. If related elements of the code are dispersed in a program, this makes it necessary to refer to multiple locations within a source listing in reviewing or modifying the source code. The following are specific guidelines

- *Place **`include`** directives at the beginning of each program.* **`#include`** compiler directives for header or other files should be located at the beginning of each program. If it is necessary to include files in the middle of a program, this must be clearly tagged with a comment.

- *Place all external function prototypes in physical proximity.* External function prototyping should be in one place, e.g., a header file. Prototypes should not be in each individual file where the function is referenced. For functions with static scope, the prototypes should be in the same module where they are defined and used, and the function should be declared as static.

The following discussion applies to C++ only

- *Segregate base from derived classes.* In C++, it is desirable to segregate base classes from derived classes.

4.4.1.8 Minimizing Use of Literals

Following guidelines are applicable to both C and C++

Literals, also called hard-coded numbers or hard-coded strings, are more difficult to identify than names to which a constant value or a string is assigned at the beginning of the module. Safety systems should utilize symbolic values (using the **`const`** identifier or if necessary, **`#define`**) instead of literals that have some extrinsic meaning or that may be changed in the future. The following specific guidelines apply:

- *Parentheses.* In safety systems, all expressions for **`#define`** should be place in parentheses,

even for a single number. The reason for using parentheses on a single number is that **#define** value may be changed later to an expression and consistency is always desired. It makes systems maintenance easier. As mentioned earlier, defining a variable with the **const** identifier is preferable to **#define**.

- *Enumeration.* When there are several sequential integer numbers, enumeration constants are preferred to separate **#define** statements (Porter, 1993). Enumeration makes it easier to modify when a new number needs to be inserted to the sequence.

For example, in the following statements:

```
#define temp1_sensor    (10)
#define flow1_sensor    (11)
#define flow2_sensor    (12)
```

The equivalent enumeration constants are:

```
enum instrument_labels
{
      temp1_sensor = 10,
      flow1_sensor
      flow2_sensor
};
```

To add an additional temperature sensor before **flow1_sensor,** all the numbers after **temp1_sensor** need to be changed in the **#define** statements. However when using enumeration only one change is needed: inserting the new label between **temp1_sensor** and **flow1_sensor**.

The new code will be:

```
#define temp1_sensor          (10)
#define temp2_sensor          (11) /* add new operation */
#define flow1_sensor          (12) /* 11 changed to 12  */
#define flow2_sensor          (13) /* 12 changed to 13  */
```

The equivalent enumeration constants are:

```
enum instrument_labels
{
      temp1_sensor = 10,
      temp2_sensor,              /* this is the only change */
      flow1_sensor,
      flow2_sensor
};
```

If literals are used, comments should be associated to facilitate search and replace efforts.

4.4.2 Data Abstraction

Data abstraction is the combination of data and allowable operations on that data into a single entity, and establishment of an interface which allows access, manipulation, and storage of the data only through the allowable operations.

4.4.2.1 Minimizing the Use of Global Variables

Following discussion applies to C

Generic guidelines apply to C. Because of the potential for unintended side effects, use of global variables in safety related programs should be limited (Parnas, 1990). Readability is enhanced when variables are declared, set, and used in the same routine. If global variables are to be used, the following language-specific guidelines can mitigate the associated safety concerns.

- *Keep global variables and associated functions in the same file.* If a limited number of functions need to share a certain variable, those functions can be included in the same file and the shared variable given file scope.

- *Declare global variables in one header file.* When a global variable has to be used, it should be declared in one header file. There should not be multiple reference **`extern`** declarations for a variable. The following example shows how multiple references create maintenance problems and safety concerns:

```
static int i;
main()
{
      extern int i;
      {
            extern int i; /* Scope? */
      }
}
```

- *Initialize global variables in one place.* As noted earlier. global variable initialization should occur in exactly one place in the program.

4.4.2.2 Minimizing the Complexity of Interfaces

Following guidelines are applicable to both C and C++

The generic guidelines apply. Interfaces are a frequent cause of software failures (Thayer, 1976). Complex interfaces are difficult to review and maintain and are therefore not desirable in safety-related programs. The following are specific guidelines:

- *Limit the number of parameters.* In the C and C++ languages, the number of parameters of a function or a macro should be minimized. Large numbers of parameters can make interfacing complex.

- *Use structures.* When there many parameters and some of those parameters are related, they should be defined in a structure, and a pointer to the structure should be passed as a parameter to reduce stack usage.

- *Avoid expressions in parameter lists.* Since the order of parameters being evaluated is unspecified in the ANSI C standard, the expressions should be eliminated in parameter passing to a subroutine or a macro, as shown in the following example:

```
calculate_area(length=2, width=length+2);
```

Because the second parameter, "width," may be evaluated first when the routine is called, it may produce an unintended result. A possible correction for the above function call is:

```
length = 2;
width = length + 2;
calculate_area(length, width);
```

4.4.3 Functional Cohesiveness

Cohesiveness is the manner and degree to which the tasks performed by a single software module are related to one another (IEEE, 1990). Functional cohesiveness refers to a clear correspondence between the functions of a program and the structure of its components.

Following guidelines are applicable to both C and C++

The generic guidelines apply to C and C++. Review and maintenance are when a given function implements only one well understood purpose.

Following discussion applies to C++ only

The rationale for the design of class libraries should be obvious and related to the objective. Objects defined in C++ should have a single identifiable purpose. Specific guidance is a design level issue which is beyond the scope of this document.

4.4.4 Malleability

Following guidelines are applicable to both C and C++

Malleability is the ability of a software system to accommodate changes in functional requirements (Parnas, 1990). Malleability extends data abstraction with the motivation toward isolating areas of potential change. The generic guidelines apply to both C and C++. There are no additional language-specific guidelines.

4.4.5 Portability

Portability is the ease with which a system or component can be transferred from one hardware or software environment to another (IEEE, 1990). From the perspective of safety, the benefits of portability are the adherence to standard programming constructs that yield predictable and

consistent results across different operating platforms (Witt, 1994). Thus, code that is reused or converted to run on a different platform will be easier to maintain and will be more exhaustively tested.

The following portability-related guidelines relevant to C and C++ have been discussed previously:

- Minimizing the use of built-in functions (section 4.3.1)
- Minimizing the use of compiled libraries (section 4.3.2)
- Minimizing interface ambiguities (section 4.1.2.5)
- Minimizing dynamic binding (section 4.1.2.12)
- Minimizing the use of tasking (section 4.1.3.1)
- Minimizing the use of interrupt driven-processing (section 4.1.3.2).

The following additional specific guidelines will be discussed in this section:

- Minimizing anonymous data types
- Avoiding reserved words and keywords
- Minimizing hardware dependencies.

4.4.5.1 Minimizing Platform-Dependent Data Types.

Following guidelines are applicable to both C and C++

This topic has been partially discussed in previously (section 4.1.2.6 Use of Data Typing). Implementation-dependent data types may create problems across different platforms or compilers. The related guideline discussed in that section is the use of the integer and floating point data types. A typical example of this data type is **`int`**, which is 16 bits in some compilers and 32 bits in others.

4.4.5.2 Avoiding Reserved Words

Following guidelines are applicable to both C and C++

The following are portability-related guidelines on the use of reserved words in C and C++:

- *Avoid underscores.* Identifiers with starting underscore or underscores should not be used. According to the ANSI C standard ((ANSI 9989-1990), section 7.1.3) all identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use. Identifiers that begin with an underscore are reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces. Identifiers

starting with double underscores `__LIKE_THIS` and identifiers starting with an underscore and followed by an upper case letter `_SUCH_AS_THIS` are reserved words. Identifiers starting with an underscore `_like_this` are reserved for file scope variables. C++ reserves identifiers with double underscores for implementation and libraries. Using identifiers with starting underscore and double underscores can cause unspecified results if they are reserved words (such identifiers can also cause unspecified results later even if they are not reserved words for the current revision of the compiler).

- *Avoid use of C++ keywords even though that language is not used.* C programmers should avoid using names that are keywords in C++ since C programs may later be converted to C++ programs. Examples are `catch`, `class`, `delete`, `friend`, `inline`, `new`, `operator`, `private`, `protected`, `public`, `template`, `this`, `throw`, `try`, and `virtual`.

- *Do not use the names of functions in the standard library.* The names of the functions in the standard library should be treated as reserved words (Plum, 1991).

4.4.5.3 Minimizing Hardware Dependencies

Following guidelines are applicable to both C and C++

- *Define hardware-dependent address symbolically.* In a control system, it may be possible to avoid directly accessing hardware by means of a vendor supplied device driver. However, it may be necessary or desirable for the safety system software to directly interface to the hardware for the purposes of traceability. If writing to hardware is necessary, the addresses should be clearly documented and defined in a manner that minimizes the possibility of change errors. This may be using symbolically as defined earlier in this section (or by means of class definitions (in C++) for potential future changes.

- *Use `volatile` attribute for data items that are mapped to hardware.* Data items that are mapped to actual hardware must have the `volatile` attribute. This attribute ensures that the compiler will not use optimization and leave the value in a CPU register, but will read it from the memory location each time it is set or used (Harbison, 1987, p. 265). The rationale for the use of volatile is that the value may have changed since the last time it was set or used by the CPU (e.g., a bit set to busy subsequently was set to not busy). When such an item is referenced, its pointer should be a pointer-to-volatile.

- *Avoid the use of bit fields.* Bit fields are dependent on the compiler and the "little-endian/big-endian" nature of the CPU. They should therefore not be used. Shifting and masking should be used instead. Additional guidelines on the use of bit masks in place of bit fields are found in section 4.1.2.5.

- *Do not measure time intervals by counting clock cycles.* Generating delays by counting clock cycles should also be avoided since the timing of a clock cycle can will differ on a different platform.

References

American National Standards Institute, *ANSI C Standard, American National Standard for Programming Languages—C*, ANSI/ISO 9899-1990.

Binkley, D.W., "C++ in Safety Critical Systems", NIST-IR 5769, National Institute of Standards and Technology, November, 1995.

Cargill, T., *C++ Programming Style*, Addison Wesley, 1992.

Chillarege, R., "Orthogonal Defect Classification", *IEEE Transactions on Software Engineering*, 1992.

Cuthill, B., "Applicability of Object Oriented Design Methods and C++ to Safety Critical Systems", *Proceedings of the Digital System Reliability and Nuclear Safety Workshop*, NUREG CP-0136, NIST SP 500-216, 1993.

U.S. Department of Defense, Software Development Standard, MIL-Std-2167A, August, 1986, Appendix C.

Eckel, B., "Exception Handling in C++", *Embedded Systems Programming,* Vol.8, No.1, January, 1995.

Harbison, S.P., and G.L. Steele, *C: A Reference Manual,* Prentice Hall, Englewood Cliffs, NJ, 1987

Institute of Electrical and Electronics Engineers, IEEE Std 100-1977, *IEEE Standard Dictionary of Electrical and Electronic Terms.*

Institute of Electrical and Electronics Engineers, IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology.*

Kernighan, B.J and P. J. Plauger, *The Elements of Programming Style, Second Edition*, McGraw-Hill, New York, 1978.

Koeman, S. and S. Ross, "Optimize Your Code to Run Faster and Jump Higher with the Visual C++ 2.0 Compiler," *Microsoft Systems Journal*, 1995.

Liao, Y., "Requirements Directed Automatic Instrumentation Generation for Program Monitoring and Measuring," *In IEEE Transactions on Software Engineering*, 1991.

Meek, B.L., "Early High-Level Languages," *In Software Engineer's Reference Book*, J.D. McDermid, ed., CRC Press, Inc., 1993.

Parnas, D.L., A.J. van Schouwen, and S.P. Kwan, "Evaluation of Safety Critical Software," *Comm. ACM*, Vol. 33, No. 6, p. 636, June, 1990.

Plauger, P.J., "Under Construction", Embedded Systems Programming, Vol.8, No.4, Apr. 1995, pp. 125-128.

Plum, T. and D. Saks, *C++ Programming Guidelines*, Plum Hall, 1991.

Porter, A., *The Best C/C++ Tips Ever*. Osborne McGraw-Hill, New York, 1993.

Spuler, D.A., *C++ and C Debugging, Testing, and Reliability*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

Thayer, R., "Software Reliability Study," Rome Air Development Center report RADC TR 76-238, March 1976.

Witt, B.I. and F.T. Baker, and W.W. Merritt, *Software Architecture and Design*, Van Nostrand Reinhold, New York, 1994.

5 PLC Ladder Logic

This chapter discusses use of Programmable Logic Controller (PLC) Ladder Logic in safety systems. The chapter is organized in accordance with the framework of Chapter 2. Section 5.1 discusses reliability-related attributes of PLC Ladder Logic; Section 5.2 discusses robustness-related attributes of Ladder Logic; Section 5.3 discusses traceability-related attributes; and Section 5.4 describes maintainability-related attributes. A summary matrix showing the relationship between generic and language specific guidelines, together with weighting factors, is included in Appendix B. Language-specific weighting factors were based on the special nature of the language with its industrial control and hardware orientation together with limited data types.

At present, Ladder Logic is the principal problem solving (application) language for PLCs[23]. Although programming considerations are largely common, the variety of PLC models and the absence of a single standard that unambiguously defines Ladder Logic complicate the issue of defining some guidelines and providing examples. Most of the programming examples in this chapter and Appendix A use the Allen Bradley PLC-5 variety of Ladder Logic. However, the use of this PLC as an example should neither be interpreted as an endorsement or criticism of that product line.

5.1 Reliability

The reliability of a PLC Ladder Logic program means its ability to perform its required functions under stated conditions for a specified period of time (IEEE, 1990). Reliability depends on the runtime predictability of the following:

- Memory utilization
- Control flow
- Timing.

PLC Ladder Logic-specific guidelines are described in the following sections.

5.1.1 Predictability of Memory Utilization

The key element in predictability of memory utilization is to avoid the use of dynamic memory allocation. However, PLC Ladder Logic does not specifically allow for dynamic memory allocation. In general, memory required by the program is static at runtime. For each variable that the program

[23] A PLC is a special purpose computer for industrial control applications. More complete descriptions of both PLCs and the Ladder Logic programming language are provided in Appendix A.

uses, there is a specified memory location in a data table file. Each program is stored in a program file whose size is determined during compilation or translation. Thus, the generic guidelines are not relevant for Ladder Logic programs.

The only memory allocation that is not defined prior to runtime is memory utilization by the "operating system" (PLC firmware) for stack and queue purposes. However, this memory allocation is beyond the scope of the PLC Ladder Logic controller. In general, stack allocation should not be a cause of program crashes due to restrictions imposed by the Ladder Logic programming environment. In some PLC models these restrictions are limits on the number of parameters passed to a subroutine or on nesting levels, in other PLC models, other controls are used. The intent of these is to prevent the PLC programmer from causing failures due to memory management problems.

5.1.2 Predictability of Control Flow

Predictability of control flow is the capability to determine easily and unambiguously what path (i.e., which set of branches and in what order) the program will execute under specified conditions. This subsection discusses guidelines related to the following attributes:

- Maximizing structure
- Minimizing control flow complexity
- Initializing variables before use
- Single entry and exit points for subprograms
- Minimizing interface ambiguities
- Use of data typing
- Accounting for precision and accuracy
- Order of precedence of arithmetic, logical, and functional operators
- Avoiding functions or procedures with side effects
- Separating assignment from evaluation
- Proper handling of program instrumentation
- Controlling class library size
- Minimizing use of dynamic binding
- Controlling operator overloading.

5.1.2.1 Maximizing Structure

The generic guidelines apply. Use of *goto* or equivalent statements resulting in an unstructured shift of execution from one branch of a program to another should be avoided because such programs are difficult to trace and understand.

Ladder Logic language allows the programmer to use *goto* statements. In Ladder Logic language, there is no mechanism to force the programmer to develop a structured program. A sample use of

the *goto* (JMP) command is shown in Figure 5-1. Whether *goto* statements should be banned in a project depends on the characteristics of the selected PLC. Some versions of Ladder Logic allow the maximization of structure by the use of block structured code and calls to subroutines. When available, these constructs should be utilized.

However, not all PLC Ladder Logic implementations support subroutines, especially in smaller models. Fewer still support parameter passing to subroutines or subroutines with local memory. In the case of a PLC without subroutine support, the jump to label illustrated in Figure 5-1 may be the only mechanisms available to provide control flow over program segments.

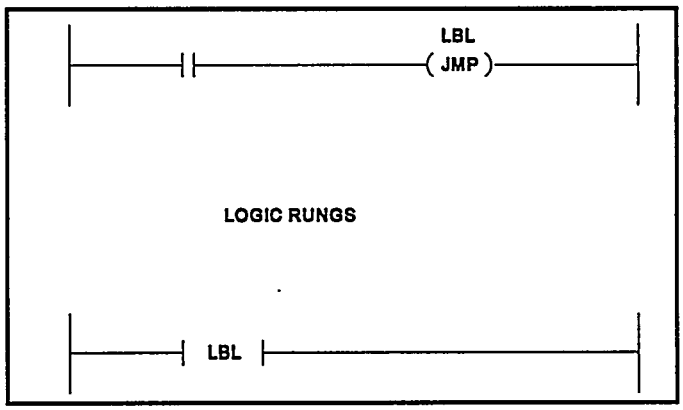


Figure 5-1 Use of *goto*

If *goto* statements are used, it is necessary to justify why such statements are needed and why alternative programming methods could not be used. The following specific guidelines are applicable if the *goto* (or JMP) is used:

- *Use watchdog timers or scan counters with backward jumps.* The PLC does not limit direction, so that the program can jump backwards. This backward movement could result in an internal watchdog timer expiration, causing the PLC to enter a fault state. This is another reason to require a timer or a scan counter to protect the integrity of the program (see guidelines below).

- *Ensure that data initialization has occurred before making the jump.* Since logic between the JMP and the LBL instructions are not scanned by the PLC, data table words and bits can be left in an non-initialized state. This could breach a safety-critical application.

5.1.2.2 Minimizing Control Flow Complexity

The generic guidelines are applicable. The control flow in Ladder Logic is controlled by "if..then" structures, making it is easy to predict run-time behavior of a single statement. Even a relatively complex control flow structure, as shown in Figure 4.2, is reviewable in PLC Ladder Logic. However, it is not always so easy to predict behavior on the program level, when many rungs are involved. A further complication is the complexity/feature set of the specific Ladder Logic implementation being used. There are significant differences in various models of PLCs.

The specific guidelines related to control-flow are as follows:

- *Decomposition.* The Ladder Logic program should be subdivided into cohesive subroutines.

- *Nesting level limits.* Care should be taken to ensure that nesting levels are not excessive. The maximum nesting level may be defined on a project-specific basis. For some PLCs, there is a limit on the maximum number of levels.

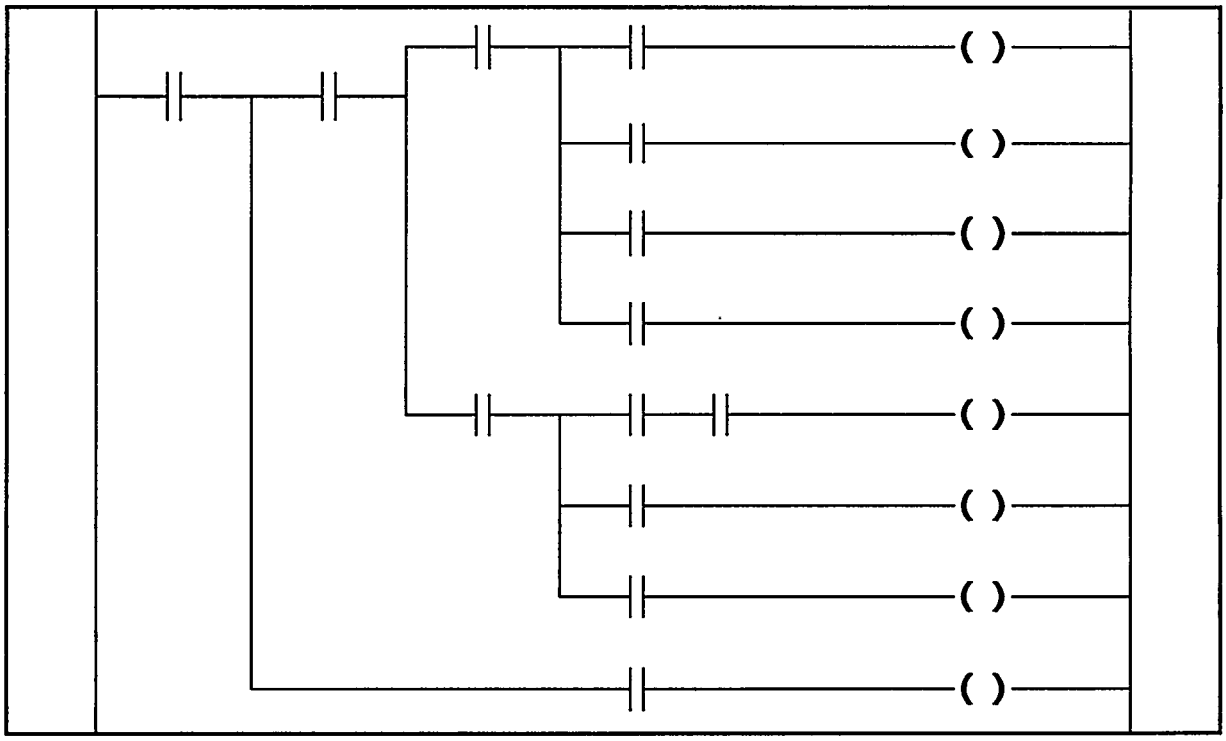


Figure 5-2 Sample of "Complex" Control Structure

- *Limitation for use other than Boolean functions.* PLC Ladder Logic should be limited to its primary intended purpose, i.e., interlocks and other Boolean applications. The above diagram is a good example of how Ladder Logic used in such a manner can be quite clear and easy to understand, even when expressing a complex boolean relationship. The same cannot be said for the use of Ladder Logic for mathematical functions or other purposes. In such cases, the code can be more complex and difficult to understand. If Ladder Logic is needed for such code, extensive documentation is necessary to make its purpose clear in a production system.

- *Impact of the underlying PLC data base.* Predictability of the behavior of entire PLC programs depends not only on the Ladder Logic program itself but also on the interaction with the PLC data base. It is not unusual for PLC programs to consist of dozens of rungs of logic applied to a single global variable base. There is a significant potential for programming errors. Proper and strict management of this variable base, or PLC memory map, and adherence to a methodology for using these variables are required for the safe programming of PLCs. These guidelines are discussed later.

5.1.2.3 Initialization of Variables Before Use

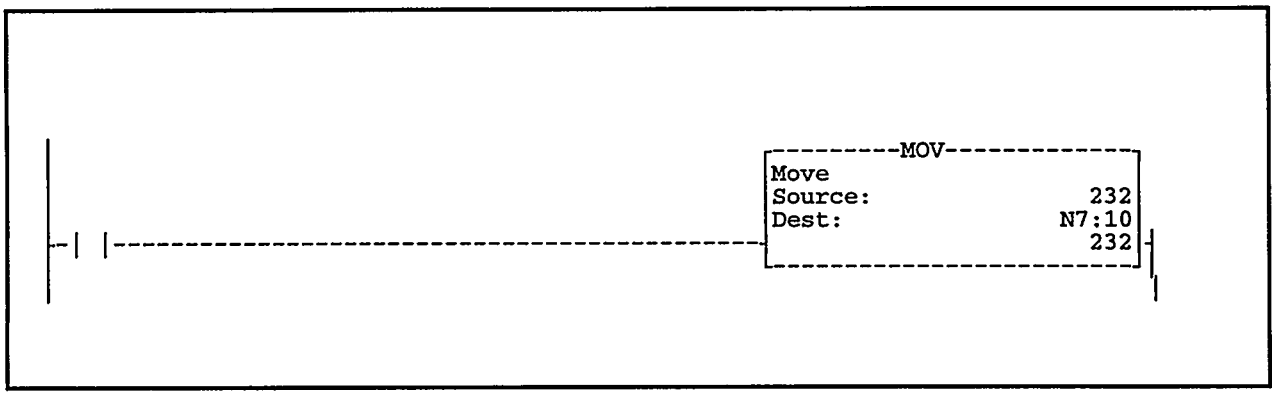
Proper variable initialization is critical for Ladder Logic programs. However, the generic guideline is applicable in a manner somewhat different from other high-level languages because of the differences in which initialization must occur in different Ladder Logic implementations. The following are specific guidelines.

- *Initialization of variables in Ladder Logic programs.* Where supported, variables should be initialized in the Ladder Logic code. Explicit initialization of variables in Ladder Logic, or any of the other PLC Languages, is one of the requirements of the IEC 1131-3 PLC Language Specification. Unfortunately, few if any currently available PLC systems support this concept at the source code level. It is anticipated that the feature will become more common in future implementations.

- *Initialization at program load time.* Many, but not all, PLC development environments allow the programmer to set initial values for PLC variables, which are then subsequently uploaded to the PLC. Others simply initialize the variable pools to zero. Both the PLC programmer and auditor should be aware of how the particular PLC system chosen for a safety-critical application operates in this regard, which should be noted in the PLC program documentation. Relying on the development environment to upload initial values of variables does not automatically ensure that all variables were correctly initialized. Also, the programmer normally has the capability to initialize the data table files manually, not through explicit assignment in the Ladder Logic program.

- *Initialization at power up.* Initialization should be performed every time the system is powered up, restarts operation, or recovers from a failure. An initialization subprogram can handle all the program initialization issues, not only variables, when the PLC is turned into RUN mode. This procedure is recommended unless other means for ensuring correct initialization are in place.

  The following is an example of initializing some words to an explicit value (e.g, the boiling point of a liquid) into the calculation:



Another example is the executive program that calls an initialization subroutine shown in Figure 5-3. Many PLCs have a mechanism similar to the SYSTEM_INITIAL shown in the figure: a flag from the operating system signals the first scan of the PLC. Some PLCs further distinguish this first scan as a either a *Cold Start*, when initialization of variables may be necessary, or a *Warm Start*, in which all variables have successfully retained their values since the PLC was powered down. Specific initialization actions are required in these circumstances, depending on the application. However, critical variables should be explicitly initialized in the program in a start up scan subroutine.

- *Accounting for mode changes.* Initialization may also be a concern when an operator changes the mode of operation. The program should not rely on assumed prior conditions to initialize after a mode change.

5.1.2.4 Single Entry and Exit Points in Subprograms

The generic guidelines apply. Ladder Logic implementations supporting subroutines generally allow only a single entry point to those subroutines. When the program jumps to such a subroutine, the entry point will always be the first rung. However, Ladder Logic allows the use of multiple exits by placing a RETURN rung at different locations along the execution path. An example of multiple exits is shown in Figure 5-4; an equivalent program with a single exit is also presented. It should be noted that the end of program statement acts as a RETURN rung so that it is not necessary to explicitly include one. When passing parameters, however, the program needs the RETURN statement complete with the parameter return address.

In the case of a PLC system without explicit subroutine support, it is even more critical that all subprograms (implemented with JMP to label) have a single entry and exit point. Not only will this simplify understanding of the program, but it will also contribute to correct operation. On many PLC systems, overlapping or nested JMP commands could cause counter-intuitive and difficult-to-understand results at run time.

Guidelines for a single exit requirement can be established in the programming manual. Use of multiple exit points may be justified by the developer by showing that a single exit causes more problems than it fixes. When using multiple exit points, it is necessary to ensure that the state of the data tables will be unambiguously known at all exit points.

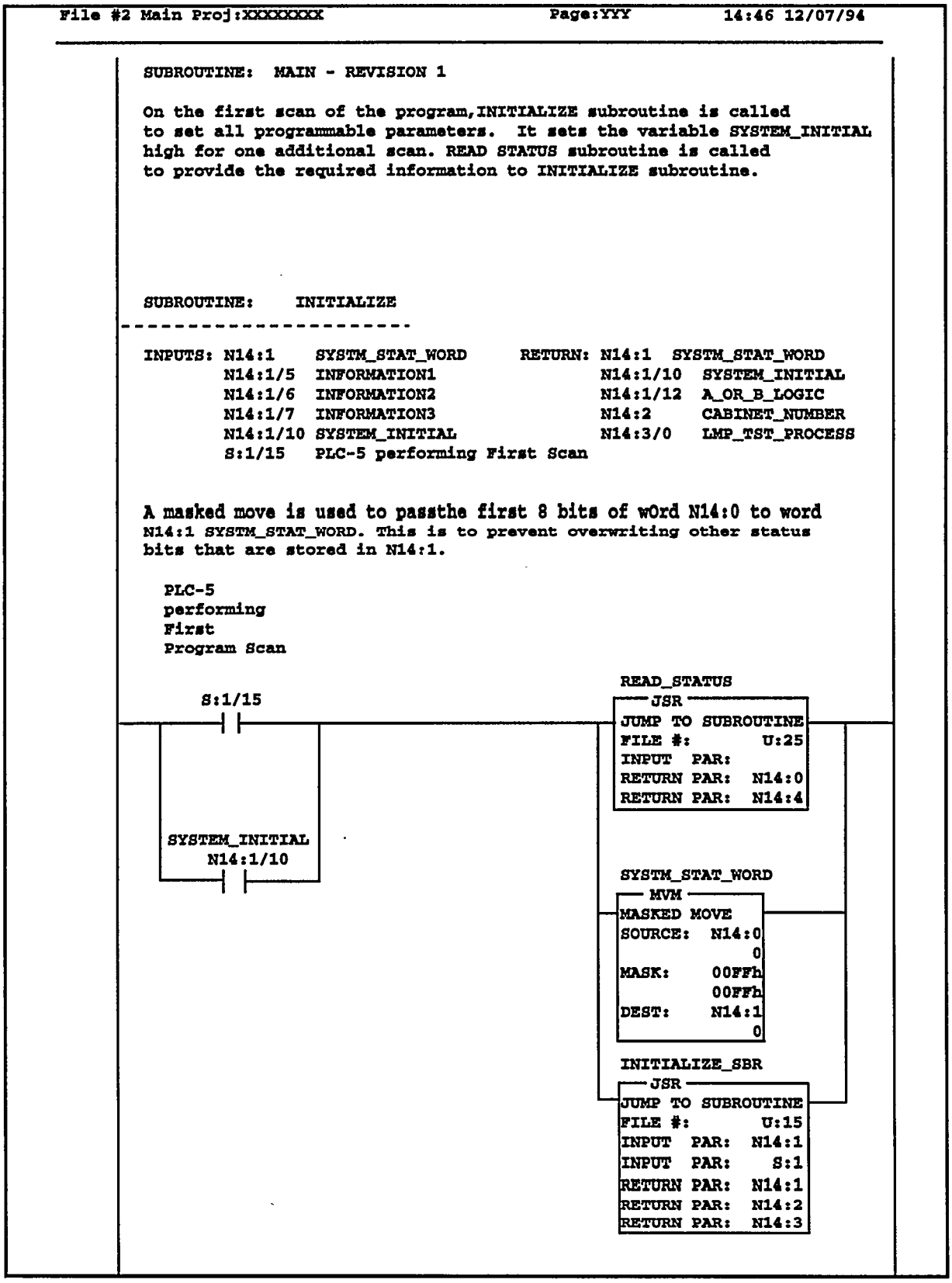


Figure 5-3 Use of an Initialization Subroutine

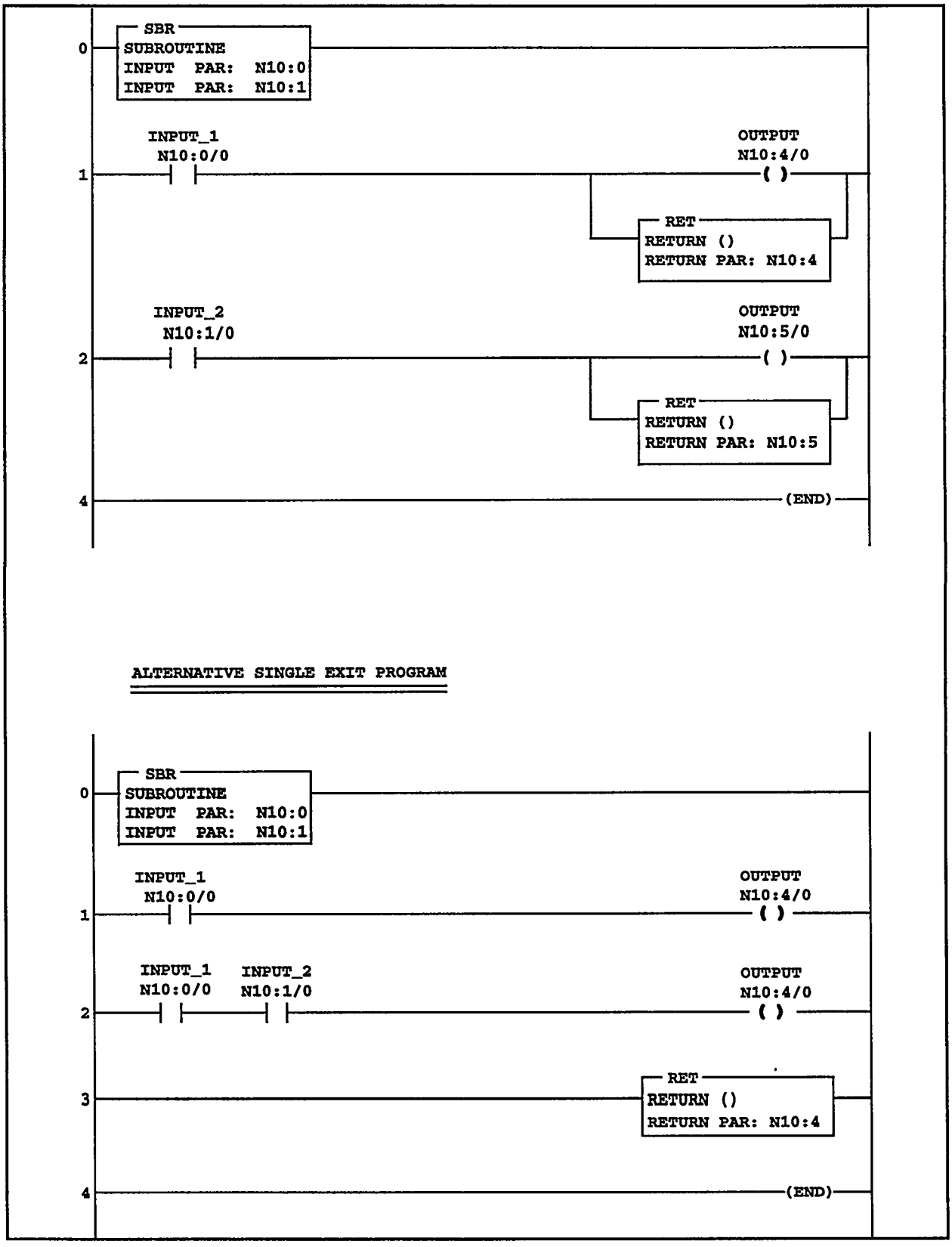


Figure 5-4 Ladder Logic Multiple RETURN

5.1.2.5 *Minimization of Interface Ambiguities*

The generic guidelines have limited applicability. Interface errors account for a significant portion of coding errors. Unfortunately, Ladder Logic has limited support for avoiding such errors. The following are specific measures that can be used:

- *Validity checking.:* The preferred approach is testing for the validity of input arguments before they are passed to the data table addresses used by the subroutine. Typically, such validity checking would be a range check done in the rung previous to the subroutine jump (JSR) call. As an alternative, it can be done at the beginning of the subroutine. In the example in Figure 5-4, each input parameter is in the range [0,1], but is stored as a 16-bit integer. A validity test is required to verify if the actual input is limited to the valid range.

- *Comments.* Internal comments and documentation of interfaces are important to avoid interface ambiguities and errors.

- *Type Checking.* Type checking can be used to detect some basic types of interface incompatibilities. However, it is the least effective since most variables are integers.

5.1.2.6 *Use of Data Typing*

The generic guidelines have limited applicability. In general, most Ladder Logic implementations are weakly typed. It is therefore not possible to gain the advantages of strong data typing. The following are specific guidelines.

- *Ensure that the data table properly accounts for variable types.* The data tables must be constructed to account for the differing lengths and storage characteristics of data types. For example, in the TSX PLC line sold by AEG/Schneider, identifiers W3, DW3, and FW3 all refer to the same location in memory, but are treated as a 16-bit integer, a 32 bit integer (in conjunction with the next location, W4), or a 32-bit floating point value (again with W4) respectively. Care must be taken, in the event that DW3 is used as a 32 bit integer, that neither W3 nor W4 is used as 16-bit integers elsewhere in the program, as this would result in corrupted data. Data types supported by the Allen Bradley PLC5 line are floating point, integer, binary, BCD/HEX, and ASCII. A problem can exist in certain instructions where the result of a calculation is incompatible with the resulting data table constructs, such as a negative integer being written into a BCD (or decimal) data table area. In this case, the data would be stored incorrectly, which could result in a latent failure that would be manifested subsequently. However, should the number being written into the resulting word be too large in quantity, and a file type instruction is being used, the risk exists that the PLC will fault (typically, a 'BAD OPERAND' fault would occur) immediately.

- *Ensure that type conversion will not result in an error.* For example, a floating point word/file transfer to an integer data type will result in rounding, and in fact, some floating point words may be truncated.

- *Develop project-specific guidelines.* The nature and extent of data typing varies from PLC implementation to implementation. It is therefore imperative that project-specific guidelines on the use of available data types and appropriate safeguards be developed. These guidelines should reflect specific PLC characteristics, and compliance with these guidelines should be monitored.

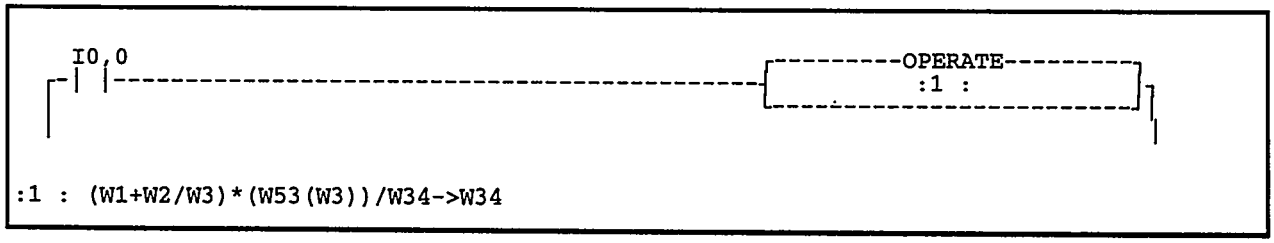
5.1.2.7 Precision and Accuracy

The general guidelines are applicable. The specific guideline is to ensure that the accuracy required by the algorithm is supported. Most Ladder Logic programs handle integer and bit variables. Algorithms that require floating point arithmetic must be analyzed on a case by case basis to verify that the processor and language provide the accuracy required by the algorithm.

5.1.2.8 Order of Precedence of Arithmetic, Logical, and Functional Operators

Ladder Logic implementations vary in how they handle order of precedence in arithmetic and logical expressions. Many implementations perform arithmetic operations by means of dedicated ADD, SUBTRACT, MULTIPLY and DIVIDE blocks, etc, as illustrated in the Allen-Bradley and Modicon example programs illustrated here. These blocks only accept a predetermined number of parameters, and so order of precedence is not an issue in systems of this type.

Other PLC systems do allow complex mathematical statements by means of "operation blocks" or compute and transfer (CPT) blocks. Here, the order of precedence of arithmetic operators can be an issue. Unfortunately, there is no consensus among PLC implementations of this type as to the order of precedence of arithmetic operators. Hence, the liberal use of parenthesis (when available) is recommended to force the desired execution order. An example follows:



The operate block in this Ladder Logic example contains the complex expression footnoted as :1:. In the case of the Allen Bradley PLC5 controller, a complex expression in a COMPUTE (CPT) block instruction can also be entered. As an example, a unit conversion could be done in one CPT block

[(N7:0*2) - 32]. Again, parentheses are needed. Order of evaluation of expressions on systems that support this type of construct will vary by make and manufacturer. In this instance, order of evaluation of the calculations was explicitly indicated by the use of parentheses. Both the programmer and the auditor should be aware of what the requirements are for the specific PLC system used.

Order of execution of logical elements is controlled by the Ladder Logic network itself. Whereas all Ladder Logic implementations execute each network of Ladder Logic sequentially, the order of execution of each network of logic varies from implementation to implementation. The specific nature of the PLC system used in a safety-critical application must be explicitly known by both the programmer and auditor. Use of Ladder Logic constructs that depend for their correct operation upon the specific nature of the Ladder Logic network execution order should be avoided.

5.1.2.9 Avoiding Functions or Procedures with Side Effects

Generic guidelines are applicable.

5.1.2.10 Separating Assignment from Evaluation

Some Ladder Logic implementations do not allow external assignment or even expression evaluation as part of conditional statements. On these systems, conditional statements are restricted to simple variable comparisons, and the generic guidelines do not apply.

However, expression evaluation within comparison blocks is allowed on many PLC systems. For example, on the Allen Bradley PLC5, the CMP instruction accepts expressions for data comparisons. The Modicon 984 line has no separate compare instruction, but utilizes a side effect of the subtract block to implement comparison functions. On these systems, it is not possible to separate assignment from evaluation of conditional statements.

In such cases, the specific guidelines are as follows:

- *Use buffer variables or output coils.* A conditional statement in a PLC requiring an assignment should use a designated dummy variable as a buffer. This variable is used for no purpose other than as a memory buffer for unwanted assignments. This practice is easily auditable, and prevents confusion of assignment and evaluation of conditionals. Many (but not all) PLCs require that each network of Ladder Logic contain an output coil, even though the boolean result of the network is meaningless in the context of the application.

- *Develop project-specific guidelines for separating assignment from evaluation.* The features and functionality of the PLC system used for a safety critical application regarding the separation of assignment from evaluation should be documented and conformance should be monitored.

5.1.2.11 *Proper Handling of Program Instrumentation*

The generic guidelines described in Chapter 2 are applicable. The following are specific guidelines.

- *Do not perform on-line modification.* Most PLCs provide a facility that allows the modification of the PLC program while the PLC is executing that program. The operational consequences of utilizing this feature during operation can be quite dangerous. First, a programmer could accidentally introduce errors into a running PLC program by using this feature. Secondly, the added communications load on the PLC processor during the program change transfer could also result in delays that prevent needed actions from happening in time.

- *Do not activate on-line monitoring facilities for time critical operations.* There should be no use of debuggers, instrumentation, or monitors during PLC operation of time-critical functions unless such monitoring is part of the baseline design and its impact on timing has been accounted for. If such monitoring is necessary, it should be done in an off-line mode or using a simulator/emulator. If operations are not time critical, then on-line monitoring may be performed, but with caution and only under conditions where it can be guaranteed that monitoring of non-time-critical functions will not affect time-critical functions.

5.1.2.12 *Control of Class Library Size*

Ladder Logic does not support classes and objects. Therefore, the generic guidelines are not applicable.

5.1.2.13 *Minimizing Dynamic Binding*

Ladder Logic does not support dynamic binding. All structures must be defined by the programmer before compilation. Therefore, the generic guidelines are not applicable.

5.1.2.14 *Control of Operator Overloading*

The generic guidelines are not applicable. Ladder Logic prohibits operator overloading and does not support polymorphism.

5.1.3 Predictability of Timing

Predictability of timing is crucial in a safety system used in real-time control. Timing-specific concerns relevant to PLCs include:

- Minimizing the use of tasking
- Minimize the use of interrupt-driven processing
- Input/output timing
- Avoidance of self-modifying code

5.1.3.1 Minimizing the Use of Tasking

While some PLC systems do not support multitasking in any form, many support it either implicitly or explicitly. Implicit multitasking occurs where only one Ladder Logic program can be run, but the firmware manages handling the Ladder Logic program scan, remote I/O scan, block data transfers, and other communications asynchronously (i.e., each as an independent task). Limited multitasking allows the PLC programmer to create a timed interrupt, a distinct Ladder Logic program (or section of Ladder Logic code) designated to be executed at fixed intervals (usually expressed in msec), regardless of the state of the main program. Other PLCs have complete multitasking capabilities, with each task having a defined periodicity and separate I/O scan.

The generic guidelines on minimizing the use of tasking apply at the application level. Where multitasking is supported, caution and prudence must be exercised. The decision of whether or not to use explicit multitasking (i.e., the simultaneous running of multiple Ladder Logic programs in a single PLC) should not be taken lightly. Multitasking is an attractive programming model and may be simpler at the application level than coding a single task to perform the same functions. However, worst case execution times, latencies, and coordination of data access may introduce uncertainties that are unacceptable in safety applications. System-level alternatives, such as the use of multiple PLC's should be considered if design of a single task is unduly complex.

Specific guidelines for PLC multitasking are as follows:

- *Account for processing capacity.* The PLC program must limit PLC CPU utilization and provide generous margins to account for variation. CPU bandwidth usage should be explicitly calculated and shown to be within specified margins. The results of these calculations should be included in the PLC documentation along with the periodicities of the various tasks derived from them. Manufacturer's guidelines for CPU bandwidth utilization should be strictly followed. For implicit multitasking, the Ladder Logic application should allow a sufficient margin for PLC firmware overhead tasks and for variations in scan times due to hardware latencies. Where explicit multitasking is used, margins must also include variations in the application tasks. For example, a 10 msec timed interrupt task may normally execute in one msec. However, under some cases, 10 msec might be required.

This situation will prevent other applications and system overhead tasks from being executed, which will cause a PLC failure. Worst-case conditions must be defined, and measurements of execution times under these conditions for each task must be made. If it is not possible to characterize such worst case conditions authoritatively, multitasking should not be used.

- *Account for concurrent access to global variables.* Another safety related issue in regard to multitasking in PLCs is that, in many cases, each task accesses the same global variable base rather than a separate variable base for each task. The potential for programming errors when global variables can be accessed at different periodicities is significant. For example, an input that is updated by a 500 msec auxiliary task can be directly referenced by a 10 msec fast task. Both of these tasks can read or write to the same internal bits and words. The PLC memory map must be carefully designed, documented, and verified to ensure that concurrent data access has been properly implemented. If it is not possible to model and represent this concurrent access authoritatively, multitasking in conjunction with a global database should not be used.

5.1.3.2 Minimizing the Use of Interrupt-Driven Processing

Ladder logic programs in themselves are not normally implemented using an interrupt driven architecture. However, they do exist within an interrupt driven runtime environment. The indirect impact of interrupt driven processing must be considered in the design of the Ladder Logic application. The following guidelines apply:

- *Account for interrupts in critical response times.* PLC response times can be affected by timer interrupts, local input interrupts, I/O scan interrupts, and other event-driven interrupts. Such interrupt processing may not be under the control of the application programmer. However, since this adds execution time and overhead time (for stack maintenance, etc.) to the overall system response, it must be considered where response times are critical. This issue is further discussed in the following section on I/O timing.

- *Use of interrupts for exception handling and recovery.* Interrupt-driven processing can be an asset to safety when used to recover from processor hangs (via a watchdog timer) or more general processor failures (via a fault routine). These issues are discussed in Section 4.2.

5.1.3.3 Input and Output Timing

The programmer must ensure that the order of program execution is such that variables are updated prior to their use and that the values of inputs or the result of the previous step are current. Timing issues that should be verified in an audit or review of a real-time PLC system depend strongly on factors specific to the methods used by various PLC operating systems for scanning the real world

I/O. Generally speaking, these methods can be classed into four categories:

1) PLC has no separate I/O scan - I/O is updated as required by each rung of Ladder Logic ("Immediate I/O update").

2) PLC I/O scan occurs asynchronously from Ladder Logic scan ("Asynchronous"). This allows values of inputs to change during the course of a single Ladder Logic scan.

3) PLC I/O scan occurs asynchronously from Ladder Logic scan, but input and output values are "captured" in a buffer to eliminate the possibility of variance during the Ladder Logic scan ("Captured Asynchronous").

4) PLC I/O scan is fully synchronous with the Ladder Logic scan ("Synchronous").

In addition, PLC systems vary widely in the delay time (i.e., latency) between when an event related to a sensor occurs and when it is seen by the Ladder Logic system. Similarly, there is a latency between when an actuator is commanded by the Ladder Logic program and the actual activation. These delay times are influenced by:

- The type of sensor signal used
- The input modules' input filter delay
- The I/O scan type mentioned above
- The data rate between the PLC processor and its I/O racks.

Thus, the PLC program design and documentation should explicitly address the I/O impact of response time. Timing issues that may need to be reviewed include:

- *Accounting for multiple scans of the same variable.* In a multitasking software system, an input variable might be read by segments of the program in different scans on PLC systems that allow this (e.g., Immediate I/O and Asynchronous I/O types).

•*Accounting for the effect of hardware-induced latency of input and output signals.* This delay and its characteristics should be known (i.e., measured) and documented as part of the PLC program documentation.

•*Accounting for the effect of sensor induced latency.* Sensors themselves have different response times in differing states. For example, if a proximity switch has a latent response time on both sides (blocked and not blocked), then the software constructs need to be cognizant of this delay, especially when this data is used in conjunction with other data and certain programming methodologies such as one shots.

• *Accounting for the effect of I/O data rates.* Input/output data rates can vary from less than 38.4 kilobits per second (KBPS) to greater than 12 megabits per second (MBPS).

•*Synchronization of replicated PLCs.* Multiple PLCs in safety systems might be used in redundant configurations based on hot backup (dual redundant) m out of n voting or median selection (for triple redundancy and higher). Some of these applications might require that the programs executed on different PLCs be synchronized. If this is indeed the case, care should be taken to ensure selection of a redundant PLC system that supports the desired degree of synchronization. Hot backup, or triply redundant, PLCs have varying types of synchronization, ranging from none to twice per PLC scan as well as explicit synchronization of PLC program execution and variable pool data after the execution of each network of Ladder Logic. As the PLC programmer has little or no control over the synchronization algorithms used, the usage of synchronization of Ladder Logic programs on PLCs of this type is not a direct application-level issue. However, the strengths and limitations of the redundancy management and synchronization design should be well documented and understood. The impact in the design should be explicitly documented, and a rigorous testing program (also beyond the scope of this document) need to be considered.

5.1.3.4 Avoidance of Self-Modifying Code

Most Ladder Logic implementations do not provide any features that allow the program to modify itself. However, modification of run time environment parameters is possible. The following specific guidelines apply to these parameters:

•*No changes to system configuration parameters.* System configuration parameters should be accessed only by the appropriate routines. This can be verified by the use of cross reference tables generated by the programming tool to determine which subroutines are accessing the configuration variables. However, cross reference information WILL NOT show usage of data table areas accessed by indirect and indexed addressing programming techniques. Configuration parameters depend on the specific processor used and should be identified in the design documents.

• *No changes to task periodicities or running tasks.* If multitasking is to be used, there should be no changes to task periodicity, even if it is possible to modify these periodicities from the application program. Some PLCs also allow other types of control over PLC operation, e.g., stopping the PLC program execution or stopping/starting individual tasks. These features should not be utilized in safety critical systems.

5.2 Robustness

Robustness refers to the capability of the software to survive abnormal or other unanticipated conditions. The intermediate attributes of robustness are:

•Transparency of functional diversity
•Controlled Use of Exception Handling
•Input and Output Checking
•Error Containment.

These are discussed in the following sections.

5.2.1 Transparency of Functional Diversity

There are no specific guidelines for functional diversity. The generic guidelines apply.

5.2.2 Exception Handling

The generic guidelines are not directly applicable due to the unique software architecture of PLCs and the interaction with the hardware. The following are specific guidelines for exception handling supported by PLCs:

•Use of system status information for recovery
•Accounting for shutdown behavior
•Use of watchdog timers.

These are described below.

5.2.2.1 Use of System Status Information for Recovery

When available, system status information should be used as part of the detection and recovery process. The nature and extent of the PLC system status monitoring varies among manufacturers and models. Some PLCs provide Ladder Logic software commands which output status bits that indicate abnormal conditions of execution (not restricted to hardware faults). Examples of these problems are arithmetic overflow, full communication queues, bad addresses, and program assembly errors. These bits can be used by the Ladder Logic program to initiate exception handling similar to that for hardware faults. Most PLCs immediately shut down if a RAM memory checksum error or other serious system error occurs, thereby eliminating the need for a status bit for this condition. Figure 5-5 shows a monitoring routine in an Allen Bradley PLC-5 that checks the status of error bits and annunciates to the operator that the system experiences problems. The information can also be used by the programmer to write an exception handling routine which either handles the problem or directs the Ladder Logic application program to a predefined state, such as shutting down the controlled system.

Specific guidelines for use of system status information are:

- *Completeness.* All relevant information should be used to detect and determine the appropriate recovery action.

- *Correctness.* The recovery action should be appropriate for the condition.

- *Observability.* The Ladder Logic program should annunciate and log the condition.

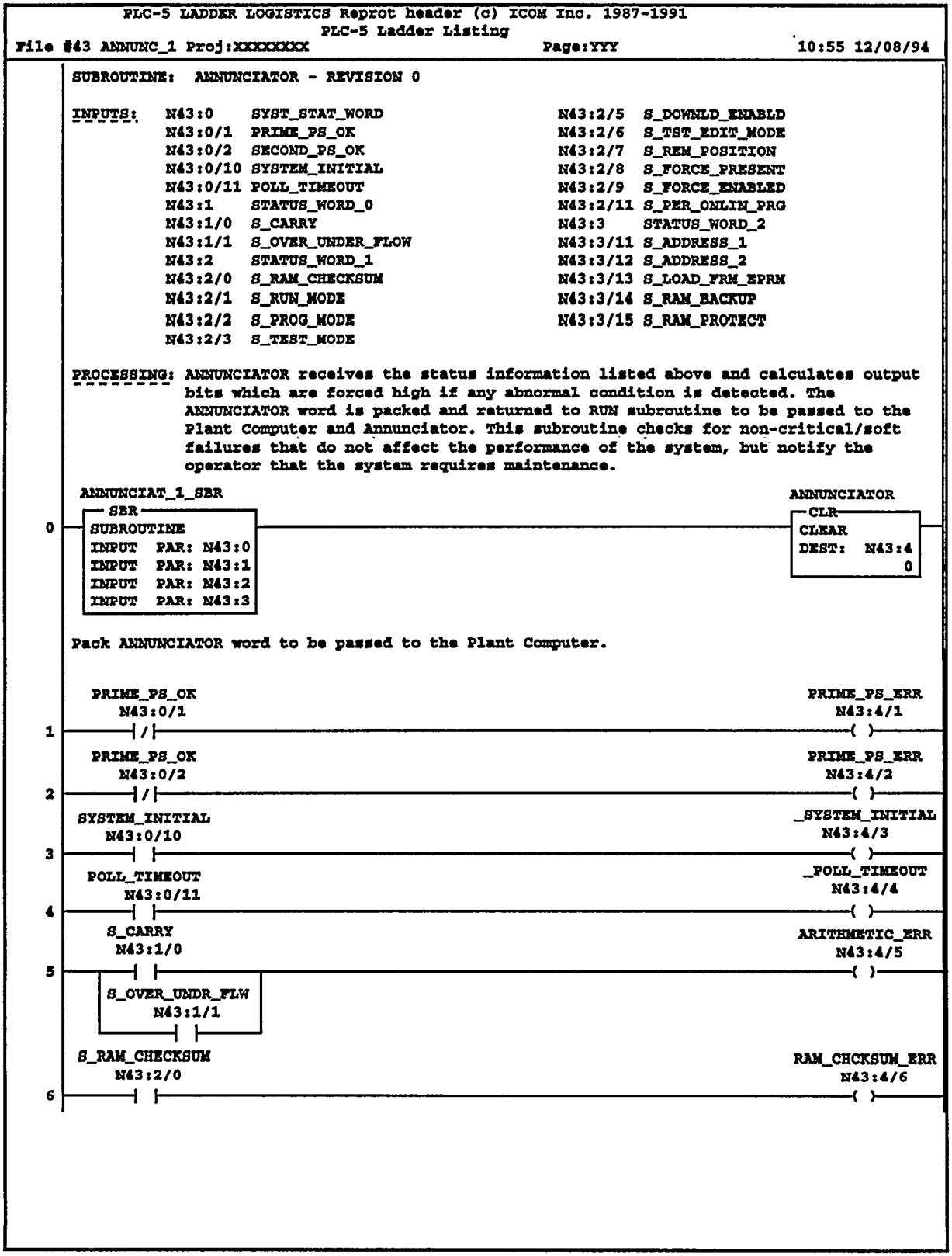


Figure 5-5 Health Monitoring Routine Sample Program

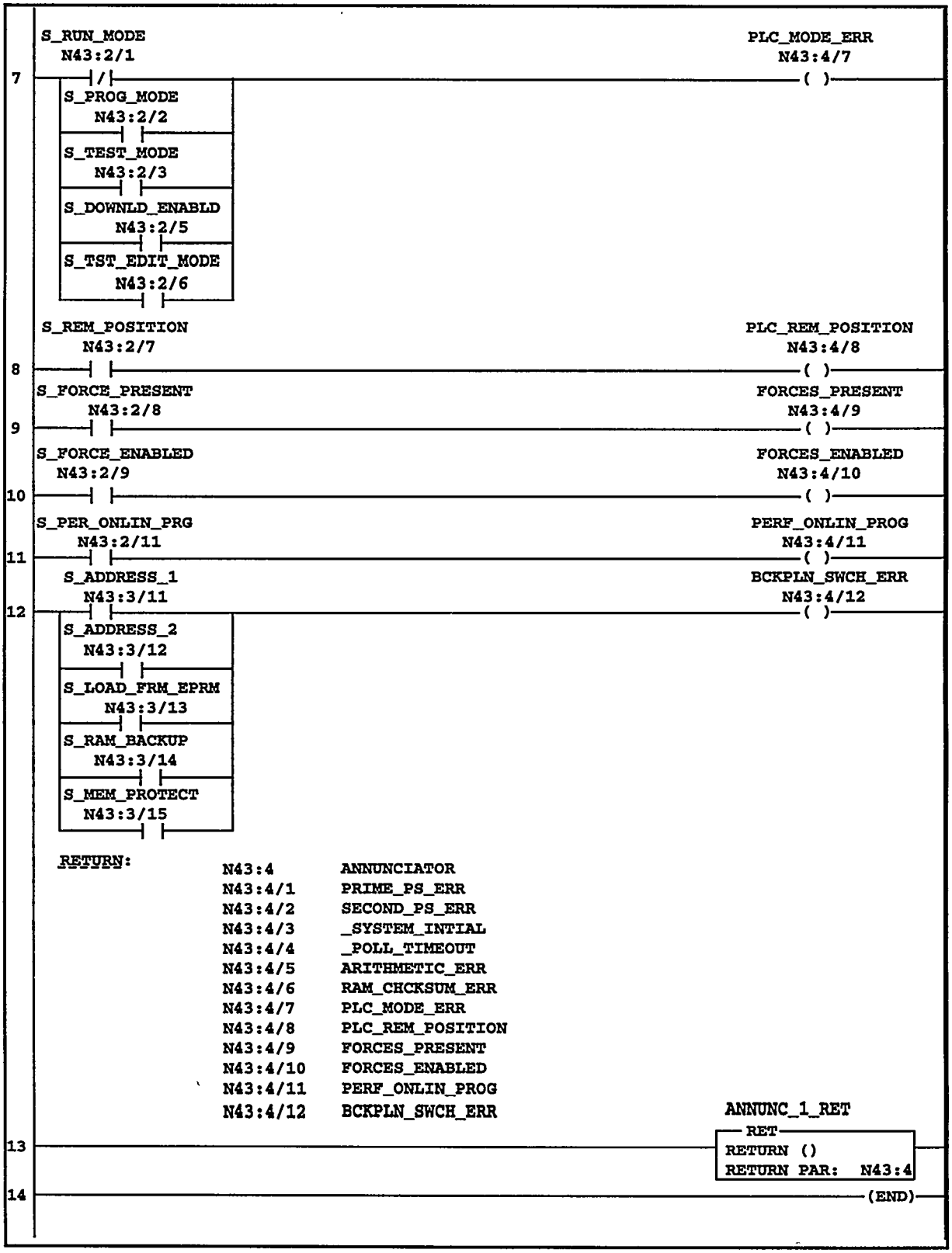


Figure 5-5 Health Monitoring Routine Sample Program (continued)

5.2.2.2 Accounting for Shutdown Behavior

The Ladder Logic program should properly account for PLC behavior at shutdown. Generally, all outputs turn off, but this is not always true. The PLC system is designed so that such a shutdown places the system in a fail-safe condition.

Some PLCs have the capability to run a designated Ladder Logic subroutine which the processor automatically executes when it encounters a condition that will cause execution of the main Ladder Logic routine to stop. In the PLC5, this subroutine is called a "fault routine". It allows the designer to decide on the appropriate action, including shutting down the system in a safe manner. An example of a simple fault routine is shown in Figure 5-6. The routine annunciates to the operator that the system is experiencing problems and brings the system to a halt. Another example, shown in Figure 5-7, clears the major fault error bits and restarts operation by forcing the PLC to perform a startup procedure. Should the fault still exist, then the fault word will be set to reflect this, and the fault routine will be executed again. It may be desirable to limit the number of times the fault routine runs in some cases.

The following specific guidelines apply to exception handling fault routines:

- *Completeness.* The fault routine cannot be relied on to detect all instances of program crashes. Additional provisions that may be required by the specific safety requirements of the application for PLC major faults must be specified.

- *Observability.* The fault routine should annunciate and log the condition. The execution of the fault routine should not be masked.

- *Validity checking.* The conditions under which the fault routine is running may have corrupted program memory, data files, or I/O. The fault routine must ensure the validity of its environment before proceeding to execute.

- *Fail safe properties in the absence of the fault routine.* The fault routine cannot be relied upon to operate under every major failure condition. The PLC may be so disabled that this is not possible. Thus, the system design should ensure a safe state in the absence of the successful execution of the fault routine.

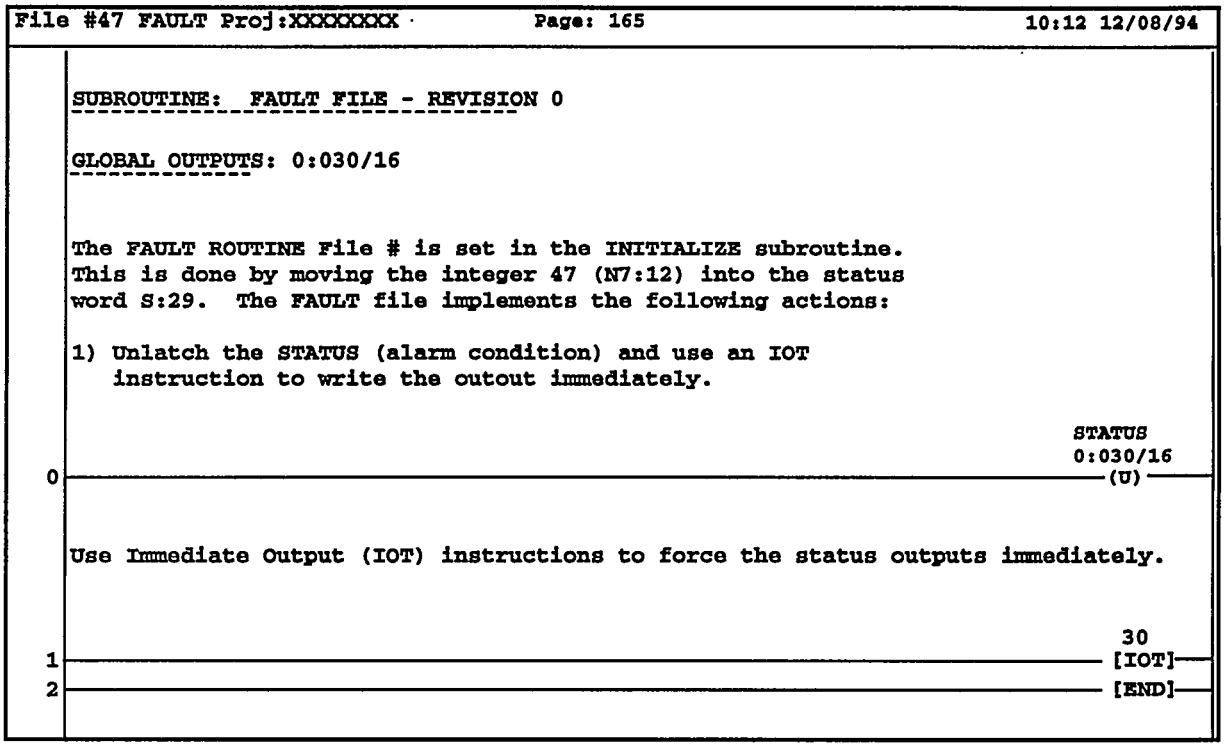


Figure 5-6 Fault Routine That Alarms and Halts Sample Program

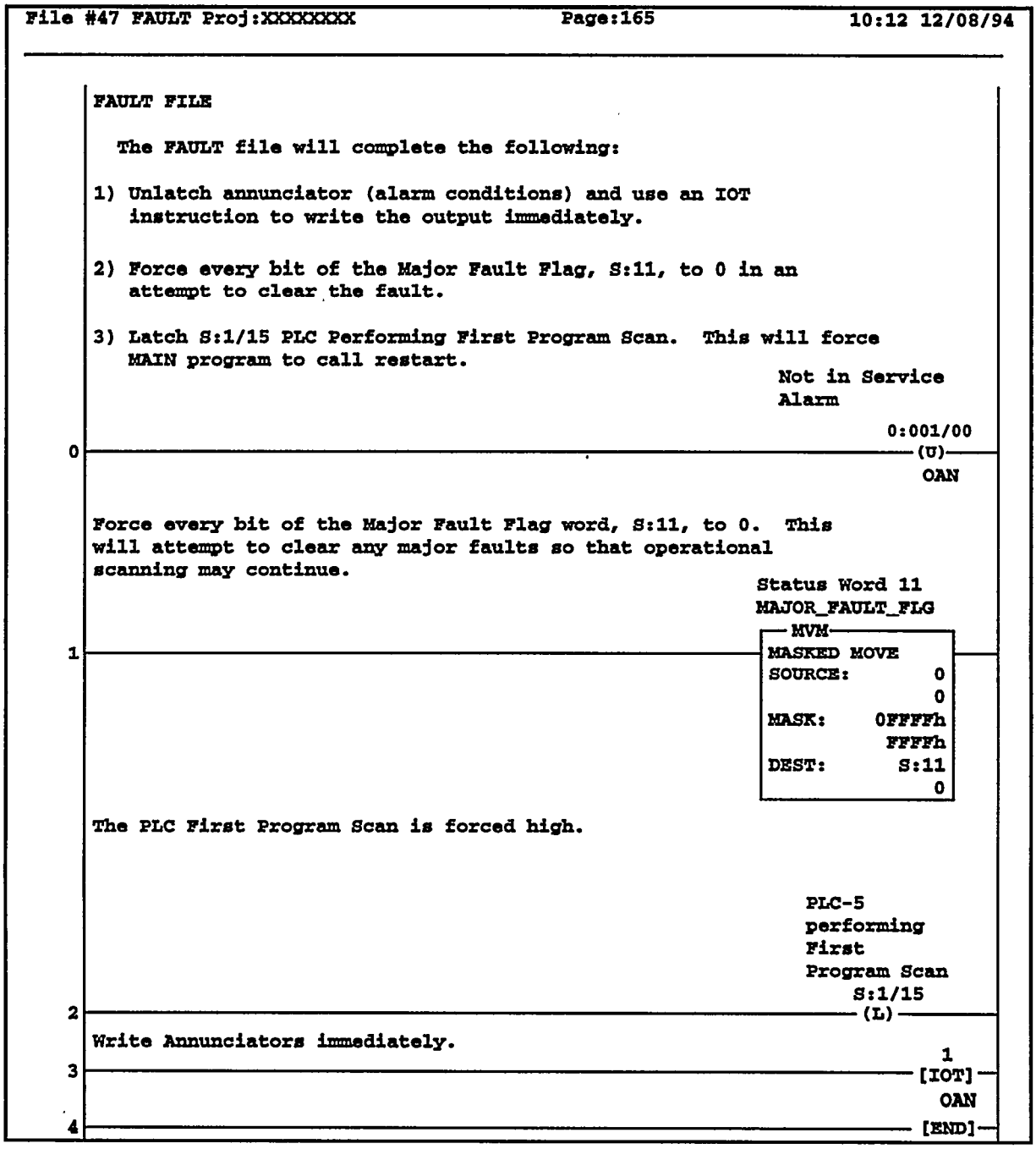


Figure 5-7 Fault Routine That Restarts Operation (Sample Program)

5.2.2.3 Watchdog Timer

The PLC system provides an internal watchdog interval timer which is either fixed or set by the program (depending on PLC manufacturer). The fixed watchdog timer is typically utilized to protect against a stopped or hung CPU. The timer expiration will shut down the PLC explicitly. The

software-based watchdog timer is typically utilized to protect against excessive scan times caused by infinite loops and related failure modes. If under program control, the timer interval should be set during initialization. If the program scan time exceeds this value, the interval timer expires. Once the timer interval expires, the PLC halts and declares an error condition. This provides a mechanism for identifying each scan during which the program exceeds its expected execution time.

Both the Ladder Logic program and the system design should contain provisions to recover from the timer expiration condition. Ladder Logic provisions include programming the fault routine to handle the watchdog timer fault bit. System design measures can include an external watchdog timer, independent of the PLC, that will handle the fault (e.g., by alarming) in case the PLC crashes and cannot execute the fault routine. The external timer is a second line of defense in the event of a failure of the Ladder Logic recovery.

5.2.3 Error Containment

The generic guideline has limited applicability. Depending on the capabilities of the PLC, it may be possible to separate local variables from global variables that provide one line of defense. The second line of defense is data validation when variables are passed among ladder logic routines, or when input or output occurs. This was discussed earlier under avoiding interface ambiguities.

5.3 Traceability

Attributes specifically related to traceability include the use of built-in functions and compiled program libraries.

5.3.1 Use of Built-in Functions

The generic guideline applies. Ladder Logic includes built-in function blocks for frequently used functions. Ladder Logic applications rely on the PLC operating system and the supported function set.

The robustness of the PLC operating systems is a function of the quality of the development process. Generally speaking, PLC operating systems are produced under strict software quality controls, and are extensively tested. The quality and integrity of operating systems must be affirmed by the commercial grade dedication process that qualifies the use of the PLC in safety-related applications.

The function set is defined by the PLC manufacturer and these functions are implemented by the PLC firmware. The quality and integrity of these built-in functions must be affirmed by the commercial grade dedication process that qualifies the use of the PLC in safety-related applications.

The built-in functions provided by the PLC are usually simple building blocks and do not obscure the traceability between the code and the design specification.

5.3.2 Use of Compiled Libraries

The generic guideline applies. Compiled libraries should be used with caution. Some PLCs support external libraries as optional function blocks written in C or PL/M. In the case of Allen Bradley PLC-5, they are called "custom application routines" or CARs. They perform functions such as mass flow control. These routines are 68000 native code, which the PLC 5 executes. Data is passed back and forth via the PLC data table. Add-on libraries may also be written in Ladder Logic, available from the manufacturer and other vendors. The following specific guidelines apply:

- *Accounting for interfacing and integration issues.* Where functions from these libraries are used, special care must be taken to review the integration of these functions into the PLC Ladder Logic program, such as unintended side effects.

- *Development process.* The same testing, validation, documentation, and visibility into the development process must be applied to the function blocks as the Ladder Logic software resident on the parent PLC.

- *Assessing accuracy and robustness.* The accuracy and robustness of the libraries must be understood as part of the dedication process. This understanding can generally be gained through testing. However, if source code is unavailable, the testing of necessity must be at the functional or "black box" level. Careful judgement in assessing the results of such testing is necessary.

- *Timing issues.* The latency in passing data to the routines and receiving data from the routines must be understood and documented.

Coprocessors offered by some PLC manufacturers are related to compiled libraries. Coprocessors are separate processing boards installed in PLCs that accept conventional programming languages such as C or BASIC. The software programs written in these languages and executed on a different processor can be used by the ladder logic as function blocks. When coprocessors are used, the following additional guidelines apply:

- *Accounting for interface and integration issues.* As was the case with compiled libraries, special care must be taken to review the integration of coprocessors into Ladder Logic, particularly with respect to the use of memory and for unintended side effects. Additional issues are the extent to which hardware error checking is incorporated when data are passed across a bus or via direct memory transfers. Additional validity checks in software may be necessary. These considerations should be documented.

- *Development process.* As was the case for compiled libraries, the same testing, validation, documentation, and visibility into the development process must be applied to the function blocks resident in coprocessors as the software resident on the primary PLC.

- *Failure behavior and robustness.* The coprocessor hardware platform should have the same hardware failure behavior robustness as the "parent" PLC. If not, the software design should account for the differences.

5.4 Maintainability

The software maintainability lower-level attributes in this section are limited to those affecting safety. These include the following:

•Readability
•Abstraction
•Functional cohesiveness
•Malleability
•Portability.

5.4.1 Readability

The generic guidelines apply. Readability is essential for review and maintenance of PLC Ladder Logic safety systems. The graphical notation of Ladder Logic can facilitate understanding the operation of a single Ladder Logic network. Understanding a complete Ladder Logic program, however, requires the reader to understand the interactions between many Ladder Logic networks operating on a global variable database. In many cases, the interaction occurs between Ladder Logic networks that are pages apart in the documentation. Thus, the programs and databases must be structured to facilitate understanding by individuals other than those who wrote the code. The following specific guidelines apply:

- *Overview documents.* Since there is no ladder logic overview function, the review of any program for readability should include a general overview document. A program flow chart can be used to document control flow. Documentation should not just explain the purpose of each network but the purpose of each section of PLC program. The documentation must be maintained together with the code as changes are made.

- *Documentation of PLC data files.* An important component of documentation readability concerns the documentation of usage of data files — particularly if they are global — with a data flow description among the data tables.

5.4.1.1 Notation

The generic guidelines are not applicable. Ladder logic notation is determined by the characteristics of the specific programming package used. This notation is not readily modifiable by the end user.

5.4.1.2 Conformance to Indentation Guidelines

The guidelines are not applicable to Ladder Logic.

5.4.1.3 Descriptive Identifier Names

Ladder Logic supports the use of descriptive identifiers or tagnames, with lengths between 7 and 32 characters being common. In addition to the identifier, each variable can be described by an address description. A typical address description has 5 lines of 15 characters each.

The following are specific guidelines:

- *Inputs and outputs.* The identifiers should be as similar as possible to the names used externally (e.g., P&ID numbers). Use of the same variable name for different purposes is not allowed in Ladder Logic.

- *Consistency with project notation.* Ladder Logic names should be consistent with design documents.

5.4.1.4 Comments and Internal Documentation

The generic guidelines apply. Ladder Logic supports internal documentation by means of "rung descriptions" and "section headers."

The following are specific guidelines:

- *Revision level.* An important internal documentation feature is the revision level. In some PLCs, if the revision level is recorded as a comment, it will be disassociated from the code when it is downloaded to the PLC. To avoid configuration management problems in such systems, it is recommended that the revision level be recorded as part of the program itself by storing it in memory as a variable. Figure 4.3 is an example which shows the subroutine version marked as a comment (not the preferred practice in this case) and not as a memory location.

- *Interfaces.* Detailed and unambiguous descriptions of subroutine interfaces and functions are another important documentation feature that should be verified. As shown in Figure 4.3, each subroutine should have a detailed description of the input parameters, global variables (if any), the processing performed by the subroutine, output parameters returned, effect on global variables, and side effects (if any).

- *Calling hierarchy.* The level of documentation required for incorporation in the program depends on the complexity of the program/subroutine and on the description provided in other accompanying documents such as the software design description. Two important issues to be documented are (1) the hierarchy of subroutines and who is calling whom, and (2) the flow of data and information among subroutines. These two items, especially the second one, are important to understand the system and enable independent review. Some programming shells provide a database and cross references of all data-table variables used by the program. The designer or an auditor can use these tables to track the flow of information.

5.4.1.5 Limitations on Subprogram Size

The generic guidelines apply. Due to the limited number of Ladder Logic rungs that can fit on a single page of documentation, limiting the size of subprograms is important. It is difficult for a program auditor to follow operation of any program over more than a few pages. However, Ladder Logic as a language does not enforce any limitations on subroutine size. Moreover, some PLCs only support the division of programs via JMP to label instructions as there is no subroutine support. Thus, decisions on program size limitations are dependent on the properties of the individual Ladder Logic implementation and the project needs. The following are specific guidelines:

- *Use subroutines and subprograms.* For Ladder Logic implementations supporting subroutines and nesting, there should be a limit on the maximum number of rungs per routine. Even for PLCs without subroutine capabilities, it should be possible to subdivide the application into a set of manageably sized subprograms. (The distinction is that after a subroutine is executed, control is transferred back to the calling program without an explicit JMP statement). An upper limit might be 50 rungs, but even limits as low as 10 rungs may be appropriate where visibility is important.

- *Avoid arbitrary program division.* The basis for subdividing programs should be by function, responsibility, or class of data. This guideline is related to functional cohesiveness described below.

5.4.1.6 Minimize Mixed Language Programming

The guidelines on minimizing mixed language programs are partially applicable. IEC 1131-3 compliant systems support mixed language programming among the five defined languages in the

IEC 1131-3 specification. The reason why there are five languages is that each has strengths and weaknesses. Ladder Logic, for example is an excellent tool for expressing Boolean relationships between entities, as in an alarming function. However, it is not as clear as Sequential Function Charts (SFCs) for sequencing operations, nor is it as readable as Structured Text (ST) for complex mathematical operations.

Thus, readability and maintainability of PLC programs are enhanced when each of these languages, if available, are utilized for their strengths. However, a judicious balance must be struck. The following are specific guidelines:

- *Ensure that proper tools are within the development organization.* Such tools include compilers, debuggers, cross reference generators, testing, and documentation aids. A multiple language safety application should not be contemplated without adequate support for maintenance and enhancements for *all* languages used in the applications.

- *Use each language according to its strengths.* Mixed languages should be used because the resulting application is easier to maintain or more robust. Additional languages should not be introduced gratuitously into a safety application. Justification for the use of each additional language should be included in the documentation.

5.4.1.7 Minimize Obscure or Subtle Programming Constructs

Each make and model of PLC supports a number of obscure and sometimes counter-intuitive programming constructs in their Ladder Logic implementations. These are normally peculiar to specific implementations. There should be project guidelines relating to the specific characteristics of the PLC. It may be advisable to consult the manufacturer's technical support organization to obtain such information. The following are guidelines common to multiple PLCs (however, they may not be applicable to all PLCs):

- *Avoid use of overlapping JMP to label statements or to labels that precede the JMP in the code.* Different systems will execute overlapping or backwards jumps differently, and sometimes in unpredictable ways.

- *Minimize indirect addressing.* Although program constructs can be more concise using these addressing techniques, the addresses and functionality presented are not obvious. Without the proper tools and documentation, however, the underlying logic could be overlooked. Such indirect addressing should be used sparingly and with adequate documentation.

- *Use indexed addressing for repeated elements only.* Indexed addressing should be used where there are repeating elements (e.g., thermocouples on a single hot leg). They should not be used for grouping elements with diverse meanings (e.g., a temporary storage variable at one location, the value of a sensor at the second, etc.).

5.4.1.8 Minimize Dispersion of Related Elements

In general, PLC programs are stored by their development environments as a single file or group of files. This precludes the dispersion of related elements among several files from being an issue with the majority of PLC implementations.

However, there are PLC systems that do not conform to this general statement. When dealing with a system of this type, it is important that logically related elements of the program remain in a single file so as to minimize any confusion in locating and understanding them.

5.4.1.9 Minimize Use of Literals

The generic guideline is partially applicable. Most, but not all, Ladder Logic implementations support an area of the global variable pool that is writable by the development environment but not by the PLC program itself. The actual nature of these "Constant" variables (to use the IEC 1131-3 nomenclature) varies from implementation to implementation. When available, the use of variables from this constant pool is preferred to the use of literals. However, Constant variables may not be available on all PLC systems; in such systems, literals are necessary.

5.4.2 Data Abstraction

This principle depends on the following specific base attributes:

- Modularity
- Information hiding
- Minimizing the use of global variables
- Minimizing the complexity of the interface and defining allowable operations.

PLC Ladder Logic does not provide the advanced features of object-oriented languages, such as C++, to support abstraction. However, Ladder Logic provides some tools that can help achieve abstraction.

5.4.2.1 Modularity

The generic guidelines are applicable. Some Ladder Logic implementations support modularity through the subroutine structure; however, the language does not enforce use of subroutines and design of cohesive functions. Even in the absence of this supporting language features, all Ladder Logic programs should be organized as a number of distinct subprograms, each with a particular function, dedicated variable area, and each fully documented. Passing of information between these

subprograms should be accomplished via a well documented and consistent methodology (guidelines are discussed under global variables).

In the event that subroutines are not available on the PLC system chosen for a particular project, the Ladder Logic program should be arranged into a series of subprograms, each with a particular function, in order to enhance the understanding of the program.

5.4.2.2 Information Hiding

The generic guidelines apply to those Ladder Logic implementations that support the concept of information hiding through the use of local variables that no other subroutine can access or alter. The Ladder Logic program should be designed to use parameter passing to subroutines through formal parameter interfaces. Even if the parameter is a global variable that is visible inside the subroutine, it should be passed to the subroutine as a parameter.

For PLCs that do not allow subroutine parameter passing or local variables (at the time of this writing, most do not), information hiding through formal parameters cannot be supported. However, as described in the next section, there are techniques using the global PLC data tables that can be used.

5.4.2.3 Minimizing the Use of Global Variables

The generic guidelines apply only to those PLCs and implementations of ladder logic that support local variables. Global variables can be accessed from any part of the Ladder Logic program. Thus, they can cause side effects or unintended behavior through deliberate or inadvertent modification by various programmers working on different parts of the program. Local variables should be separated from global variables for those Ladder Logic implementations that support local variables. In most PLC systems, local variables are *static* memory locations, that is, they maintain their value after the subroutine returns. However, support for local variables is not common in current PLC Ladder Logic implementations; most currently use a single global variable pool. The following guidelines apply to the management of the global data memory area when local areas are not supported:

- *Separate variables by usage.* Usage of variables within this pool can be controlled by the PLC programmer to separate the handling of local and global variables. This can be achieved by setting aside distinct areas of memory for use only by single PLC subprograms (i.e., local memory areas). Passing of variables to and from subprograms should be accomplished by "transfer" variables used for this purpose only. The method for such transfers should be consistent in all of the application subprograms.

- *Use transfer variables.* Interface to subroutines on PLCs that do not support parameter passing is via the use of global variables. It is recommended that, on systems of this type,

that specific variables be explicitly designated for the input and output parameters associated with each PLC subprogram.

- *Use support tools and documentation for global memory areas.* A careful examination of the PLC memory map, with the aid of the cross-reference features normally found in the PLC program development environment, is mandatory to ensure safe PLC programming. The exact features, layout, and composition of a PLC cross-reference listing vary between PLC programming packages. For example, ICOM software has a feature that applies local/global flags to data table files. (It is not part of the PLC firmware, and does not serve any purpose when using another programming software package.) In general, these listings show which PLC variables are being used, in what part of the program they are being used, and whether they are being read from or written to.

- *Ensure proper index variable bounds.* Some PLCs support treating the variable pool (or a section of it) as a large array and allow indexing into this array. Expressions using this indexing should be carefully audited to ensure that the index value remains within the value of the array under all circumstances.

5.4.2.4 Minimizing Interface Complexity

The specific guidelines related to interface complexity are the same as the transfer variable, global memory area partitioning, and documentation guidelines discussed above.

5.4.3 Functional Cohesiveness

The generic guidelines apply. Every subprogram should have one clearly discernable purpose with input and output parameters related to that purpose. Two or more different functions should not be combined in a single subprogram.

5.4.4 Malleability

Malleability is the ability of a software system to accommodate changes in functional requirements. Ladder Logic allows programmers to create code which is hard to change. However, the guidelines related to modularity, minimizing obscurity, interfacing, global memory management, and portability can be used to achieve malleability.

5.4.5 Portability

Portability is a safety concern required by the need to minimize changes when replacing or upgrading

equipment. The features, functionality, syntax and semantics of the various implementations of Ladder Logic for PLCs and PLC-like systems vary widely, more so than any of the other languages considered in this report. It is difficult, therefore, to make sweeping statements about safety-related aspects of portability. Nevertheless, over the plant life, it is unlikely that the same runtime environment will be supported since every vendor only supports its own equipment and upward compatibility (i.e., programs executed on an older processor will also execute on the newer processor ladder) is not always provided. When new processors are introduced, the instruction set is usually so different that the application should be re-written anyway to take advantage of the new firmware. The objective of maximizing portability is to reduce the likelihood that changes will introduce dangerous faults.

Unfortunately, conforming to the IEC 1131-3 standard at present will not guarantee portability. Currently, this standard is vague in many areas where PLCs vary. Moreover, not all PLC manufacturers have committed to supporting the standard even in its current form. However, as has happened in other areas of computing, pressure for standardization will grow. As this occurs, conforming to an extended IEC 1131-3 standard will enhance portability.

Although portability of the Ladder Logic program itself may not be possible, the *design* and *approach* can be made portable. Candidate areas for such unified approaches are common PLC functions including:

- Analog programming
- Alarm handling
- Fault/exception handling
- Operator interface
- Closed loop control programming
- Variable frequency drive interfacing
- Computer communications
- Data logging.

A consistent approach to these areas will provide common code and will result in greater portability to new runtime platforms.

5.5 Security

Security in this context refers to the protection of computer software from accidental or malicious access, modification, or destruction. The discussion in this section is restricted to security measures associated with the Ladder Logic language and its associated program development environment.

The main concern of security when handling PLC systems is that an unauthorized person might gain access to the program and:

- Change the program or the data in the PLC memory
- Change the PLC configuration
- Download a wrong program
- Leave the system in the wrong "mode" after maintenance
- Force inputs and/or outputs.

Security concerns are particularly acute when program or hardware maintenance is performed. The key issues are password protection and physical access. The latter is not a language feature, but it is mentioned here because the PLC environment is vulnerable to security infringements by improper change of ROM components.

Software maintenance on a PLC can be performed either by connecting an external PC to the PLC, or from a user interface station that might run a Supervisory Control and Data Acquisition (SCADA) package that interfaces with the PLC. The nature and level of this type of password protection vary from PLC programming package to PLC programming package. In some cases, the interfacing software packages provide password protection with multiple levels of access rights that allow people with different skills and authority to perform only the functions for which they are authorized. Other PLC systems come with keys and locks that only allow modification of the PLC program after the key is inserted. However, PLC programming packages have no security provisions whatsoever.

The auditor should verify that the design requires minimum operator access to software. Whenever operator access is necessary, the system should be designed to include security measures in the application proper, rather than relying exclusively on interfacing software.

Some PLCs have implemented a security system which is part of the PLC firmware. This will limit interaction with the PLC memory contents based upon access rights (Allen Bradley, 1991). Because it is firmware-based, the passwords are also resident in the memory of the PLC. If this feature is to be exploited, the runtime software package used to develop the ladder logic must support it.

References

Allen Bradley, *PLC-5 Programming Software – Programming*, Publication 6200-6.4.7 November 1991.

Allen Bradley, *PLC-5 Programming Software – Software Testing and Maintenance*, Publication 6200-6.4.10 November 1991a.

ICOM PLC-5 Ladder Logistics, User's Manual, 1989.

International Electrotechnical Commission (IEC), *Programmable Controllers General Information,* IEC Standard 1131, Part 1, 1992. (Available in the U.S. from the American National Standards Institute, New York.)

International Electrotechnical Commission (IEC), *Programmable Controllers Programming Languages,* IEC Standard 1131, Part 3, 1993. (Available in the U.S. from the American National Standards Institute, New York.)

SoHaR Incorporated, *Generic Attributes for High Level Languages, Task 1*, SoHaR Inc., Contract RES-04-94-046, Beverly Hills, CA, October 1994.

Institute of Electrical and Electronic Engineers, ANSI/IEEE 729-1983, *Glossary of Software Engineering Terminology*, 1983.

6 Sequential Function Charts

PLC Sequential Function Chart (SFC) programs do not resemble traditional high-level languages. Instead, SFCs are program structure tools that present a visualization of the underlying control flow. The SFC structure includes both steps and transitions; each step and transition is implemented in an underlying IEC 1131 language (ladder logic, structured text, instruction lists, or functional block diagrams). The charts provide a higher level of abstraction that hides lower level details handled in the underlying languages. An introduction and basic description of SFCs in the context of IEC 1131 is contained in Appendix A.3. As noted in that section, SFCs are best used in applications where the execution can be partitioned into distinct steps.

This chapter discusses the applicability of the generic attributes to PLC SFCs. The chapter is organized in accordance with the framework of Chapter 2. Section 6.1 discusses reliability-related attributes of SFCs; Section 6.2 discusses robustness-related attributes of SFCs; Section 6.3 discusses traceability-related attributes; and Section 6.4 describes maintainability-related attributes. A summary matrix showing the relationship between generic and language-specific guidelines, together with weighting factors, is included in Appendix B. Language-specific weighting factors were based on the limited nature of the language, which has no variables, data types, or subroutines.

6.1 Reliability

Reliability is either (1) ability to perform the required functions under stated conditions for a specified period of time (IEEE, 1990) or (2) the probability of successful operation upon demand (IEEE, 1977; p. 584). The reliability of an SFC program depends on the run-time predictability of the following:

- Memory utilization
- Control flow
- Timing.

SFC-specific guidelines derived from these generic attributes are described in the following sections.

6.1.1 Predictability of Memory Utilization

SFC programs do not directly allocate memory. Thus, the generic guidelines are not applicable at the SFC level. However, they are applicable at the underlying language level. The previous chapter has a discussion of these issues for Ladder Logic.

6.1.2 Predictability of Control Flow

Predictability of control flow is the capability to determine easily and unambiguously what path (i.e., which set of branches and in what order) the program will execute under specified conditions. Related base attributes are:

- Maximizing structure
- Minimizing control flow complexity
- Initializing variables before use
- Single entry and exit points for subprograms
- Minimizing interface ambiguities
- Use of data typing
- Accounting for precision and accuracy
- Order of precedence of arithmetic, logical, and functional operators
- Avoiding functions or procedures with side effects
- Separating assignment from evaluation
- Proper handling of program instrumentation
- Controlling class library size
- Minimizing use of dynamic binding
- Controlling operator overloading.

Guidelines related to predictability of control flow for SFCs are discussed in this section.

6.1.2.1 Maximizing Structure

The generic guidelines are applicable. Use of *goto* statements or equivalent execution control statements that result in an unstructured shift of execution from one branch of a program to another are difficult to trace and understand. Although SFCs allow the programmer to use *goto* statements, they should not be used in safety-critical applications with one exception: handling out-of-sequence events in an abnormal situation. This situation was discussed in Section 5.2.

6.1.2.2 Minimizing Control Flow Complexity

The generic guidelines are applicable. Although SFCs have a limited syntax, it is possible to create SFCs that are quite complex. Hence, the following guidelines:

- *Limit the number of parallel paths.* The number of parallel paths at the beginning and end of a logic zone should normally be limited to seven (Hughes, 1989; p. 178).

- *Limit use of SFC to sequential operations.* Use of SFCs in non-sequential applications (e.g.,

state machines) will result in a large number of directed links and divergence of sequence selections, resulting in an overly complex SFC. This should be avoided.

Use of Macro-steps as a means of simplifying the appearance of SFCs was discussed in Section 5.4.

6.1.2.3 Initializing Variables Before Use

SFCs do not handle initialization because they do not have variables. Thus, the generic guideline is not directly applicable at the SFC code level, but is applicable at other levels. The following are specific guidelines:

- *Accounting for initialization as part of the program design:* SFC-specific variables, when they exist, are typically initialized and maintained by the PLC system, and so there are no application program initialization issues concerning them. These variables are maintained in the same data table, using the same data types that the PLC uses. Thus, initializing variables used in the languages that define the step actions and the transition conditions is an issue. The SFC Initial Step is an appropriate place for code performing this initialization

- *Initialization of process steps and transitions:* Within each process step and transition, there are initialization issues with the lower level IEC 1131 language (e.g., Ladder Logic).

6.1.2.4 Single Entry and Exit Points for Subprograms

The generic guideline is applicable. The SFC grammar allows only single entry and exit points (called *transitions)* from each process step. Macro-Steps, as well, may only have single entry and exit points. However, it should be noted that the control language in each one of the process steps or transitions may involve multiple entry points . The previous chapter discusses these issues for Ladder Logic.

6.1.2.5 Minimizing Interface Ambiguities

SFC does not support any interfaces. However, there is an issue of interfaces *between* steps with respect to latching bits. In order to have a bit stay on between steps, the bit has to be latched since all non-retentive bits are reset in the post scan. However, latching bits can cause a problem during initialization as well as during runtime if the bits are not reset immediately[24]. The specific guideline is therefore to avoid use of latching bits.

---

[24]An incident that occurred to one reviewer is that a main motor bit was latched 'ON'. When a circuit breaker tripped, the motor came on immediately because the bit was not reset explicitly. This condition could have caused a major accident.

6.1.2.6 Use of Data Typing

The generic guideline is applicable to the underlying languages, not to SFCs themselves. Some SFC implementations do not have variables (Allen Bradley, 1989); therefore, there are no data types. Other SFC implementations have variables associated with each step. In one such system, each step has a step bit (X0, X1, etc.) that is on when that particular step is active. Each step may also have a step timer (X0,V, X1,V, etc.) that indicates the length of time that step has been active. However, the data types of these step-associated variables are fixed.

However, the underlying IEC 1131-3 languages do have varying degrees of support for data typing. For example, as was described in the previous chapter, PLC Ladder Logic provides few data types. The specific guideline with respect to SFC programs is to use data types to the maximum extent possible.

6.1.2.7 Accounting for Precision and Accuracy

Some SFC implementations do not have variables; therefore, the guidelines are not applicable for SFCs. However, the guidelines are applicable for the languages used within each step.

6.1.2.8 Order of Precedence of Arithmetic, Logical, and Functional Operators

The mail issue regarding order of precedence in SFC is what occurs when multiple transitions in a divergence of sequence selection are evaluated as true simultaneously (i.e. on the same PLC scan). Depending on how the SFC is implemented, the leftmost sequence may be selected, or all valid sequences may be selected.

All transition conditions involved in a divergence of selection sequence should be programmed to be mutually exclusive in order to exclude the possibility of multiple transitions involved in such a structure being evaluated as true simultaneously. This is actually a requirement of the IEC 848 SFC standard.

However, the guidelines are applicable for the underlying IEC 1131 languages used within each step. If Ladder Logic is used within a step, the applicable guidelines are found in the previous chapter.

6.1.2.9 Avoiding Functions or Procedures with Side Effects

Generic guidelines are applicable.

6.1.2.10 Separating Assignment from Evaluation

As noted above, SFCs vary in their support for variables and assignment; therefore, the guidelines are not applicable for SFCs. However, the guidelines are applicable for the underlying IEC 1131 languages used within each step. If Ladder Logic is used within a step, the applicable guidelines are found in the previous chapter.

6.1.2.11 Proper Handling of Program Instrumentation

Program instrumentation generally depends on the programming support environment for the PLC and not on the SFC itself; therefore, the generic guidelines are largely inapplicable. However, the guidelines are applicable for the underlying IEC 1131 languages used within each step. For ladder logic, the issue of program instrumentation discussed in the previous chapter (Section 5.1.2.11) are applicable.

As mentioned above, some SFC implementations have variables associated with the execution state and execution time of steps. These variables are a form of instrumentation. Tracking usage of these variables, as well as all others in the PLC, is a major aspect of ensuring PLC program safety.

6.1.2.12 Controlling Class Library Size

Neither SFC nor the underlying IEC 1131 languages support classes and objects; therefore, the generic guidelines are not applicable.

6.1.2.13 Minimizing Dynamic Binding

Neither SFC nor the underlying IEC 1131 languages allow dynamic binding. All structures must be defined by the programmer before compilation. The generic guidelines do not apply.

6.1.2.14 Controlling Operator Overloading

Neither SFC nor the underlying IEC 1131 languages allow operator overloading or polymorphism. The generic guidelines are not applicable.

6.1.3 Predictability of Timing

Predictability of timing is crucial in a safety system used in real-time control. This section discusses SFC-specific issues related to:

- Minimizing tasking
- Minimizing interrupt processing
- Divergence of sequences
- Simultaneous sequences
- Accounting for scans and post scans.

6.1.3.1 Minimizing the Use of Tasking

At the source code level, SFC does not support multitasking; therefore, the generic guidelines are not applicable. However, it should be noted that the operating system in the PLC firmware may include a multitasking kernel which may support execution of multiple independent SFCs. Such multiple independent SFCs should be avoided in safety applications.

6.1.3.2 Minimizing the Use of Interrupt Driven Processing

The generic guidelines have limited applicability. SFCs themselves do not support interrupts. Should a condition occur which requires immediate attention, the SFC program cannot service the request due to the sequential nature of execution. This issue is discussed further in the section on exception handling.

It should be noted, however, that the firmware or runtime environment program associated with the SFC might use interrupts. It is therefore necessary to demonstrate that the system/software can meet all of its timing and safety function requirements under the most demanding conditions of interrupt occurrence.

6.1.3.3 Divergence of Sequence

The following are specific guidelines for divergence of sequence A divergence of sequence selection is represented in SFC by a single horizontal line under a step, followed by multiple parallel transitions. Appendix A explains divergence of sequence.

- *Define mutually exclusive transition conditions.* All transition conditions involved in a divergence of selection sequence should be programmed to be mutually exclusive in order to explicitly exclude the possibility of multiple transitions involved in such a structure being evaluated as true simultaneously. This programming style is mandated by the IEC 848 SFC standard.

- *Ensure convergence of sequence following divergence of sequence.* Any divergence of sequence selection must eventually be followed by a convergence of sequences, where the alternate sequence paths reunite. This should be checked by the auditor, as well, although most SFC editors enforce this.

- *Account for limits on the number of transitions.* Limits on the number of transitions that can be placed in a divergence of sequence selection vary from implementation to implementation. These limits should be accounted for in the design.

6.1.3.4 Simultaneous Sequences

In the event that multiple transition conditions evaluate as true simultaneously (i.e., on the same PLC scan), different implementations of SFC will result in different behavior.

- *Avoid dependence on execution order.* On some systems, the leftmost branch is selected; on others, all of the sequences following true transition conditions are selected. Therefore, it is considered poor programming practice to have the proper operation of a simultaneous sequence depend upon the order of processing of active steps in these sequences within a single scan. The PLC program auditor should check for this.

- *Use simultaneous sequences only where synchronization is required.* Simultaneous sequences are used when parallel processes need to be synchronized at their beginning and their ending. Where asynchronous sequences that do not require this kind of synchronization are desired, they should be coded as independent SFC Charts.

6.1.3.5 Accounting for Post-Scan Timing

Post-scan timing is unique to the SFC language. After a true transition, the processor scans a step once more to reset all timer instructions and other variables and controls (Hughes, 1989; p. 178). This extra step is called the post-scan. The new active step is scanned for the first time only during the next scan. The following are specific guidelines related to this characteristic of SFCs:

- *Post-scan timing requirements.* The time required for the post-scan should be characterized and shown to be in accordance with the safety requirements of the PLC and overall safety system.

- *No timers in transitions.* The processor never postscans a transition program file. Therefore, timers should not be set in a transition because they will not be reset.

6.2 Robustness

Robustness refers to the capability of the program to survive off-normal or other unanticipated conditions. This section discusses guidelines on functional diversity and exception handling.

6.2.1 Transparency of Functional Diversity

SFCs are well suited to implementing diverse algorithms or implementations given that the need for such diversity has been established. An AND path can force several different process steps to evaluate the same condition. An additional step can vote. An OR path can be used to cause a transition if it is desired to program a system such that any number of diverse parallel algorithms cause the transition. The following are specific guidelines:

- *Order of execution.* The design should account for the safety impact of the order of execution of diverse process steps. The ordering on the SFC should reflect the intention of the design.

- *Interfaces.* The safety system design should account for all local and global variables necessary to support replicated processing in transition files. As part of the implementation, it should be verified that no variables in transition files will be initialized or overwritten.

6.2.2 Exception Handling

The level, nature, and functionality of SFC exception handling varies significantly among SFC implementations. Exception handling functionality in SFC ranges from none at all, through activation of a designated fault sequence under certain conditions, to the ability to completely override the activation status of an SFC chart under control of portions of the PLC program not in the SFC (Allen-Bradley, 1989; PLC Direct, 1994; Telemecanique, 1994). It is necessary for project and PLC SFC-specific guidelines to be created for exception handling to account for these specific characteristics.

Although there are significant variations, the following guidelines apply to most SFC implementations:

- *Use of GOTO or JMP statements to handle the interruption of control flow.* Sequential function charts do not support interrupt processing due to the sequential nature of execution. Thus, should an abnormal condition or exception occur which requires immediate attention, SFCs do not allow servicing of the request. GOTO or JMP statements can provide a method of handling this abnormal asynchronous condition. For example, should a mixing sequence not be completed because a valve failed to open, the mixer contents would have to be dumped. Due to the sequential nature of SFC, it is not possible to exit the current transition and start executing the dumping step without using JMPs or GOTOs. Although JMP or GOTO statements can be used for this purpose, their use for normal control flow should be minimized.

- *Avoiding conflicts.* It must be determined that the two events, transition and exception-handling, do not conflict with each other.

- *Behavior of the exception-handling mechanism during a process step.* The exact behavior of process steps interrupted by fault routines should be characterized and shown to be in accordance with the safety requirements of the PLC and overall safety system. For example, a fault routine may not interrupt a process step unless initiated by the PLC. This behavior must be understood explicitly.

- *Behavior of the exception handling mechanism during a transition.* The exact behavior of transitions interrupted by PLC fault routines should be characterized and shown to be in accordance with the safety requirements of the PLC and overall safety system. The transition and exception handling mechanism must be evaluated as to whether they conflict with each other.

- *Restart behavior.* Care must be taken in design for power up and fault recovery conditions. The exact behavior of SFC restart after an exception should be characterized and shown to be in accordance with the safety requirements of the PLC and overall safety system. For example, pre-scan and post-scan firmware logic employed when using SFCs only operates when the step is entered and exited. The SFC reset instruction can be used to shut a system down, however, there is no control for orderly shutdown should a fault occur. This behavior may not be acceptable in a safety application.

6.2.3 Input and Output Checking

Data corruption in a process step or transition can have serious consequences if allowed to propagate to other process steps. SFCs do not have explicit input and output checking mechanisms. However, the generic guidelines apply to the underlying program steps and transitions.

The specific guideline is that input and output checking (error containment) should be handled at the language level and not at the SFC level. The likelihood of error propagation can be reduced if a process step uses reasonableness checks prior to setting variables used by other steps. Similarly, the possibility of error propagation is reduced and safety is enhanced if a module using values set by another module performs checks on acceptability before operating on these variables. When the checks indicate that some assertions have been violated, exception handling can be used to bring the system to a state defined in the higher level design. Specific guidelines for PLC ladder logic were described in the previous Chapter.

6.3 Traceability

Traceability refers to attributes of safety software which support verification of correctness and completeness compared with the software design. The intermediate attributes for traceability are:

- Readability
- Minimizing use of built-in functions,
- Minimizing use of compiled libraries.

Because readability is also an intermediate attribute of maintainability, it is discussed in the next section. The following paragraphs discuss the latter two attributes.

6.3.1 Use of Built-In Functions

The SFC language does not explicitly support built-in functions. However, the underlying IEC 1131 languages used in process steps and transitions do support such functions. The use of built-in functions raises safety concerns for the following reasons:

- The requirements for built-in functions may not be the same as those for developing safety systems.

- The exception handling of the built-in function may not be as well characterized as portions explicitly developed for the safety system.

- The specific built-in functions may vary from one PLC platform to another thereby raising portability and maintainability concerns.

Because of these concerns, the use of built-in functions should be minimized. When built-in functions are used, the developers should conduct thorough testing and develop a means for tracking errors. The details and acceptance criteria of such a testing and verification program are beyond the scope of this document.

6.3.2 Use of Compiled Libraries

SFC does not support the use of external libraries. However, its runtime environment does consist of libraries of compiled components the underlying languages may also support compiled libraries. The concerns in the previous section also apply to compiled libraries. When compiled libraries are used, the developers should conduct thorough testing and develop a means for tracking errors. The details and acceptance criteria of such a testing and verification program are beyond the scope of this document.

6.4 Maintainability

This section discusses safety-related maintainability attributes for SFCs. These include:

•Readability
•Data abstraction
•Functional cohesiveness
•Malleability
•Portability.

6.4.1 Readability

Readability allows software to be understood by qualified development personnel other than the original developer. Readability is essential for safety because it facilitates reviews and reduces the likelihood of errors during maintenance.

SFC was specifically designed as a notation for representing a sequence of operations. As such, it fits a developer's cognitive model of machine sequencing. Thus, SFC programs *for sequencing operations* are readable. In general, the SFC construct adds an additional level of abstraction to the programming language.

The following specific guidelines are related to readability:

•Conformance to indentation guidelines
•Descriptive identifier names
•Comments and internal documentation
•Limitations on subprogram size
•Minimizing mixed language programming
•Minimizing obscure or subtle programming constructs
•Minimizing dispersion of related elements
•Minimizing use of literals
•Controlled use of macro-steps.

6.4.1.1 Conformance to Indentation Guidelines

Because of the structure and notation of SFC, indentation guidelines are not applicable.

6.4.1.2 Descriptive Identifier Names

The generic guidelines are applicable. Many SFC systems allow the naming of steps and transitions. Identifiers are used to label the steps and transitions of the SFC. Each identifier refers to a program file containing a process step or transition. The identifiers should be defined so that they provide adequate information on the nature and content of each file. Specific guidelines should be developed for each system and project, and the project-specific guidelines should be followed in the actual SFC programs.

6.4.1.3 Comments and Internal Documentation

The generic guidelines apply. The following are specific guidelines:

- *Descriptions of steps.* Clear and unambiguous descriptions of process steps need to be provided. These descriptions should include the processing performed by the step, timers set and reset, and other operations. The description should be, in accordance with the design, traceable to higher-level requirements and design documents.

- *Description of interfaces.* The interfaces for each step and transition should be described in the preamble. This description should include a complete identification of the input parameters, global variables (and any side effects), and output parameters. These descriptions should be traceable to higher level design documents.

- *Description of transition conditions.* The transition conditions should be clearly stated. All input variables and global variables should be identified. These descriptions should be traceable to higher-level design documents.

6.4.1.4 Limitations on Subprogram Size

The generic guidelines are applicable. SFC implementations vary in the limitations on the amount of code that can be in a single step. These limitations range from a single network of Ladder Logic to no limit whatsoever (other than memory capacity of the PLC). The following are specific guidelines:

- *Limitation of a single step to a single function.* The code in a single step should be limited to performing a single action. Since each SFC step is typically a subroutine using a PLC supported language, the rule for subroutines should apply to steps - one function which is clearly definable. Multiple actions in a step are to be discouraged.

- *Limitation on transitions to a single expression.* SFC transition conditions are limited to a single expression.

6.4.1.5 Minimization of Mixed Language Programming

The generic guidelines are *not* applicable. Each of the IEC1131-3 programming languages for PLCs is specific to a particular aspect of the control problem domain. PLC programs that are simple have lower incidence of programming errors, and are more maintainable than those that use the IEC 1131-3 languages for their intended purposes.

The following are specific guidelines on the use of SFCs in a mixed IEC 1131-3 language application:

- *Use SFC for sequencing.* SFC is specifically intended for the programming of machine sequences. The SFC notation for this purpose is clearer than Ladder Logic or Structured Text.

- *Do not use SFC for interlocking or evaluation of logical relationships.* Ladder Logic is well suited for interlocking and other applications requiring evaluation of Boolean relationships. SFC is not suited for this purpose.

- *Do not use SFC for mathematical operations or evaluation of mathematical relationships.* Structured text excels over SFC, Ladder Logic, or function blocks for mathematical relationships.

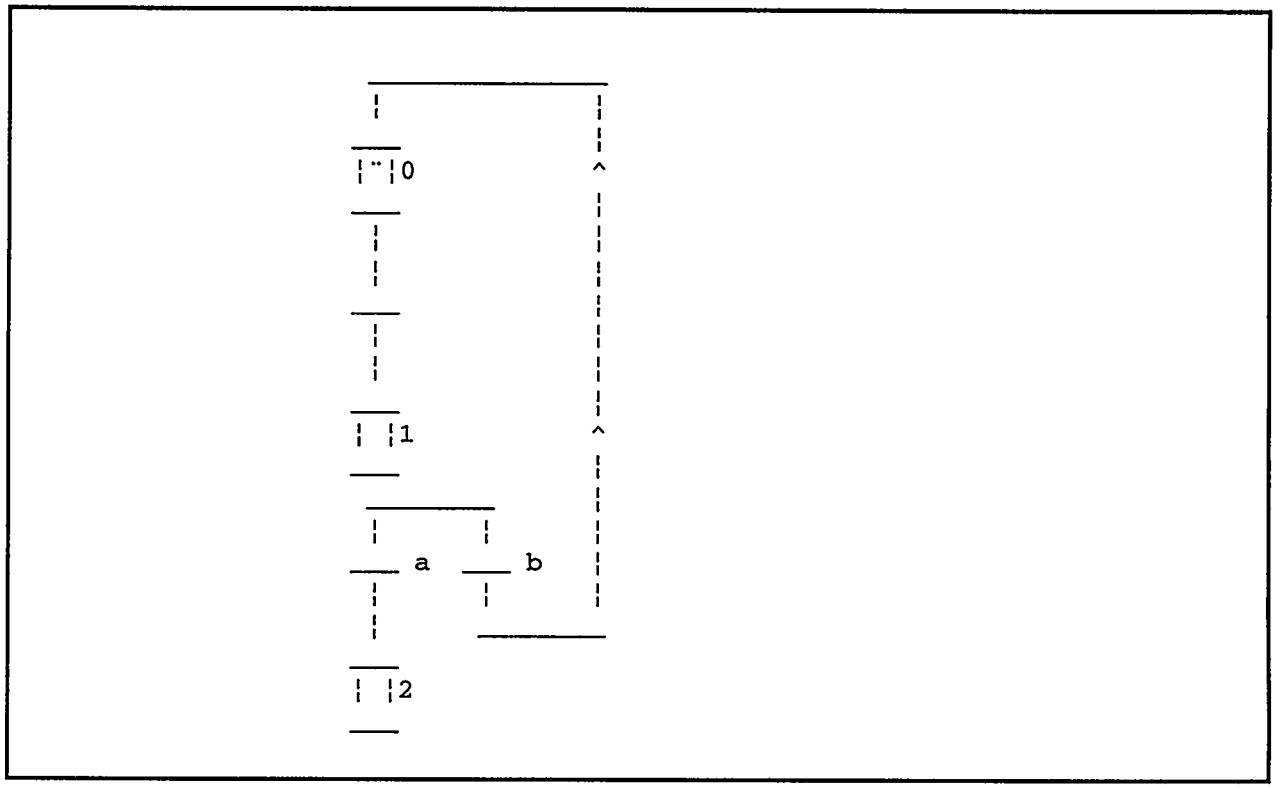
6.4.1.6 Minimize Obscure or Subtle Programming Constructs

The guidelines associated with this generic attribute have limited applicability. The following are specific guidelines:

- *Avoid nesting of subroutines within an SFC step.* An SFC step suggests that a certain PLC subroutine will be executed at that step. When the end of program statement or RET statement is executed for that subroutine, the transition file is then checked, and the flow continues from there. Calling nested subroutines of any language from within the called SFC step is obscure because of the assumption that an SFC step is one subroutine.

- *Do not use SFC constructs that are not related to sequencing.* SFC as a language is intended for the programming of control sequences. There are some SFC constructs that allow other uses for SFC. These constructs should be avoided.

- *Avoid backward directed links in parallel paths.* This is demonstrated in the following SFC construct (which should be avoided). The transition condition labeled 'b', when active, allows the re-activation of step 0. This can lead to multiple steps in the same sequence being active simultaneously. SFC programs that allow this can be difficult to program and maintain.



6.4.1.7 *Minimize Dispersion of Related Elements*

The guidelines associated with this generic attribute are applicable. Dispersion can be an issue with SFC because of its graphical organization. Few details are presented at the SFC level, and specific variables associated actions are contained within many step and transition files. A further degree of dispersion can occur because a step can be organized as several subroutines, each of which could be in a separate file. Project-specific guidelines on how to structure SFC programs to minimize the dispersion of safety-critical components should be developed and adhered to during development.

6.4.1.8 *Minimize Use of Literals*

The generic guidelines are not applicable because the SFC language does not include literals. Use of literals can occur in the underlying IEC 1131 languages. Specific guidance for PLC ladder logic is contained in the previous chapter.

6.4.1.9 *Controlled Use of Macro-Steps*

Macro-steps (nested SFCs), when available in the SFC implementation, can enhance readability by combining several smaller steps and transitions into a single larger step. However, the misuse of

macro-steps can make SFC programs difficult to understand. Macro-step use should be controlled by project guidelines to ensure that undue complexity is not hidden through excessive use of such nesting.

6.4.2 Data Abstraction

As described in Chapter 1, data abstraction is the combination of data and allowable operations on that data into a single entity, and the establishment of an interface which allows access, manipulation, and storage of the data only through the allowable operations. It reduces or eliminates the potential side effects of changing variables either during runtime or in software maintenance activities (Parnas, 1972). SFC programs provide an abstraction of the control sequence to be executed by the PLC. This section includes guidelines on:

•Minimizing the use of global variables
•Minimizing the complexity of the interface defining allowable operations.

These attributes are discussed further in the following subsections.

6.4.2.1 Minimizing the Use of Global Variables

The generic guidelines have limited applicability because many PLCs allow only global variables. Nevertheless, as noted previously, there are some implementations which do support a distinction between local and global variables. If local variables are supported by the underlying language of the process step or transition, they should be used for all internal operations.

6.4.2.2 Minimization of the Complexity of Interfaces

The generic guidelines are applicable. The primary interface issues are in the interaction between process steps and transition files. These must be addressed through the underlying IEC 1131 languages.

6.4.3 Functional Cohesiveness

The generic guidelines are applicable. The following are specific guidelines.

- *A single function for each step.* Every step should have one clearly discernable purpose related to the time in which it should be executed. Two or more different steps should not be combined in a single step if they handle different functions or processes.

- *Use macro-steps for related functions.* When there are several related functions that are to be performed in series, macro-steps can be used.

6.4.4 Malleability

Malleability is the ability of a software system to accommodate changes in functional requirements. To implement a malleable software system, it is necessary first to identify what is expected to be constant and what is expected to be changed, and then to segregate what is expected to be changed into easily identifiable areas where alterations can be made with a minimum of collateral changes. The segregation into steps provides some malleability.

6.4.5 Portability

The advent of IEC-1131 software standards will create a common platform and a standardized approach. However, this will be breached by hardware vendors trying to add extensions which only they can interpret. This extensibility can be useful for an application, but useless in the desire for standardization.

Only IEC 1131-3 compliant SFC systems should be used. Without the use of IEC 1131-3 constraints, an SFC will NOT be portable between platforms. The implementations of SFC are varied. For example, European implementations, or GRAFCET, (Blanchard, 1985) differ from domestic implementations. Allen-Bradley's SFC is not a complete implementation of the IEC 1131 standard; it also has unique features (Allen-Bradley, 1989).

References

Allen Bradley, *PLC-5 Programming Software - Programming*, Publication 6200-6.4.7 November, 1991.

Blanchard, M. *Le GRAFCET de noveaux concepts*, CEPAD (France), 1985.

Bossy, J.C., P. Brard, P. Fagere, and C Mwerlaud, *Le GRAFCET sa pratique et ses applications,* Educalivre, France, 1979.

Hughes, T.A., *Programmable Controllers,* Instrument Society of America, Research Triangle Park, NC, 1989.

Institute of Electrical and Electronic Engineering, *IEEE Standard Dictionary of Electrical and Electronic Terms*, IEEE Std 100-1977.

Institute of Electrical and Electronic Engineers, *Glossary of Software Engineering Terminology*, ANSI/IEEE Std 729-1983.

Institute of Electrical and Electronic Engineers, *IEEE Standard Glossary of Software Engineering Terminology,* IEEE Std 610.12-1990.

International Electrotechnical Commission (IEC), *Preparation of Function Charts for Control Systems,* IEC Standard 848, 1986. (Available in the U.S. from the American National Standards Institute, New York.)

International Electrotechnical Commission (IEC), *Programmable Controllers General Information,* IEC Standard 1131, Part 1, 1992. (Available in the U.S. from the American National Standards Institute, New York.)

International Electrotechnical Commission (IEC), *Programmable Controllers Programming Languages*, IEC Standard 1131, Part 3, 1993. (Available in the U.S. from the American National Standards Institute, New York.)

PLC Direct Corp., *PLC Direct Technical Overview*, 1994

Telemecanique, *XTEL PLC Programming Software Version 5.5*, 1994

7 Pascal

This chapter describes guidelines for the application of Pascal in safety systems and is organized in accordance with the framework of Chapter 2. Section 7.1 discusses reliability-related attributes; Section 7.2 discusses robustness-related attributes; Section 7.3 discusses traceability-related attributes; and Section 7.4 describes maintainability-related attributes. A summary matrix showing the relationship between generic and language-specific guidelines, together with weighting factors, is included in Appendix B.

Although Pascal was standardized by the IEEE 770 and ANSI X3J9 committees and is documented by several standards (NIST, 1985), the language has several major variants. The most significant of these is Pascal developed by Borland International Corp. running under versions of the Microsoft MS-DOS and Windows operating systems (Microsoft, 1992; Borland, 1991). These are of significance for this report because of their current and potential continued use as platforms for testing Class 1E equipment. Guidelines that are specific to these latter variants are indicated as such in this chapter.

Language-specific weighting factors were based on the key characteristic of Pascal designed for safety, that is, strong data typing. Other factors were determined to be neutral from this perspective. Recursion and interrupt handling through the run-time environment are felt to be important in the negative sense; their use should be constrained and limited.

7.1 Reliability

This section discusses specific guidelines associated with intermediate attributes related to reliability. The intermediate attributes are as follows:

- Predictability of memory utilization
- Predictability of control flow
- Predictability of timing.

These attributes are discussed in the following sections.

7.1.1 Predictability of Memory Utilization

Base-level attributes related to the predictability of memory utilization in Pascal are as follows:

- Avoiding dynamic memory allocation
- Minimizing memory paging and swapping
- Avoiding recursion

- Use of handles with pointers
- Avoiding the use of direct memory access.

Specific guidelines for these base attributes are discussed in the following subsections. It should be noted that the final three guidelines are applicable to Pascal but are not included in the generic guidelines. The final two guidelines are specific to Borland Pascal.

7.1.1.1 Avoiding Dynamic Memory Allocation

The generic guideline on avoiding dynamic memory allocation is applicable to Pascal. Dynamic memory allocation should be avoided in safety systems written in Pascal.

The strong typing of ANSI Standard Pascal makes each array type with different bound a distinct type. This can make handling variable-length data items, such as strings, a problem. Kernighan (Kernighan, 1981) has pointed out that the way around this problem is to ensure that all strings of a program are set to strings of predetermined lengths, with an associated string type for each length. In a safety system, this approach is preferable to an alternative approach using dynamic memory allocation. This issue is discussed further in the section on data typing.

The use of dynamic memory can be detected through the Pascal statements containing **`new`** (to allocate), **`dispose`** (to free memory), and the Pascal pointer (^). An alternative form is **`GetMem/FreeMem`**. It should be noted that these two methods do not allocate memory on the heap in the same way. The use of these functions interchangeably could conceivably destroy the heap thereby losing all the data and crashing the computer (Borland, 1991). Care must be taken to avoid "dangling pointers," i.e., pointers to space which has been freed or deallocated.

If dynamic memory allocation is necessary in a safety application, the application program should not use multiple variables pointing to the same memory location. The danger is that when the shared memory space is deallocated, another variable may still point to the released memory space unless each one is explicitly set to null by the application program. If an application (e.g. a linked list) necessitates such multiple accesses, it must be justified and documented.

The following is an example of dynamic memory allocation using Borland Pascal 7.0:

```
{------------------------------------------------------------ Example 1 }
{ declaration }
      VAR StrPtr : ^STRING;
          GenPtr : POINTER;

{ Then, that string pointer is allocated space within the program}

      New(StrPtr);

{ The string pointer is copied to the general one }

      GenPtr := StrPtr;

{------------------------------------------------------------ Example 2 }
{ The program assigns this value to an ARRAY of variant records.
  One of the elements of the record is of type POINTER: }

      TYPE YYSType = record case Integer of
              1: ( yyInteger : Integer);
              2: ( yyPointer : Pointer);
          end;
```

If dynamic memory use is essential, the software should always release dynamic memory as soon as possible.

7.1.1.2 Minimizing Memory Paging and Swapping

The generic guideline on minimizing paging and swapping is applicable to Pascal programs. There are no Pascal-specific guidelines.

7.1.1.3 Avoiding Recursion

This guideline is not generic; however, it is applicable to Pascal. Recursive programs should not be used in safety systems *unless it can be definitively shown that there is always a terminating condition within a deterministic time and number of iterations, and that the memory will not be exceeded at the maximum level of recursion.* The number of recursions can be large, even infinite, because the terminating condition may not occur.

There are two types of recursion in Pascal: self-recursion and mutual recursion. Self-recursion can be recognized by having a procedure call within a procedure of the same name. In mutual recursion, two routines call each other. In the following example, functions A and B will call each other until some termination criterion is met (unspecified in this example). Mutual recursion is rarely detected by compilers.

```
function B(x : integer) : char ; forward ;
function A(y : integer) : char ;
begin
  ... B(I) ...
end ;
function B(x : integer) : char ;
begin
  ... A(j) ...
end ;
```

7.1.1.4 Use of Handles with Pointers

The following guideline is applicable to Borland Pascal.

If pointers must be used, handles should be used whenever possible. Handles allow memory management to recapture and compact free memory[25]. The memory block should be locked to protect moveable blocks and should be unlocked as soon as possible thereafter. When data in a moveable block needs to be changed, locking the block while the change is being made and then unlocking the block protects the data. When a block is locked the block cannot be moved. Once the block has been unlocked, memory management can then move the blocks for compaction. If the handle is not unlocked in a timely manner, memory management is unnecessarily hampered.

This guideline is illustrated in the following example (Borland, 1991).

```
ItemGlobalHandle := GlobalLock(GlobalHandle) ;
ItemGlobalHandle^[0] := 255 ;   {Process data
                                 using ItemGlobalHandle}
GlobalUnlock(ItemGlobalHandle) ;
if ((DataRecord.bitOptions or DDE_Release) <> 0 ) then
    GlobalFree(ItemGlobalHandle);
end ;
```

Improper locking and unlocking of handles or failure to lock handles is a frequent source of errors in Macintosh programming, which uses dynamic relocation and compaction of memory.

7.1.1.5 Avoid Use of Direct Memory Access

The following guideline is applicable to Borland Pascal under Windows and in Protected Mode under DOS.

---

[25] Compacting memory is a design issue that must be handled with care.

Direct memory access should not be used except in situations where hardware devices have memory-mapped control registers that must be read or written. Although Borland Pascal permits access to memory directly, this is not a safe practice under Windows at any time. Windows should manage memory issues or the programs may crash (Borland, 1991). Protected mode does not allow direct addressing. Instead, memory selectors should be used.

If direct memory access has to be used, it should be encapsulated, where possible, to avoid errors.

7.1.2 Predictability of Control Flow

This section discusses base-level attributes related to the predictability of memory utilization in Pascal. These guidelines are

- Maximizing structure
- Minimizing control flow complexity
- Initializing variables before use
- Single entry and exit points for subprograms
- Minimizing interface ambiguities
- Use of data typing
- Accounting for precision and accuracy
- Order of precedence of arithmetic, logical, and functional operators
- Avoiding functions or procedures with side effects
- Separating assignment from evaluation
- Proper handling of program instrumentation
- Controlling class library size
- Minimizing use of dynamic binding
- Controlling operator overloading.

These attributes and their relevance to safety are discussed in the following sections. It should be noted that the avoiding-side-effects guideline is applicable to Pascal but not included in the generic guidelines.

7.1.2.1 Maximizing Structure

The generic guideline on maximizing structure applies to Pascal. Maximizing structure means not using `gotos` (jumps in program control). Three language-specific guidelines are related to `goto` statements, `if ... else if` statements, and case statements.

- *Avoid **goto** statements except as early exits from loops.* The use of **goto** clouds the structure of the code in that it can obscure program flow logic and result in unreachable code. The following is an example[26] of a fragment of a Pascal program containing **goto** statements resulting in unreachable code.

```
B_Label:    statement_1;
            goto A_Label;
            statement_2;            {unreachable code}
            statement_3;            {unreachable code}
            statement_4;            {unreachable code}
A_Label:    statement_5;
            statement_6;
            statement_7;
            goto B_Label;
            statement_8;            {unreachable code}
```

The rationale for the early loop exit exception to this guideline can be seen in the following example. In Pascal the loops can be labeled in order to clarify the meaning of multiple loops and the code structure. In the following example **the_first_loop** and **inner_most_loop** are loop names.

[26] This rather trivial example is only included for the purpose of illustration.

```
label : the_first_loop, after_the_first_loop, the_inner_most_loop,
      after_inner_most_loop ;
...
the_first_loop :
for i := 100 downto 1 do
  begin
    for alpha  := 1 to 26 do
       begin
        for numbers := 5 to 11 do
            begin
             the_inner_most_loop:
             for steps  := 1 to 10  do
                begin
                ...
                if sample <= 10e-6 and bc_flag
                    then goto after_the_first_loop ;
                ...
                if bc_flag or not op_flag
                then goto after_inner_most_loop ;
                ...
                     end ; {the_inner_most_loop}
                   {loop name for readability}
          after_inner_most_loop : j := 5 ;
          end ;
      end ;
  end ; {the_first_loop}                {loop name for readability}
after_the_first_loop : i := 1 ;
```

It should be noted that standard Pascal allows only integers as labels, while Borland Pascal has an extension to the language that also allows character strings as labels (Jensen, 1974; Borland, 1991). It should also be noted that Borland Pascal 7.0 uses the keywords **`break`** and **`continue`**, so that **`goto`**s with these constructs are not necessary.

- *Use of if ... else if and case statements.* The use of **`if ... else if`** is shown in the following example:

```
if  condition_1 then
      statement_1 ;
else if  condition_2 then
      statement_2 ;
else if  condition_3 then
      statement_3 ;
else
      statement_4 ;
```

The final **else** statement allows the handling of conditions not anticipated in the first three conditions; it also serves as a default. This construct should be used in all situations even if it can be guaranteed that the conditions specified by the other **else if** statements are exhaustive.

The **case** statement serves as a switch for multiple branches and allows one evaluation for the multiple branches. It is an alternative to the **if** statement *under the circumstances that all conditions within the case statement are exhaustive* (Jensen, 1974, p 31; Grogono 1983, p 161). It is a run-time error (of unspecified behavior) if the case selector does not equal one of the case conditions. Some implementations of Pascal allow for a default selector, e.g., *otherwise*. However, if a default selector is used, the program is non-portable.

```
case thermal_alarm of
       core        : core_thermal_alarm(sensor_value) ;
       inlet       : inlet_thermal_alarm(sensor_value) ;
       outlet      : outlet_thermal_alarm(sensor_value) ;
end  ;
```

7.1.2.2 Minimizing Control Flow Complexity

The generic guideline with respect to nesting levels applies to Pascal. Specifically, control flow complexity results from the use of too many nested levels of branching or looping. As noted in the generic report, there should be explicit organizational or project-specific limits on nesting. There are no specific guidelines with respect to Pascal.

7.1.2.3 Initializing Variables before Use

The generic guideline with respect to initialization of all variables applies to Pascal. Run-time predictability requires that memory storage areas set aside for process data be set to known values prior to being accessed (i.e., set and used). Variables should be initialized to some known value at

the beginning of an execution cycle before they are used. In Pascal all pointers must be initialized to **NIL**.

The key characteristic of Pascal associated with this guideline is the lack of compile time initialization. The lack of compile time initialization means that variables must be initialized explicitly by assignment statements. Because initialization occurs at the beginning of the program, initialized variables must be visible at the highest level of the calling hierarchy. The result is that most variables to be initialized will have global scope (Kernighan, 1981). This is problematic because excessive use of global variables conflicts with the data abstraction and visibility guidelines described below.

The following guideline is applicable to Borland Pascal

When using separately compiled units with shared variables, initialization should occur in one and only one place.

7.1.2.4 Single Entry and Exit Points for Subprograms

The generic guidelines apply to Pascal. Standard Pascal is a block-structured language in which procedures and functions are defined by begin and end statements. This guideline is enforced by the language (ANSI, 1983; p 66).

The following guideline is applicable to Borland Pascal

Borland Pascal provides the capability for multiple exit points. This capability *should generally not be used* in safety-critical systems. When multiple exit points are unavoidable, the rationale should be documented; and return value assignments must precede every exit point.

```
{ In standard Pascal, acceptable }
        function F: Boolean;
          begin
            if condition
              then F:=true;
              else
                begin
                  ...
                  F:= ...
                end
          end;

{ Borland Pascal (and some others), alternative form, not acceptable in
   safety system }

        function F: Boolean;
          begin
            if condition then
              begin
                F:=true; exit;  { first exit }
              end;
            ...
            F:= ...
          end;  { second exit }
```

7.1.2.5 Minimizing Interface Ambiguities

The generic guideline with respect to interface ambiguity minimization applies to Pascal. Interface ambiguities minimization can occur in both functions and procedures. The following additional guideline applies:

- *Alternate data types in subroutine formal argument lists.* Inadvertent switching of parameters of the same type can be avoided by not listing the same types in consecutive order when possible, as shown in the following example.

```
      process_sensor_data(sensor_id : integer,
            value : string[255],
            calib_date : integer,
            calib_tech : string)
```

7.1.2.6 Data Typing

The generic guidelines for data typing apply to Pascal. Pascal is a strongly typed language, and the code should take advantage of this feature to the maximum extent possible. The following are specific guidelines.

- *Use subtypes.* When defining data types, it is generally good practice to use subtypes of the predefined types to define the range explicitly, thus bounding the errors. When an object is assigned a number outside its range, a run-time error is raised (Jensen, 1974; Grogono 1983). The limits on data types should not be excessively constrained, forcing an unnecessary error to be generated.

- *Minimize the use of implicit type conversions.* All type conversions in Pascal are implicit. Therefore, the programmer and the reviewer must be vigilant for these unannounced conversions. An example with string assignments where the receiving string (right hand side of an assignment statement) is a different size than the assigned string (left hand side).

The following is an example showing implicit type conversions in equations:

```
i : integer ;
r : real ;

r := i + r ;            {implicit conversion from integer to
                        real -- allowed}
i := i + r ;            {illegal}
```

Pascal ensures that expressions involving arithmetic evaluations or relational operations have a single data type or the proper set of data types for which conversion difficulties are minimized. It is not possible to assign the result of a **`real`** expression to an **`integer`** variable (Grogono, 1983, p. 37).

- *Limit the use of indirection (pointers).* Limiting the use of indirection, such as array indices and access types, in Pascal to situations where there are no other reasonable implementation alternatives and performing validation on indirectly addressed data prior to setting or use, ensure the correctness of the accessed locations.

7.1.2.7 Accounting for Precision and Accuracy

Precision and accuracy generic guidelines apply to Pascal. Precision and accuracy issues include the meaning and use of fixed point and floating point numbers, round off-errors, type declarations and digital accuracy, and portability. The accuracy and precision necessary are a function of the project

requirements in concert with the computer, the compiler, the hardware, the sensors, the observability and the control requirements. The issues raised must be factored into the design of the software. These are discussed in the generic guideline chapter of this report.

Within the rules of precedence, order of evaluation of expressions in Pascal is implementation-defined. This may lead to unexpected results in the presence of optimized code being generated by the compiler. This is especially an issue with floating point computations. A compiler might replace ((1.0+x)-x) with 1.0 at compile time, when the floating point rounding error is what the program is trying to compute (note that the above optimization is always guaranteed to be correct for integer types).

7.1.2.8 Order of Precedence of Arithmetic, Logical, and Functional Operators

The generic guidelines for order of precedence apply to Pascal. The default order of precedence of such operations as left to right with exponentiation, multiplication, and addition should not be depended on. Hence, the following specific guidelines:

- *Use parentheses.* Arithmetic, logical, and other operations should use parentheses or other mechanisms for ensuring that the order of evaluation of operations is explicitly stated.

- *An expression should not depend on the order of evaluation.* The Pascal standard permits operands of an expression to be evaluated differently from the left to right order in which they are written. For example, in the statement:

  ```
  i := F(J) div G(J) ;
  ```

  where **F** and **G** are functions of type Integer, **G** may be evaluated before **F**, since this enables the compiler to produce better code. If **F** and **G** have side effects, in particular, changing the value of **J**, (perhaps inadvertent — as described in the next section), the order of execution may have an effect that the programmer had not intended and that may lead to a subtle and difficult to find the bug (Borland, 1991; p 241).

7.1.2.9 Avoiding Functions or Procedures with Side Effects

Generic guidelines are applicable. The following specific guideline applies to Pascal:

Global variables should not be set or changed by procedures and functions for which that variable is global in scope. This means using local variables within functions and subroutines for variables that should not be visible outside the function or procedure, and using the **var** only for those variables that the procedure should be changing.

7.1.2.10 Separating Assignment from Evaluation

The generic attributes apply to Pascal programs. Since there is no embedded assignment operator for expressions in base Pascal, embedded assignment can only occur via side-effect producing functions, which were discussed in Section 2.1.2.9.

7.1.2.11 Proper Handling of Program Instrumentation

The generic guidelines are applicable to standard Pascal. Borland Pascal and Turbo Pascal have extensive instrumentation capabilities that can be implemented transparently in the source code using the debugger supplied by the company. The additional guideline is to ensure that compiler switches are set in a manner that does not disable debugging, such as $D-.

7.1.2.12 Controlling Class Library Size

The generic guidelines for this attribute are applicable to Borland Pascal but not to ANSI standard Pascal, which is not object oriented.

7.1.2.13 Minimizing Use of Dynamic Binding

The generic guidelines for this attribute are applicable to Borland Pascal but not to ANSI standard Pascal, which is not object oriented. The following specific guideline applies.

Dynamic binding and methods should be avoided if possible.[27] The rationale for this guideline is that dynamic binding forms unpredictable relationships which are hard to debug and difficult to test for all possible configurations. If a class declares or inherits any virtual methods, then variables of that type must be *initialized* through a constructor call before any call to a virtual method. Thus, any object type that declares or inherits any virtual methods must also declare or inherit at least one constructor method.

Dynamic method calls are dispatched at run time, as opposed to virtual methods whose invocation is known at compile time. For all other purposes, a dynamic method can be considered equivalent to a virtual method. An object is *instantiated*, or created through the declaration of a variable or typed constant, or by applying the standard procedure **new** to a pointer variable of an object type.

---

[27]Methods are functions and procedures that are used to manipulate and retrieve data from the data objects in the methods' class. Methods are by default *static*, but can, with the exception of constructor methods, be made *virtual* through the inclusion of a **virtual** directive in the method declaration. The compiler resolves the calls to static methods at compile time, whereas calls to virtual methods are resolved at run time. The latter is sometimes referred to as *late binding* or *dynamic binding*.

It is important to note that assignment to an instance of an object type does not entail initialization of the instance.

The following are examples of constructors:

```
constructor Field.Copy(var F : Field) ;
begin
  Self := F ;
end ;
constructor Field.Init(FX,FY, FLen : Integer ; FName : String) ;
begin
  X := FX ;
  Y := FY ;
  Len := FLen ;
  GetMem(Name, Length(FName) + 1 ) ;
  Name^ := FName ;
end ;
constructor StrField.Init(FX,FY,FLen: Integer; FName : String) ;
begin
  Field.Init(FX, FY, FLen, FName) ;
  GetMem(Value, Len) ;
  Value^ := '' ;
end ;
```

The following are examples of destructors:

```
destructor Field.Done :
begin
  FreeMem(Name, Length(Name^) + 1 ) ;
end ;

destructor StrField.Done ;
begin
  FreeMem(Value, Len) ;
  Field.Done ;
end ;
```

Dynamic binding uses the heap and is therefore susceptible to the same types of memory problems described in Section 7.1.1.1, Avoiding Dynamic Memory Allocation. Therefore, as with pointers, dynamic memory should be avoided if possible. All cases requiring dynamic binding should be documented and justified.

7.1.2.14 Controlling Operator Overloading

Pascal does not have operator overloading features; therefore, the guideline is not applicable.

7.1.3 Predictability of Timing

Predictability of timing is crucial in a safety system used in real-time control. Concerns over object-oriented base attributes discussed in the previous sections (e.g., package library size, dynamic binding, and operator overloading) also apply to timing. In addition, specific concerns related to interrupts are discussed in Section 7.1.3.2.

7.1.3.1 Minimizing the Use of Tasking

Pascal does not have tasking features; therefore, the generic guidelines are not applicable.

7.1.3.2 Minimizing the Use of Interrupt Driven Processing

The generic guidelines for interrupt-driven processing apply to Pascal. It is not generally desirable in safety-critical systems because it can lead to nondeterministic maximum response times and can lead to unanticipated system states. Use of a deterministic approach to the monitoring and control of multiple input sources is normally preferred. However, there may be some situations where interrupt-driven processing has a significant design advantage over alternatives, for example, to handle the acceptance and processing of plant input. When interrupt service routines are needed, only the minimum processing needed to buffer the input should be performed by the interrupt driver. All non-time-critical processing (e.g. units conversions) should occur in the main line code.

The following is the form of an interrupt handler in Borland Pascal under MS-DOS on Intel processors:

```
procedure IntHandler(Flags, CA, IP, AX, BX, CX, DX, SI, DI,
      ES, BP : Word);
interrupt ;
begin
      ...
end ;
```

Interrupt routines must be designed with care. Masking of interrupts, nested interrupts, and interrupt processing in general can all cause non-deterministic behavior. Also, some form of locking or mutual exclusion may be required when using interrupts.

In case of code that directly accesses hardware, it must be noted that Pascal lacks the **volatile** attribute, so it is not possible to guarantee that memory accesses are not deleted and that they occur in the specified order.

7.2 Robustness

Robustness refers to the capability of the software to survive off-normal or other unanticipated conditions. The intermediate attributes for robustness are as follows:

- Controlled use of diversity
- Controlled use of exception handling
- Input and output checking.

This section describes Pascal-specific guidelines for the base-level attributes of software diversity and exception handling.

7.2.1 Transparency of Functional Diversity

There are no Pascal-specific guidelines for functional diversity. The generic guidelines apply.

7.2.2 Exception Handling

Standard Pascal does not have exception handling. Therefore, this guideline is not applicable. Borland Pascal has specific types of error handling, which are not as general as full exception handling. The following guidelines apply to Borland Pascal:

- *Exit handling.* Exit handling can be used to recognize run-time errors explicitly and plan for their resolution, and for post-mortem analysis. Borland Pascal provides a method of declaring run-time errors and of building the appropriate exit handling code. This is *exit handling*, not exception handling. It is considered good practice to recognize these conditions explicitly and plan for their resolution.

```
procedure TestExit ;
var
  ExitSave : Pointer ;

procedure MyExit ;
far    ;
begin
  ExitSave := ExitProc ;       {Always restore old vector first}
  ...
end ;

begin
  ExitSave := ExitProc ;
  ExitProc := @MyExit ;
  ...
end ;
```

- *Use of IOresult.* The built-in function **IOresult** returns MS-DOS error codes when performing input and output operations through the operating system. This function is used with input/output checking disabled (the $I compiler directive). Under these circumstances, use of **IOresult** (for input and output made through the operating system) can result in more robust code. For example, in the following code fragment, the procedure **FileIOCheck** would call the **IOresult** built-in function, determine whether the file-open was successful, and take appropriate action, such as bypassing a routine and informing the operator, if it was not successful (Borland, 1991).

```
{$I-}              {disable I/O Checking }
Assign(F, Filename);
Reset(F):
FileIOCheck;
```

It should be noted that input/output checking should normally be enabled. If it is disabled, as in the example above, an error checking routine should be performed immediately after the operation.

7.2.3 Input and Output Data Checking

The generic attributes for input and output data checking are applicable to Pascal.

7.3 Traceability

Traceability refers to attributes of safety software that support verification of correctness and completeness compared with the software design. The intermediate attributes for traceability are

- Readability
- Use of built-in functions
- Use of compiled libraries.

Because readability is also an intermediate attribute of maintainability, it is discussed in Section 7.4. Pascal-specific guidelines for the latter two attributes are discussed in the following subsections.

7.3.1 Controlling Use of Built-in Functions

The generic guidelines on the use of built-in functions apply to Pascal. Pascal functions defined in the standard are portable to other compilers. The distinction between built-in functions and intrinsics that may be implemented inline by the compiler is not always self-evident. Some "functions," e.g., *ord,* are really intrinsics. Some, such as *sqrt,* are really library functions.

The use of some built-in functions may be necessary or expedient. The decision is a design-level issue that is beyond the scope of this report. However, for functions determined to be desirable for inclusion in safety systems, the testing and related generic guidelines apply. An example of a function whose behavior should be tested and understood because it is not uniform across compilers is **mod** (modulo) (Grogono, 1983; p. 36).

7.3.2 Use of Compiled Libraries

The following guidance is specific to Borland Pascal

The generic guidance relating to limiting the use of compiled libraries is applicable to Pascal. Although there is no reference to compiled libraries in the Pascal language specification (ANSI, 1983), Borland Pascal has extensive support for compiled libraries and for dynamic linked libraries, which are part of the Microsoft Windows operating environment.

Borland Pascal units are program modules that make it possible to perform separate compilation. A unit can contain code, data, type, and/or constant declarations, and can use other units. The unit has a public section called *interface* and a private section called *implementation* (Borland, 1991). Units are necessary because of a 64K code segment limit (Borland, 1991). However, because they are compiled separately, they do not have the same visibility rules as text-based files, which are included prior to compilation. Thus, global types, variables, and definitions must be compiled into a separate global-level unit. Beneficial uses of units (even if not essential) include providing

common and enforceable data type declarations and module initialization. Constant definitions enhance safety and are not a violation of the guideline. Units can also be used to include well-tested and trusted libraries from the development organization. However, units used to include externally developed code and dynamic link libraries should be minimized.

Units can be recognized by the reserved word "unit" appearing at the beginning of the Pascal source code. The following is an example program that uses a precompiled unit called **Mathfunc.**

```
program calculate
{$R MATHFUNC}
uses Mathfunc;
type
...
```

The following is the beginning of the source code unit for the **Mathfunc** unit.

```
unit Mathfunc;
interface
function add (X, Y): real;
function multiply (X, Y): real;
...
implementation
function add...
function multiply...
```

In addition to precompiled units written in Pascal, it is also possible to link in code written in other languages, such as C, in Windows Dynamic Linked Libraries (DLLs) in a separate compilation unit called a *library*. This unit is identified by a reserved word "library" at the beginning of the source file. The functions which may be accessed by another routine can be recognized by the reserved word "export." The following is an example:

```
library Mathfunc;
function Power(x,y: real): Real; export;
begin
      Power:=Exp(y*ln(x));
end;

{ more functions here }
```

That a routine uses such library functions can be determined through the word "external." The following is an example of "external."

```
unit Mathfunc;
const Place: integer := 21;
interface
function add (X, Y): real;
function multiply (X, Y): real;
...
implementation
function add; external 'Mathfunc' index Place; {assuming this is the 21st
                                                  Function in the library }
function multiply...
```

There are several different types of libraries that could be used, depending on whether the application is running under MS-DOS only or MS-DOS and Windows; additional libraries may be used for object classes shipped with the language (applicable to both the MS-DOS and Turbo versions). The decision as to which libraries are necessary and which are expedient is a design-level issue that is beyond the scope of this report. However, for libraries determined to be desirable for inclusion in safety systems, the testing, configuration control, and related guidelines apply.

7.4 Maintainability

This section discusses the Pascal-specific attributes of the following intermediate attributes related to maintainability:

- Readability
- Data abstraction
- Functional cohesiveness
- Malleability
- Portability.

Base-level attributes and Pascal-specific guidelines are discussed in the following sections.

7.4.1 Readability

The following base attributes are related to readability:

- Conformance to indentation guidelines
- Descriptive identifier names
- Comments and internal documentation
- Limitations on subprogram size
- Minimizing mixed language programming
- Minimizing obscure or subtle programming constructs

- Minimizing dispersion of related elements
- Minimizing use of literals.

The Pascal-specific guidelines associated with these attributes are discussed in the following subsections.

7.4.1.1 Conformance to Indentation Guidelines

The guidelines developed for the generic indentation attribute are applicable to Pascal.

7.4.1.2 Descriptive Identifier Names

The guidelines developed for the generic descriptive identifier names attribute are applicable to Pascal. The following additional guidelines apply:

- *Separate words in compound names with underscores.*

```
Rads_Per_Second
Core_Temperature
```

- *Choose names that are as self-documenting as possible.*

- *When separate compilation units exist, utilize prefixes.* (The following guidance is specific to Borland Pascal.) Where there are multiple modules, it is possible to have a convention specifying that every export from a module have an identical descriptive prefix on the name. This allows a person reading the code to see immediately where a particular imported function, procedure, or variable came from.

7.4.1.3 Comments and Internal Documentation

The guidelines associated with the generic attributes are applicable.

7.4.1.4 Limitations on Subprogram Size

There are no Pascal-specific guidelines. The guidelines associated with the generic attributes are applicable.

7.4.1.5 Minimizing Mixed Language Programming

There are no Pascal-specific guidelines. Since there is no separate compilation in ANSI standard Pascal, there can be no mixed language programming. The guidelines associated with the generic attributes are therefore not applicable.

However, in Borland Pascal, separate compilation is supported and use of mixed language programming is, therefore possible (although non-portable). Since, generally speaking, there are differences in calling conventions and data types between languages, mixed languages should be used with caution, if at all.

7.4.1.6 Minimizing Obscure or Subtle Programming Constructs

There are no Pascal-specific guidelines. The guidelines associated with the generic attributes are applicable. The guidelines on side effects, global variables, and order of evaluation are also related.

7.4.1.7 Minimizing Dispersion of Related Elements

The guidelines associated with the generic attributes are applicable. In addition, when elements are dispersed throughout the code, it is hard to check, validate, and maintain the code.

The following guideline is specific to Borland Pascal.

- *Use compilation units to group related elements.* Pascal has a strict order in which it accepts declarations (i.e., label, **const**, type, **var**, **procedure** and function declarations, and finally the main procedure). Thus, it is difficult to keep the declaration, initialization, and use of types and variables close together in large programs in standard Pascal (Kernighan, 1981) . However, where separate compilation is supported, related variables and procedures can be kept in separately compiled units.

7.4.1.8 Minimizing Use of Literals

The guidelines associated with the generic attributes are applicable. In addition, the following Pascal specific guidelines apply:

- *Use constants for numeric literals.* The use of numeric literals as hard coded constants,

```
Area :=  3.14159265*sqr(radius) ;
```

instead of constant identifiers such as,

```
const
      pi : real := 3.14159265 ;
```

decreases readability and complicates maintainability, particularly if the literal is associated with a process parameter which may be tuned or a conversion factor which may be changed upon recalibration of an instrument. It is far easier to change one value set at the beginning of a source code file than it is to guarantee that all literals associated with such a parameter have been changed completely and correctly throughout all relevant source code files. When constants are not used, uniform comments should be associated with each constant to facilitate search and replace operations.

7.4.2 Data Abstraction

Data abstraction is the combination of data and allowable operations on that data into a single entity, and the establishment of an interface which allows access, manipulation and storage of the data only through the allowable operations. This principle results in the following specific base attributes:

•Minimization of the use of global variables.

7.4.2.1 Minimization of the Use of Global Variables

The guidelines associated with the generic attributes are partially applicable. Standard Pascal does not support external variables (local variables whose values persist in memory after the execution of the routine has ended). Thus, any values which are necessary in the next invocation of a function or procedure must be maintained at a higher scope. Moreover, as pointed out earlier, variables which must be initialized early in program execution of necessity must be visible at a relatively high position in the program hierarchy. Finally, there are appropriate uses for global variables, i.e., maintaining the state of data that must be accessed by many functions. The alternative is to pass such values as parameters which increases the complexity of the function interfaces.

Nevertheless, global variables obscure the passage of data between subprograms and defeat the benefits of data abstraction. They are a primary mechanism for side effects and the resultant subtle bugs. Thus, a balance must be struck between the characteristics of Pascal, which tend to encourage use of global variables (related to initialization and persistence of variables), and the principles of data abstraction.

7.4.2.2 Minimization of Complexity of Interfaces

The generic guidelines are applicable to Pascal. No language-specific attributes apply.

7.4.3 Malleability

The generic guidelines apply. Malleability is the ability of a software system to accommodate changes in functional requirements (Witt, 1994). Malleability extends data abstraction with the motivation toward isolating areas of potential change. To implement a malleable software system, it is necessary to identify what is expected to be constant and what is expected to be changed, and to isolate what is expected to be changed into easily identifiable areas where alterations can be made with a minimum of collateral changes.

7.4.4 Functional Cohesiveness

The generic guidelines are applicable. No additional guidelines apply.

7.4.5 Portability

The generic guidelines have limited applicability. From the perspective of safety, the benefits of portability are the adherence to standard programming constructs that yield predictable and consistent results across different operating platforms (Witt, 1994). However, the limitations of the standard base Pascal language make it difficult to write real time control programs without extensions. Some of the difficulties were discussed in this chapter (no external variables, no separate compilation units, no default ("otherwise") in a case construct, etc.). As a result, almost all Pascal compilers have language extensions to varying degrees. Thus, portability is difficult to achieve in Pascal.

References

ANSI/IEEE770X3.97-1983, *American National Standards Committee Pascal*, 1983 .

Borland International Corporation, *Borland Turbo Pascal 4.0*, Scotts Valley, CA, 1987.

Borland International Corporation, *Borland Pascal for Windows Programmer's Guide,* Scotts Valley, CA, 1991.

Coad, P., "OOD Criteria, Part 1," *Journal of Object-Oriented Programming*, 4: 69–70.

Grogono, P., *Programming in Pascal*, 2nd Edition, Addison-Wesley Publishing Company, Reading, MA, 1983.

Hutcheon, A., "A Study of High Integrity Ada," (UK) Ministry of Defence contract: SLS31c/73 Language Review, Document Reference SLS31c/73-1-D, Version 2, July 9, 1992.

Jensen, K. and N. Wirth, *Pascal User Manual and Report, Second Edition*, Springer Verlag, New York, NY, 1974.

Kernighan, B. W. and P.J. Plauger, *The Elements of Programming Style*, McGraw-Hill, New York, NY, 1974.

Kernighan, B. W. and P.J. Plauger, *Software Tools*, Addison-Wesley, Reading, MA, 1976.

Kernighan, B.W., *Why Pascal is Not My Favorite Programming Language*, April 2, 1981, Available from Internet Universal Resource Locator (URL): http://www.ee.ryerson.ca:8080/~elf/hack/pascal.html .

National Institute of Standards and Technology, *FIPS PUB 109 Pascal*, 1985. (Available from National Technical Information Service).

Page-Jones, M., *The Practical Guide to Structured System Design*, New York Yourdon Press, Prentice-Hall, New York, NY, 1980.

Pyle, I. C., *The Ada Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Software Productivity Consortium, *Ada Quality and Style Guidelines for Professional Programmers*, Van Nostrand Reinhold, New York, NY, 1989.

Witt, B. I, F. T. Baker, and W. W. Merritt, *Software Architecture and Design.* Van Nostrand Reinhold, New York, NY, 1994.

8 PL/M

This chapter discusses guidelines for the application of PL/M in safety systems. This chapter is organized in accordance with the framework of Chapter 2. Section 8.1 discusses reliability-related attributes; Section 8.2 discusses robustness-related attributes; Section 8.3 discusses traceability-related attributes; and Section 8.4 describes maintainability-related attributes. Appendix A.4 provides additional information on the language including its history and variations across different processors. A summary matrix showing the relationship between generic and language-specific guidelines, together with weighting factors, is included in Appendix B.

Intel Corp., the company which originally sponsored development and promoted the use of PL/M, discontinued support of their last PL/M compiler (PL/M-386) in December 1994. Since then, the use of PL/M in real-time control systems has diminished, and the number of programmers with proficiency in this language is also declining. Thus, conservative use of the language and its features is advisable in development of safety-related applications.

8.1 Reliability

Reliability implies that the software executes to completion, produces expected results, and that the output is within the required response time. Other attributes of *reliability* are as follows:

- Predictability of memory utilization
- Predictability of control flow
- Predictability of timing.

Further discussion on the relevance of these attributes as they relate to safe use of PL/M is found in the sections below.

8.1.1 Predictability of Memory Utilization

PL/M and the supporting development environment provide compile-time features for enforcing the predictability of memory utilization. These features do not depend upon run-time support portions of the compiler.

Unlike most other computer architectures, Intel's PL/M software development environment encourages the separation of data and instructions into distinct contiguous segments (Intel, 1990; Intel, 1992). The PL/M compiler generates relocatable object modules in which the various types of memory are kept separated. At program link time, all program instructions are collected and stacked together, followed by data constants, read-write variables, and stack allocation information.

After linking, Intel requires one last step before the program module is made executable in devices with nonvolatile RAM. The last step, known as Locate, maps the various collected memory segments by type into their final absolute memory addresses. All program instructions are mapped into a ROM segment, or an EEPROM segment where they remain nonvolatile until reprogrammed. RAM variables and the system stack are likewise mapped into an address space containing the read-write memories. The Locate step is not required where PL/M programs are being used with an operating system in volatile RAM. A loader performs the locate function in these cases.

8.1.1.1 Minimizing Dynamic Memory Allocation

The generic guideline applies. The PL/M language does not have built-in functions equivalent to the C `alloc` and `malloc`, which dynamically allocate RAM at run-time. Any dynamic allocation of RAM must be explicitly handled by the PL/M programmer. Such allocation is nevertheless discouraged and should be identifiable as part of a review.

8.1.1.2 Minimizing Memory Paging and Swapping

The generic guideline applies. In embedded systems where the bulk of PL/M has been used, the concepts of memory paging or process swapping are not likely to be used. In such systems, generally all programs reside in fixed read-only memory. Likewise, sufficient read/write data memory should be designed into a system. Removable or moving magnetic media are usually only used for data collection, monitoring, and secondary storage.

If memory paging and process swapping are proposed for use in an embedded safety system, the design should be reviewed and reconsidered in light of the above.

8.1.1.3 Minimizing Memory Bank Switching and Shadow Memory

The PL/M linker and locator programs can be manipulated to produce sections of binary code that have the same address space as other program modules, usually by means of a hardware bank-switching mechanism devised by the system hardware designers. This mechanism is commonly used in smaller micro-controller architectures (limited to 64k) when the complete address space has been consumed.

Use of hardware bank-switching, and its associated software housekeeping, should be avoided if at all possible because it is a source of unreliability. Great care must be taken to ensure that program and data code is where it is thought to be. Interrupts and exceptions may cause the vectoring of the program to an address page that has been switched out of working memory.

8.1.2 Predictability of Control Flow

Control flow defines the order in which statements in a program are executed. Control statements determine sequential execution of code, conditional branching, iteration and looping, and procedure invocation (Meek, 1993). A predictable control flow allows an unambiguous assessment of how the program will execute under specified conditions. Attributes related to safe control flow include the following:

- Maximizing structure
- Minimizing control flow complexity
- Initializing variables before use
- Single entry and exit points for subprograms
- Minimizing interface ambiguities
- Use of data typing
- Accounting for precision and accuracy
- Order of precedence of arithmetic, logical, and functional operators
- Avoiding functions or procedures with side effects
- Separating assignment from evaluation
- Proper handling of program instrumentation
- Controlling class library size
- Minimizing use of dynamic binding
- Controlling operator overloading.

These attributes and their relevance to safety are discussed in the following sections.

8.1.2.1 Maximizing Structure

The generic guideline applies. The PL/M language supports structured programming. Although PL/M does have a `goto` statement, in almost all cases a structured programming construct can be found to replace or eliminate it. Structure is maximized by eliminating *`goto`* statements and using appropriate block structured code instead. The PL/M constructs of `DO..CASE`, `DO..WHILE`, iterative `DO` and `IF..THEN..ELSE` permit branching with a defined return without introducing the uncertainty of control flow associated with the `goto` statement.

Guidelines, recommendations, and examples for enhancing a safe program using PL/M's structured constructs are provided below.

- *DO..END Blocks.* The simple DO..END statement pair is a building block of structured programming. The DO block in PL/M is sometimes confused with the active DO statements described below. The following example of a simple DO block is provided to clarify program blocks:

```
DO;
      Statement 1
      Statement 2
      Statement 3
         ----
      Statement n
END;
```

- *DO CASE Blocks.* The DO CASE statement in PL/M is a simpler construct than the CASE or SWITCH statement found in other languages and it must be used with care. The main problem with the PL/M CASE statement is that it is unbounded. It is quite easy to generate an out-of-bounds CASE value that will then branch into incorrect code. The code segment in the example below will produce unexpected and possibly disastrous results if ETEST is not in the range of 0 to 4.

```
ETEST = 5;
DO CASE ETEST;
     TEST = TEST + 1;          /* case 0 */
     TEST = TEST * TEST;       /* case 1 */
     ;                         /* case 2 (null stmt)*/
     TEST = TEST - 1;          /* case 3 */
     CALL NOTEST;              /* case 4 */
END;  /* End of DO CASE ETEST */
```

The reason for this construct is that the PL/M compiler generates an array of addresses (pointers) for each of the cases defined. Each address in the array points to a section of code for the particular CASE element. At the end of each code element, an absolute branch statement takes the code to the next statement after the DO CASE. If evaluation of the CASE index results in an out-of-range value, that incorrect value attempts to access a pointer to a nonexistent array element fetching a pointer to "garbage". Left unbound by the IF... THEN...ELSE statement, the DO CASE would subsequently perform a "wild" branch to the location pointed to by the erroneous pointer.

In contrast, other languages have a bounded CASE-like statement. The SWITCH statement in C, for example, will yield a `default` statement, or act as a null statement if the evaluated switch index does not match a valid case statement. For programmers with a background in C who are about to embark on a PL/M project, this statement may be a source of potential problems.

This shortcoming of PL/M can be corrected by containing the DO CASE statement within a condition (i.e., an IF statement) that checks whether the DO CASE index is within the valid range. In the following example, if ETEST is negative or greater than 4, the ELSE clause will catch and handle the exception. The DO CASE statement will be ignored when ETEST is out of range.

```
IF (ETEST >= 0) AND (ETEST < 5)     /* Confine cases to [0..4]*/
    THEN DO CASE ETEST;
        TEST = TEST + 1;            /* case 0 */
        TEST = TEST * TEST;         /* case 1 */
        ;                           /* case 2 (null stmt)*/
        TEST = TEST - 1;            /* case 3 */
        CALL NOTEST;                /* case 4 */
    END;                            /* End of DO CASE ETEST */
ELSE CALL TEST_NUMBER_EXCEPTION     /* handle exception */
```

An alternative to this construct is to limit the use of the DO CASE statement to binary (i.e., true/false) conditions.

- *DO WHILE Blocks and IF Statement.* Relational comparisons normally result in 0FFH being set for TRUE and 00H being set for a FALSE condition. DO WHILE only looks at the least significant bit to determine TRUE (=xxxxxxx1B) or FALSE (=xxxxxxx0B) condition. This may cause confusion when using both the DO WHILE statement and the IF statement as shown in the following examples:

```
Improper assumptions: 00H is FALSE;   01H..0FFH is TRUE
                      00H is FALSE;   0FFH is TRUE;
                      01H..0FEH undefined.

Correct assumption:  xxxxxxx0B is FALSE;  xxxxxxx1B is TRUE.
```

- *Procedure Activation.* In PL/M, there are three ways in which a procedure can be activated. In the first two a procedure is invoked by name, and there is no problem (in these forms the parameter list is optional):

```
CALL name [(parameter list)];     /* untyped procedure form */
name [(parameter list)];          /* typed procedure form   */
```

A third type of procedure invocation is possible: by location. This method contains risks, as the compiler does not fully check the number of parameters passed, nor does it provide automatic type conversion for these parameters. The invocation form for call by location is as follows:

```
CALL location[.member-identifier] [(parameter list)];
```

The location value can be a structure reference, but it cannot be subscripted. Use of the call-by-location method of invocation is *not* recommended. If this style must be used, detailed attention must be given to the parameter list. Since both type conversion and parameter checking occur at compile time, checking these constructs can prevent problems.

- *goto Statement.* The `goto` statement should be avoided because it leads to unstructured code. Programming teams should be challenged to develop a complete software program without using a single `goto` statement. There is almost always a way to structure code so that a `goto` statement is not needed. `goto` statements sometimes crop up when a programmer becomes frustrated with the handling of exception or error handling code. Generally, it is better to handle errors and exceptions locally rather than to branch out of the middle of the block. Exception handling is further discussed below.

- *Comments /* ... */.* The method in which PL/M implements comments can sometimes cause problems. In certain cases, unmatched comment pairs inadvertently "comment out" sections of source code statements. If this occurs in code segments that are infrequently used, such as safety handling exceptions, the fault can go unnoticed for a long period of time. In the following example, statement2 has been inadvertently commented out by the missing terminator of statement1. The compiler will not object as it is only scanning for the next comment terminator `"*/"`.

```
    ....
statement1;   /* This is a comment about these...
statement2;   /* ...three statements and how statement 2... */
statement3;   /* ...has been accidentally commented out.    */
    ....
```

In the PL/M-80 and PL/M-86 compiler, unbalanced comment pairs are not caught and

flagged by the compiler when they occur at the end of a compiled module. In the following case, statement3 does not produce code because it is inadvertently commented out. The compiler also does not object and does not produce a warning or error. In this case, we have a compiler weakness or shortcoming that does not object to unbalanced comment delimiter pairs.

```
        ....
    statement1;   /* This is a comment about these...          */
    statement2:   /* ...three statements and how statement 3...
    statement3;   /* ...has been accidentally commented out.
END;
```

8.1.2.2 Minimizing Control Flow Complexity

All generic guidelines under this heading apply to PL/M. Excessive nesting can usually be avoided by the use of functions, subroutines, or CASE statements in place of in-line branches. Guidelines specifying a limit on the nesting levels should be included in the project's programming handbook.

8.1.2.3 Initialization of Variables Before Use

The generic guideline applies in PL/M. In embedded systems, uninitialized variables can often be the source of latent software bugs.

In PL/M, the variables initialized prior to execution are part of the CONSTANT segment and are normally stored with the CODE segment. If a variable requires an initial value, but is not a constant, then it must be initialized by the software. PL/M compilers do not contain built-in facilities to provide initialization of variables automatically. The compiler will help partition the code into data segments, but the user must write the code to move the data from a ROM segment into a RAM segment to initialize it at run time. The reason is that most PL/M applications do not run under a standard operating system, which would normally handle the initialization on program loading.

Certain debugging tools can mask initialization problems during development. In-circuit emulator systems may test and initialize emulation memory as part of the power-up sequence. Hence, when a user program executes in the emulation environment, every variable has unknowingly been initialized to a known value (usually zero). When this same debugged code is moved to the actual operating platform, the RAM values will likely be random. This condition can result in latent flaws with safety significance -- particularly in rarely used exception and error handling code.

One method of avoiding the above condition is to clear all RAM areas to zero intentionally and explicitly as part of the software initialization process. In embedded systems, the software often performs some self-test on the hardware system well before the main program is entered. The

pseudocode shown in the example below illustrates how PL/M startup code can provide proper "housekeeping" before beginning to execute.

```
Power$On$RESET:
  /*---- Gain control of the System ----*/
     Disable Interrupts;
     Bring all peripherals to known state;
     Perform system self-tests;

  /*---- Setup operating environment ----*/
     Set up interrupt vectors;
     Initialize peripheral devices;
     Clear all RAM to zeros;
     Initialize program RAM variables;
     Enable appropriate interrupts;

Main$Program$Loop:      /* Drop into Main Program */
     Statement_1;
        ....
```

8.1.2.4 Single Entry and Exit Points in Subprograms

The generic guideline applies in PL/M. Multiple entry and exit points in a subprogram introduce uncertainties in the control flow similar to the use of *goto* statements. Control flow predictability is enhanced when there is only a single entry point, and a single exit point from a subprogram. Because predictability of execution flow is important to safety, multiple entry points in procedures or functions should not be used even if the language supports them.

- *No calls to locations.* When PL/M procedures are invoked by name, they can only have one entry point, which is the name assigned to the procedure itself. However, PL/M also allows a call to a location. This is dangerous as the compiler will not guarantee that the destination location is even a procedure or that it has a valid RETURN statement. Repeated invocations to this errant location will continue to PUSH data onto the system stack without a corresponding POP of the same data off the stack on exit. The result will be a system crash as the stack grows out of bounds.

  The example below illustrates how a second entry point can be dangerously assigned to a procedure.

```
DO$IT$ALL: PROCEDURE (A, B);
           Statement_1;
             ....
             ....
DO$SOME:   Statement_k; /* Label entry point */
             ....
             ....
           RETURN;
```

For safety related reasons, it is recommended that the procedure call-by-location not be used. A better method to accomplish the above is shown below. Here, two procedures are defined instead of one with multiple entry points. Both procedures now have only one entry point and one exit point.

```
DO$SOME:   PROCEDURE
           Statement_1;
             ....
           Statement_n;
           RETURN;

DO$IT$ALL: PROCEDURE (A, B);
           Statement_1;
             ....
             ....
           CALL DO$SOME;
           RETURN;
```

8.1.2.5 Minimizing Interface Ambiguities

Interface errors in argument lists and messages passed to other program entities account for many coding errors. These errors may appear syntactically correct to the compiler and hence go unnoticed until runtime. An example of such an error is reversing the order of arguments when calling a procedure. Unfortunately, PL/M offers limited safeguards to prevent such problems (i.e., a linker check for the number and type of parameters).

The following specific guidelines apply:

- *Use templates during code development.* A template can provide a useful mechanism for preventing argument list errors. In the example below, each procedure when written includes a calling sequence template stored as a comment in the procedure's header block. Each time a procedure invocation is to be coded, the programmer should COPY the calling template (three lines in the following example) and PASTE it where the invocation should occur. The comment delimiters are then removed, and the associated parameters become part of the program. Once the invocation has been coded, the remaining commented declaration lines can be deleted. By having all of the information at hand at the coding point, the programmer does not risk guessing at the parameter specifications. Templates should also be built for system procedures and built-in functions. The following example shows a procedure CALL template:

```
/*****************************************************************/
/* Calling Template:                                             */

/*  CALL FIRE$LASER (CHANNEL, DURATION, POWER$LEVEL);            */
/*             DECLARE CHANNEL BYTE;                             */
/*             DECLARE DURATION, POWER$LEVEL REAL;               */

/*                                                               */
/*****************************************************************/
FIRE$LASER: PROCEDURE (CHANNEL, DURATION, POWER$LEVEL);
             DECLARE CHANNEL BYTE;
             DECLARE DURATION, POWER$LEVEL REAL;
```

- *Parameter Validity Checking.* In any language, including PL/M, active checks can be placed in the code to ensure that proper parameters have been passed. In the FIRE$LASER example below, checks can be placed at the beginning of the procedure to ensure that all parameters passed are valid. A compound IF statement is used to verify data before the actual procedure logic is invoked in the following example.

```
FIRE$LASER: PROCEDURE (CHANNEL, DURATION, POWER$LEVEL);
   DECLARE CHANNEL BYTE;
   DECLARE DURATION, POWER$LEVEL REAL;
   DECLARE DURATION$LOW LITERALLY '0.0';  /* Minimize literals in...*/
   DECLARE DURATION$HI LITERALLY '3.0';  /* ...code by declaring...*/
   DECLARE POWER$LOW LITERALLY '0.0';    /* ...them centralized... */
   DECLARE POWER$HI LITERALLY '100.0';   /* ...in the header.      */

     IF (    ((CHANNEL = 1) OR (CHANNEL = 2))
         AND ((DURATION > DURATION$LOW) AND (DURATION < DURATION$HI))
         AND ((POWER$LEVEL) > POWER$LOW AND (POWER$LEVEL < POWER$HI))
        ) THEN DO;
              ... /* Code to fire the laser */
       END;
END; /* End of FIRE$LASER */
```

In areas of safety-critical applications, this overhead is justified to ensure that parameters passed are within acceptable range. Although these parameters may have been checked elsewhere, these checks add an extra level of safety if some of the calling code is modified incorrectly during maintenance in the future.

8.1.2.6 Use of Data Typing

The generic guideline applies. Acceptance of data that is different from that intended for use by a subprogram or procedure can cause failures. The PL/M language provides for simple data typing of variables and constants. In PL/M the data types are fixed and predefined. Simple data typing provides for memory length and simple data pattern format checking. Thus, the data types BYTE and unsigned `char` or `WORD` and `int` can occupy the same number of bits, but have different meanings when being evaluated. For example, `WORD` is 0...65535, but `int` is -32768..32767.

In PL/M, only the constant data type is checked for a maximum and minimum range. This is only to ensure that the compiler can properly fit the data value into the specified data type. No user-specified range check is made. *Strong Data Typing*, which allows a user not only to specify a data type but also to place valid range bounds on that data type, is not supported.

Specific guidelines are as follows:

- *Actively check all mathematical and index values prior to use.* As PL/M does not support strong data typing, this must be implemented manually. Calculated values should be checked for their potential to overflow or underflow. Index values should be checked to ensure that they do not attempt to access out-of-bound array or matrix elements. Memory pointers should also be checked to ensure that they point to valid memory areas.

- *Avoid automatic or implicit type conversions.* For clarity, readability, and comprehension, explicit type conversions should be used.

- *Avoid mixed mode operations.* Mixed mode operations should also be avoided for the same reasons as stated above.

- *Limit the use of indirection with indices, pointers, and based variables to situations where no other reasonable alternatives exist.* Validation should be performed on indirectly addressed data to ensure correctness of the accessed locations.

- *Add explicit range checking.* Adding explicit data checking when the data has not been validated previously can be prudent. In the example below, the variable `DURATION` is verified by the procedure `CHECK$DURATION` to ensure that its value is within a valid range. Line 34 of this example uses a compound `If` statement to ensure that all laser parameters are in range before allowing the laser instrument to fire. The `ELSE` clause of this same statement on line 36 locally handles the case of one of these parameters being out of range.

```
 1          STRONG$DATA$TYPE: DO;

 2   1      DECLARE TRUE LITERALLY '0FFH';
 3   1      DECLARE FALSE LITERALLY 'NOT TRUE';

 4   1      CHECK$DURATION: PROCEDURE (DURATION) BYTE;
 5   2            DECLARE CHK$FLAG BYTE, DURATION WORD;
 6   2            DECLARE DURATION$LOW LITERALLY '0';
 7   2            DECLARE DURATION$HI LITERALLY '3';

 8   2            IF ((DURATION > DURATION$LOW) AND
                      (DURATION < DURATION$HI))
                    THEN CHK$FLAG = TRUE;
 9   2            ELSE CHK$FLAG = FALSE;
10   2            RETURN (CHK$FLAG);
11   2      END CHECK$DURATION; /* End of Procedure */

12   1      CHECK$POWER$LEVEL: PROCEDURE (POWER$LEVEL) BYTE;
13   2            DECLARE CHK$FLAG BYTE, POWER$LEVEL WORD;
14   2            DECLARE POWER$LOW LITERALLY '0';
15   2            DECLARE POWER$HI LITERALLY '100';

16   2            IF ((POWER$LEVEL > POWER$LOW) AND
                      (POWER$LEVEL < POWER$HI))
                    THEN CHK$FLAG = TRUE;
17   2            ELSE CHK$FLAG = FALSE;
18   2            RETURN (CHK$FLAG);
19   2      END CHECK$POWER$LEVEL; /* End of Procedure */

20   1      CHECK$CHANNELS: PROCEDURE (CHANNEL) BYTE;
21   2            DECLARE (CHK$FLAG, CHANNEL) BYTE;
22   2            DECLARE CHAN$A LITERALLY '3';
23   2            DECLARE CHAN$B LITERALLY '23';
24   2            DECLARE CHAN$C LITERALLY '19';
25   2            IF ((CHANNEL = CHAN$A) OR
                      (CHANNEL = CHAN$B) OR
                      (CHANNEL = CHAN$C))
                      THEN CHK$FLAG = TRUE;
26   2            ELSE CHK$FLAG = FALSE;
27   2            RETURN (CHK$FLAG);
28   2      END CHECK$CHANNELS; /* End of Procedure */

29   1      LASER$SETUP$EXCEPTION: PROCEDURE;
              /* ...exception handling code here... */
30   2      END LASER$SETUP$EXCEPTION;

31   1      FIRE$LASER: PROCEDURE (CHANNEL, DURATION, POWER$LEVEL);
32   2           DECLARE CHANNEL BYTE;
33   2           DECLARE (DURATION, POWER$LEVEL) WORD;

34   2           IF (     (CHECK$CHANNELS(CHANNEL) )
                      AND (CHECK$DURATION(DURATION) )
                      AND (CHECK$POWER$LEVEL(POWER$LEVEL) )
                    ) THEN DO;
                       /* ... Code to fire the laser */
35   3                 END;
36   3           ELSE CALL LASER$SETUP$EXCEPTION; /* handle exception */
37   2      END FIRE$LASER; /* End of FIRE$LASER */
38   1      END STRONG$DATA$TYPE; /* End of Program */

END OF PL/M-386 COMPILATION
```

8.1.2.7 *Precision and Accuracy*

The generic guideline applies. The software application must provide adequate precision and accuracy for the intended safety application. Safety concerns are raised when the declared precision of floating point variables is not supported by analysis, particularly when small differences between large values are calculated. The following are specific guidelines:

- *Account for different hardware.* The same data types, when used by different compilers, may have different precision. For instance, the data type `WORD` is a 16-bit number in PL/M-86 and PL/M-286, but becomes a 32-bit number in PL/M-386. Likewise `DWORD` is a 32-bit number in PL/M-86/286 and a 64-bit number in PL/M-386.

- *Account for optimization in floating point computations.* Unexpected results can occur during compiler code optimization. This is especially an issue with floating point computations. A compiler might replace `((1.0+x)-x)` with 1.0 at compile time, when the floating point rounding error is what the program is trying to compute. Note that the above optimization is always guaranteed to be correct for integer types.

- *Verify numeric precision in ported code.* In porting code containing calculations, the range of precision of the data types should be investigated and verified. This is particularly true when porting code downward to a less powerful platform. Even though the data types may be syntactically equivalent, their precision may be inadequate for the function to be ported.

- *Express precision in terms of numeric ranges.* Comment block procedures with precise numeric ranges (rather than data types) are shown in the following example.

```
/*   Designed for the PL/M-386 platform.              */
DECLARE DELTA$VOLTS WORD;    /* Range: 0..(2**32)-1  */
DECLARE VOLT$1 HWORD;        /* Range: 0..(2**16)-1  */
DECLARE LED$V BYTE;          /* Range: 0..255        */
```

If the code in this example were to be run on both an 80286 and an 8086-based platform, the values for `DELTA$VOLTS` and `VOLT$1` would have be changed from `WORD` to `DWORD`, and from `HWORD` to `WORD`, respectively, in order to maintain the same mathematical precision. This becomes a simpler task if the intended data range has been expressed in comments by the original designer of the procedure, such as in the example shown below.

```
/*   Designed for the PL/M-86 or 286 platforms.      */
DECLARE DELTA$VOLTS DWORD;  /* Range: 0..(2**32)-1  */
DECLARE VOLT$1 WORD;        /* Range: 0..(2**16)-1  */
DECLARE LED$V BYTE;         /* Range: 0..255        */
```

In the above example, expressing the variable only by data type leaves the issue of changing the data type ambiguous. Without this information, the programmer inadvertently or unknowingly may leave DELTA$VOLTS as data type WORD in the porting process.

8.1.2.8 Use of Parentheses Rather than Default Order Precedence

The generic guideline applies. The default order of precedence of arithmetic, logical, and other operations varies among languages. Developers or reviewers may make incorrect precedence assumptions when explicit parentheses are not used. In moving between languages with similar statement definitions such as "C" and PL/M, developers and reviewers are particularly vulnerable to these wrong assumptions about order of operations.

The explicit use of parentheses and other mechanisms for ensuring a clear statement of the order of evaluation of operations should be used. In some cases, complex statements should be broken down into two or three simple statements to enhance clarity and readability and to ensure that the compiler properly evaluates the statement expressions. This is particularly the case in floating point computations when compiler optimization is used. Such expressions should be broken up into multiple statements because the ordering of statements is usually preserved, even by optimizing compilers.

8.1.2.9 Avoiding Functions or Procedures with Side Effects

Generic guidelines are applicable.

8.1.2.10 Separating Assignment from Evaluation

Separation of assignment statements from the evaluation of expressions is particularly important in PL/M because the syntax defines two meanings for the token "=" (equal sign). The equals sign can represent the logical relational operator "equals," or it can represent the assignment of a value to a variable. PL/M attempts to compensate for this by defining an embedded assignment token of ": =" (colon, equals). The latter is explained below. Embedded assignment can also occur by invoking a typed procedure within an expression.

Embedded assignment statements should be separated from the evaluation of expressions. The PL/M language documentation (Intel, 1990) explicitly states that:

> *"...the rules of PL/M do not specify the order in which subexpressions or operands are evaluated. When an embedded assignment changes the value of a variable that also appears elsewhere in the same expression, the results cannot be guaranteed."*

Intel does not guarantee the order in which the following ambiguous expression will be evaluated. In addition, the compiler may even interpret the statement differently in various levels of compiler optimization. The expression:

```
A = (X:=X+4) + Y*Y + X;
```

could result in A being assigned either of the following:

```
(X+4) + Y*Y + (X+4);
(X+4) + Y*Y + X;
```

The ambiguity can be removed by separating out the embedded assignment statement, and recoding explicitly as the programmer intended it to be:

```
 X = X + 4;

A1 = X + Y*Y + X;        or,

 X = X + 4;

A2 = X + Y*Y + (X-4);
```

In summary, safety concerns dictate that assignments be separated from evaluation in order to avoid ambiguity and to improve readability of the code. Modern compilers do well in constructing optimized code. The inclusion of a large number of terms in an expression in source code statement rarely results in more efficient machine code than the same logic broken out into two or more lines of code.

8.1.2.11 *Proper Handling of Program Instrumentation*

The generic guideline applies. Program instrumentation is used to collect and output certain internal state values of a program during execution. Program instrumentation is one method that allows a developer to check that particular aspects of a specification have been correctly implemented (Liao, 1991). Use of program instrumentation is often the only method for observing the operation of systems containing proprietary and/or protected operating systems. Fortunately for the vast majority of PL/M users, nonintrusive real-time methods of obtaining the same information exist through use of the in-circuit emulator development tool.

In-circuit emulators (ICE) allow detailed data about a program's execution to be collected in a non-invasive manner while the program executes in real-time. Since no code is necessarily added to the program, the program being executed under the ICE unit can be the exact code to be run in the final system.

If an ICE system is not available, or for some reason program instrumentation appears preferable, the following guidelines and recommendations are offered:

- *Minimize run-time perturbations.* Instrumentation that interferes with the normal execution flow and timing rhythms is undesirable in safety applications because it will change the normal operation pattern of the program. Less intrusive methods should be employed, such as collecting data in memory and later processing them in a background task.

- *Instrumentation source code should remain visible.* PL/M does not provide any compiler features that generate hidden or concealed code for a "debug" mode of operation. Compiler directives may be used, however, to compile program instrumentation conditionally into the code. This is generally acceptable if the two models do not depart as discussed above.

- *Conform to software instrumentation and test guidelines.* Program review is facilitated and safety enhanced if instrumentation and test procedures are described in the project- specific handbook. Program instrumentation and test are often detailed in a separate test specification. These test specifications should describe the program instrumentation and its scope in detail.

8.1.2.12 *Control of Class Library Size*

The generic guideline does not apply. Because PL/M is an older language, it does not contain any of the features or concepts related to object-oriented methods, including classes, inheritance, operator overloading, and polymorphism. Object-oriented characteristics can be enhanced by controlling limits on subprogram and module sizes.

8.1.2.13 Minimizing Dynamic Binding

PL/M does not support dynamic binding of code segments. As PL/M is primarily an embedded language that executes from nonvolatile ROM, the dynamic binding of code during run time is not supported. However, bank switching, which is a hardware form of dynamic binding, sometimes appears. Hence, the following specific guideline for this issue.

The PL/M object code linkers and locate programs do allow for the generation of overlay or shadow ROM code (see section 8.1.1.2) by the use of hardware bank switching techniques. These represent a risk and should therefore be eliminated. Bank switching is difficult to test and debug, particularly in the areas of fault and interrupt handling.

Most cases of bank switching appear in modifications to a system when the complete address space becomes full. From a safety standpoint, bank-switching is never worth the risk and effort. It is preferable to upgrade the hardware to the next microcomputer architecture containing a larger memory address space.

8.1.2.14 Control of Operator Overloading

The generic guideline does not apply. The PL/M language does not support the concepts of polymorphism or operator overloading.

8.1.2.15 Compiler Optimization and Hardware Flags

PL/M-86 and later compilers are capable of performing extensive optimizations on the object code generated by earlier passes of the compiler. Such optimization changes the exact sequence of machine code produced from a given sequence of PL/M source statement.

One of the impacts is that the microprocessor hardware flags cannot be predicted or determined for any given point in a program. As an apparent carry-over from the early unoptimized PL/M-80 compiler, the language provides built-in functions that attempt to return the current value of the hardware flags. These built-in functions should be used with caution if used at all. They are listed in the following table.

Table 8-1 Optimization and Hardware Flags.

Hardware flag bits	CARRY, SIGN, ZERO, PARITY
Carry-rotation functions	SCL, SCR
Decimal adjust function	DEC
Hardware register	FLAGS
Arithmetic operators	PLUS, MINUS

Functions that use these hardware flags should be programmed in assembly language so that predictable control can be achieved. It is also recommended that, where warranted, a library of these functions be developed in one module so that they might be isolated and better maintained.

8.1.3 Predictability of Timing

Predictability of timing is crucial in a safety system used in real time control (Kopetz, 1993; Leveson, 1992). Response to asynchronous interrupt inputs must be predictable to ensure that safety-related procedures are allowed to complete execution within their precise window of time according to specification. In addition, output values must be computed and prepared according to precise timing requirements.

8.1.3.1 Minimizing the Use of Tasking

Tasking is undesirable in safety systems unless there is a compelling justification. The PL/M language does not provide any language facility for implementing concurrent processing. Intel does, however, provide a compatible real-time operating system kernel known as iRMX.

If an operating system kernel such as Intel iRMX is used, it should be provided with complete source code. Although the user documentation for such a system may be extensive, developers need to have access to all aspects of this controlling code to avoid safety-related problems that may be hidden from view.

8.1.3.2 Minimizing the Use of Interrupt-Driven Processing

Use of interrupts to handle the acceptance and processing of plant and operator inputs can reduce average response times. It also usually leads to nondeterministic maximum response times. Improper use of interrupt-driven processing has been implicated in at least one fatal accident (Leveson, 1992). Documents and standards related to digital system safety generally discourage or prohibit the use of interrupt-driven processing to facilitate analysis of synchronization and run-time behavior and to avoid the nondeterministic response times inherent in interrupt-driven processing.

However, use of interrupts may be necessary to capture asynchronous data within a certain deadline. Not doing so may allow the external data to change or become overrun with other new data. The following specific guidelines are applicable.

- *Interrupt handlers should be as short and simple as possible.* The processing associated interrupts should be minimized. The interrupt handler should only access, queue, and flag data for processing at a later time. There should be only a single path of execution with no delays or waiting involved.

- *Avoid nested interrupts.* Nested interrupts should not be permitted in safety systems.

- *The interrupt handler should not set or otherwise alter shared data.* In general, the interrupt handler should write data into a dedicated memory area or buffer. However, if the handler must access shared data, some form of locking or mutual exclusion may be required when using interrupts.

The following is a descriptive example of an interrupt driven system. This basic design has been used in a number of successful biomedical and process control instruments. A hardware timer provides a system "heartbeat" of 30 ms. This heart beat time is arbitrarily chosen and could be set to any reasonable time-slice interval.

Every 30 ms the timer interrupts the background task and performs any time critical tasks. The interrupt duty cycle is designed to not exceed 50 percent.

Hardware signals are latched and generate a level two interrupt. Interrupt handlers are designed to be low in overhead. They execute as a fast "store, flag, and return." In other words, on interrupt they:

Fetch the waiting input data,
Store it in a queue,
Set a data available flag, and
return to processing.

This approach eliminates the use of interrupt processing and yet acknowledges asynchronous input data quickly.

Every 30 ms the level one timer interrupts. The level one task then performs the following:

- Checks critical areas of the system for validity.
- Looks for new queued input data.
- Calculates any new controlled output values.
- Outputs new values (if any).
- Returns from Interrupt.

When interrupt processing has been completed, the system returns to background processing. Tasks that are not time critical are continuously processed in a priority order in this task. Examples include writing data to a display buffer, storing data in a data cartridge and similar tasks.

Tasking has been minimized in this system. In addition, and most important, the tasking that does exist is explicitly controlled; it is not delegated to a black box operating system kernel. Interrupts are used as necessary to capture (but not process) real-time events. They then terminate as rapidly as possible. The timer-interrupt routine is efficient enough to complete all of its tasks within 15 ms.

8.2 Robustness

Robustness (or survivability) refers to the capability of the software to continue execution during abnormal or other unanticipated conditions. Robustness is an important safety system attribute because unanticipated events can occur during an accident or excursion. The ability of the software to continue monitoring and controlling under such circumstances is vital. The intermediate attributes for robustness are as follows:

- Controlled use of software diversity
- Controlled use of exception handling
- Input and output checking.

These attributes and their relevance to safety are discussed in the following sections.

8.2.1 Controlled Use of Software Diversity

The decision to employ diverse software implementations is a design-level function. The PL/M languages offer no features that require more than the generic concerns under this heading.

8.2.2 Controlled Use of Exception Handling

Exception handling deals with abnormal system states and input data. Exception handling provisions in some languages facilitate the establishment of alternate execution paths in the event of conditions that, although unexpected, result in states that can be defined in advance. Problems can arise in the use of exception raising and handling, however, because execution flow during exception conditions is often difficult to trace.

Attributes that pertain to safe exception handling include the following:

- Local handling of exceptions
- Preservation of external control flow
- Uniformity of exception handling.

PL/M has no native facilities that support exception handling. Synchronous exceptions can be handled locally, but asynchronous ones may require an interrupt or trap handler to process them. Asynchronous exceptions can only be handled by interrupt or trap handlers. The effect of handling the exception in this way can be localized to the module containing the handler, and flags can be used to communicate the error to other modules. Sometimes polling can be used to turn an asynchronous condition into a synchronous one.

8.2.3 Input and Output Checking

Input and output data should be validated before being used. Corruption of data, whether due to a transient failure of a sensor, a flipped memory bit, or an invalid calculation, can have serious consequences on subsequent processing if the error is allowed to propagate. PL/M does not offer any specific language features to accomplish this checking. However, data can be validated as part of the application software as shown in the following example.

The example incorporates both input/output checking and local exception handling. This procedure checks and confines the input and output data to specific ranges. In addition, the exceptions raised from data being out of range are handled by a local procedure.

Lines 6 through 18 in the example are nested local procedures that perform input and output data checking. Also, the procedure `HANDLE$EXCEPTIONS` provides a local facility for handling the exceptions encountered in this procedure.

The reason for using a procedure to accomplish this is that procedures provide isolation and localization of the exception code. They also increase readability which promotes review and maintenance. Although not shown in this example, the complete limits and default values for the input and output data should be explicitly defined within the local procedure with a series of `DECLARE...LITERALLY` statements.

Use of this format also provides some of the positive attributes of data abstraction and encapsulation. All data and procedures necessary to handle data I/O checking and exceptions are contained within procedure `CALCULATE$VELOCITY`.

On line 20 of the example, the data input values are checked and adjusted. If any are out of range, an exception can be raised that will be handled later in the procedure. Between lines 20 and 21, the full calculation of velocity will occur. Line 23 then checks the results of the computations and adjusts them before making the data available as output from this procedure.

During execution of this procedure, data input and output exception flags may have been raised by either local procedures `IN$CHECK` or `OUT$CHECK`. Perhaps further processing of these noted exceptions is necessary. A message may have to be sent to another module warning of a possible degradation of the system. This might be done in local procedure `HANDLE$EXCEPTIONS`.

If necessary in the design, an exception flag can be returned from the typed procedure `CALCULATE$VELOCITY`.

```
                /*********************************************************/
 3   1          CALCULATE$VELOCITY: PROCEDURE (CHAN$1, CHAN$2, TIME) BYTE;
 4   2              DECLARE (CHAN$1, CHAN$2, TIME) REAL;
 5   2              DECLARE V$EXCEPT BYTE;

                    /*****************************************************/
                    /* Local Procedure: IN$CHECK                         */
                    /* Checks that input data is within valid range.     */
                    /* Substitutes Max/Min data for out of range data ... */
                    /* .. so that calculations can continue.             */
                    /*****************************************************/
 6   2              IN$CHECK: PROCEDURE BYTE;
 7   3                  DECLARE I$EXCEPT BYTE;
                          /* ... other statements ...  */
 8   3                  RETURN (I$EXCEPT);
 9   3              END IN$CHECK;

                    /*****************************************************/
                    /* Local Procedure: OUT$CHECK                        */
                    /* Checks that output data is within valid range.    */
                    /* Adjusts as necessary so that computation and...   */
                    /* ... control can continue as normal.               */
                    /*****************************************************/
10   2              OUT$CHECK: PROCEDURE BYTE;
11   3                 DECLARE O$EXCEPT BYTE;
                          /* ... other statements ...  */
12   3                 RETURN (O$EXCEPT);
13   3              END OUT$CHECK;

                    /*****************************************************/
                    /* Local Procedure: HANDLE$EXCEPTIONS                */
                    /* ...code to handle the out-of-data-range exception */
                    /* ...locally so that calculations can continue.     */
                    /*****************************************************/

14   2              HANDLE$EXCEPTIONS: PROCEDURE BYTE;
15   3                 DECLARE C$EXCEPT BYTE;
                          /* ... Handle local exceptions here ... */
16   3                 C$EXCEPT = FALSE;
17   3                 RETURN (C$EXCEPT);
18   3              END HANDLE$EXCEPTIONS;

                    /*-----------------------------------------------------*/

19   2              DECLARE (EXCEPT$IN, EXCEPT$OUT) BYTE;

20   2              EXCEPT$IN = IN$CHECK;     /* Check data about to be used */
                       /* ...Perform all processing of data here... */
                       /*      ... other statements ...             */
21   2              EXCEPT$OUT = OUT$CHECK;  /* Check data just computed */

22                  V$EXCEPT = TRUE;
23   2              IF (EXCEPT$IN OR EXCEPT$OUT)THEN V$EXCEPT = HANDLE$EXCEPTIONS;

24   2              RETURN (V$EXCEPT);  /* exception flags can also be...    */
                                         /* ...returned to caller if desired. */
25   2          END CALCULATE$VELOCITY;
```

The above design preserved the flow of the control logic while handling any exceptions. No `goto` statements have been used to branch to other outside exception handling code, thus transferring flow to another control path.

8.3 Traceability

As defined earlier, traceability refers to attributes that support and allow verification of correctness and completeness when compared to the software design specifications. The intermediate attributes for traceability are as follows:

- Readability
- Use of built-in functions
- Use of compiled libraries.

Readability is an intermediate attribute shared by traceability and maintainability; it is discussed under that heading in Section 8.4 below. The latter two attributes and the PL/M features relevant to safety are discussed in the following section.

8.3.1 Use of Built-in Functions

Generic guidelines apply to PL/M. Concerns over the use of built-in functions can be addressed by controlling the use of built-in functions through organizational or project-specific guidelines. Regression test cases make it possible to establish conformance with expected results for new releases of compilers and runtime libraries. Therefore, regression test cases, procedures, and results of previous testing for allowable built-in functions should be maintained. Test cases should assess behavior for out-of-bounds and marginal conditions in the specific runtime environment. Examples of these conditions include negative arguments on square root functions and improperly terminated strings. The built-in functions included with PL/M-386 are shown below.

LENGTH, LAST, SIZE	LOW, HIGH
DOUBLE, REAL, FLOAT, FIX	INT, SIGNED, UNSIGN
ABS, IABS	BYTE, WORD, HWORD
CHARINT, SHORTINT, INTEGER	SELECTOR, OFFSET, POINTER
Rotate (ROL, ROR)	Log Shift (SHR, SHL)
Arith Shift (SAL, SAR)	Move (MOVB, MOVW, MOVHW)
Compare (CMPB, CMPHW)	Find (FINDB, FINDW)
String Mismatch (SKIP)	Translate String (XLAT)
Set String (SETB, SETW)	Copy Bit (MOVBIT)
Find Bit (SCANBIT)	Time Delay (TIME)
Lock Set (LOCKSET)	Interrupt ENABLE, DISABLE
CAUSE$INTERRUPT	HALT
CARRY, SIGN, ZERO, PARITY	PLUS, MINUS
Decimal Adjust (DEC)	STACKPTR, STACKBASE
INPUT, OUTPUT	SET$REAL$MODE
GET$REAL$ERROR	WAITFORINTERRUPT

8.3.2 Use of Compiled Libraries

The generic guidelines apply to PL/M. Compiled libraries are routines written and compiled by a group or organization, usually outside and removed from the current development group. Compiled libraries are often sold by third-party providers and are available only in object-code format with detailed calling and usage documentation. For the most part they are documented "black boxes" with their internal methodologies and algorithms hidden. Concerns for such libraries are similar to those for built-in functions.

8.4 Maintainability

Attention given to maintainability issues in program design makes it easier and safer to make changes to the program. These issues reduce the likelihood of errors inadvertently being introduced during the change or upgrade process. Addressing these issues at design time is really an investment in the future robustness of the program.

The following attributes are related to maintainability as it affects safety:

- *Readability.* These are attributes of the software that facilitate the understanding of the software by project personnel.
- *Data Abstraction.* This is the extent to which the code is partitioned and modularized so that the collateral impact and probability of unintended side effects due to software changes are minimized.
- *Functional Cohesiveness.* This is the appropriate allocation of design-level functions to software elements in the code (i.e., one procedure, one function).
- *Malleability.* This is the extent to which areas of potential change are isolated from the rest of the code.
- *Portability.* The major safety impact is the avoidance of nonstandard functions.

These attributes are discussed in detail in the sections below.

8.4.1 Readability

The attribute of good *readability* allows the software to be understood by qualified personnel other than the original author of the code. Readable source code adds to the documentation of the program itself (self-documenting). Studies have shown that manual code reading is more effective than structural testing or functional testing for finding code faults (McGarry, 1992). Therefore, it seems

that good readability will enhance the probability of locating faulty or weak code that could cause failures in operation or problems during maintenance. The following attributes make source code more readable:

- Conformance to indentation guidelines
- Use of descriptive identifier names
- Comments and internal documentation
- Limitations on subprogram size
- Minimizing mixed language programming
- Minimizing obscure or subtle programming constructs
- Minimizing dispersion of related elements
- Minimizing the use of literals.

PL/M aspects of these attributes are discussed below.

8.4.1.1 Conformance to Indentation Guidelines

Appropriate indentation facilitates the identification of declarations, control flows, nonexecutable comments, and other components of source code. Indentation guidelines are generally part of a project specification, organizational style, or standards document. In the paragraphs below, indentation issues, guidelines, and recommendations are discussed as they pertain to PL/M program blocks and control flow blocks.

- *Program blocks.* Program blocks separate sequences of statements. In PL/M, the `DO` and `END` statements define the limits of a program block. In PL/M, program blocks can be nested. Each program block, therefore, provides a natural method of expressing the program logic by indenting. It is recommended that, for clarity and understanding, the program segments and blocks be indented consistently throughout the program.

- *Control flow blocks.* Program control statements of `DO ...WHILE, DO CASE`, iterative `DO`, and `IF...THEN...ELSE` also provide natural indentation segments.

8.4.1.2 Descriptive Identifier Names

The generic guidelines apply. , an identifier is the name of a variable, procedure, symbolic constant, or statement (label). Identifiers can be up to 31 characters long. The first character must be alphabetic, and the remainder may be either alpha or numeric.[11] There is no distinction between upper and lower case letters. The "$" (dollar sign) can be used to improve readability; it is not

[11] This applies to early versions of PL/M such as PLM-80. Later versions also allow the underscore character and either alpha, numeric, or the underscore as the first character.

evaluated by the compiler as an identifier. An identifier containing a dollar sign is equivalent to the same identifier without the dollar sign.

The following are language-specific guidelines:

- *Distinguish procedure and variable names.* Variable names should be distinguished from procedure names by some convention (this can be project-specific). It is often convenient to give a hierarchy number to a module in addition to a name. The hierarchy number is used primarily for documentation purposes and with the prefix/suffix notation. Use of an identifier prefix (or suffix) allows information about the identifier to be attached or carried.

- *Loop variables should be given some standard nomenclature.* As these variables are often local counters and have no other meaning except their local use as a counter or index, programmers may be tempted to choose any nondescript name that comes to mind. A standard nomenclature, as in lines 5 and 6, allows these variables to be identified readily.

- *Label data from an external source.* In general, data that is received from an external source, such as a sensor or data port, should have a name descriptive of that source. `VIBRATION$X, VIBRATION$Y, VIBRATION$Z` is a better descriptive label than `IO$PORT$1, IO$PORT$17,` and `IO$PORT$23`. The declaration of these might be as shown below.

```
DECLARE VIBRATION$X BYTE;    /* X-axis vibration component from Port 01H */
DECLARE VIBRATION$Y BYTE;    /* Y-axis vibration component from Port 017H */
DECLARE VIBRATION$Z BYTE;    /* Z-axis vibration component from Port 023H */
```

- *Avoid reserved words or words similar to existing reserved word.* PL/M, being an older language, does not support features such as *overloading* and *pre-compiled headers.* Reserved words or even identifiers containing reserved words should never be used as identifiers. It is best to give wide berth to identifiers similar to reserved words. These identifiers may become reserved words in the course of the code's lifetime due to compiler changes.

8.4.1.3 Comments and Internal Documentation

Weak or lacking internal program documentation and comments raise safety concerns. Sparse, incomplete, or outdated program comments can impede code review and mislead those performing program modification and maintenance.

Comments are important elements of safety software that should be maintained with each revision of the source code, no matter how minor the change.

Although the concerns with comments in PL/M are essentially generic language ones, the following example may be helpful to reviewers in judging the adequacy of comments in the target of their review This example shows basic information about the module as well as where additional information can be found. Note how the comments indicate that the outline of the software documentation has been designed and space has been allocated in section 4.2.2 for detailed documentation of this module.

```
RANGING$LASER: DO;      /* Module */
/*****************************************************************/
/*  Module 4.2: RANGING$LASER
/*  Revision #: 2.2
/*  Revision Date: December 12, 1993
/*  Revised by: Sally Newprogrammer, Approved by: Sarah Boss
/*
/*  Function: This module contains all of the software functions
/*            necessary to initialize, aim, arm, and fire the
/*            main system ranging laser unit. All routines, data,
/*            and declarations necessary to operate the laser are
/*            contained in this module.
/*
/*  Documentation: This module is documented in further detail in
/*                 section 4.2.2 of "ABC Systems Software Manual"
/*                 3-100422 Rev C (December 1993)
/*
/*  Include Files: File LASER.EQU should be included in any
/*                 external module which uses the procedures
/*                 contained within.
/*
/*  Associated Hardware: Apex 150 Ranging Laser #43-4568-01A
/*
/*  Module author: John C. Programmer
/*  Original Date: January 14, 1983
/*****************************************************************/
     .... statements ...

END;  /* End of Module RANGING$LASER  */
```

In the above example, the complete module has been encapsulated; therefore the only outside references are contained in the include file named "LASER.EQU." Other modules may not be so self -contained and may require other types of header information. For instance, utility subroutines or procedures are often used many places in a program. Routines such as BCDTOBINARY, DISPLAY$TIME, etc. often have a "WHERE USED:" comment section in their header block.

The following example illustrates a comment header block for procedures. The function is described narratively. The inputs are described in real measure units. The range of valid arguments is also

shown. Since this is a utility subroutine, the locations where it is used throughout the program are shown.

```
/*************************************************************/
/*  Procedure: AIM$LASER (X, Y, Z) BYTE PUBLIC;
/*  Revision Date: December 1, 1992
/*  Revised by: Sally Newprogrammer, Approved by: Sarah Boss
/*
/*  Function: This procedure physically aims the laser unit base
/*            on coordinate input information X, Y, and Z. Servo
/*            information is calculated, and the servos activated
/*            by calling private procedure SET$SERVO located in
/*            this module. If the status return for the servo
/*            operation is OK, a TRUE indication is returned to
/*            the Calling program.
/*
/*  Inputs: Coordinates are in units of millimeters passed as real values.
/*          Precision must be to three decimal places. Valid ranges are
/*          as follows:
/*                       X:  0.000 .. 100.000
/*                       Y:  0.000 ..  24.750
/*                       Z:  0.000 ..  75.000
/*
/*  Where used: INIT.PLM: INIT$LASER
/*              MAIN.PLM: GET$RANGE, DEACTIVATE$LASER
/*              TEST.PLM: TEST$1, TEST$5, TEST$19
/*
/*  Documentation: Section 8.2.9 of "ABC Systems Software Manual"
/*                 3-100422 Rev C (December 1993)
/*
/*  Module author: John C. Programmer
/*  Original Date: January 14, 1983
/*************************************************************/

    AIM$LASER: PROCEDURE (X, Y, Z) BYTE PUBLIC;
        DECLARE (SX, SY, SZ, STATUS) BYTE;
        DECLARE (X,Y,Z) REAL;

        SX = SET$SERVO (CHANNEL$1, X);  /* Return status of servo move */

        SY = SET$SERVO (CHANNEL$2, Y);
        SZ = SET$SERVO (CHANNEL$3, Z);
        /* ...other statements... */
        RETURN (STATUS);                /* Combined status of servos */
    END;

END AIM$LASER;  /* End of AIM$LASER Procedure */
```

Other items that might be included in comment header blocks and in line comment blocks include the following:

- Performance requirements for the procedure
- Unusual external interfaces and associated information
- Error handling and exception behavior and related information
- Inputs and outputs of the module and their range of values
- References to appropriate design documentation and charts
- Purpose and expected results of blocks of in-line code
- Expected results at branching junctures within a code segment
- Expected actions and results of exception code
- Detailed in-line comments explaining unusual constructs and deviations from normal program practices.

8.4.1.4 Limitations on Subprogram Size

Only generic guidelines apply.

8.4.1.5 Minimizing Mixed Language Programming

The generic guidelines apply. Generally speaking, mixing programming languages is a source of error because of different calling conventions, register usage, and data representations. None of the Intel PL/M languages support in-line assembly language coding.

However, mixed language coding and linking is sometimes necessary. When functions must be developed in a second language, they should be isolated and designed as loosely coupled as possible. If at all possible, parameters should be passed to the routine rather than accessed as a global entity.

Where separate assembly code must be used, macros should be defined to hide calling convention details.

8.4.1.6 Minimizing Obscure or Subtle Programming Constructs

The generic guidelines apply. Obscure or subtle coding techniques should be avoided if at all possible. If they cannot be avoided and justification for their use exists, they should be isolated and well commented. An example follows:

```
/*----NON-STANDARD CODE FOLLOWS-----------------------------------*/
/* The following code is used to increase performance by using    */
/* a left shift by 3 to replace a multiply by 8.                  */
/*----------------------------------------------------------------*/

OPERAND1 = SHL (OPERAND1, 3);   /* OPERAND1 = OPERAND1 * 8        */

/*----End of Non-Standard Code Section----------------------------*
```

In this example, the code is clearly marked as nonstandard code. The surrounding comments describe exactly what the code is attempting to accomplish. The end of the code block is also clearly marked.

8.4.1.7 Minimizing Dispersion of Related Elements

When related elements of code are dispersed in a program, it is necessary to refer to multiple locations within the source listings during reviews and maintenance. Review is facilitated and safety is enhanced if project-specific guidance is provided on the placement of related elements in the code. Since the PL/M language is not complex, most cases of code dispersion occur with the use of the DECLARE statement and general utility procedures.

- *Control dispersion of DECLARE statements.* The DECLARE ... LITERALLY statement is often used to give more meaningful names to numeric constants. These descriptive names are then used throughout the program to enhance readability. Therefore, they should be placed in a source-code file to be included in all program modules. All of these values are then localized to one file making them easier to change. Compiler directives can then be set as desired in each module, either to print or not to print the contents of this include file.

  Similarly, the DECLARE ... EXTERNAL statement is used to declare a data type (and length) for a variable or constant declared to be PUBLIC elsewhere. For procedures which are dispersed throughout the program — such as those called from the main program — a separate file of external declarations should be maintained and included in files as needed. Some degree of control over these dispersed elements is thus maintained. An exception to this is discussed in the paragraph below.

- *Dispersion of general utility procedures.* Procedures that are general to the program and used throughout to provide some minor function are referred to as general utility procedures. These procedures are similar in nature to the built-in functions. General utility procedures should be grouped together in one or more modules. For code review or maintenance purposes, all of these routines will then be conveniently located in one listing. As a further convenience in identifying these general subroutines, they may be prefixed with a lower case character as in: `u$BCD$TO$BINARY`, or `s$MULT$32` (see also Section 8.4.1.2).

  The general utilities module(s) should maintain an `$INCLUDE` file of external declarations for these publicly declared routines. This file should be included in any module that calls or invokes any of these general procedures. Thus, dispersion of these declarations is localized to one source-file module.

- *Use of header files for imports and exports.* Header files should be used to group module exports. Imports should only use header files.

In summary, code element dispersion should be minimized where possible by proper grouping and use of included files. These `$INCLUDE` files should have adequate header comment documentation describing the purpose of the include file and where each element is used.

8.4.1.8 Minimizing the Use of Literals

The generic guidelines apply. Use of literals in the PL/M source code impacts safety because it decreases readability and complicates the maintainability of code. Use of literals often causes different representations of the same value to be dispersed throughout one or more program modules. It is far easier to change one set of values located at the beginning of a file, or included with the file with an `$INCLUDE` statement, than to guarantee that all literal values associated with an item have been successfully located and properly changed.

Literals are often used by programmers because they show an actual value which is easier to use during debug time. This often occurs when a certain bit pattern must be passed to a hardware port to accomplish some I/O task, such as turning an LED indicator on or off. This code may be convenient for a brief time while hardware and software team members debug a hardware unit. This convenience is short lived, however, as the following two examples illustrate.

The first example below shows a section of code that is intended to turn on an LED indicator and later turn it off. During a coding session, it is relatively easy for a programmer to glean information from an electrical schematic diagram quickly, then directly code this information into the program. Suppose later that some change has been made to the hardware requiring all of the code associated with this LED to be modified. Using a text editor search for "OUTPUT(3)" would not find the second occurrence, which is coded as "OUTPUT(03H)."

```
OUTPUT(3) = 00000100B;     /* Turn power LED on  */
    ....                     ....
OUTPUT(03H) = 0FBH;        /* Turn power LED off */
```

The next example shows a better method of handling the above situation with literals. PL/M has a DECLARE...LITERALLY statement that allows literals to be assigned to a label. In this example all literal data are grouped together in one place, and all of the commands and data associated with that I/O device are defined. Should a change be made later to the hardware system, all of the necessary software changes can be accomplished by changing just three DECLARE...LITERALLY lines of code.

```
/* Commands and data for Power LED device */
DECLARE PWR$LED LITERALLY '03H';
DECLARE LED$ON  LITERALLY '04H';
DECLARE LED$OFF LITERALLY 'NOT LED$ON';
    ....
    ....
OUTPUT(PWR$LED) = LED$ON;
    ....
    ....
    ....
OUTPUT(PWR$LED) = LED$OFF;
```

In addition, the code is more readable and somewhat self-documenting. In larger programs, the declaration of these literals would probably occur in a file that would be included with the INCLUDE compiler control statement. The sequence for the example above might appear as follows:

```
$INCLUDE (IODEFS.PLM) /* Commands and data for Power LED device
*/
    ....
    ....
OUTPUT(PWR$LED) = LED$ON;
    ....
    ....
    ....
OUTPUT(PWR$LED) = LED$OFF;
```

Literals that are exported by a module should be grouped in the module's header file.

8.4.2 Data Abstraction

Data abstraction involves combining both the data and the allowable operation on that data (procedures or functions) into a single entity. Furthermore, data abstraction calls for the establishment of an interface that allows access to, manipulation of, and storage of the data only through allowable operations. Data abstraction is an important contributor to safety in that it reduces or eliminates the side effects of variables being changed inappropriately during run time or inadvertently or incorrectly changed during software maintenance.

The PL/M language pre-dates the current concepts of data abstraction. Hence, PL/M does not have any built-in mechanisms for implementing data abstraction directly. However, it will also be shown that the PL/M program module can provide an appropriate and acceptable container for data abstraction as discussed in Section 8.4.3.

8.4.2.1 Minimizing the Use of Global Variables

The generic guidelines apply. It is desirable to limit the scope of variables in safety-related programs. Variables that are made available to all program segments increase the potential for unintended side effects. However, global variables may be the simplest way to represent some sort of global state or other data that must be accessed by most or all functions. The alternative is to pass the variable as a parameter, which increases the complexity of the procedure and function interfaces. Global variables may also be necessary to share data from separately compiled modules.

The following are specific guidelines related to global variables.

- *Initialization of global variables.* All global variables used in a program should be initialized in exactly one place.

- *Imports and exports from separately compiled modules.* All exports from a module should be explicitly global, and everything else should be made local to the module by being explicitly declared static. Exports from a particular module should be specified in one and only one header file. All importing modules should use the header file. They should not import variables, functions or procedures independently from the header file by using externals. Headers should use prototypes unless there is a good reason not to, in which case, the reason should be documented.

- *Use macros for local variables in emulators, simulators, and debuggers.* In-circuit emulator (ICE) tools, debuggers, and simulators complicate use of local variables because of the length of their identifiers. One such emulator, the Intel I²ICE system, uses a naming convention as follows:

```
[:module.-name.][procedure-name.][variable-name][expr[,expr]]
```

However, it is also possible to construct a temporary macro which would reference this variable with just one or two characters while debugging this code section.

8.4.2.2 *Minimizing Interface Complexity*

The generic guidelines apply. Interfaces between procedures, functions, and program modules are often a source of software failures. If an interface becomes too complicated, it will be difficult to review, understand, and maintain. Complex interfaces are not desirable in a safety-related program and should be avoided. Specific guidelines include the following:

- *Limit the number of arguments used in the calling program.* Requirements for a large number of arguments can cause confusion and errors in a safety-related program. If a programmer must set up a large number of parameters to invoke a procedure, some of the choices may not be properly thought out. It is better to have a programmer understand the meaning of the parameters than to require that they be blindly and rotely specified.

  Procedures that require a number of arguments may be indicative of a design in which excessive functionality has been allocated. A better design may be two or more smaller procedures, each of which accomplishes a narrower task. The example in the section on data abstraction illustrates this point by showing how one or more method procedures allows a user to understand more clearly how laser ranging data are obtained. This method requires the programmer to think through how the instrument obtains ranging data.

- *Do not use ambiguous or terse expressions.* Use of meaningless expressions for modes or options can confuse the programmer. Both of the example procedure invocations below will accomplish the same results. However, the second form is better because it immediately provides information on the parameters. A person reading and checking code is more likely to question the correctness of a parameter choice in the second invocation than in the first.

```
(1)   CALL FIRE$LASER (2, 3.0, 1000);

(2)   CALL FIRE$LASER (CHANNEL$1, MSEC$3, ONE$WATT);
```

- *Explicitly state restrictions and limitations.* Lack of easily understood restrictions and limitations on the use of allowable operations can also complicate an interface. The above example can be expanded to remove ambiguities about parameter usage and limitations. In the following example, a table of valid parameter settings for invoking the `FIRE$LASER` procedure is provided. In this example, we assume that the laser manufacturer only recommends these settings for this model. By declaring a list of valid settings, an improper invocation of the procedure is less likely.

```
/* VALID PARAMETER SETTINGS FOR THIS LASER UNIT */

/* There are only 3 Laser channels defined for this instrument */
DEFINE CHANNEL$1 LITERALLY '1';
DEFINE CHANNEL$2 LITERALLY '2';
DEFINE CHANNEL$3 LITERALLY '3';

/* There are 5 power settings defined for this instrument */
DEFINE ZERO$WATT LITERALLY '0';
DEFINE QUARTER$WATT LITERALLY '249';
DEFINE HALF$WATT LITERALLY '502';
DEFINE THREE$QTR$WATT LITERALLY '754';
DEFINE ONE$WATT LITERALLY '998';

/* The pulse width should always be set to 3 milliseconds */
DEFINE MSEC$3 LITERALLY '2998';
```

8.4.2.3 Use Modules to Facilitate Data Abstraction

PL/M modules can be used to enhance maintainability through limiting data visibility and achieving a measure of data abstraction in PL/M. The following example of a laser ranging instrument demonstrates this concept. To use the laser ranging instrument, the calling program need only turn on the instrument, aim the instrument through an allowable range, and activate and receive the range data. The methods used to obtain the range data are hidden from the calling program. The calling program cannot misuse the instrument by tinkering with the laser power levels and pulse durations. In addition, the calling program can aim the laser unit only through a valid domain of coordinates.

The laser functions are collected in a separate source module. In doing so, the procedures that are public and available to the code outside of this module are controlled. Procedures not declared EXTERNAL will remain hidden and private to this module. No other code except the laser control code will be placed in this source module. In the example below, lines 3 through 11 define the current constant parameters for the laser instrument. If these values change in the future, due to hardware modifications, they can be easily modified. Line 12 has local variables that contain the current power settings and pulse duration times for the instrument. The variables are local to this module and cannot be "seen" or used by other routines outside this module, that is, these variables are hidden or encapsulated within this module.

Lines 13 through 21 in the example contain two support procedures that are used only by the procedures contained within this source module. The two procedures FIRE$LASER and SET$SERVO

```
   1          RANGING$LASER: DO;       /*  Module */

   2          /*------ Private Procedures & Data ----------*/

   3   1          DECLARE ZERO$WATT LITERALLY '5';   /* Zero watt = 5 counts  */
   4   1          DECLARE ONE$WATT LITERALLY '123';  /* One watt = 123 counts  */
   5   1          DECLARE MSEC$0 LITERALLY '0';      /* 0 milliSec */
   6   1          DECLARE MSEC$3 LITERALLY '3000';   /* 3000 uSec = 3 milliSec */
   7   1          DECLARE ON LITERALLY '0FFH';
   8   1          DECLARE OFF LITERALLY '00H';
   9   1          DECLARE T1 LITERALLY '23H';
  10   1          DECLARE T2 LITERALLY '41H';
  11   1          DECLARE T3 LITERALLY '84H';

  12   1          DECLARE (L$POWER, L$DURATION) REAL;

  13   1          SET$SERVO: PROCEDURE (CHAN, AMOUNT) BYTE;
  14   2            DECLARE AMOUNT REAL;
  15   2            DECLARE (CHAN, STATUS) BYTE;
                     /*   ...other statements...    */
                     /* check for valid coordinates */
  16   2            RETURN (STATUS);
  17   2          END; /* SET$SERVO */

  18   1          FIRE$LASER: PROCEDURE BYTE;
  19   2            DECLARE STATUS BYTE;
                     /* ...other statements... */
  20   2            RETURN (STATUS);
  21   2          END;

                  /*------ Public Procedures & Data -----------*/

  22   1          OPERATE$LASER: PROCEDURE (ON$OFF) PUBLIC;
  23   2              DECLARE ON$OFF BYTE;

  24   2              IF (ON$OFF = ON) THEN
  25   2                  DO;
  26   3                      L$POWER = ONE$WATT;
  27   3                      L$DURATION = MSEC$3;
  28   3                  END;
                      ELSE
  29   2                  DO;
  30   3                      L$POWER = ZERO$WATT;
  31   3                      L$DURATION = MSEC$0;
  32   3                  END;
                  /* other statements */
  33   2          END;

  34   1          AIM$LASER: PROCEDURE (X, Y, Z) BYTE PUBLIC;
  35   2              DECLARE (SX, SY, SZ, STATUS) BYTE;
  36   2              DECLARE (X,Y,Z) REAL;

  37   2              SX = SET$SERVO (1, X);
  38   2              SY = SET$SERVO (2, Y);
  39   2              SZ = SET$SERVO (3, Z);
                        /* ...other statements... */
  40   2              RETURN (STATUS);
  41   2          END;

  42   1          GET$RANGE: PROCEDURE REAL PUBLIC;
  43   2              DECLARE RANGE REAL;

  44   2              IF (FIRE$LASER) THEN
  45   2                  DO;
                            /* ...Calculate RANGE... */
  46   3                  END;
  47   2              ELSE RANGE = 0;  /* Error */
  48   2              RETURN (RANGE);
  49   2          END;

  50   1      EXCEPTION$LASER: PROCEDURE EXTERNAL;
               /* ...handle laser exception here... */
  51   1      END EXCEPTION$LASER;

  52   1      END;  /* End of Ranging$Laser Module */

END OF PL/M COMPILATION
```

are hidden from other code outside this module and are thus protected from being used by other code outside this module. Thus the laser can neither be aimed in an inappropriate direction nor

inadvertently fired. Lines 22 through 51 of the example shown are public procedures. These are the methods available to code outside this module that allow the data to be properly manipulated and the laser instrument to be used safely.

Thus module RANGING$LASER is the closest we can come to generating a software object in PL/M. We have forced a procedural language in a disciplined manner to behave like and produce some of the benefits of, an object-oriented language. The short main program in the following example demonstrates how this object will work. The main program is defined on line 1 of the example. Lines 2 through 11 declare and define the external procedures located publicly within module RANGING$LASER. These are the only procedures (methods) available to the main program to manipulate and operate the laser instrument. Lines 14 and 15 define the meaning of ON and OFF commands to the laser. These could be placed in a common INCLUDE file in a larger program. Line 16 defines a set of directional coordinates for the laser, and line 17 is a variable to contain the distance data received from the instrument.

The laser can now be properly manipulated. It can acquire range data safely by using the code in lines 18 through 24. Simply, if the coordinates of the target are valid, as determined by method AIM$LASER returning TRUE, the code within the IF...THEN clause will execute, turn the laser ON, fire the laser and obtain range data, and turn the laser unit OFF. If the coordinates are invalid, the error exception handler EXCEPTION$LASER is called to correct, notify, or otherwise handle the erroneous situation.

The code in this main program has no way of inadvertently changing the laser power levels and pulse duration times.

```
PL/M COMPILATION OF MODULE MAINPROGRAM

   1           MAIN$PROGRAM: DO;      /* Main Program Module */

   2           /*-------- Declare External Procedures --------*/

   3    1      OPERATE$LASER: PROCEDURE (ON$OFF) EXTERNAL
   4    2          DECLARE ON$OFF BYTE;
   5    2      END OPERATE$LASER;

   6    1      AIM$LASER: PROCEDURE (X, Y, Z) BYTE EXTERNAL;
   7    2          DECLARE (X,Y,Z) REAL;
   8    2      END AIM$LASER;

   9    1      GET$RANGE: PROCEDURE REAL EXTERNAL;
  10    2          DECLARE RANGE REAL;
  11    2      END GET$RANGE;

  12    1      EXCEPTION$LASER: PROCEDURE EXTERNAL;
  13    2      END EXCEPTION$LASER;

               /*-------- Main Program Segment --------*/

  14    1          DECLARE ON LITERALLY '0FFH';
  15    1          DECLARE OFF LITERALLY '00H';
  16    1          DECLARE (X1, Y1, Z1) REAL INITIAL (4.1, 5.7, -6.1);
  17    1          DECLARE DISTANCE REAL;

  18    1          IF (AIM$LASER (X1, Y1, Z1)) THEN
  19    1             DO;
  20    2               CALL OPERATE$LASER (ON);     /* Turn on laser unit
*/
  21    2               DISTANCE = GET$RANGE;        /* Fire & get range value
*/
  22    2               CALL OPERATE$LASER (OFF);    /* Turn off laser unit
*/
  23    2             END;
  24    1          ELSE CALL EXCEPTION$LASER;       /* or handle exception
*/

  25    1      END;  /* End of Main Program Module */

END OF PL/M COMPILATION
```

8.4.3 Functional Cohesiveness

There should be a clear correspondence between the function of a program and the structure of its components. Review and maintenance of program codes are facilitated when every function is implemented in a procedure and when that procedure implements only one function.

As a guideline for using PL/M in safety-oriented systems, it is further recommended that program modules contain only procedures of like functions. Each PL/M module can limit the scope of variables and procedures within that module. The following is an example of a recommended structure:

```
MODULE$1: DO;
   /* Global Declarations for MODULE$1 */
   PROCEDURE$1A:
   END;
   PROCEDURE$1B:
   END;
END;

MODULE$2: DO;
   /* Global Declarations for MODULE$2 */
   PROCEDURE$2A:
   END;
   PROCEDURE$2B:
   END;
END;
```

Each module above can contain one or more related functions or methods. The scope of the variables and procedures defined in each module is limited to that module unless it is explicitly defined as `PUBLIC`. Therefore, each PL/M module can *cohesively* contain related procedures and variables, and it can make available to functions outside of this module only those entities that are explicitly declared as `PUBLIC`. This concept is also discussed in the section on data abstraction.

8.4.4 Malleability

Malleability is a measure of the ease with which a software system can accommodate changes in its function. Malleability depends upon data abstraction, encapsulation, and cohesiveness built into the program. It extends those attributes in order to isolate and identify areas of potential change. Most of these issues have already been discussed. Two topics that may be of interest to reviewers are covered below.

8.4.4.1 Isolation of Alterable Functions

PL/M functions that are likely to be altered should be placed in separate `DO;-END;` modules within the source code file to which they belong. Potentially alterable functions should, in most cases, remain in the same module with related functions and code. Attempts to place all potentially alterable functions in one file may result in a collection of unrelated procedures that only have

alterability in common. Such attempts may destroy the cohesiveness and data abstraction attributes designed into the system. Functions likely to be altered should be isolated and marked as such with comments within the module in which they were designed.

8.4.4.2 Isolation of Hardware-Specific Functions

Another area of possible change and alterability in embedded systems is hardware-specific functions, such as those specific to a peripheral device or a model of an attached instrument. If, during maintenance, a different or upgraded peripheral device replaces an existing device, the change over will be easier and safer if the code is localized to a subset of modules or functions.

It is recommended that code for these peripheral devices be written in the form of device drivers, and that they be loosely coupled to the remainder of the system. The associated CALLs to these device drivers should remain transparent so that the calling code is not impacted by a change in the device driver code.

8.4.5 Portability

The benefits of portability are that programming constructs yield predictable and consistent results across different operating platforms. Thus, code that is to be reused or converted to run on a different platform will be easier to maintain. Attributes related to portability discussed elsewhere in this report include the following:

- Minimizing the use of built-in functions
- Minimizing the use of compiled libraries
- Minimizing dynamic binding
- Minimize tasking
- Minimize asynchronous constructs such as interrupts.

PL/M code is processor specific, and thus has inherently limited portability. Also, it is an obsolescent language, and any new applications should plan for migration to another language (see Appendix A.4).

References

U.S. Department of Defense, DoD Std 2167A, *Software Development Standard*, Appendix D, 1986.

Institute of Electrical and Electronics Engineers, IEEE-Std-7-4.3.2-1993, Appendix F, *IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations.*

Intel Corporation, *PL/M Programming Manual*, 9800268B, Chandler, AZ, 1977.

Intel Corporation, *PL/M-86 Programming Manual*, 9800466-02B, 1980.

Intel Corporation, *8086 Software Tool Box, Volume II*, 122310-001, December 1984.

Intel Corporation, *PL/M-86 User's Guide*, 121636-004, August 1985.

Intel Corporation, *8086 Software Tool Box*, 122203-002, January 1985.

Intel Corporation, *PL/M-96 User's Guide for DOS Systems*, 481644-001, December 1988.

Intel Corporation, *PL/M Programmer's Guide*, pg 5-34, 452161-002, May 1990.

Intel Corporation, *PL/M-386 Programmer's Guide*, 611052-001, 1992.

Kopetz, H., "Real Time Systems," In *Software Engineer's Reference Book*, J.D. McDermid, ed., CRC Press, Inc., Cleveland, OH, 1993.

Liao, Y., "Requirements for Directed Automatic Instrumentation Generation for Program Monitoring and Measuring," In *IEEE Transactions on Software Engineering*, 1991.

Leveson, N.G, and C.S. Turner, *An Investigation of the Therac-25 Accidents*, University of California, Irvine Technical Report 92-108, Irvine, CA, 1992.

McGarry, F., "The Impacts of Software Engineering," briefing presented to the NRC Advisory Committee on Reactor Safeguards (ACRS), August 21, 1992.

Meek, B.L.,"Early High-level Languages," In *Software Engineer's Reference Book*, J.D. McDermid, ed., CRC Press Inc., Cleveland, OH, 1993.

9 Ada 95

This chapter discusses Ada 95-specific guidelines. The experience with Ada 95 is somewhat limited so the guidelines in this chapter represent a merging of Ada 95 guidelines which have evolved thus far with Ada 83 guidelines which should still apply. The Ada 95 information is based, in large part, upon a report (Saaltink, 1996] of an analysis performed for the Canadian Department of National Defence. The Canadian report was intended for the guidance of developers, not reviewers or other personnel involved in the certification or approval of high integrity systems. The recommendations and language of the Canadian report reflect this viewpoint. Since this is not the viewpoint of the current report, the language and recommendations of the Canadian report have been adapted to the current report's needs.

As pointed out previously (in the chapter dealing with Ada 83), there is a question about the limited number of Ada 95 compilers, and whether or not any of them is sufficiently mature to be used in safety-critical applications[12]. Since there is a considerable Ada 83 legacy in Ada 95, many of the guidelines presented earlier for Ada 83 apply equally to Ada 95.

Section 9.1 identifies reliability-related attributes; Section 9.2 discusses robustness-related attributes; Section 9.3 discusses traceability-related attributes; and Section 9.4 describes maintainability-related attributes. [A summary matrix is contained in Appendix B, together with language-specific weighting factors. These factors were influenced by Ada's strong typing and exception handling capabilities.] Because of the number of guidelines in this chapter, the guidelines for each intermediate attribute are summarized in a box at the end of the section discussing that attribute.

9.1 Reliability

The intermediate attributes of reliability related to Ada are as follows:

- predictability of memory utilization
- predictability of control flow
- predictability of timing
- predictability of mathematical or logical result

Ada-specific guidelines are described in the following subsections.

---

[12]Ada 95 differs from Ada 83 in several major areas, making Ada 95 potentially more suitable over the long term for developing safety-critical systems. The most important improvements are (a) providing object-oriented features, (b) new features for more responsive task communication such as protected types for emulating the monitor structure, and (c) hierarchical library structuring. Where appropriate in the text, references have been made to some of the differences between Ada 83 and Ada 95 which affect safety.

9.1.1 Predictability of Memory Utilization

Base-level attributes related to the predictability of memory utilization in Ada are as follows:

- minimizing dynamic memory utilization
- minimizing memory paging and swapping

Specific guidelines for these attributes are discussed in the following subsections. These guidelines show that dynamic memory utilization may be invoked in many subtle ways in Ada95.

9.1.1.1 Avoiding Dynamic Memory Utilization

The generic[13] guidelines apply to Ada. Dynamic memory allocation should be avoided. Errors resulting from dynamic memory allocation can include (SPC, 1989, pp 76, 112 - 113):

1. Memory leaks that can cause the software to run out of memory. This problem is very likely to occur in Ada since an access object (pointer) ceases to exist when its scope is exited, but the allocated memory it points to remains allocated.

2. Corruption of data due to multiple pointers to the same areas. Such corruption can be difficult or impossible to correct or even detect. This error condition can lead to the system crashing, frequently due to an exception being raised at a point distant from where the data were corrupted. This makes tracing the cause of the crash very difficult.

The following are Ada-specific guidelines related to memory allocation. The final four guidelines are mitigation approaches and are relevant if dynamic memory allocation is determined to be unavoidable by the system designers. Also discussed at the end of this section are allocators and other approaches serving to manage dynamic memory when the developer believes dynamic memory is required. This discussion is intended to assist the reviewer in assessing the safety of the developer's approach.

- *Avoid explicit dynamic memory allocation.* The Ada primitive `new` causes memory to be allocated during execution. The following Ada code is an example of the use of dynamic memory for a linked list:

```
type Cell;
type Link is access Cell;
type Cell is
      record
            Value: Element;
```

[13]It should be noted that "generic guidelines" refers to the non-language specific guidelines of Chapter 2, not to the Ada construct "generic".

```
                    Next : Link;
                end record;

        L: Link := null;                        -- initialization unnecessary
        L:= new Cell;                           -- allocation of memory
```

- *Avoid dynamically created tasks.* Tasks should be elaborated only at system initialization. Dynamically created tasks also cause dynamic memory allocation in Ada. The dynamic memory utilization problem is aggravated in this case because the generic subprogram the programmer might have utilized to deallocate objects in memory, `Unchecked_Deallocation`, does not apply to tasks or to objects that have tasks as components. This issue of dynamic tasks is discussed further in Section 3.

- *Avoid recursion.* Recursion also uses dynamic memory space. Therefore, recursive procedures or functions should not be used. Recursion depth can be very large, even infinite if the terminating condition does not occur. An unanticipated large number of recursive calls can use up available memory (SPC, 1989; Hutcheon, 1992). Recursion can frequently be recognized by having a subprogram call within a subprogram of the same name, as seen in the following example.

```
procedure RECURS_EXAMPLE(arg1: in type1 arg2: in type2) is
      arg1a:  type1;
      arg2a:  type2;
begin
      sequence of statements
      RECURS_EXAMPLE(arg1a=>arg1 arg2a => arg2);
      more statements
end RECURS_EXAMPLE;
```

Mutual recursion involving two or more subprograms can also occur. Depending on the arrangement and physical location of the source code for these subprograms, mutual recursion can be difficult to detect from source code. For example:

```
      procedure P(....)  is
      begin
            .....
            Q(....);
            .....
       end P;
```

and

```
procedure Q(....) is
begin
      .....
      P(....);
      .....
end Q;
```

- *Use aliased objects and components subject to restrictions.* Aliased objects are the variables that may be designated by general access types. The type-safe access to variables provides a possible way of constructing "safe" pointer algorithms. There are a number of language facilities that interfere and should be restricted (Saaltink, 1996):

1. Objects depending upon records with discriminants can be initially constrained but have the discriminants inappropriately updated through access variables; hence, they should be avoided.

2. Objects with aliased components can change the representation in ways that make reading or writing through access variables unsafe. Do not use `'aliased` on objects or components of objects that have the following representation specifications applied:

   a. `'size`
   b. `'component_size`
   c. `pragma pack`
   d. `'alignment`

3. Aliased objects that cannot be accessed by general access types can be created because of Ada95's *accessibility rules,* which guarantee that it is impossible to access objects after they are deallocated. Most of these checks can be done at compile time, but interactions with generics mean that dynamic checks are sometimes present. The use of aliased types or access types as formal generic parameters is not recommended, unless it can be shown that these checks will not fail.

4. Access types designating unconstrained subtypes allow the creation of unconstrained views of constrained objects. Operations on such views, such as assignment and comparison, may require unpredictable temporary storage in some implementations. This capability has to be used with caution.

To make aliased variables and general access types safe for use, it is necessary to prohibit discriminants for all but limited objects, ensure that all aliased objects are at the outermost accessibility levels, avoid generics, and prohibit representation clauses on any type or object that has aliased components. There are also restrictions on the variables that may designate aliased objects. See Section 5.3.10.3 of (Saaltink, 1996).

- *Avoid/restrict the use of access types.* Ada95 provides three kinds of access types: *pool specific* access types, that provide access to dynamic data created and manipulated by programs; *general* access types that can also designate aliased variables and components of variables; and *access-to-subprogram* types whose values designate subprograms. Pool-specific and access-to-subprogram access types should not be used in high-integrity systems and general access types require some care in their use (Saaltink, 1996). Variables of access types (also known as pointers in other languages) have a type safety in Ada95 that is superior to that found in most other languages, but there are still sufficient problems with both access types that their use in critical or secure software is not recommended. Also, caution is advised when access types are used in arrays or records because Ada95 specifies a default initial value of `null` for access types.

  Pool-specific access types should not be used in high integrity systems. This prohibition is based on the unsafeness of dynamic memory techniques, and possible erroneous behavior under some storage pool allocation schemes. There is a possibility that type-specific storage pools (created by the `for Access_Type'Storage_Size use XXX` construct) are viable in some implementations, but the implementer should ensure that there is no risk of pool exhaustion or fragmentation.

  Ada95's generalized access types (those declared with the `access all` construct) can access constants and variables as well as allocated objects, and in this environment provide a type-safe and accessibility-safe way of accessing data objects that cannot be replicated. Some anomalies exist in the interaction with discriminated types, private types, and aliased components of arrays or records that have been declared with representation clauses. Since these other structures are prohibited for high integrity systems, general access types are usable in this constrained environment. If used, it should be with caution, because they adversely affect data-use analysis, and accessibility checks may be dynamic.

  Generalized access types may be used with caution if the objects and aliased objects are declared in the bodies of the same scope as the type of aliased object to which they can refer. The passing of the access value, or of an address that represents the access value, outside of the declaring package body, is not permitted. These restrictions are designed to make data-use analysis of accessed objects tractable, and to eliminate security data access violations.

  Access-to-subprogram types should not be used in high integrity systems. This is based upon the intractability of analysis for this paradigm. In addition, this paradigm has less compile-time support than either tagged types or generic instantiations, and therefore may be more error-prone in actual usage. Also, formal access parameters of subprograms should not be used.

  There are three alternatives to the use of access-types and dynamic storage pools.

1. Variables and subprogram parameters can be used to implement algorithms. Formal parameters are rarely copied on call or return, unless type conversions also happen as part of the call or return.

2. Create an array of the base type and use the index as the access variable. This method is safer than generalized access types because the access variable (index) can be kept or used by units that have no visibility to the storage pool (array), but these units should call subprograms that have visibility to the storage pool to actually manipulate the data element.

3. Create an array of aliased components and use `access all` types (generalized access types) to manipulate the data. This method permits the direct usage of pointer algorithms, but also permits access to invalid objects or null values. These situations can be checked with the `'Valid` attribute, so pointer manipulation using these constructs is permissible, but not recommended.

- *Avoid user-defined finalization.* User-defined finalization relies upon extending the tagged type `Ada.Finalization.Controlled`. Since they are tagged types, all controlled types suffer from the difficulties described in Section 5.3.9 of (Saaltink, 1996). The requirement to finalize controlled types makes the closing of any master scope very difficult to analyze. This is because the order of execution of finalization code is not specified tightly, and the order of execution and time to complete all finalizations are bounded but not exact.

  System-level finalization happens whenever a scope is left. It consists of operations such as removal of stack frames, return of dynamic memory acquired by the run-time system, return of storage pools declared in this scope, removal of exception handlers for this frame, termination of any tasks dependent upon this scope, and finalization of any controlled objects. Adherence to the guidelines in this document should eliminate any operation that will cause unpredictable system-level finalization.

- *Do not instantiate generic units during runtime.* If generic units are used, they should be instantiated only during initialization (Jones, 1988). However, as will be described in the section on traceability (Section 9.3.3), generic units are not desirable in safety significant software.

- *Minimize use of local large composite objects.* A memory allocation problem on the stack can occur if large composite objects are declared as local objects of a subprogram. Avoid the use of dynamic arrays as in `P(array(<>) of ...)`.

- *The `others` clause for aggregates (see Section 9.1.4.1) should not be used.* Any expression appearing in a `discrete_choice_list` in an aggregate should be static, so that the size of the aggregate object can be statically determined.

An aggregate may occur only in a context that imposes an "applicable index constraint" (see ARM 4.3.3(10)) that can be statically determined, to avoid dynamically sized, temporary storage. It is recommended that the implementation of aggregates be inspected to ensure that no dynamic storage is used for aggregates.

- *Minimize use of unconstrained types.* Unconstrained types such as record types with unconstrained dynamic bound, and string types should be used with caution because of the impact on memory allocation. Other restrictions (Saaltink, 1996) on the use of unconstrained and indefinite objects are :

1. An object of a discriminated type with default discriminants takes the initial (default) discriminant values, but this may be changed by assignment at any time. Discriminated types with default discriminants should not be used in high integrity systems.

2. Objects of unconstrained string type with static bounds that are constrained by an `initial` value (i.e., assigned to a static string as part of the declaration) are, however, acceptable.

3. Other objects of unconstrained array types that are constrained by the initial value should not be used unless it can be shown that the initial value is static, and that the implementation never uses dynamic memory techniques in accessing this object.

4. Parameters of unconstrained types in subprogram calls are acceptable, since all actual references to the parameter become references to the original objects that should be constrained (because no unconstrained objects may be declared). Some caution is advised, however, because an implementation might use dynamically allocated temporary objects for some operations.

5. Unconstrained and indefinite function return types should not be used, since no constrained object exists to determine bounds before the return value is created, which means that dynamic memory techniques are probably used. This includes class-wide function return types.

6. For high integrity systems, the declaration of objects of class-wide types should not be used, since an object of a class-wide type may be initialized by any extension of the type of the object. Such an object should be constrained by an initial value, which in the general language can be determined only dynamically. The use of dynamic storage management is likely.

7. Parameters of class-wide types should be used with caution. In contrast to parameters of unconstrained subtypes, there is no simple way to determine the underlying type of the actual object that the parameter represents. Furthermore, some operations on the parameter may require dynamically allocated memory. Some further issues are discussed in Section 5.3.3.4.2 of (Saaltink, 1996).

Alternative constructs may be seen by considering the following procedure:

```
procedure swap(Left: in out String; Right: in out String)is
      C: String := Left;  -- Prohibited for high integrity subsets
      begin -- swap
            Left   := Right;
            Right  : = C;
      end Swap;
```

The declaration of C should not occur, since its storage cannot be statically allocated. An acceptable swap procedure can be declared as

```
procedure swap( Left: in out String; Right: in out String )is
      C :Character := ASCII.Nul;
begin -- swap
      if    Left'First/= Right'First
        or Left'Last   /= Right'Last then
            -- raise some exception or log an error message, etc.
            -- if exceptions not used, extend procedure
            -- using error codes to notify of failures
      else
            for i in Left'Range loop
                  C := Left (i);
                  Left(i) := Right(i);
                  Right (i) := C;
            end loop
      end if;
end Swap;
```

This example moves one component at a time to avoid dynamic storage effects.

For class-wide parameter alternatives, consider the following procedure

```
type T is tagged record ... end record;
procedure swap(Left: in out T; Right: in out T ) is
      C : T  : = Left;
begin -- swap
      Left   : = Right;
      Right  : =  C;
end Swap;
```

The above procedure is a primitive dispatching subprogram that will be called for any extension of

T that has not overridden `swap`. Unfortunately, the routine as written will copy only those fields defined for T, and leave untouched any additional fields defined by extensions. We could pass `Left and Right` in as class-wide parameters, which makes the subprogram non-dispatching, or can convert the parameters to the class-wide (indefinite) types, as shown below.

```
type T is tagged record ... end record;
procedure swap(Left: in out T; Right: in out T) is
      C:  T'Class  :=  T'Class(Left);  --  Prohibited  for  high  integrity
      subsets
begin -- swap
      T'Class(Left)  := T'Class(Right);
      T'Class(Right) := C;
end Swap;
```

In the above example, the definition of `C: T'Class` is prohibited because the compiler cannot determine until run time what the storage requirements are for this object.

The recommended workaround is to use the dispatching call given in the first example, and to ensure that every extension has overridden such primitive subprograms. This will require extra-lingual methods, such as annotations, to ensure that each level has overridden the original definition and has provided a specific routine.

```
type Tn is new T with record ... end record;
procedure swap( Left: in out Tn; Right:  in out Tn) is
      C        :  Tn  : = Left;
begin -- swap
      C      : = Left;
      Left   : = Right;
      Right  : = C;
end Swap;
```

See Section 9.1.4.3 for other issues related to data typing.

Mitigation Approaches: Guidelines

- *Use `length` clauses if dynamic memory allocation is necessary.* If dynamic memory allocation is necessary in a safety application, a `length` clause reserves in advance a pool of specified size of dynamic memory for any allocated objects of a given datatype. To take full advantage of this feature, the program should keep track of the number of objects currently allocated from the pool and ensure that this number does not exceed the capacity of the pool.

- *Provide handlers for the predefined exception* `STORAGE_ERROR` *if dynamic memory allocation is necessary.* If dynamic memory allocation is necessary in a safety application, providing handlers for the `STORAGE_ERROR` exception allows for graceful recovery from the situation of running out of dynamic memory. Without such handlers, the exception is propagated to the run-time executive and will most likely result in a crash of the system. The handlers should be provided for all program unit bodies in which memory is dynamically allocated, as well as in recursive subprograms (SPC, 1989; pp 77-78).

- *Explicitly handle dynamic memory deallocation if dynamic memory allocation is necessary.* Any automatic garbage collection facility provided by a compiler should not be used because it may affect timing. The `pragma CONTROLLED` is provided so that the program can disable automatic garbage collection (reclamation of unused memory)[14]. If dynamic memory allocation is necessary in a safety application, the application program should take full control for dynamic memory allocation and deallocation. Avoid the use of dynamic arrays, as in `procedure P(A:array(<>) of ...)`.

- *Do not assign values of dynamically allocated access objects to other access objects.* If dynamic memory allocation is necessary in a safety application, the application program should not use multiple variables pointing to the same memory location. The danger is that when the shared memory space is deallocated, another variable may still point to the released memory space unless each one is explicitly set to null by the application program. If an application (e.g. a linked list) necessitates such multiple accesses, it should be justified and documented.

```
procedure update_X is
   type three_D_Type is
      record
         x_coord : array(1..100) of float;
         y_coord : array(1..100) of float;
         z_coord : array(1..100) of float;
      end record;
   type three_D_pointer_type is access three_D_Type;

   procedure Dispose is
      new Unchecked_Deallocation(object => three_D_type,
      Name   => three_D_pointer_type);
      ...
      p,q                  : three_D_pointer_type;
      three_D_display  : other_3D_type;  -- a 3-D subtype defined elsewhere
      ...
      begin
      p:=new three_D_pointer_type;-- dynamically allocate access objects p and q
```

[14] It should be noted that according the language definition, there is no mandatory garbage collection requirement. It is up to the compiler implementation to provide such a facility.

```
      ...                              -- p is assigned a value somewhere in the code
      q:=p;                            -- q has been set to the value of p
                                       -- this is the source of the problem
      ...
      Dispose(p);    -- p has been set to null - now q contains an illegal value
      ...
      three_D_display :=q.x_coord;
                  -- annunciator_display will have unintended contents.
                  -- program may continue execution with undetected error
      ...
      three_D_display := p.x_coord;
                  -- CONSTRAINT_ERROR exception will be generated by this
               statement
end update_X;
```

The above example instantiates a procedure called `Dispose` to handle integers from the generic procedure `Unchecked_deallocation` for deallocating dynamically allocated memory units. It then allocates two access objects (`p` and `q`) on the stack, sets the value of `p`, sets the value of `q` based on `p`, deallocates `p` but leaves `q` pointing to inaccessible memory. Somewhere later in the code, the value of `q` is used in an assignment statement. The result may be technically invalid, but if it is within the constraints of the type, it will be displayed with no external manifestation of an error condition. On the other hand, if the explicitly deallocated access object (`p`) is used in a different assignment statement, the error will be detected and an exception will be raised. While neither condition is desirable, an undetected incorrect data value is far worse than a detected incorrect data value which causes an exception to be generated (and hopefully handled without causing an unacceptable system state). The above example demonstrates not only the potential dangers in dynamically allocated variables but also the need to understand the detailed behavior of the `Unchecked_deallocation` procedure and how its use can lead to subtle errors. Important points of its behavior include:

(a) After completion of its execution, the value of the given parameter is `null`.
(b) If the given parameter is `null`, the call has no effect.
(c) If the given parameter is not `null`, the memory pointed by it is returned to the heap.

This last point is of the greatest significance to the above example. Because Ada has no runtime support such as a reference counter, it is possible to define two or more access objects (pointers) to a given location and free the space using only one of those access objects . The other access object(s) would still have an illegal access value(s) and might cause a hazard if used in subsequent processing.

Mitigation_Approaches:_Use_of_Allocators_and_Explicit_Storage_Management

Allocators are one way that dynamic memory can be assigned by Ada. Memory assigned by an allocator comes from a common pool of memory, usually called the "heap," or may come from user-defined or system-defined storage pools.

Dynamic memory may also be assigned in the following circumstances:

- when temporary objects are needed, for example, during type conversions or aggregate assignment; or
- when unconstrained variables are elaborated, initialized or updated.

The dynamic memory for these uses often comes from the stack, but some implementations may use "heap" memory.

Allocators assign dynamic memory to access values in Ada95. There are three distinct methods of managing dynamic memory in Ada95:

1. The normal heap.
2. Storage pools created and managed by the system. Each pool is a dedicated storage area, not affected by other storage pools or the heap.
3. User-defined storage pools. Storage pools with user-defined storage management algorithms.

The risks of storage exhaustion or fragmentation of global memory, possible storage leaks, and potential covert channels cause the prohibition of dynamic memory in high integrity systems. The normal heap and storage pools created and managed by the system are not suitable in these environments. User-defined, dedicated storage pools for each type avoid most of the difficulties that exist in the system-level storage pools and may be appropriate. The syntax used for creating these pools is

```
type p_access_Type is Access P_Type;
for P_Access_Type'Storage_Size use nnnn;
```

Before being able to consider using user-defined storage pools, one should be sure that

- the objects being assigned are of a constrained base type (hence of one fixed size);
- The implementation dedicates one storage pool to each type, ensuring that no unintended mixing, hence possible fragmentation, can occur;
- Each new type declared using these paradigms has its own storage pool declared; and
- No type conversions are permitted from such a type to general access types. Conversion to access types using other storage pools is prohibited by the language.

One should also ensure that the implementation guarantees dedicated allocation and use of the memory in this case.

There are at least two ways of managing dynamic storage explicitly. The first approach uses an array of aliased components and an explicit storage manager, and can be specified as follows:

```
generic
      type T is private;
      type T_ref is access all T;
      pool_size:  Integer;
      package dynamic_Ts is
            exception Usage_Error;
            function Pool_Available return Integer;
            function Create ( initialValue: T) return T_ref;
            procedure Release (p: T_ref );
      end dynamic_Ts;
```

Users are expected to call `Pool_Available` to see if enough storage is available; then call `Create` to obtain storage. Storage that is no longer required can be freed by calling procedure `Release`. A possible implementation of these routines, which gives constant-time Create and Release implementations, is as follows:

```
package body dynamic_Ts is
      type Pool_Array is array (1 .. pool_size) of aliased T;
      type Free_List.Array is array (1 .. pool_size) of T_ref;
      pool: Pool.Array;
      free_list: Free_List_Array;
      space_available: Integer;
      function Pool_Available return Integer is
      begin
            return space_available;
      end Pool_Available;
      function Create(initiaiValue: T) return T_ref is
            P: T_ref;
      begin
            if space_available <= 0 then raise Usage_Error;
            end if;
            P := free_list(space_available);
            space_available := space_available - 1;
            p.all  := initialValue;
            return p;
      end Create;

      procedure Release(p: T_ref)  is
      begin
```

```
            if space_available >= pool_size then raise Usage_Error;
            end if;
            space_available := space_available + 1;
            free_list(space_available) := p;
      end Release;
begin
      for i in 1  .. pool_size loop
            free_list(i)  := pool(i)'Access;
      end loop;
      space_available := pool_size;
end dynamic_Ts;
```

A second possible approach is to use array indices instead of access variables; users would then write `pool(i)` instead of `i.all` to dereference their indices. Otherwise, the management of the storage can be similar to the above example. A possible specification of such a package is:

```
generic
      type T is private;
      pool_size:  Integer;
package dynamic_Ts is
      exception Usage_Error;
      subtype T_ref is Integer range 0 .. pool_size;
      type Pool_Array is array (1  .. pool_size) of aliased T;
      pool:  Pool_Array;

      function Pool_Available return Integer;
      function Create(initiaiValue: T) return T_ref;
      procedure Release (p: T_ref);
end dynamic_Ts;
```

This second approach has several advantages: access types are not used, and the data referenced by `pool(i)` is obviously an element of the pool (in the first variant, `p.all` might be any object of type T). On the other hand, dereferencing an access value may be faster than indexing into an array.

These packages offer an alternative to the use of allocators and `Unchecked_Deallocation` and have some advantages over them. For high integrity systems, however, it is probably more appropriate to design storage managers at a higher level than this. For example, if the access values are used to form linked lists, it might be more appropriate to provide a package that manages the links as well as the basic allocation. Such packages can have interfaces and implementations that eliminate the possibility of certain types of errors.

9.1.1.2 Minimizing Memory Paging and Swapping

The generic guidelines are applicable on the system level. Ada itself contains no features for memory paging and swapping.

9.1.2 Predictability of Control Flow

Base level attributes related to the predictability of control flow in Ada are as follows:

•Maximizing structure
•Minimizing control flow complexity
•Single entry and exit points for subprograms
•Proper handling of program instrumentation.

These attributes and their relevance to safety are discussed in the following subsections.

9.1.2.1 Maximizing Structure

Maximizing structure means minimizing the explicit transfer-of-control statements that change the control flow from the basic set of sequential, conditional, and loop constructs. Most such statements can result in unreachable code. The following guidelines are applicable.

- *Do not use* `goto` statements. The generic guideline on maximizing structure by avoiding `goto` statements applies to Ada. The use of `goto`s can obscure program flow logic. This statement should be used only when there is no alternative. In Ada, where certain types of transfer of control have been incorporated into the language under other names such as `exit`, there is no real reason to use a `goto` in an Ada program (Sanden, 1994). `If` statements, `case` statements, and `loop` statements can be used to express the desired flow of control. In some cases, `return` statements, `exit` statements, or `raise` statements can be used. Consider the following example:

```
<<B_Label>>  statement_1;
             goto A_Label;
             statement_2;            -- unreachable code
             statement_3;            -- unreachable code
             statement_4;            -- unreachable code
<<A_Label>>  statement_5;
             statement_6;
             statement_7;
             goto B_Label;
             statement_8;            -- unreachable code
```

- *Use only one `exit` statement per loop.* At least one `exit` statement is needed in loops without iteration schemes (LRM, 1995). Thus, only one `exit` statement should generally be used for the loop within the loop or for any nested loops. The following loop is well-structured and is as readable as most alternatives

```
loop
      Get_input;
      exit when Input_exhausted;
      Process_data;
end loop;
```

- *Use only one `return` statement per function.* Multiple return statements can make the meaning of a subprogram confusing. Thus, function subprograms should have only one `return` statement and procedure subprograms should either use the normal exit at the end of the body or have only one `return` statement if the end of the body is inaccessible, for example, an infinite loop just before the end of the body.

A `return` statement can be used when it is determined that no further useful action is possible in a subprogram body. A `return` is effectively a forward jump to the end of the subprogram body. `Return` statements can make the normal flow of control of the subprogram more apparent, and thus increase the readability of the code. However, a deeply nested, `return` statement is easily overlooked, and the assessment or maintenance of code can suffer.

The `return` statement is used when leaving a master scope. Significant run-time activity can occur in conjunction with this statement: specifically, termination of nested tasks, finalization of controlled types, removal of exception handlers, return of temporary storage used by the run-time environment. Almost all these activities are not applicable in the set recommended in this document because language constructs that cause this behavior are prohibited. Activities still possible are exception handlers in the program unit and anonymous storage pools, both of which require clean-up. In these cases, the return should leave only the outermost scope.

While maximizing structure is highly desirable for normal program flow, different rules apply to exception handling as discussed in Section 9.2.2. When exceptions are raised, other considerations (e.g., timing, intermediate operations, etc.) dominate. The guidelines on exception handling discuss `raise` statements in more detail.

9.1.2.2 Minimizing Control Flow Complexity

The generic guideline applies to Ada. The language-specific guidelines for minimizing control flow complexity are as follows:

- *Limit nesting levels.* As noted in the generic report, there should be explicit organizational or project-specific limits on nesting. These limits may be determined in part with respect to a particular language and execution platform. The style guidelines for Ada published by the Software Productivity Consortium recommend a maximum nesting level of three to five (SPC, 1989; pp 83 - 84).

- *Use `if..elsif` instead of nested `if..else`.* Use of an `if..elsif` in place of nested `if..else` statements helps avoid program structural and logical errors (Barnes, 1984; p 62) as shown in the following example:

```
-- Use
if condition_1 then
  statement_1;
elsif condition_2 then
  statement_2;
end if;

-- instead of
if condition_1 then
  statement_1;
else
    if condition_2 then
      statement_2;
    end if;
end if;
```

Always provide an `else` branch to `if` statements if there is a remote chance that the conditions specified by the other `if` statements are exhaustive.

- *Use `case` statements for multiple branches.* The `case` statement serves as a switch for multiple branches and allows one evaluation for them. It is a powerful alternative to the `if` statement when the branch to be taken depends upon the value of a discrete expression, and it is preferred if more than two conditions or branches are called for in the software design. To avoid a syntax error, the `when others` construct should be included if there are any possible values not given in other alternatives, as seen in the following example (SPC, 1989; p 85):

```
-- Use
      case thermal_alarm is
            when core    => core_thermal_alarm(sensor_value);
            when inlet   => inlet_thermal_alarm(sensor_value);
            when outlet  => outlet_thermal_alarm(sensor_value);
            when others  => do_something;
      end case;
```

```
-- instead of
if thermal_alarm = core then
  core_thermal_alarm(sensor_value);
elsif thermal_alarm = inlet then
  inlet_thermal_alarm(sensor_value);
elsif thermal_alarm = outlet then
  outlet_thermal_alarm(sensor_value);
else
  do_something;
end if;
```

It should be noted that the `case` statement is not an all purpose replacement for the `if.. then...else` construct. A `case` statement is only possible if the cases depend on the different values of one expression with a limited range of possible values. (In the example on this page, `thermal_alarm` is an enumerated type with a limited set of possible values.) In that situation, the `case` construct is always preferable over an `if.. then...else` unless the number of branches is very small.

Timing analysis for a `case` statement is implementation-dependent; there are several different implementation strategies. The most common strategy for `case` statements with a relatively few elements that are contiguous is to build a jump table. In these cases, the evaluation of the condition and transfer to the code for that case occurs in nearly constant time.

9.1.2.3 Single Entry and Exit Points for Subprograms

Although the generic guideline is applicable with respect to one *normal* entry and exit point per subprogram[15], the guideline has limited applicability due to Ada's exception handling and tasking features. In Ada, however, subprogram declarations and implementations may be separated; this characteristic of Ada gives rise to additional guidelines arising out of the possibility that a subprogram is called before the body has been elaborated. In addition, some restrictions are required on parameter modes, default expressions, subprogram calls, and function results. Ada-specific guidelines are:

- *One normal entry and exit per subprogram.* Subprograms (procedures and functions) should have one normal (as opposed to exception) entry and one normal exit. The word `return` should appear exactly once in each function and not be used in a procedure. In exceptional cases, however, multiple exits can be used if they increase readability.

- *Limit the number of exception entry/exit points.* The number of these points should be kept as low as possible. Each of these exception propagation exit/entry points should be

[15]It is more appropriate to refer to entry and exit points in program unit bodies rather than in subprograms in the case of the Ada language.

documented clearly. The propagation of an exception raised in a subprogram to the caller of the subprogram should be limited or not used at all because such propagation creates an additional exit point for the first subprocedure and an additional entry point for the caller's exception handler. More points on propagation of exceptions are discussed in Section 2.2.2.

- *Avoid multiple task entry points.* Each active program unit (i.e., task) may have multiple interaction points with other active program units. The number of these interaction point should be designed to minimize program complexity both within the task and the entire program. Additional points on tasking are described in Section 2.2.

- *Elaboration of a subprogram body should be shown to occur before a call to that subprogram.* (Saaltink, 1996) Ada's separation of subprogram declarations from subprogram bodies introduces the possibility that a subprogram is called before the body has been elaborated. If this occurs, the exception `Program_Error` is raised. Care should be taken to avoid the possibility of such failures (and the need for run-time code to check for this condition). In many cases, the subprogram declaration and definition can occur together, which eliminates the possibility of this failure. In most other cases, it is possible to ensure that no calls of a subprogram appear between its declaration and definition. For subprograms declared in a library-level package, `pragma (Elaborate_Body)` can be used to ensure that no code is executed between the elaboration of the specification and the elaboration of the body.

- *Indefinite return types from functions should not be used.* (Saaltink, 1996) A function may have an indefinite return type. In such a case, the anonymous return object cannot be created until a `return` statement is executed. In addition to possible control flow problems, the storage for this object will need to be allocated dynamically, and the caller will need to deallocate this object. In addition, tagged return types are problematic if the tag of the returned object is not the same as that of the return type. The language rules are intricate, and this feature should be avoided unless the returned object is known to have the tag of the return type.

- *Indefinite formal parameters in procedures and functions should be used with care.* As was the case with indefinite return types, any expression involving parameters of indefinite type may require dynamic storage of unknown size for temporaries. The following restrictions are imposed to eliminate these cases as follows:

  1. Subtype conversions in subprogram calls should not be used.

  2. Concatenation of one-dimensional arrays should not be used.

  3. Discriminated records should not be used.

The remaining situations, class-wide types and unconstrained arrays, have reasonable implementations that do not need dynamic storage, but the implementation should be checked to confirm this assumption. Access parameters are not allowed in high integrity systems, since they can be used to pass or copy the access to an object whose lifetime is incompatible with the target of the eventual assignment. Allocated objects in Ada95 always have the lifetime of the access type definition, and generalized access types have explicit accessibility checks, but access parameters could be used to avoid such checks and are therefore considered inappropriate.

Some indefinite return types can be avoided simply by using a constrained return type, without changing the function. For example, if a function always returns a string of ten elements, its return type can be specified as `String (1..10)` instead of `String`. Alternatively, it may be possible to return a general access value (e.g., return the definite type `access all String` instead of `String`), or to replace the function with a procedure which returns the value as an `out` parameter.

Instead of declaring an access parameter, a parameter may be declared to be of a previously declared generalized access type.

The use of default expressions for some parameters is a convenience, but comes at a cost. Implementations may differ in how they support default parameters, making time and space analysis more difficult. Traceability of source code to object code is also impeded. Default expressions are therefore to be discouraged.

Formal parameter modes provide a significant capability to provide extra protection to data that is passed into subprograms. Data that is passed in as `in` cannot be updated within the subprogram. Procedure parameters of mode `out` should either be initialized in order to guarantee that the corresponding actual parameter receives a valid value when the procedure returns, or an analysis performed to show that a valid value is always assigned.

Aliasing should not be used in procedure calls. To facilitate this, it is useful to document any non-local variables used in a subprogram. (For a subprogram appearing in a package specification, it should not be necessary to mention any variables declared in the subpackage body that are used in the subprogram, as callers outside the package normally do not have access to these variables.) An `aliased` variable declared outside a subprogram should not be used in the subprogram body.

For some types, it is unspecified whether the parameters of that type are passed by copy or by reference. The choice can affect the semantics of a program; thus, programs using such types might be non-portable.

In cases where the parameter passing mechanism is not specified (i.e., may be by copy or by reference), bounded errors can arise if parameters are aliased with other parameters' or with globals. Such aliasing should be avoided.

9.1.2.4 Proper Handling of Program Instrumentation

The generic attributes apply to Ada programs. However, there are no specific Ada guidelines for this attribute.

9.1.3 Predictability of Timing

An Ada-specific guideline related to timing predictability was discussed with regard to recursion in Section 3.1.1.1. Additional related guidelines are:

- Minimizing the use of tasking
- Minimizing the use of interrupt-driven processing
- Characterization of timing for the Ada runtime environment
- Control of memory management from the application

These guidelines are discussed in the following sections.

9.1.3.1 Minimizing the Use of Tasking

The generic guidelines for tasking apply to Ada. The use of tasking in safety-critical applications should either be avoided entirely or should be highly constrained because of the following reasons: scheduling policy and timing uncertainties, implementation differences, race conditions, asynchronous tasking problems, priority issues, and abort issues.

Since Ada is a language that defines a model of concurrency within the language, there is considerable interest in using this model to manage the concurrency that occurs in a real system. To date, this has not been possible; the complexity of the model and the lack of precision made exact specification and analysis impossible. For Ada95, a limited subset consisting of library-level tasks and protected types, but excluding entries, looks promising but has not yet been fully analyzed.

Some compiler vendors are attempting to certify run-time systems supporting limited forms of concurrency, and research groups are studying models of concurrency and restricted subsets of Ada's features that allow for strong predictions of program behavior.

A system can be designed to use a cyclic scheduler rather than tasks. Some programs that might naturally be expressed using tasking can be rewritten (using loop inversion or some similar technique), where each task is replaced by a package that represents the task state, and has a procedure that makes some progress in the task's activities. A main loop can interleave the steps of these packages to simulate the concurrent activity of the tasks.

It may be possible to implement a customized concurrency mechanism to be used instead of the predefined tasking features.

Although tasking should generally be avoided in safety-critical software, there may be cases where it is the only reasonable solution. The following guidelines will reduce the risks associated with tasking identified above, but do not totally mitigate them.

- *Ensure that the concurrent software design is as simple as possible, but no simpler.* That is, there should be no more tasks than necessary and there should be no more task synchronization and communication than necessary.

- *Avoid abort statements.* Programs should avoid using the abort statement. The following is an example of an abort command:

```
abort A_Short_Task, Temperature_Tracking(3), Sensor_Data_Collection.all;
```

  Aborting a task can have many consequences, not all of which are obvious. If a task is aborted, then all tasks dependent on it are aborted. Furthermore subprograms and blocks that were called by it will also be aborted. If the task was suspended, the abort will cause it to appear to have been completed. Delays are canceled by aborts, and tasks are removed from entry queues. Accept and select statements will be left waiting for partners. Aborting a task in rendezvous has complex consequences that depend on the situation (SPC, 1989, p. 121; Barnes, 1984, p. 239).

- *Avoid dynamic tasking.* All tasks should be elaborated only at system initialization. Dynamic tasking complicates the predictability of the run-time behavior of a program for at least the following reasons:

  1. Allocated task objects referenced by access variables allow generation of *aliases*, that is, multiple references to the same task object. Anomalous behavior can arise when references to an aborted task are made using an alias.

  2. A dynamically allocated task that is not associated with a name (i.e., a "dropped pointer") cannot be referenced for the purpose of making entry calls, nor can it be the direct target of an abort statement (SPC, 1989, pp. 76-78, 111-112). Note that there may exist tasks that need not be referenced, such as a task which performs some periodic function in the background.

  Tasks created at runtime by means of the allocator `new` should not be used at all for the following reasons:

  3. Runtime-created tasking complicates debugging, understanding, and control flow tracing.

4. Runtime-created tasking allocates memory from the heap and can lead to an insufficient memory condition.

- *Use `delay` statements only for waiting, not synchronization.* `Delay` statements should not be used to set a starting time or to synchronize tasks. Synchronization and control should be handled through a rendezvous between tasks. The `delay` statement only sets a *minimum* time period, not a maximum period. For example, `delay 3.0` means a delay of at least 3 seconds. The only guarantee is that the delay will be for a minimum time period (Barnes, 1984; p 251). Timing uncertainties are associated with differing implementations by compiler vendors, interactions with underlying operating systems (or real-time kernels), and the design of the hardware platform.[16]

- *Minimize the number of `accept` and `select` statements.* Both the number of `accept` and `select` statements per task and the number of accept statements per entry should be minimized to the extent possible without unduly complicating internal program logic and complexity. The rationale for this guideline is to simplify the concurrent design (SPC, 1989; p 119). With more `accept` or `select` statements, the verification of the design and state of each calling program and each entry call causing the execution of different code sequences dependent on the task's local state can become a very involved effort.

- *Avoid certain variations of `select` statements.* Conditional entry calls, selective waits with `else` parts, timed entry calls, and selective waits with `delay` alternatives should be avoided because they pose a risk of race conditions (SPC, 1989; p 119). The only circumstance under which they should be used is when the possibility of race conditions can be conclusively shown not to exist.

- *Use `terminate` alternatives with every selective wait.* Multiple task exits (as opposed to returns) are frequently necessary to avoid deadlocks (SPC, 1989; p 122) . Every Ada selective wait statement not requiring an `else` part or a `delay` alternative should have a `terminate`. However, unnecessary or redundant `terminate` statements should be deleted from tasks to reduce possible confusion.

- *Account for exception handling during task interactions.* An exception raised during a rendezvous (i.e., in the body of an `accept` statement) affects both the calling and the called task[17]. The exception should be handled either in the body of the `accept` statement or in the affected task. More discussion of exception handling is in the next section.

- *Minimize use of the PRIORITY pragma.* The program should not depend on the order in

---

[16] Ada 95 adds the function **`delay until`**

[17] If the rendezvous is nested, i.e. if the `accept` statement appears in the body of another `accept` statement , yet another task is affected.

which tasks are executed or the extent to which they are interleaved. PRIORITY should be used only to distinguish general levels of importance. The rationale for this guideline is that the Ada tasking model is based on preemption and requires that tasks be synchronized only through the explicit means provided in the language (i.e., rendezvous, task dependence, and pragma SHARED). The scheduling algorithm is not defined by the language and may vary from time slice to preemptive priority. Some implementations provide several choices that a user may select for the application. It should be noted that this pragma may limit portability. The number of priorities may vary between implementations. In addition, the manner in which tasks of the same priority are handled may vary between implementations even if the implementations use the same general scheduling algorithm.

9.1.3.2 Minimizing the Use of Interrupt-Driven Processing

The generic guidelines for interrupt-driven processing apply to Ada. It is not generally desirable in safety-critical systems because it can lead to nondeterministic maximum response times and unanticipated system states. Use of a deterministic approach to the monitoring and control of multiple input sources is usually preferred. However, there may be some situations where interrupt-driven processing has a significant design advantage over alternatives (e.g., to handle the acceptance and processing of plant emergency input). The following mitigating guidelines apply:

- *Declare interrupt values using named constants, and isolate them from other declaration clauses.* The actual value for an interrupt is implementation defined. The isolation of the interrupt value named constants will not affect performance and provides portability between similarly supported implementations (SPC, 1989; p 145).

- *Isolate interrupt receiving tasks into implementation-dependent package bodies when possible.* The handling of interrupt entries is not specified by the Ada Language Reference Manual (DoD-STD-1815A). They are implementation dependent; that is, specific to a compiler and its target machine. If the code is moved to a different implementation (which may happen either during the initial development or during maintenance), the interrupt-handling features may not be supported. The reason why this guideline is qualified with "when possible" is that the isolation of interrupt entries creates an additional rendezvous that will often double the interrupt latency time. Where this is unacceptable, the interrupt entries should be proliferated with a resulting decrease in portability.

- *Pass the interrupt to the main tasks via a normal entry.* This allows any implementation-dependent features to be isolated from the higher level (and presumably more complex and worthy of preserving) software that actually handles the interrupt.

- *Do not use task entry points for interrupt processing.* Task entry points should not be used for interrupt handling, unless the user-written low-level code is known to be safe (Jones, 1988).

9.1.3.3 Characterize Timing for the Ada Run-Time Environment

The run-time environment (RTE) that is loaded together with the Ada source code into the target system is a key component affecting timing. The RTE is generally delivered by the compiler vendor and is not developed as part of the safety application. Nevertheless, a process of testing and validation of the RTE to ensure that it is deterministic, is functionally correct, and will satisfy timing requirements is an important part of the safety development process. Characterization of the Ada RTE for suitability in the safety application is primarily a test and verification issue which is beyond the scope of this document.

9.1.3.4 Avoid Automatic Memory Management

As noted in Section 2.1, a major source of timing uncertainty is automatic garbage collection (memory reclamation) by the run time environment (if supported). Thus, it should be disabled in time-critical systems by use of the `pragma controlled` where deterministic response time requirements exist.

9.1.4 Predictability of Mathematical or Logical Result

Base level attributes related to the predictability of the mathematical or logical result in Ada95 are as follows:

- Initialization of variables before use
- Minimizing interface ambiguities
- Use of data typing
- Accounting for precision and accuracy
- Order of precedence of arithmetic, logical, and functional operators
- Avoiding functions or procedures with side effects
- Separating assignment from evaluation
- Controlling class library size
- Minimizing use of dynamic binding
- Controlling operator overloading.

These attributes and their relevance to safety are discussed in the following subsections.

9.1.4.1 Initialization of Variables before Use

The generic guideline with respect to initialization of all variables applies to Ada. Variables should be initialized to some known value at the beginning of an execution cycle before using them. The effect of uninitialized objects can be serious because the current (uninitialized) value may not be a valid value, causing exceptions or erroneous behavior. Scalar objects should be shown to be initialized before they are read. The simplest approach is to explicitly initialize all objects. Initialization of variables may be done in three ways:

1. Explicit initialization during the run time of each variable. This method physically separates object declaration from initialization, which makes static analysis more difficult.

2. Initialization of each variable as the object is declared. This initialization occurs at elaboration time, but it is visible in the declarative code and is executed in the order of declaration. This method is our preferred way of initializing. Note that this is not possible for limited types, which should use explicit initialization.

3. Default initialization. Default initialization occurs at elaboration time, and results in run-time code to do the initialization. The order of initialization of subcomponents is not predictable, traceability of object code is hindered, and these default initializations may depend upon default initializations of components. Default initializations are therefore discouraged in the highest integrity systems and their appearance in such systems challenged.

A compiler cannot be depended on to reset variables automatically (Gottfried, 1993; SPC, 1989, pp. 103-104). However, even if the compiler could be relied on to initialize values, the safety concern would still exist because the compiler cannot be expected to initialize all objects with suitable values.

Ada provides a variety of syntaxes for data initialization upon elaboration of a variable as shown in the following example:

```
subtype Number_Of_Widgets is Natural range 0 .. 1_000;
Accumulator : Number_Of_Widgets := 0;
type Coefficients is array (1 .. 3, 1 .. 3) of Weights;
Example_Coefficients : Coefficients
  := (       ( 1.0,   0.5,   0.1),
             ( 0.5,   1.0,  -0.3),
             ( 0.1,  -0.3,   1.0)  );

type Complex_Numbers is record
  Real_Part              : Float := 0.0;
  Imaginary_Part: Float := 0.0;
end record;
Zero : Complex_Numbers; -- Automatically initialized to(0.0, 0.0)
                        -- when elaborated (unreliable)
Square_Root_Of_Minus_1 : Complex_Numbers
  :=  (Real_Part => 0.0, Imaginary_Part => 1.0);
type A is array (1 .. 100) of Character;
      AA :A := (others => 'x');
                        -- Aggregegate initialization:
                        -- multiple elements
                        -- of an array can be given initial values
                        -- by means of the construct 'others ==>'

type B is array (years, months) of Integer;
```

```
BB: B := (others => (others => 0));
                    -- Without this construct,
                    -- it would be very impractical to initialize
                    -- a large array.
```

Parameters of mode `out` can be initialized using only the first method, and should be used with caution

If several variables are declared together, the initializing expression is evaluated once for each variable that is declared. This may result in different variables having unexpectedly different values, if the initialization functions have side-effects.

The following are Ada-specific initialization guidelines.

- *Initialize in function body if initialization occurs via a function call.* If initialization occurs via a function call, initializations should be done in a program body rather than in the variable declaration since the function body may not have been elaborated when the variable declaration was encountered (SPC, 1989; pp 103-104).

- *Restrict use of aggregate assignments for initialization of large objects.* Aggregates are the mechanism for atomically initializing or assigning values to compound structures, such as records or arrays. Aggregates are one of the two ways to change a discriminant for unconstrained discriminated objects (the other being assignment of an existing object). As well as ensuring that the target object is completely covered by the assignment and that discriminants and component values match, aggregates provide notational shorthand and clarity to the assignment of values to compound objects. As shown in the above example, aggregates are a useful way of initializing large arrays. However, the initialization of large objects via aggregates should occur with caution. The reason for this guideline is that some compilers accomplish aggregate assignments by first building a temporary version of the object with the specified values in system memory and then copying the contents into the actual object. If the size of the temporary version exceeds available memory, the result could be a system crash.[18] In such cases, testing should be done to ensure that the aggregate assignment can be performed acceptably under operational conditions. An alternative is to perform initialization in the program unit body rather than in the objects' declarations for large objects.

[18]Such a situation actually occurred in the experience of one of the writers of this section. In an image processing application, a 1024 x 1024 array of pixels was initialized by an aggregate of the form `((others => 0), others => 0)`. This caused the entire system, including the operating system and the other jobs being executed concurrently, to crash without any error messages. Determining the cause was complicated by the fact that the Ada code was syntactically and semantically correct.

Within the aggregate definition, the assignment of values to components is not deterministic, and the component associations for calling functions will call such a function once for each associated component. The choices made by a given implementation will be determinable, and should be checked.

There are two cases in Ada where *explicit* initialization of a variable need not be done to comply with the guideline. First, all objects of access type (i.e., pointers) are automatically initialized to `null` by the compiler. Second, type definitions for records may contain default initialization values for all components; whenever objects of those record types are elaborated, their components are set to the defaults in the absence of an explicit initialization (DoD-STD-1815A; Section 3.7).

9.1.4.2 Minimizing Interface Ambiguities

The generic guideline to minimize interface ambiguity applies to Ada. Ada automatically provides features that eliminate many interface errors. For example, constraint checking is performed on values of actual input parameters to ensure they are not out of range. Another example is that the indices of the first and last elements in an array or array slice-parameter are automatically passed in with the actual array parameter. Nevertheless, the language does not eliminate interface ambiguities.

The following are specific guidelines:

- *Specify argument modes.* Arguments with procedures and entries should have their modes specified in their declarations rather than relying on the default mode (SPC, 1989; p 68). Specifically,

```
procedure Quadratic(a, b, c: in  Float;    root1, root2 : out Float);
```

rather than

```
procedure Quadratic(a, b, c : Float; root1, root2 : out Float);
```

While the latter declaration is acceptable syntax (and in that sense, is unambiguous to the compiler), explicit use of modes avoids confusion to programmers as well as to reviewers.

- *Restrict use of the* `in out` *mode.* The `in out` mode should be used only for parameters whose value will be changed by the procedure. It should not be specified for parameters used exclusively as either `in` or `out` parameters. When used in place of an `in` mode, it is possible to modify a value that should be constant unintentionally. Using `in out` for an `out` mode causes fewer problems, but it does obscure the intent of the parameter. This

mode is frequently used in the case of an output parameter whose value is read inside a subprogram; when this situation leads to a compilation error, many programmers will change the mode from `out` to `in out` rather than taking the trouble to declare and use a local variable.[19] For example, programmers will code as follows:

```
procedure Find_Max (In_The_List : in Some_Array_Type;
                    Maximum     : in out Element_Type) is
begin
    Maximum := Element_Type'first;
      for List_Index in In_The_List'range loop
            if In_The_List(List_Index) > Maximum then -- value read here
                  Maximum := In_The_List(List_Index);
            end if;
      end loop;
end Find_Max;
```

instead of coding:

```
procedure Find_Max (In_The_List : in Some_Array_Type;
                    Maximum     : out Element_Type) is
          Local_Max : Element_Type := Element_Type'first;
begin
      for List_Index in In_The_List'range loop
            if In_The_List(List_Index) > Local_Max then
                  Local_Max := In_The_List(List_Index);
            end if;
      end loop;
      Maximum := Local_Max;
end Find_Max;
```

- *Use named parameter associations.* Named parameter associations should be used by the calling routine for functions, procedures, and task entries whenever there are two or more parameters of the same type in the parameter list. Using named parameter associations improves readability and reliability (Booch, 1983; p 106). The following example shows the use of named parameter associations for a quadratic equation evaluation procedure.

```
Quadratic(a      => second_order_coefficient,
```

[19]In Ada 95 reading an **out** mode parameter is allowed. According to the Ada 95 rationale, too many programmers were forgetting to copy the value of the local variable into the output parameter at the end of procedures.

```
            b      => first_order_coefficient,
            c      => constant_term,
            root_1 => first_root,
            root_2 => second_root,
            OK     => status) ;
```

- *Refer to the target data type rather than the pointer's type when referencing data.* When data referenced by a pointer are to be read or modified in a subprogram and the value of the pointer itself is not to be used, the declaration and call of the subprogram should refer to the target data type rather than the pointer's data type as shown below.

```
type Target_Type is array (1 .. 100) of Component_Type;
     type Pointer_Type is access Target_Type;
     The_Data : Pointer_Type := new Target_Type'(others => 0);
     -- Better subprogram declaration
     procedure Print(The_Data : in     Target_Type);
     -- Better subprogram call
     Print(The_Data.all);
     -- Worse subprogram declaration
     procedure Print(The_Data : in     Pointer_Type);
     -- Worse subprogram call
     Print(The_Data);
```

  This practice removes ambiguity about which data are to be processed in a subprogram, that is, the data being pointed to or the pointer. For `in` mode parameters, this practice removes the possibility of modifying data meant to remain unchanged, since it is possible to modify data pointed to by an `in` mode access type parameter. The practice also allows checking for out-of-range data values. However, care should be taken when passing a large object by value to avoid memory overflows.

- *Avoid aliased parameters.* Aliased parameters should be avoided. They can arise from using the same actual parameter for more than one formal parameter (and calling both by reference), using overlapping array slices, referencing global variables, and using pointers referencing the same data for different actual parameters. Results can be dependent on compiler-specific implementations such as the order of evaluation of actual parameters. Even when called by value, passing the same actual to two formal parameters or passing a global variable to a procedure is discouraged.

The presence in Ada of implementation-dependent features, such as representation items, representation attributes, enumeration representation clauses, and the `pragma pack`, is necessary in many situations. These features can, however, cause problems in high-integrity systems so their use should be reviewed carefully. Guidelines related to these features and some of the workarounds that might be appropriate for reviewers to consider follow.

- *Representation items/clauses should be used cautiously.* Representation items are used to specify or constrain how data and other entities are represented on the underlying hardware. There can be many legitimate reasons for specifying this representation, for example, because of special requirements of certain I/O hardware, to interface with code written in another language, or to reduce the storage requirements of a program.

  Implementations may have difficulty in supporting some representations, and there can be unforeseen consequences of representation choices. If possible, any entity which has had its representation specified should be used in the most constrained ways possible, and in the smallest scope. If necessary, data can be converted to a type with a standard representation before being used in any sophisticated way.

- *Special restrictions apply to the use of* `pragma pack`. `Pragma pack` is a representation specification that causes the compiler and run-time system to modify the representation of objects to which the pragma has been applied. A run-time system will usually attempt to pack arrays or records as tightly as possible, i.e., so that a minimum number of storage units will be used to represent that item.

  `Pragma pack` is a language construct that changes how other language features work. Specifically,

  - It usually takes more time and CPU resource to access an individual element of a packed structure than it would take to access the equivalent element in an unpacked structure;

  - Run-Time code is often used to access records; and

  - `Pragma pack` applied to records that have `'alignment` or `aliased` applied to the overall object or to components may be erroneous.

  The following restrictions in the use of `pragma pack` in high integrity systems apply.

1. Any declaration and use of packed objects should be contained within the body of a single Ada package. This means that a new type should be declared within the package body, and `pragma pack` applied to that type. The packed object(s) will be declared within the same scope and will be of the same type of the derived, packed type declaration.

2. For objects of a packed data type:

   - Avoid components with `'alignment` or `aliased` applied or with type-related representation items applied. Type-related representation items include enumeration representation clauses, record representation clauses, external tags, `pragma import`, `pragma export`, `pragma convention`, specification clauses for

the `'Read` or `'Write` attributes, input clauses, `pragma Atomic`, and `pragma Volatile`.

- Other, conflicting type-related representation items should be avoided.

- Conversion between packed and unpacked composite types should not be used.

Type conversion between packed and unpacked types can be simulated by a loop doing component-by-component assignments.

- *Use of representation attributes should be scrutinized.* The use of representation attributes as a specification mechanism to control the storage, alignment, or representation of a type, component, or object should be used with extreme caution. Projects that use these attributes in this way should ensure that:

1. the compiler and run-time environment support the attribute in how it is used;

2. the compiler and run-time environment use static techniques in the management of objects that follow the representations specified;

3. the effect of the representation clause is minimized, usually by restricting the type specification and representation specification to a package body, and explicitly converting between such objects and objects that do not contain such representation clauses; and

4. the attribute is not used in conjunction with other attributes or pragmas that may affect components or enclosing types and objects in ways that may produce erroneous behavior or exceptions.

Type conversion to or from a type for which representation attributes have been specified should be prohibited.

Representation attributes, such as `'alignment` and `aliased` are representation specifications that causes the compiler and run-time system to modify the representation of objects to which they have been applied. A run-time system will usually attempt to modify the alignment, representation, or storage of data objects to meet the specified representation. These attributes may be applied to a subtype, or to individual objects of a subtype.

Representation attributes have characteristics that are very implementation dependent, and the possibility of interactions among these attributes is significant. When representation attributes must be applied as a specification of a characteristic of a type, subtype, or object, they should be applied within a very constrained region, such as a package body, and care should be taken to ensure that other, possibly conflicting representation items, are not being applied to components, composite types, or objects containing this type.

The specification of representation attributes takes an expression or a name as their argument. In the general case, these expressions and names reflect dynamic values. Using dynamic quantities or values that are determined at run time forces the compiler to use dynamic storage management techniques or to provide the possibility of exceptions that cannot be detected at compilation time. For high integrity systems, only static expressions and named numbers should be permitted, and projects should ensure that the compilation system uses static storage management techniques when the attributes are specified statically.

An exception to this is the `'small` attribute for fixed point types. This attribute is mandatory for fixed point types and should reflect the `'delta` of the type declaration.

- *Enumeration representation clauses should be avoided.* This guideline may be violated in the body of a package where interfacing to the external world is necessary. Immediately recast the value into an enumeration-type object of the same class, but with default representations, after explicitly ensuring the validity of the data by the use of the `'Valid` attribute.

- *Enumerated types with representation clauses as the basis for an iterative Ada construct, such as the `for` loop should be avoided.*

Enumeration representation clauses let the developer control the internal representation of enumeration literals. Any integer can be assigned to an enumeration literal. The use of such a clause may have several effects on a program unit that employs them. Specifically,

1. The size or potential packing of objects of the enumerated type may differ from that of either the default representation, or other types in the same class.

2. The time to perform a `'succ` or `'pred` will change.

3. Opportunity exists to carry values that do not correspond to one of the represented values within variables. This may happen because compilers often do checks based on the initial and final value of a range, but will not check for the legality of all possible values (because of the cost of the computation).

Instead of applying a record representation clause, a derived integer type with a range that includes all needed values can be declared and used to import or export values to the external world. Such a type has the benefit that its legal value set includes all values in its range. To find or test for values that correspond to the enumerated type, create an array of the values indexed by the enumeration type, as described below.

For example,

```
type Command_Choices is
      (RESET, MOVE_ARM, LOWER_HEAD, RAISE_HEAD, SCAN);
type Command_Representation is new Integer range 0 .. 16#FF#;
Reset_Representation     : constant Command_Representation := 16#0F#;
Move_Arm_Representation  : constant Command_Representation := 16#1F#;
Lower_Head_Representation: constant Command_Representation := 16#2F#;
Raise_Head_Representation: constant Command_Representation := 16#3F#;
Scan_Representation      : constant Command_Representation: = 16#6F#;

Command_to_Command_Choices is
      constant array (Command_Choices) of Command_Representation :=
      (RESET       => Reset_Representation,
       MOVE_ARM    => Move_Arm_Representation,
       LOWER_HEAD  => Lower_Head_Representation,
       RAISE_HEAD  => Raise_Head_Representation,
       SCAN        => Scan_Representation);
```

Now, conversions from the internal representation consist of one explicit array lookup, and conversions from external to internal consist of at most a linear search of the array. For enumeration types with a small set of choices and nearly a compact external representation, a case statement works efficiently in the conversion. For enumeration types with a large set of choices and a strict ordering in the representation, binary search techniques may substantially improve the time bounds on the conversion. In either case, invalid data can be readily detected and explicitly handled without the risk of `Constraint_Error`.

9.1.4.3 Use of Data Typing

The generic guidelines for data typing apply to Ada. Ada was made a strongly typed language in order to provide the potential for increased safety. Code should take advantage of this feature to the maximum extent possible. A type in Ada95 consists of

- a set of values for all objects of that type,
- a set of subprograms that are directly associated with that type, and
- a representation of the type in terms of value, size, packing, and alignment.

The typing mechanism in Ada95 is known as *strong typing* and is a significant contribution to software engineering. Values that have been associated with one type cannot be quietly converted to another type; they should be explicitly converted. This requirement forces designers to be explicit about conversions and makes them visible in the source code. Conversions can occur only between types that have a common parentage, i.e., all numeric types, or types that have been derived from the same parent type.

Ada95 has added the notion of *statically matching subtypes* to Ada83. When a compiler determines

that one subtype does not statically match another type, it will prevent any usage of the first type in inappropriate places. This allows compile-time detection of what would have been an execution-time error in Ada83. For the following declaration:

```
N: constant = 10;
subtype Short_String_Type is String ( 1.. N);
```

The following declarations do *not* statically match `Short_String_Type`

```
subtype Another_String_Type is String(1..N-i);
subtype Another_String_Type is String (2..N);
```

The following declarations *do* match `Short_String_Type`,

```
subtype Another_String_Type is Short_String_Type;
subtype Another_String_Type is String(1..N);
```

It was seriously considered during the Ada9X project to force the last statement to not statically match the basic type declaration, but the ensuing incompatibility forced this to remain a statically matching subtype. The major issue is that a project could accidentally make a subtype match its parent, and ensuing procedure calls or generic instantiations that do not make sense will be permitted. It is recommended that extra precautions be taken to declare such subtypes as new, derived types.

The following specific guidelines are related to the full use of data typing:

- *Constraint checking.* Run-time constraint[20] checking allows the detection of anomalous conditions. Specifically, when an object is assigned a number outside its range, then a `CONSTRAINT_ERROR` is raised. The `pragma suppress` disables run time constraint checking and should *not* appear in any Ada programs used to generate the safety-system machine code (SPC, 1989; p 102). Out-of-range values should be detected as soon as possible in safety- critical systems, so their point of occurrence may be localized before they have a chance to propagate and corrupt further calculations.

[20]Most Ada 83 implementations provide another predefined exception, `NUMERIC_ERROR`, for detection of overflows and underflows. This run-time check also should not be suppressed. In Ada 95 the `NUMERIC_ERROR` exception is incorporated into the `CONSTRAINT_ERROR` exception (Ada 95 LRM , 1995).

- *Limit range of scalar datatypes.* Scalar data types with the narrowest possible range of values should be used in order to detect erroneous data calculations. For example, the predefined subtype, `Positive` should be used for variables that are always greater than zero instead of the predefined type `Integer`. If the upper limit of possible values for the variable is known, a subtype of Positive should be used. A corollary is that the predefined types in `package Standard` should be used for variable definitions only when the possible range of values for the variable is completely unknown or is the same as the range for a predefined type (SPC, 1989; p 34).

  The practice of using the most-limited bounds on ranges can lead to difficulties in the case of real types and subtypes. This practice may result in the raising of spurious constraint errors and in needless interruption of normal program execution. The following example illustrates this difficulty:

```
with Trig_Functions;
      PI : constant := 3.14159265;
      subtype Angles is Float range 0.0 .. 2 * PI;
      subtype Args is Float range -1.0 .. 1.0;
      -- Spherical trigonometric function to calculate
      -- angular distance between two points
      function Angular_Distance     (Side_B  : Angles;
                                     Side_C  : Angles;
                                     Angle_A : Angles) return Angles is
         Cos_Distance : Args := 0.0;
      begin
         Cos_Distance :=
            Trig_Functions.Cos(Side_B) * Trig_Functions.Cos(Side_C) +
            Trig_Functions.Sin(Side_B) * Trig_Functions.Sin(Side_C) *
            Trig_Functions.Cos(Angle_A);
         return Trig_Functions.Acos(Cos_Distance);

      end Angular_Distance;
```

  Although mathematically correct, execution of this function will sometimes cause constraint errors when the two points are very close together; this is because, in such cases, the `Cos_Distance` calculated may be slightly greater than 1.0, the upper limit of datatype `args`, due to limited precision. When such cases are encountered, the recourse should be to rework the algorithm rather than extend the bounds of the subtype(s) and thus weaken the benefits of constraint checking. An added test may be used to check for this condition:

```
...
    begin
      Temp := (
```

```
        Trig_Functions.Cos(Side_B) * Trig_Functions.Cos(Side_C) +
        Trig_Functions.Sin(Side_B) * Trig_Functions.Sin(Side_C) *
        Trig_Functions.Cos(Angle_A));
    if ... then                -- test for constraint error on Temp
       Cos_Distance := Args(Temp);
       return Trig_Functions.Acos(Cos_Distance);
    else
       ...                     -- Flag the constraint error
    end if;
end Angular_Distance;
```

- *Range checking in subexpressions.* Some Ada 83 implementations constraint-check intermediate as well as final values of expressions. In the example below, a constraint error exception would be raised by some implementations at the point where A and B are added together:

```
type Example_Type is range 0 .. 10;
A, B, C : Example_Type;
      . . .
A := 7;
B := 5;
C := (A + B) / 2;       -- Constraint error could be raised here
```

This problem has been removed from Ada 95 implementations.

- *Minimize type conversions.* In Ada, all type conversions are explicit and may be found in the source code. However, the use of type conversions, and particularly of unchecked type conversions (a bit-for-bit copy without any checks for such problems as mismatched type size), is strongly discouraged. Type conversions partially negate the benefits of strong typing.

  The Ada95 mechanism for change of representation uses subtype conversion to copy from objects with one representation to compatible objects with a different representation. If the objects are simple objects, this mechanism poses few difficulties, except for the case of enumerated types with enumeration representation clauses.

  If the objects to be converted are composite types, this change of representation introduces non-determinism and run-time code, and should not be used. Change of representation for enumerated types is not recommended.

  Type conversion on arrays with dynamic bounds is prohibited. Conversions between general

access types should satisfy accessibility rules. These rules are usually statically determinable, but the interaction with generics makes some cases subject to run-time checks. It is recommended that conversion between different access types be avoided.

Type conversion between access types having different storage pools is prohibited. Subprogram `out` and `in out` parameters, and function results should not include an explicit or implicit type conversion on subprogram call or exit. Assignment and subtype conversions for composite types, where one or both have representation clauses applied, should be by component.

- *Avoid use of unchecked conversions.* A predefined generic library function called `Unchecked_Conversion` is provided by Ada to facilitate interaction with hardware and/or lower level software. However, using this facility may lead to assigning illegal values to an object. This is against the Ada strong typing philosophy and should not be used in safety-critical systems unless absolutely necessary. Documentation of the rationale for each unchecked conversion should be included in the code.

- *Limit use of access objects.* Programs should limit the use of objects declared as access types (pointers) to situations in which there are no better alternatives. In general, such indirection leads to confusion. The problem can be compounded with dynamic allocation and multiple access objects used for the same address as pointed out in Section 2.1.1.

- *Avoid declaring variables in package specifications.* Keeping variable declarations out of package specifications and instead defining subprograms to access the data will result in greater data abstraction and less coupling. This practice can have maintainability benefits as well. The example below demonstrates the point by showing a part of a compiler. Both the package handling error messages and the package containing the code generator need to know the current line number. Rather than storing this in a shared variable of `type Natural`, the information is stored in a package that hides the details of how such information is represented and makes it available with an access routine.

```
package Compilation_Status is
   type Line_Range is 1 .. 2_500_000 ;
   function Source_Line_Number return Line_Range ;
end Compilation_Status ;
-------------------------------------------------------------
package body Compilation_Status is
   -- define Line_Range variable
   function Source_Line_Number return Line_Range is
      -- define function
end Compilation_Status ;
-------------------------------------------------------------
with Compilation_Status ;
package Error_Message_Processing is
```

```
      -- Handle compile-time diagnostic.
end Error_Message_Processing ;
------------------------------------------------------------
with Compilation_Status ;

package Code_Generation is
   -- Operations for code generation.
end Code_Generation ;
```

9.1.4.4 Accounting for Precision and Accuracy

Precision and accuracy generic guidelines apply to Ada. Ada supplies many more features to control the precision and accuracy of calculations, than most other languages.

The following Ada-specific guidelines apply to precision and accuracy:

- *Allow for only the minimum accuracy specified in the program.* Ada enables the users to specify the minimum accuracy of numerical types. This minimum accuracy also specifies the accuracy of arithmetic operations on the types. It does so in a way that depends on information in the type declarations rather than the characteristics of the computer running the program or of the compiler. Thus, a program is obtained that will run with a minimum guaranteed degree of accuracy on any machine for which the program can be compiled.

  When the development hardware or test hardware differs from the target hardware, it is of vital importance to realize that Ada guarantees *minimum* accuracy. Because of their implementation of arithmetic operations, some systems make more efficient use of the machine; therefore, this may provide slightly better accuracy than required by Ada. However, these slight differences may mask small errors that can accumulate during the course of a computation to give significantly incorrect results. That two different machines use the same number of digits in the mantissa of a floating point number does not imply they will have the same arithmetic properties. Therefore, only the *minimum* accuracy should be incorporated into the design and implementation. No safety-system program should depend on an accuracy better than the minimum (SPC, 1989; p 136). These issues should be factored into the design of the software.

- *Use appropriate operators for relational tests.* Relational tests should use <= and >= on real values rather than <, >, = and /= (SPC, 1989; Section 7.2.9).

- *Use Ada attributes for checking of small values.* Ada attributes should be used in comparisons and checking for small values (SPC, 1989; Section 7.2.10). For example:

```
if abs(X - Y) <= Float_Type'small
                  -- Test for absolute "equality"
if abs(X - Y) <= abs(X) * Float_Type'epsilon
                  -- Test for relative "equality"
```

- *Use Ada attributes for checking for special values.* It is important that the code test carefully around special values (SPC, 1989; Section 7.2.11). For example, the following statement should be used for a test around zero:

```
if abs(x) <= Float_Type'small -- Preferred test for value of 0.0
```

9.1.4.5 Order of Precedence of Arithmetic, Logical, and Functional Operators

The generic guidelines for order of precedence apply to Ada. The following are Ada-specific guidelines.

- *Use parentheses.* Arithmetic, logical, and other operations should use parentheses to ensure that the order of evaluation is explicitly stated for operators of different precedence (SPC, 1989, pp 79 - 80). For example:

```
-- Use
Root := ((-B) + Square_Root((B ** 2) - (4.0 * A * C)))/(2.0 * A);

-- instead of
Root := (-B + Square_Root(B ** 2 - 4.0 * A * C))/(2.0 * A);


-- Use
C := (not A) or B ;

-- instead of
C := not A or B ;        -- may be mistaken for "not (A or B)"
```

The reasons for using explicit parentheses is not only to avoid misinterpretation. Absence of parentheses may also cause the results of an expression to differ because an optimizing compiler may alter the order of expression evaluation for operators with the same precedence. Any program that depends upon a specific order of evaluation is considered erroneous. By *erroneous*, we mean that the Ada compiler may not detect the violation, so

the effect of running such a program is unpredictable (Booch, 1983; p 153).

Most expressions in Ada have precise semantics and will yield answers within the error bounds applicable to the type, or raise an exception if the result cannot be represented. Certain implementation freedoms in expression evaluation, however, can affect the outcome of the evaluation of an expression:

- Implementations are allowed to represent intermediate results with extra precision or a wider range of values.

- Implementations are allowed to re-associate operators with operands within a sequence of operations at a given precedence level, even if this leads to an exception where none would have been raised in the "canonical" order. An example of such an expression is

```
A  := B*C/D;
```

  On a typical 32-bit processor, if `B=10_000`, `C=10_000`, and `D=100`, the above expression would raise an exception if `B*C` was evaluated first, but not if `C/D` was first evaluated. On the other hand, `C/D` will truncate and may give less fidelity to the answer.

- Implementations are allowed to suppress exceptions if the result of an expression can be shown to be unused (presumably the evaluation is eliminated as dead code in these cases).

- The precision of the results of some floating point or fixed point operations is not fully specified and may vary among implementations.

Care should be taken, therefore, to ensure that the results of the evaluation of an expression are correct.

In the following example, the addition of `B` and `C` will cause a numeric overflow; therefore, it is vital that the subtraction be performed first.

```
a)  X := A - B + C;      -- Evaluation order may be changed by
                         -- optimizing compiler

b) X := (A - B) + C;     -- Evaluation order certain
```

- *Account for full evaluation in logical expressions.* In Ada, all expressions are fully evaluated even if the final value is known earlier. For example, to evaluate logical expression `X AND Y`, both `X` and `Y` will be evaluated even if the value of `X` is `FALSE`

(making the evaluation of `Y` unnecessary). This may lead to subtle errors, if the legality of the evaluation of `Y` depends on the value of `X`. If the desired effect is not full evaluation (i.e., evaluation will stop as soon as the first not true condition is found), alternative constructs such as `AND THEN` or `OR ELSE` should be used.

9.1.4.6 Avoiding Functions or Procedures with Side Effects

Generic guidelines are applicable. Also see Subsection 3.2.2.3 for Ada-specific guidelines. Some care is required in subprogram calls. Subprogram calls are a well-understood and analyzed area of programming. The major mistake in traditional languages is that the interface contract between the caller and the subprogram itself is insufficient to guarantee that the number, type, and alignment of all parameters is always correct. Ada95's dependency, compilation environment (library) requirements, and strong type checking virtually eliminate all forms of incorrect invocation of a subprogram (Saaltink, 1996).

Another issue is that Ada95 permits the same name to be used with different parameter profiles, or for subprograms with the same name and profile to be imported into one scope. This can cause confusion or misunderstanding for human readers, and may cause tools that parse source code to fail, or to identify the wrong subprogram being called. Ada's type checking rules guard against interface errors by ensuring that each parameter has an appropriate type.

Named parameter associations are useful in cases where some parameters have default values or where there are many parameters. A named association makes the relationship between actual parameters and formal parameters clear, can enhance readability, and facilitates maintenance.

As pointed out previously, a subprogram declaration should be elaborated before a call to it is executed; otherwise the exception `Program_Error` is raised. This situation can arise only if the subprogram declaration appears textually before its body and the call occurs in the intervening text, or if the declaration and body appear in different compilation units. Also, since there is considerable freedom for an implementation to perform actions in different orders (e.g., evaluation of parameters, order of conversion and copy-back), care should be taken that these different orders do not affect the outcome of the program.

For scalar parameters, the constraints on the actual parameter may not match the constraints on the formal. The copy-back of the formal parameter's value to the actual parameter might, therefore, lead to a `Constraint_Error`.

The following constraints on subprogram calls are therefore imposed (Saaltink, 1996):

- A called subprogram body should be already elaborated.
- Side-effects in parameter evaluations should be avoided.

- Formal and actual subtype should statically match, or analysis should show that results are always within range, or the formal should have an indefinite type, in order to avoid constraint errors.

- Overlapping between parameters should be avoided as should overlapping between a parameter and a variable referred to by the procedure (as a global).

If multiple subprograms of the same name are used (whether overloaded, overridden, or homographic), it should be shown that the intended subprogram is in fact called. If a subprogram call propagates an exception, actual parameters that are passed by copy will not be updated. Parameters that are passed by reference may have inconsistent intermediate values if the execution of the subprogram was interrupted and should be used with care.

Side effects in parameter evaluations can be avoided by the use of extra variables that hold the parameter value. The computation of this value can then appear as a separate statement prior to the call. Overlapping and aliasing can be avoided by creating copies of a parameter (unless its type is limited). The use of distinct subprogram names eliminates overloading, overriding, and homographs (except for predefined subprograms), and obviates the need to analyze a call to determine its target.

9.1.4.7 Separating Assignment from Evaluation

Assignment statements (e.g., `extern_var := 100`) should be separated from evaluation expressions (e.g., `if sensor_val < temp_limit`). In Ada the separation can be violated when functions with side effects are used as part of the evaluation. In the example below, the guideline is violated when execution of `func(a)` sets a global variable:

```
if (func(a) < templimit) then
```

The generic guideline for this attribute is, therefore, satisfied when the guidelines to avoid side effects are followed.

9.1.4.8 Control of Class Library Size

This attribute is not applicable to Ada 83, however, it is applicable to Ada 95. Two specific issues that arise with respect to to this attribute are the use of system libraries and the use of child library packages. The former is discussed in Section 9.3.2 and the latter is discussed below.

Child packages have complete visibility to the private part of the parent package in their own "private" declarative region and body. When types, objects, and subprograms are declared in the private part of a package, all child packages have direct access to those entities. This provides three distinct accessibility levels to entities declared by the package:

1. The public view available to units that `with` the package;

2. The subsystem view of the public and private declarations in the package specification, available to the package body, the private part and body of child packages and private child packages; and

3. The package body.

Thus, child packages allow a subsystem to be extended without forcing the basic definition of the system (in the parent unit) to be modified or recompiled. This is a useful facility, but can be abused as described below.

Difficulties with child library packages are as follows:

- Subsystems written, analyzed, and tested can be later extended in ways not intended by the originator. This can be argued to be a benefit of such packages or to be a hindrance. From the position of tractable analysis, the possible addition of child library packages means that significantly more source code should be included in all static analyses of the parent unit. The only control possible here is management controls.

- *Avoid derivations from the partial view of a type (see Section 9.2.3) and avoid the extension of any tagged type outside its declarative region.* Child packages may `with` units that are not descendants of the parent (or an ancestor) but that `with` the parent and thus have public visibility. The child package then would have both views simultaneously, which can cause difficulties, such as common names of object components or primitive subprograms that occur in the private part of the parent package, and in `with`ed units.

9.1.4.9 Minimizing Dynamic Binding

This attribute is not applicable to Ada 83, however, it is applicable to Ada 95.

Subprograms having a parameter of a tagged type and declared in the same package specification as the tagged type are "primitive subprograms" of the type, and can be dispatching. Such a subprogram having a parameter of tagged type `T` can be called with a parameter of type `T'Class;` in this case, the actual subprogram body called is determined by the tag of the parameter.

It should be noted that Ada95's dispatching mechanism offers the implementer the choice of early or late binding; a call to a primitive procedure can be resolved at compile-time or the parameter can be class-wide and the call resolved at run-time. Adherence to this guideline may therefore be monitored easily.

Tagged types can also be used without dispatching. This non-dispatching usage is accomplished by

making all primitive subprograms parameterized by `type'class`. Caution is urged, however, because equality, assignment and attributes are always dispatching for tagged types.

9.1.4.10 Control of Operator Overloading

The generic guidelines apply to operator overloading. Overloading introduces the possibility of simplification (e.g., `Put: (x: T)` procedure for many types `T`), making it easy to write a program by reducing the human memory load. However, overloading can make reading the code more difficult. The advice of the Ada Quality and Style guide seems reasonable: use overloading judiciously, for widely used subprograms that perform similar actions of arguments of different types, and preserve the conventional meaning of overloaded operators. Operator overloading should be controlled by project guidelines and the Ada specific guidelines discussed below. In Ada the following functions can be overloaded:

```
and   or    xor
=     <     <=    >     >=
+     -     &     abs   not
*     /     mod   rem   **
```

Operator overloading can be a benefit to readability and complexity by allowing a single operator to be useful for different data types. An example of operator overloading is shown below. The body of the function would check to see if the sum is less than 360.0 and greater than or equal to 0.0 and, if not, returns the sum modulus 360.0.

```
function "+"(LEFT, RIGHT: Angles_In_Degrees) return Angles_In_Degrees;
```

Ada-specific guidelines for operator overloading are as follows:

- *Order of procedure.* The code should avoid operator overloading when the inherent precedence of the operator is different from that desired.
- *Consistency of semantics.* The code should preserve the conventional meaning of the operators (SPC, 1989; Section 5.7.4).

```
function "*"(LEFT, RIGHT : Matrix) return Matrix;
```

Such a function should define the "*" operator consistent with the expected matrix multiplication function.

Caution is urged in the use of overloading for operators (Saaltink, 1996). Operators are used in

expressions, and tools that analyze expressions usually assume that the operator will have certain characteristics, such as no side-effects. User-defined operators should preserve these assumptions. It is likely that additional annotations will be required for tools that perform static analysis on user-defined functions.

9.2 Robustness

The intermediate attributes for robustness are as follows:

- Controlled use of diversity
- Controlled use of exception handling
- Input and output checking.

9.2.1 Controlled Use of Software Diversity

As noted in Chapter 2, use of diverse software implementations is a design-level decision. A discussion of the factors affecting the use of diversity are beyond scope of this document. The generic attributes and guidelines for both internal and external diversity apply to software written in Ada. However, there are no Ada language-specific guidelines.

9.2.2 Controlled Use of Exception Handling

Exception handling provides for alternative execution paths to handle unexpected and abnormal situations that can be anticipated. The generic guidelines for exception handlers described in Chapter 2 are applicable to Ada programs. The reviewer should be sensitive to the fact that the use of exceptions involves a fairly complex trade-off analysis and that the reviewer may therefore disagree with the implementer's choice about using or not using exceptions.

Exceptions furnish a convenient means for a subprogram to signal a failure, and encourage robust programming by making such error conditions hard to ignore (even if a program fails to check for the error condition, the normal flow of the program is disrupted).

There is a price to be paid for the use of exceptions. There is a run-time penalty in supporting exceptions (handlers must be set up and torn down). The analysis of programs with exceptions is complicated by the many possible paths (there is an "exception path" from any subprogram call to the nearest enclosing `handled_sequence_of_statements`). The timing analysis of programs using exceptions is also difficult (unless full details of the implementation of the exception mechanism are known).

There may need to be extra code to handle the transfer of control inherent in the raising of an exception, if that transfer leaves a master scope that declares an object that needs finalization. Since using objects that require finalization is discouraged, this situation should not arise in a critical system.

The propagation of an exception to a subprogram call can expose the implementation choice as to whether certain parameters were passed by copy or by reference. Any `out` or `in out` parameters passed by copy will not be updated because the subprogram was left abnormally. The use of a specific error value, in contrast, ensures that all `out` and `in out` parameters are updated before the return is made. If exceptions are used, one should be careful to consider the possibility of this implementation dependence.

A number of language-defined checks occur at run time: for example, checks for arithmetic overflows; array indices lying in the index range of the array; division by zero; or storage exhaustion. The failure of any such check raises an exception. In these circumstances, the raising of an exception rather than proceeding with an incorrect value or dangerous operation is a great benefit. For Ada 95, the language rules in Section 11.6 of the ARM give implementers considerable freedom: the exception might not be raised at all, or may be raised earlier or later than specified by the "canonical" semantics. Furthermore, when an exception is raised as the result of the failure of a language-defined check, variables can be left in an abnormal state; and a subsequent attempt to read the value of such a variable is erroneous.

For these reasons, any program for which a language-defined check fails has implementation--dependent behavior and may be highly unpredictable. Any high-integrity program should not fail a language-defined check. This constraint cannot be enforced by the use of subsets (as a check-free subset is unusable), and so should be checked by analyzing the program. There are tools available for assisting this analysis.

The use of explicitly raised exceptions is neither encouraged nor discouraged, but the reviewer should note that the "cost" of using exceptions should be weighed against their advantages. In general, the guidelines stated below apply to explicit error indicators also.

The following sections discuss Ada-specific guidelines.

9.2.2.1 Local Handling of Exceptions

The generic guidelines apply. Exception handlers should be placed as close as possible to the point where the exception was raised. This is because exceptions can be difficult to localize, and it is often desirable to resume normal execution as near as possible to the point where the exception occurred after recovery actions are taken. (SPC, 1989; p 99). Also, if, in propagating the exception to a handler, the scope of a declaration is left, any entity declared by that declaration is destroyed. There can be a run-time cost for this activity, particularly in the case of controlled objects. Leaving a subprogram body by propagating an exception has an effect that depends on the parameter passing mode; any parameters passed by copy are not updated (while those passed by reference might be changed). The following are Ada-specific guidelines.

- *Minimize propagation of exceptions.* Where possible, exceptions should be handled in the subprogram in which they were raised. Automatic propagation of exceptions should be

avoided in a safety-critical application since it is implied and obscures the program logic. Any exception propagation should be intentional, not by default, and should be clearly indicated in comments. Specific guidance on the use of exception handling should be part of the coding practices documentation procedures of the organization or the specific project.

- *Localize handling of predefined exceptions.* The Ada95 ARM and the Ada83 LRM (DoD-STD-1815A) give sufficient freedom to implementers so that in many cases a predefined exception for the same cause can be raised from a number of locations. Thus, if it is possible for the same exception to be raised at more than one point in a program unit, the exception handler for each raising should be different in order to localize the exception. This is shown in the following example.

```
procedure Same_Exception_At_Different_Points is
      Dynamic_Object_A : Pointer_Type;
      Dynamic_Object_B : Pointer_Type;
begin
      begin                  -- isolate first occurrence of exception
            Dynamic_Object_A := new Target_Type;
      exception
            when Storage_Error =>
                  Text_IO.Put_Line("Heap overflow when allocating " &
                                   "A object");

      end;
      begin                  -- isolate second occurrence of exception
            Dynamic_Object_B := new Target_Type;
      exception
            when Storage_Error =>
                  Text_IO.Put_Line("Heap overflow when allocating " &
                                   "B object");

      end;
end Same_Exception_At_Different_Points;
```

9.2.2.2 *Preservation of External Flow Control*

When an exception is raised in a called subprogram declared in a package specification and is thus visible to external subprograms, the particular exception handling to be done frequently depends upon the caller. Therefore, to preserve the flow control, all exceptions that are raised to a calling subprogram should be declared in the same specification as the called subprogram. This makes them visible to the caller (SPC, 1989; Section 9.3.2), as shown in the following example.

```
package Trig_Functions is
```

```
        -- Exception raised when the combination of arguments
        -- input to a function
        -- is invalid.
      Invalid_Arguments : Exception;
        -- Exception raised when an unexpected and unchecked
        -- for constraint
        -- error is raised in any trig function.
      Unexpected_Constraint_Error : Exception;
              .
              .
        -- Function to compute the arc whose tangent is Y/X.
        -- The exception Invalid_Arguments is raised
        -- if both input parameters
        -- are essentially zero.
      function Atan (Y : in Float; X : in Float) return Angles;
              .
              .
package body Trig_Functions is
              .
              .
    function Atan(Y : in Float; X : in Float) return Angles is
    begin
      if abs(Y) < Float'small and then abs(X) < Float'small then
          raise Invalid_Arguments;
              .
              .
    exception
      when Invalid_Arguments => raise;
      when Constraint_Error | Numeric_Error =>
        raise Unexpected_Constraint_Error;
    end Atan;
```

In accordance with the guideline in the previous paragraph, unexpected occurrences of the predefined exceptions that may be raised are handled locally.

9.2.2.3 Uniform Exception Handling

When Ada code raises a defined exception, the exception processing has several courses of action: abandon the execution of the unit, try the operation again, use an alternative approach, repair the cause of the error, initiate alarms, or send messages to the operations personnel (Gall, 1975). The selection of which course of action to take should be determined on a uniform, project wide basis using the results of a safety analysis. Means of assessing and enforcing of exception handling policies should exist.

The following guidelines are suggested for uniform exception handling:

- *Clearly express and document exception handling.* All exception handling should be clearly expressed in code and uniformly documented in the program.

- *Handle predefined exceptions.* Ada has five predefined (non-user-defined) exceptions, `CONSTRAINT_ERROR`, `NUMERIC_ERROR`, `PROGRAM_ERROR`, `STORAGE_ ERROR`, and `TASKING_ERROR`. It is good practice to recognize these conditions explicitly and plan for their resolution in a uniform manner; even error conditions that the programmer believed could never arise may be caught as a predefined exception.

- *Do not raise predefined conditions explicitly.* The predefined conditions should be reserved for their intended purpose. Exceptions that are to be raised explicitly by the application program should be identified using other names.

- *Handle all program-defined exceptions.* If a condition raising an exception occurs, there should be a course of action associated with it (Booch, 1983; p 273).

- *Use exception handling for abnormal events.* Exceptions are just what they are called, and should not be used for normal processing (SPC, 1989; p 96). Exceptions should be used for abnormal or extremely unusual occurrences only. Execution of normal control sequence is abandoned after an exception is raised. The code should contain other provisions to handle normal events without the asynchronous transfer of control by an exception.

- *Minimize side effects.* While some side effects may be inevitable as a result of exception handling, they should be minimized. Critical state data should not be changed during exception processing except to the extent needed to restore mainline processing to the system.

  One side effect of exceptions is that data in a calling program unit may be corrupted. For copy-in and copy-out parameters this presents no problem, as their new values are copied back into the original objects only upon successful completion of the called unit. For program units that change data objects specified by reference parameters or that are global variables, the situation is different. The reader should consider the example of a procedure that changes the values in a large array passed to it as an `in out` mode parameter. If an exception is raised after part of the array has been processed, some of the elements in the original array object will have been processed, and other elements will retain their original values.[21]

  In safety-critical subprograms, data that are to be updated and that are passed to and from the subprogram via reference should be copied into local variables of the same data type, updated in the local variables, and copied into the output parameter objects only upon normal termination. This practice involves sacrificing time and space for safety.

---

[21] Exceptions are yet another reason why global variables should not be modified in subprograms.

- *Avoid use of compiler vendor-specific exceptions.* No exception defined by a compiler vendor can be guaranteed as portable to other implementations whether or not it came from the same vendor. Not only may the names be different, but the ranges of conditions triggering the exceptions may also be different (SPC, 1989; p 144).

  `Package Ada.Exceptions` has not been completely analyzed, and its functional predictability and storage usage are not known. The use of this package is therefore not recommended. Most of the information provided by `package Ada.Exceptions` can be furnished by a user-defined package, and calls to record the desired information can be made before an exception is raised.

- *Use `other` in exception handler definitions.* All conditions associated with exception handling should be well defined; however, `other` should be used and flagged as an unanticipated exception condition.

- *Absence of all failure of language-defined checks should be demonstrated.* For the highest levels of integrity, the failure of any language-defined check should be considered as an erroneous execution, and a demonstration that no such checks fail is recommended. Under these assumptions, suppression of checks has little impact on the predictability of a program and might increase its speed.

  On the other hand, it is usually advisable to employ a variety of techniques to ensure the reliability of a program. Even if exceptions are not expected to occur, there is a possibility that corrupted inputs and transient hardware faults could lead to exceptional conditions, that some of the assumptions on which the analysis was based are not true, or that the analysis was incorrect. There is a possibility that a language-defined check would prevent such faults from causing too much damage before being detected. An outer-level exception handler can then restart the program and attempt to recover from the fault situation. Suppression of checks eliminates this possibility.

9.2.3 Input and Output Data Checking

The generic attributes for input and output data checking are applicable to Ada. Because input and output data checking are handled through the same mechanisms in Ada, this section discusses them together. Also, because packages are Ada95's basic unit of modularity and are thus fundamental to the creation of any Ada program, the significance of packages for Ada input/output and interfaces is discussed here. Further, since private types are a vital element of Ada's support for modularity, private types are discussed in this section.

9.2.3.1 Input/Output Data

- *Never disable automatic checks on input and output data.* Ada automatically checks for certain anomalous conditions on I/O data. One such anomalous condition occurs when the

data are out of the range of the datatype; a constraint exception would then be raised. Another anomalous condition detected automatically is when the index for an array element is out of the array's bounds.

The file management packages provided with Ada compilers provide additional input and output data checking. Package `Text_IO` routines provide not only range checks, but also syntax checks, on I/O values. Packages `Sequential_IO` and `Direct_IO` check input values to ascertain if they can be interpreted as being of specified data types.

In safety-critical systems I/O data should always be regarded as untrustworthy until proven otherwise. The notions that input error checking may not be applicable if the input can be trusted and that output checking may not be necessary if downstream input checking is performed should be viewed with caution. Consequently, the automatic Ada checks on input and output data should never be disabled.

9.2.3.2 Packages

Packages allow for the partitioning of a program into parts that interact using well-defined interfaces. This can facilitate the analysis of a program and enhance its robustness by limiting the interactions among its parts.

High-integrity programs can sometimes be structured so that the code that deals directly with some critical aspect of the system can be encapsulated in a package body. This is ideal, as Ada95's language rules then guarantee that this code is called only using the interface defined in the package specification. Furthermore, any local data used in this critical code is protected from tampering. (Ada95 permits the circumvention of these protections using `'Access` or `'Address` attributes or deriving from a tagged type in the package body. The restrictions imposed in this document prevent or limit such circumventions.)

Packages define three different levels of isolation. Entities declared in the public part of a package specification are visible wherever the package itself is visible. Entities declared in the private part are visible to the package body, and also within any child packages (if the package is a library unit). Finally, entities declared in the body are visible only within the body.

These levels of isolation permit designers and implementers to implement stand-alone service packages, subsystems of cooperating packages, or packages that export all significant items. Subsystems can therefore be built with exactly the proper amount of visibility and security for the systems being designed.

Subprograms declared in a package specification usually have bodies that appear in the package body. This may create a delay between the elaboration of the subprogram declaration and the elaboration of the subprogram body. Any calls to the subprogram between these two elaborations will fail the elaboration_check and lead to Program_Error being raised.

- *Restrictions must be placed on the use of packages* (Saaltink, 1996):

  1) A package body should immediately follow its specification if it appears in a declarative region.

  2) A package specification that is a library unit should include `pragma (Elaborate_Body)`.

  3) Any initializing expressions appearing in object declarations should call only static functions.

These restrictions will minimize the elaboration delay, and reduce or eliminate the chance of failure of the elaboration check.

9.2.3.3 Private Types

Private types furnish a way for a package to make a type available to a program without revealing the definition of the type. Thus, the program outside the package cannot rely on the details of the definition, and the definition can be changed if necessary with minimal impact outside the package.

Package private parts provide two different views of types and variables:

- a public view of types, objects and their primitive subprograms; and
- a private complete view of the type, objects and primitive subprograms.

These two views give rise to some anomalies. The declaration of primitive subprograms of a type in a private part creates the possibility that one could derive from the type and create two callable subprograms with the same name. The one chosen would depend upon which view the caller had of the type. Similarly, for tagged types, it is possible in a public view to extend the record and add a field with the same name as a field in the private view. This makes static analysis much more complex, and, for some combinations of dependency, relationships may create homographs.

It can be useful to make a private type limited if it is represented in a way that makes the language-defined assignment inappropriate. If the type is declared limited, then there is a compile-time check that there are no assignments. For example, an implementation of a private type might use some large array to store data, with each value of the private type being a "handle" indicating which part of the array is used to store the data.

- *Private types may be used only if subject to severe limitations.* For the safe use of private types in the Ada95 environment, all of the following should be prohibited:

  1) Discriminated private types;

2) Private types with discriminated full definition;

3) Private types with unknown discriminants;

4) Tagged private types;

5) Derivation of any kind from a partial view of a type; and

6) Private primitive subprograms of a non-private type.

If a tagged private type is desired, one possible workaround is to move the full definition to the public part of the package.

Instead of deriving from a partial view of a type, it is possible to declare a record having a field of that type. It is then possible to define basic subprograms acting on the record type that make appropriate calls to subprograms declared in the package to act on the private type.

There may be times that private operations on public types are needed, such as to provide calls to child packages that are not available to contexts with only the public view. To provide such functionality, a package inside the private part should be declared and any such subprogram declarations placed there. These subprograms are not primitive operations, and hence will not be propagated to any derivations of the type.

```
package P is
      type T is ...
      procedure Get( Item: in out T );
      ...
      private
      package Inner is
      procedure Adjust( Item:  in out T );
      ...
      end Inner;
end P;
```

In the above example, `procedure P.Inner.Adjust` is available in the body of `P`, and in any child packages of P for any variable of `type P.T`, but is not primitive to `T`, hence will not interfere with possible derivations of `P.T`. If `T` is tagged, there will be no dispatching to `Adjust`.

- *Care should be taken when using limited types.* A variable of a limited type cannot be initialized as part of its declaration or by an explicit initializing assignment; instead, an initializing subprogram should be called.

A child package might have a non-limited view of a type; within this package assignments and comparisons are allowed. These can possibly be used to subvert the privateness or limitedness of a type.

9.3 Traceability

Traceability refers to attributes of safety software that support verification of correctness and completeness against the software design. The intermediate attributes for traceability are as follows:

•Readability
•Use of built-in functions
•Use of compiled libraries
•Use of generics.

Because readability is also an intermediate attribute of maintainability, it is discussed in Section 9.4 Ada-specific guidelines for the other attributes are discussed in the following sections.

9.3.1 Use of Built-In Functions

The generic guidelines have limited capability. The only built-in functions in Ada are those that are Ada operations. These operations may be overloaded. Because Ada does not provide an extensive number of built-in functions, each project builds or acquires (either through reusing or purchasing) additional functions. It should be noted that a separate guideline on the use of compiled libraries recommends that externally developed libraries be acquired as source code.

Externally developed software should be subjected to at least the same degree of developmental control and verification as the project-specific code. This would include assessment of the accuracy, limitations, robustness, and exception handling of the functions. Test cases, procedures, and results of previous testing should also be maintained for these libraries. The test cases should assess behavior for out of bounds and marginal conditions (e.g., negative arguments on a square root routine, improperly terminated strings for a string copy routine, and similar conditions) in the specific run-time environment.

9.3.2 Use of Compiled Libraries

The generic guidelines related to controlled use of compiled libraries are applicable to Ada. The reasons for limiting or avoiding the use of compiled libraries in safety systems are as follows:

•*Lack of visibility.* Libraries can be used to shield the applications programmer from the details of the lower level implementation; however, it is exactly that feature that prevents the applications

programmer from knowing the accuracy, limitations, robustness, and exception handling of the built-in functions. Programmers and designers should be considering how to handle error conditions such as invalid parameters, numerical instability of the calculation, non-convergence of a result, arithmetic overflow, and underflow. These different forms of failure may well require handling in different ways according to the severity of the impact of the error on the calculation. In compiled libraries, the error handling mechanisms may not provide the needed visibility to allow programmers to handle these situations (Tafvelin, 1987). This puts the burden on the reviewer to ensure that error conditions are properly handled.

•*Inconsistency in error handling.* A basic consistency data and control flow for error handling is necessary for developing and maintaining highly reliable systems. However, there is no guarantee that libraries will have consistent methods of handling exceptions.

•*Difficulties during maintenance and upgrades.* As software is maintained and new versions of compilers are acquired, libraries may become outdated.

If compiled libraries are used, then thorough testing and error tracking are necessary as described in the generic guidelines.

9.3.3 Ada Run-time Environment

The Ada RTE plays a critical role in ensuring the timing and correct execution of the compiled Ada code. However, it is not directly accessible by the programmer and falls into the category of built in functions or compiled libraries from that perspective. The reviewer concerns related to testing, error tracking, documentation, and development control described in the previous two sections also hold true for the Ada RTE.

9.3.4 Maintaining Traceability Between Source Code and Compiled Code

For a safety application, it is vital to ensure that the source code in a project baseline corresponds to the compiled object code. Traceability between source and object code is needed to avoid the uncertainty of what versions of separately compiled units are included. However, the support of the Ada language for separate compilation can pose a challenge to this traceability. When possible, the entire source (with the exception of compiled libraries, see Section 3.3.2) should be compiled on one occasion. This is the most authoritative way to establish complete traceability between source and executable.

However, it may not be possible to perform a single compilation because:

1. The source code is too large.

2. To support portability, implementation dependent source code is being placed in separate compilation units from other Ada source code.

3. It may be desirable or necessary to incorporate externally developed components in compiled rather than source form.

- *Separate compilation may be used subject to limitations.* If separate compilation is needed the following additional guidelines apply.

- *Partitioning of compilation.* Only those compilation units required for execution of a compilation undergoing compilation unit should be made visible (using a `with` clause) to each unit, i.e. the `with` clauses should not include superfluous compilation units (Jones, 1988).

- *Use of tools.* Tools should be acquired that maintain the libraries in a sufficiently transparent manner to allow such traceability without the need to compile all the source code at one time.

9.3.5 Minimizing Use of Generic Units

The Ada language includes generic units (packages or subprograms) to enhance reusability. However, their use in safety systems is problematic because they obscure the traceability between source code and executable. They are templates, not packages or subprograms, and it is not immediately clear from reading the source exactly what is running in the executable code. Use of generic units should therefore be minimized (Sanden, 1994).

Generic program units provide a powerful mechanism for parameterizing packages or subprograms by types, objects, or subprograms. This parameterization can have a significant benefit, in that complex algorithms can be written, analyzed, and tested once, then used many times. We note, however, that it can be difficult to test generics, as many aspects of the generic actual parameters can influence the effect of the generic instance. For example, the effect of a generic formal type may be different depending on whether or not the actual parameter is scalar or non-scalar, pass-by-copy or pass-by-reference, tagged or untagged, definite or indefinite, discriminated or undiscriminated, or representation attributes of the actual parameter.

Generics can be implemented in a variety of ways. Some implementations regard the generic unit as a template or "macro" which is expanded out at each generic instantiation. Others generate highly parameterized code that can be used by all instances of the generic. This "code-sharing" implementation can be tricky, in that the parameters can affect many basic aspects of the execution (for example, parameter passing modes, or storage management).

The so-called *contract model* for generics is the notion that the generic formal part of the generic declaration expresses all the requirements on the formal parameters that are needed in the generic

body. In Ada83, this contract was imperfect: certain information about a type (specifically, whether the type is definite) could not be expressed. This had the unfortunate consequence of introducing some uncertainty about the legality of a generic instantiation; an instantiation could be illegal because of the way a generic formal parameter was used in the generic body. In Ada95, most such flaws in the contract model have been eliminated. There are still some situations, however, where the requirements on a generic parameter cannot be expressed in the generic formal part.

Interactions between generics and other language constructs as expressed in the contract model, the ways that a generic unit is instantiated, and the implementation model (macro expansion or code sharing) make program units that use generics difficult to analyze. Some static analysis tools must consider all paths through generics, and require that every instantiation of a generic should be individually validated.

However, generics may be necessary in Ada — particularly predefined generic units. If generics are used, they are subject to the following guidelines.

- *Use only the parameter list for transferring data.* No global variables should be used to supplement the parameter list and used in the bodies of other subprograms. The parameter list should be comprehensive for all intended uses.

- *Document restrictions on parameters.* The use of and restrictions on generic parameters should be identified and documented (Jones, 1988).

- *Do not use shared generics.* Generics should not be used if the implementation uses shared generics.

- *Type re-export should not be used.* The re-export (i.e., derivation from the type in the package specification) of any type passed into a generic as a formal parameter type should not be used.

- *Overridden operations should be actual parameters for a type with overridden primitive operations used as actual parameters.*

- *Instantiate each generic only once for each required capability.* For example, do not instantiate a generic many times with the same type, or instantiate each time a subprogram is called. Each instantiation may add code and data, and may incur run-time cost to elaborate. The best model is to instantiate once at the library level.

- *Instantiation only during initialization.* This guideline was discussed in Section 9.1.1 on predictability of memory management.

In addition, there are a few unexpected features of generics that should be borne in mind if they are used:

- Any constraint imposed in the declaration of a generic formal object of mode in out is ignored; the constraint of the actual parameter's true declaration is used instead (ARM 12.4(12)).

- The primitive operations of a generic formal type are the predefined primitive oper-ations. User-defined overridings of these operations are ignored (ARM 12.5.1 (21)).

If generics are used, it may be necessary to analyze each generic instantiation separately. This may not be simple; the instance is a sort of macro expansion of the generic body; however, the scope and overloading rules are applied before macro expansion.

9.4 Maintainability

This section discusses the Ada-specific attributes of the following intermediate attributes related to maintainability:

- Readability
- Data abstraction
- Functional cohesiveness
- Malleability
- Portability

Base-level attributes and Ada-specific related guidelines are discussed in the following sections.

9.4.1 Readability

The following base attributes are related to readability:

- Conformance to indentation guidelines
- Descriptive identifier names
- Comments and internal documentation
- Limitations on subprogram size
- Minimizing mixed language programming
- Minimizing obscure or subtle programming constructs
- Minimizing dispersion of related elements
- Minimizing use of literals
- Controlled use of renaming

The Ada-specific guidelines associated with these attributes are discussed in the following subsections. It should be noted that the controlled use of renaming is an Ada-specific attribute that was not included in the generic guidelines.

9.4.1.1 Conformance to Indentation Guidelines

The generic indentation guidelines are applicable. The following additional guidelines apply:

- *Data structures.* Indent and align beginnings and endings of data structures.
- *Line Continuation* use different levels of indentation to distinguish between indentations for statements and for line continuation (SPC, 1989, pp. 9-11; DoD-STD-2167A, App. F).

9.4.1.2 Descriptive Identifier Names

The guidelines developed for the generic descriptive identifier names attribute are applicable to Ada. The following additional guidelines apply:

- *Follow project-specific guidelines on naming.* Project specific guidelines on the use of names for variables, type definitions, procedures, functions, records, arrays, slices, exceptions, constants, generic instantiations, access objects, and other identifiers should be developed and followed in each program. The guidelines should also address naming of items in different packages (if applicable), how names change based on scope, and other project-specific considerations.
- *Separate words.* Words in compound names should be separated with underscores as indicated in the following example (SPC, 1989; p. 17)

```
Rads_Per_Second
Core_Temperature
```

- *Use underscores with larger numbers.* Underscores should be used with large numbers to promote readability on numbers (SPC, 1989; p. 20). This is shown in the following example:

```
type Populations is range 0 .. 10_000_000_000;

type Social_Security_Numbers is range 000_00_0000 .. 999_99_9999;
```

- *Use care in abbreviations.* Abbreviations should not be used if they can be misunderstood. For example, `Time_of_Day` should be used instead of `TOD` (SPC, 1989; p 20).

9.4.1.3 Comments and Internal Documentation

The guidelines associated with the generic attributes are applicable. In addition, the following Ada-specific guidelines apply:

- *Relate the code to higher level design considerations.* Explanatory comments should not duplicate the Ada syntax or semantics, but should clarify the coded data structures or process algorithms at a more descriptive level than the code. "Comments should be technically correct and should address a reader who is an Ada programmer" (DoD-STD-2167A).

- *Use blank lines.* Related code such as declarations, loops, blocks, cases, and exception handlers should be grouped, separated with blank lines, and described with Ada comments (DoD-STD-2167A).

- *Identify "escapes" from language restrictions:* Escapes from Ada language restrictions (suppression of type checking, unchecked conversions, use of other languages, etc.) are discouraged in other portions of this chapter. However, if they are used, they should be clearly indicated in the comments together with rationale and impact.

- *Use comments when renaming.* The scope of renaming should be indicated in comments physically adjacent to the renaming statements.

- *Comment exception raising and handling.* Comments should be used to facilitate the tracing between exception raising and handling, and to provide traceability back to design documents where the exceptions and handlers were designed.

- *Identify dynamic memory allocation with comments.* As noted earlier, dynamic memory allocation is not desirable in a safety system. If used, however, there should be comments to identify when memory is allocated and released.

- *Identify tasking with comments.* As noted previously, tasking and intertasking communication poses many safety challenges. Comments should provide traceability to a design, and the design itself should clarify issues associated with timing, intertask communication, and avoidance of the risks associated with tasking.

9.4.1.4 Limitations on Subprogram Size

The guidelines associated with this generic attribute are applicable. There are no additional specific guidelines.

9.4.1.5 Minimizing Mixed Language Programming

The guidelines associated with this generic attribute are applicable. The use of machine-level[22] language or a non-Ada higher-level language should be avoided in Ada program units. The reasons for avoiding other languages are listed below.

1. There is no uniform way to implement machine-level code in an Ada source program, There will be differences in lower-level details, such as register conventions, that would hinder implementation and portability.

2. The problems with employing `pragma INTERFACE` are complex[23]. These problems include pragma syntax differences, conventions for linking/binding Ada to other languages, and mapping Ada variables to foreign language variables, among others.

3. Other languages do not provide a means of expressing low-level machine features in a high-level fashion as well as Ada does (Booch, 1983; p 264).

If use of other languages cannot be avoided, it should be minimized and controlled. The following are Ada-specific guidelines:

- *Isolate and clearly document machine language inserts.* If machine-level code inserts must be used to meet a project requirement, isolate the platform-specific implementations in a separate package. Include the commentary that a machine-level code insert is being used and state what function the insert provides and (especially) why the insert is necessary. Document the necessity of using machine-level code inserts by delineating what went wrong with the attempts to use other higher level constructs (SPC, 1989; p 146).

- *Isolate higher-level language inserts, document the `INTERFACE pragma`, and account for interface limitations*: Subprograms employing the `pragma INTERFACE` should be isolated to an implementation-dependent (interface) package. The requirements and limitations of the interface and pragma `INTERFACE` usage should be clearly documented (SPC, 1989; p 146). As noted above, the conventions used by other compilers are not specified by Ada. Thus, validating the interface and ensuring that it is free from potential interface problems can be a complex undertaking. However, a thorough examination is required for safety significant systems.

---

[22]In Ada the term "machine-level" language is equivalent to "assembly" language.

[23]A subprogram written in another language can be called if all data transfer is via parameters and function results. The `interface pragma` is the mechanism for achieving this.

9.4.1.6 Minimizing Obscure or Subtle Programming Constructs

The guidelines associated with this generic attribute are applicable. There are no additional language-specific guidelines.

9.4.1.7 Minimizing Dispersion of Related Elements

The guidelines associated with this generic attribute are applicable. There are no additional Ada-specific guidelines. In Ada, appropriately designed packages can minimize dispersion of related elements. This is so since a data structure and any subprograms operating on it can be collected in an information-hiding package in such a way as to give other parts of the software controlled access to the data exclusively via a well-defined interface.

9.4.1.8 Minimizing Use of Literals

The guidelines associated with this generic attribute are applicable. The following are additional Ada-specific guidelines.

- *Use constants instead of literals.* The use of constants supports maintainability by assuring that all values referencing a constant are automatically changed by a single change to the constant declaration. The exception to this guideline is that numeric literals may be used in well-established formulae or conversions where such values will not change and where readability will be enhanced by the use of such literals (e.g., in the quadratic equation).

- *Use attributes.* An additional Ada-specific guideline is that Ada attributes should be used wherever possible in place of literals, as indicated in the following example. This practice facilitates the propagation of consistent changes when objects related to the constant are changed.

```
MAX_LINE_LENGTH : constant := 132;
type Lines is array (1 .. MAX_LINE_LENGTH) of Character;
Line   : Lines;
-- Use
for Column in Line'range loop
    if Column = Line'first then
      ...
    elsif Column = Line'last then
      ...
    end if;
-- instead of
for Column in 1 .. 132 loop
    if Column = 1 then
      ...
    elsif Column = 132 then
```

```
        ...
    end if;
```

9.4.1.9 Controlled Use of Renaming

Renaming is frequently used to reduce the length of unwieldy, fully qualified names and to make clear ambiguous or inappropriate names. The renamed identifier can also be an aid to understanding the use of a routine. However, renaming also complicates and obscures the traceability from the procedure or function call to the source code. This makes debugging and maintenance much harder. Renaming of subprograms can cause unintended overloading that the designers, programmers, and maintainers may not realize or fully understand.

The following example (from Mil-Std-1815A) illustrates the problem:

```
function ROUGE return COLOR renames RED ;
function ROT   return COLOR renames RED ;
function ROSSO return COLOR renames ROUGE ;
```

The function RED has been renamed as ROUGE in the first line and ROT in the second. In the third line, the renaming on the first line (ROUGE) has itself been renamed to ROSSO. This renaming makes it extremely difficult to understand where a problem occurs if the function RED needs to be debugged.

The following guidelines can mitigate these problems while preserving the benefits of renaming:

- *There should be only one level of renaming.* A renamed identifier should not be renamed a second time.
- *All renaming should be done in accordance with project-specific conventions.* Project-specific conventions should be developed for variable naming and renaming.
- *Maintain a centralized list of names.* A "registry" of renaming should be maintained for each project. The scope of each renaming should also be clearly indicated in the registry.
- *Ensure that the subtype indication in the renaming declaration statically matches the subtype of the renamed object.* For example, any constraint specified by the subtype_indication of an object renaming is ignored; the constraint imposed by the object's original declaration is used instead.

Similarly, the subtypes of formal parameters in a subprogram renaming declaration may not

accurately reflect the true subtypes of the parameters of the renamed subprogram. The profile of the renaming declaration should be subtype-conformant with the profile of the renamed subprogram.

Renaming a function result might adversely affect the storage requirements of a program, depending on how the implementation manages its temporary storage. Renaming a variable may make occurrences of aliasing harder to detect.

9.4.1.10 Use representation clauses for bit mapping.

In many safety systems, there is an interface to a set of hardware discrete switches that affect the state of the system. Such bit maps are typically stored internally as integers. However, representation clauses and enumeration types can be used to effectively represent this status information in a meaningful way, which facilitates review and also reduces the possibility of coding errors as the following example demonstrates.

```
Type Line_Status_Type IS
      (Valve_1A_Open, Valve_2A_Open, Valve_3A_Open,
       Valve_1B_Open, Valve_2B_Open, Valve_3B_Open)

FOR Line_Status_Type USE
      (Valve_1A_Open     => 2#0000_0001#,
       Valve_2A_Open     => 2#0000_0010#,
       Valve_3A_Open     => 2#0000_0100#,
       Valve_1B_Open     => 2#0001_0000#,
       Valve_2B_Open     => 2#0010_0000#,
       Valve_3B_Open     => 2#0100_0000#);
```

The array should be sorted in strict ascending order. It is better to use a name than a positional association (Cohen, 1986, p. 780).

9.4.2 Data Abstraction

This section discusses Ada-specific data abstraction guidelines for the following attributes:

- Minimization of global variables
- Minimization of the complexity of interfaces
- Use of the Ada package for encapsulating programs and data.

9.4.2.1 Minimization of Global Variables

A global variable in Ada can be declared in the main procedure or in a package specification. Unless the entire program is very small, neither should be used. A variable that needs to remain in existence

and retain its value longer than the execution of a single subprogram should be declared in a package body. The package specification should include those procedures and functions that operate on the variable in the package. Such information hiding ensures that the variables are not updated in unintended ways.

9.4.2.2 Minimization of Complexity of Interfaces

The generic guidelines apply to this attribute. There are no additional Ada-specific guidelines.

9.4.2.3 Use of the Ada Package for Encapsulating Data and Related Programs

The Ada `package` feature was developed to control visibility of names and access to data. As such, it is a highly useful mechanism to prevent inadvertent alteration of data or execution by other programs. Some examples of appropriate use of the `package` construct in safety systems are contained in guidelines elsewhere in this chapter. A full discussion of this topic, however, is a design issue and beyond the scope of this document. It is covered extensively in other publications on the Ada language (Shumate, 1989; Cohen, 1986, SPC, 1989).

The only implementation-specific guideline is that the project programming guidelines and the system design itself should identify standards and conventions for :

- defining interfaces, type definitions, and data structures (including records, arrays and strings) in packages
- organization of compilation units
- use of predefined compilation units (e.g., `SYSTEM` and `STANDARD`).

9.4.3 Functional Cohesiveness

Functional cohesion measures the degree to which a subprogram performs a single, problem-related, well-understood function. The generic attributes relating to (1) a single design level function per subprogram element and (2) each identifier having a single purpose both apply to Ada . There is no additional language-specific guidance.

9.4.4 Malleability

The generic attribute applies to Ada. There is no additional language-specific guidance.

9.4.5 Portability

The generic attribute applies to Ada. From the perspective of safety, the benefits of portability are the adherence to standard programming constructs that yield predictable and consistent results across

different operating platforms. Code that has been designed to be portable will be easier to maintain when it is reused or converted to run on a different platform. The general principle is avoiding use of nonstandard, or "enhanced", constructs specific to a particular compiler by itself or in combination with the target execution platform. Where nonstandard constructs are necessary, they should be clearly identified together with the rationale, limitations, and version dependencies (SPC, 1989; pp. 127-155).

Attributes related to portability, which have been discussed elsewhere, include the following:

- Minimizing the use of built-in functions
- Minimizing the use of machine code and foreign languages
- Minimizing the use of compiled libraries
- Minimizing dynamic binding
- Minimizing tasking
- Minimizing asynchronous constructs (interrupts).

The following are additional language-specific guidelines:

•*Do not use busy loop to suspend execution.* Aside from the fact that a busy loop wastes processor resources, the timing of a standard loop cannot be determined when the code is ported to a different compiler, different machine, or even different operating systems. For example:

```
-- Use
delay 3.74 ;
-- Do not use following because of timing differences
for I in 1 .. 6874 loop
  null ;
end loop ;
```

Also, any knowledge of the execution pattern of tasks should never be used to achieve timing requirements, because of the uncertainty during porting (SPC89, p. 141).

- *Validate assumptions about the implementation of language features when specific implementation is not guaranteed or specified.* For example, there may or may not be a correlation between `SYSTEM.TICK` and package `CALENDAR` or type `DURATION`. Although such a correlation may exist, it is not required to exist (SPC, 1989; p. 141).

- *Avoid the use of package `SYSTEM` constants except in attempting to generalize other machine dependent constructs.* Since the values in this package (and its children) are implementation provided, unexpected effects can result from their use (SPC, 1989; p. 146).

The values of the constants in the SYSTEM package should not be changed. These values should be used only when absolutely necessary, and with extreme caution. The use of any portion should be contained within the body of one or more packages, and not made visible to other portions of the program. Such usage should be scrutinized to ensure that all effects are known and considered.

- *Use only pragmas and attributes defined by the Ada Standard.* The Ada LRM (Mil-Std-1815A) defines the following pragmas: `controlled, elaborate, inline, interface, list, memory_size, optimize, pack, page, priority, shared, storage_unit, suppress, system_name` and the following attributes: `address, base, callable, constrained, count, first, first_bit, last, last_bit, pos, pred, range, size, small, storage_size, succ, terminated, val, value, width`. However, the Ada standard permits an implementer (compiler vendor) to add pragmas and attributes to exploit a particular hardware architecture or software environment. Although potentially attractive, non-standard pragmas and attributes are not only non-portable, their limitations may not be as well understood nor tested as are the predefined counterparts. It should be noted that predefined pragmas and attributes in and of themselves may not be totally portable because of the latitude allowed in their interpretation by compiler implementers.

- *Avoid the direct invocation of, or implementation dependence upon, an underlying host operating system or Ada run-time support system.* Features of an implementation not specified in the Ada LRM will usually differ between implementations. Specific implementation-dependent features are not likely to be provided in other implementations. Even if a majority of vendors eventually provide similar features, they are unlikely to have identical formulations. Indeed, different vendors may use the same formulation for (semantically) entirely different features.

- *Minimize and isolate the use of the predefined* `package LOW_LEVEL_IO`. This package is intended to support direct interaction with physical devices that are usually unique to a given host or target environment. In addition, the data types provided to the procedures are implementation defined. This allows vendors to define different interfaces to an identical device (SPC, 1989; p 152).

- *Restrict and isolate variables of* `type SYSTEM.ADDRESS` *or with the attribute ADDRESS.* These are hardware-specific variables that should be kept in a "maintenance location" in the code.

References

International Standard ANSI/ISO/IEC-8652, *Ada 95 Reference Manual*, Intermetrics, Inc., Cambridge, MA, 1995.

Ada 95 Rationale, Intermetrics, Inc., Cambridge, MA, 1995.

American National Standards Institute/U.S. Department of Defense, *Reference Manual for the Ada Language*, ANSI/DoD-STD-1815A, 1983.

Barnes, J. G., *Programming In Ada*, Second Edition, Addison-Wesley Publishing Company, Menlo Park, CA, 1984.

Booch, G., *Software Engineering with Ada*, The Benjamin Cummings Publishing Company, Menlo Park, CA, 1983.

Cohen, N., *Ada as a Second Language*, Prentice Hall, Englewood Cliffs, NJ, 1986.

Gall, J., *Systematics: How Systems Work and Especially How they Fail*, The New York Times Book Company, New York, NY, 1975.

Gottfried, R. and D. Naiditch, *Using Ada in Trusted Systems, Proceedings of COMPASS 93*, May, 1993, National Institute of Standards and Technology, Washington, DC, 1993.

Hutcheon, A., et al., *A Study of High Integrity Ada*, (UK) Ministry of Defense contract: SLS31c/73 Language Review, Document Reference SLS31c/73-1-D, Version 2, 9 July 1992.

Jones, S, K. Mitchell, M. J. Mardesich, et. al., *BCAG Digital Avionics Ada Standard,* Boeing Company, Document No. D6-53339, November, 1988.

Kernighan, B. and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill, New York, 1974.

Page-Jones, M., *The Practical Guide to Structured System Design*, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1980.

Pyle, I., *Developing Safety Systems: A Guide Using Ada*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Sanden, B. I., *Software Systems Construction with Examples in Ada.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

Saaltink, M. And Mitchell, S., *Ada95 Trustworthiness Study: Guidance on the Use of Ada95 in the Development of High Integrity Systems Version 1.0*, (Canada) Department of National Defence contract W2207-5-RC02/01-SV, Document No. TR-96-5499-04, September, 1996.

Software Productivity Consortium (SPC), *Ada Quality and Style Guidelines for Professional Programmers*, Van Nostrand Reinhold, New York, NY, 1989.

Software Productivity Consortium (SPC), *Ada95 Quality and Style: Guidelines for Professional Programmers (Working Draft Version) Version 00.02.03*, Software Productivity Consortium Report No. SPC-94093-CMC, Herndon, Virginia, May 1995.

Tafvelin, S., ed, *Ada Components: Libraries and Tools*, Cambridge University Press, Cambridge, MA, 1987.

U.S. Department of Defense, *Defense Systems Software Development*, DoD-STD-2167A, Appendix D, 1 August 1986..

10 PLC Structured Text

This chapter discusses the applicability of the generic language attributes defined in Chapter 2 to the PLC Structured Text language, as defined by the IEC 1131-3 PLC language specification (IEC, 1993). Structured Text (ST) is a high-level, block-structured, text-based language for PLCs. It is similar to Pascal, but has a number of important differences derived from its PLC application. ST programs are executed repetitively in a PLC scan, and operate on a fixed PLC memory map-I/O base. The majority of ST implementations conform to the IEC 1131-3 ST specification[24]. Additional information on the general characteristics of ST are provided in Appendix A.

10.1 Reliability

Reliability of software is based upon the capability of a software component to perform its programmed tasks under stated conditions for a specified period of time. Reliability depends upon the runtime predictability of Memory Utilization, Control Flow, and Timing. ST- specific guidelines based on these concepts follow.

10.1.1 Predictability of Memory Utilization

The key element in predicting memory utilization is to avoid the use of dynamic memory allocation. The ST language, like Ladder Logic, contains no keywords or functions that allow dynamic memory allocation to take place. The memory management scheme is fixed and deterministic, resulting in a predictable memory management scheme defined prior to runtime.

10.1.2 Predictability of Control Flow

Predictability of control flow is the capability to determine easily and unambiguously what path (i.e. what set of branches and in what order) the program will execute under specified conditions.

10.1.2.1 Maximizing Structure

The generic guidelines are not applicable. ST does not support a GOTO or other keyword with an unconditional jump capability.

---

[24]There are several existing proprietary PLC high-level text-based languages that were released prior to the IEC 1131-3 specification, among them are Télémécanique's Literal Language, and TI/Siemens' Special Function Program language for the 565 series of PLCs. These languages are not in the scope of this report; while they influenced the IEC specification, they are regarded as separate languages for the purposes of this document.

10.1.2.2 *Minimizing Control Flow Complexity*

The generic guidelines are applicable. Complex control flow makes the program difficult to understand and maintain. In addition, it could be a cause of unpredictable runtime behavior. The following are specific guidelines:

•*Use the* `CASE` *construct.* In safety systems, the `CASE...OF` construct should be used to replace multiple `IF ... ELSIF ... ELSIF ... ELSE ... END_IF;` statements if possible. In the following example, `TEST_VAL` is the only term used for evaluation.

```
IF    TEST_VAL = 0 THEN DISPLAY := TC1_TEMP;
ELSIF TEST_VAL = 1 THEN DISPLAY := MOTOR_SPEED;
ELSIF TEST_VAL = 2 THEN DISPLAY := FLOW_RATE;
ELSE
   DISPLAY := 0;
   DISPLAY_ERR := 1;
END_IF;
```

should be replaced by:

```
CASE TEST_VAL OF
   0:  DISPLAY := TC1_TEMP;
   1:  DISPLAY := MOTOR_SPEED;
   2:  DISPLAY := FLOW_RATE;
ELSE
   DISPLAY := 0;
   DISPLAY_ERR := 1;
END_CASE;
```

•*Define Defaults.* When using the `CASE ... OF` statement, the `ELSE` clause should always be explicitly defined, as in the example above.

•*Minimize Use of the* `EXIT` *keyword to exit iterative constructs.* Usage of `EXIT` complicates the control flow structure. The `EXIT` keyword provides a mechanism for multiple exits from a loop, the practice of which complicates the control flow structure and is generally considered undesirable among practitioners of structured programming.

10.1.2.3 Initialization of Variables Before Use

The generic guidelines apply to internal or retentive variables, but not to variables associated with field inputs (where the initial values take on the state of the input devices). Input variables should not be initialized by an ST program prior to use, as they have external meanings which should not be overridden.

Internal variables, or variables associated with output devices, have explicit default values in accordance with the IEC 1131-3 specification. However, variables of this type should be explicitly assigned initial values when declared, for purposes of clarity. Declared CONSTANT variables must be assigned values at declaration time.

The IEC 1131-3 specification also defines RETENTIVE variables, which will hold their values over a PLC power cycle. These values should also be initialized by the program to their correct cold start values, but it should be noted that the initial values of retentive variables during a warm start condition will be their values at the time of PLC shutdown, and not the initial values as shown in the program, unless otherwise modified by the program. This should be noted in the documentation. Variables can be initialized on a warm start as well. Examples of ST variable declarations following these guidelines follow:

```
VAR
   (* Note: for exposition only – not to be used in safety software *)
   (* Variable names rather than locations should be used *)
   (* See Reliability Guidelines *)
   AT %IX6.2: BOOL;   (* Direct assignment, no variable name *)
   AT %QX5.3: BOOL:=0;(* Direct assignment, no variable name *)
         (*     Output variable explicit initialization:       *)
   LIMIT1: BOOL AT %IX6.3;       (* Input, no initial value *)
   ACT1  : BOOL AT %QX5.4 := 1;  (* Initial Value ON *)
END_VAR

VAR RETAIN  (* Retentive Variables *)
   FOO: ARRAY[0..5] OF INT := 0,1,2,3,4,5;
END_VAR

VAR CONSTANT(* Constant Variable *)
   PI: REAL := 3.1415926536;
END_VAR
```

10.1.2.4 Single Entry and Exit Points in Subprograms

The generic guidelines apply. The subprogram types defined by IEC 1131-3 for all the languages in the specification are the Function, which returns a single value, and the Function Block, which returns one or more values. These may have but a single entry point when written in ST. ST allows a Function or Function Block to have multiple exit points, by use of the RETURN keyword, however. The use of the RETURN Keyword to provide multiple exit points from a function of function block should be avoided wherever possible.

10.1.2.5 Minimization of Interface Ambiguities

The generic guidelines apply as indicated below. Interface errors account for a large portion of coding errors. An example of such errors is reversing the order of arguments when calling a Function or Function Block. A coding style that can reduce or eliminate the probability of misusing an interface enhances safety. The following techniques can be used to reduce interface ambiguities in ST:

•*Order Parameters so that Different Data Types are Alternated.* This reduces the chance that two adjacent parameters may be placed in an incorrect order. This arrangement is shown in the example below. Care must be taken not to cause undue confusion by reshuffling Function Arguments. Judicious use of structures may reduce the number of function arguments by grouping together several items of similar kind.

```
    (* Example Function Declaration *)

FUNCTION FOO

   VAR_INPUT
      PARAM1: BOOL;      (* Note alternation of param types *)
      PARAM2: INT;
      PARAM3: BOOL;
   END_VAR

   (* Body of Function *)

END_FUNCTION
```

•*Maintain consistency in parameter ordering.* Function Blocks in ST can be called with their parameters in any order, and not all parameters need be specified in the function block invocation. If parameters are not specified when a function block is called, the previous value is used, or the

initial value if the function block has not been previously called. So, for a function block BAR with input variables defined as follows:

```
FUNCTION_BLOCK BAR
   VAR_INPUT
      IN1:BOOL := 0;      (* Use more descriptive names in practice *)
      IN2:INT  := 0;
      IN3:REAL := 1.0;
   END_VAR
```

the following calls of this function block are legal in ST:

```
BAR(IN1:= 0, IN2:=23, IN3:=47.37);     (* All inputs in order *)
BAR(IN3:=23.39, IN1:=1, IN2 :=12);     (* Order changed       *)
BAR(IN1:= 1);                          (* Only IN1 specified  *)
```

This represents a tremendous potential for confusion in the interface to a function block. Therefore, it is recommended that all ST Function Block invocations in safety-critical systems assign all of the input variables for that function block in the order of their declaration in the function block definition. Exceptions to this should be noted and documented.

•*Test the validity of input arguments at the beginning of a routine, and test the validity of the results before exiting*. Such testing is important for avoiding errors that can compromise the integrity of the system. Range checking inside a Function or Function Block is preferred.

•*Avoid the use of Direct PLC I/O or Memory references.* ST, as all languages defined in IEC 1131-3, allows reference to PLC I/O via either direct reference, or via declared identifiers. I/O references are assigned to Identifiers in the variable declaration blocks of programs. As direct I/O or memory references do not have to be declared, and can result in side effects during program execution, a program that uses this construct can be difficult to understand, debug, and maintain. Use of the Direct Reference facility in ST, or any other IEC 1131-3 language, should be therefore be avoided in anything other than assignment of Declared I/O. It should be noted that the IEC 1131-3 specification forbids the use of Direct I/O and memory references in functions and function blocks, thus avoiding a serious safety problem.

10.1.2.6 Use of Data Typing

The generic guidelines apply. The following measures should be taken to reduce data typing errors:

•*Minimize the use of Type Conversions, and eliminate implicit type conversions.* The ST specification allows the use of overloaded built-in functions/function blocks, that is, functions that will operate on input data elements of various types within a generic type designator. For instance, an overloaded addition function on generic type ANY_NUM can operate on data of types LREAL (64 bit floating point), REAL (32 bit floating point) , DINT (32 bit integer), INT (16 bit integer) , and SINT (8 bit integer). Usage of these overloaded function blocks with mixed input/output data types should be avoided.

> IEC 1131-3 does not allow user-created overloaded functions/function blocks, so this is not an issue. Whereas the IEC 1131-3 specification allows for language extensibility, a potential extension that allowed the user to create overloaded functions or function blocks would be directly contrary to existing specification language, and is currently rejected by existing IEC 1131-3 compliance test suites.

•*Avoid the use of mixed-mode operations.* Operations using multiple data types should be avoided. If such operations are necessary, they should be clearly identified and described using prominent comments in the source code. Explicit use of type conversion functions should be made, if practical, to make the designer's intention clear.

•*Use a single data type in evaluations and relational operations.* Expressions involving arithmetic operations or relational operations should have either a single data type or the proper set of data types for which conversion difficulties are minimized.

•*Avoid mixing signed and unsigned variables.* Mixing signed and unsigned variables in arithmetic and logical operations raises safety concerns and should be avoided in safety systems. Explicit use of ST type conversion functions should be used to make the designer's intentions clear.

10.1.2.7 Precision and Accuracy

The generic guidelines apply. Safety software must provide adequate precision and accuracy for the intended application. At the same time, the software must also tolerate the inconsistencies from operations on floating point numbers. The following are specific guidelines for ST:

•*Use the LREAL Data Type for floating point, where available.* The LREAL (64 bit floating point) should be used for floating point calculations in safety systems, when available. The REAL Data type should be avoided because it may not provide adequate precision and accuracy.

•*Account for floating point properties in relational operations.* The equality comparisons on floating point numbers should be avoided in safety systems since the machine representation of floating point numbers may lack precision and may have a small residue error. Inequality comparisons should be utilized and equality comparisons should be avoided on floating point numbers. The following example illustrates the potential problems:

```
VAR
  VALUE: LREAL;      (* Temporary variable *)
END_VAR
...
IF VALUE = 0.0 THEN
  (* Do something *)
END_IF
```

The condition `VALUE = 0.0` in the above example is likely to be false because of rounding errors, even if the value is expected to be zero. The condition should be modified as follows:

```
VAR CONSTANT
  LREAL_TOLERANCE: LREAL := 0.00001;
END_VAR

VAR
  VALUE: LREAL;      (* Temporary variable *)
END_VAR
...
IF(VALUE <= (0.0 + LREAL_TOLERANCE)) AND
 (VALUE >= (0.0 - LREAL_TOLERANCE)) THEN
 (* Do something *)
END_IF
```

•*Account for truncation in integer operations.* If a floating-point arithmetic operation can generate truncation and rounding errors, integer arithmetic may generate such errors more often. Integer truncation errors are generated by division. In ST, as well as the other IEC 1131-3 languages, the results of integer divisions are always truncated toward zero. Truncation errors can cause safety concerns when the results with truncation are used in comparisons and conditions for control decisions. Therefore, a rounding-off technique should be utilized. As ST conversions from `LREAL` to `INT` and any variety of `INT` are automatically rounded to the nearest integer by the type conversion functions, a typical method might be to perform the division calculation with `LREAL`s, and then convert the result to an `INT`.

•*Ensure that arithmetic conversion produces a result that fits in the allocated memory location.* When it is necessary to use one of the ST type conversion functions, care must be taken to ensure that the converted value is representable in the shorter data type. For example, attempting to convert a REAL value of 12,341.56 to an SINT (8 bit signed integer) would result in an error.

10.1.2.8 Order of Precedence of Arithmetic, Logical, and Functional Operators

Generic guidelines apply. While the order of precedence of operators in the ST Language is similar to that of equivalent operators in C, there are some differences. Developers and reviewers may make incorrect precedence assumptions when the precedence is not explicitly indicated via parenthesis, particularly in complex expressions. The following specific guidelines apply:

•*Use parentheses in Boolean operators.* The Boolean operators have the lowest precedence of all the operators.

•*Use parentheses in comparisons and conditions.*

10.1.2.9 Separating Assignment from Evaluation

ST does not allow assignment inside an evaluation expression. The generic guidelines are not applicable.

10.1.2.10 Proper Handling of Program Instrumentation

No application support is necessary for instrumentation in ST programs. PLCs generally provide a development environment that allows debugging during real-time program execution, and IEC 1131-3 requires that such support facilities be present. However, the specification allows for variability, and the specific features of this debugging environment, as well as its effect on real-time response, depend on the make and model of PLC and its associated programming software. Some of the tools often included are on-line monitoring of program variables, collection of time histograms, insertion of break points, and single-step program execution. Some PLC systems allow break points during the I/O process.

The following specific guidelines apply to the use of this environment:

•*Do not modify running programs.* Most PLCs also provide a facility that allows the modification of the PLC program while it is being executed. However, the safety consequences of misuse of this feature can be severe. Not only can a programmer accidentally introduce errors into a running PLC program by using this feature, but the added communications load and system overhead associated with the program change transfer itself can cause problems with the real time response of the system.

•*Remove all development environment-related execution control commands.* Execution control commands for debugging including breakpoints, single step branches, and other features should be removed prior to downloading the operational version of the software to the PLC. Record of an explicit check for this step should be made and verified during the review process.

•*Remove all development environment-related internal monitoring commands:* Internal program monitoring commands such as watchpoints, snapshots, and data collection for histograms should be removed prior to downloading the operational version of the software to the PLC. Record of an explicit check for this step should be made and verified during the review process.

10.1.2.11 Control of Class Library Size

ST, as all IEC 1131-3 defined PLC languages, does not support classes or objects, and the generic guidelines are not applicable.

10.1.2.12 Minimizing Dynamic Binding

Dynamic Binding is not a feature supported by IEC 1131-3 languages. The generic guidelines are not applicable.

10.1.2.13 Control of Operator Overloading

ST does not allow the programmer to overload operators or functions, therefore the generic guidelines are not applicable. However, built-in functions and function blocks provided by the PLC manufacturer may be overloaded. Source code and documentation are usually not provided with such functions, and thus, if such functions have to be used, extensive testing should be performed to verify correct and continuous operation over the anticipated input domain for each data type used in such blocks.

10.1.3 Predictability of Timing

Control flow timing issues in PLC programs stem from the nature of the PLC program scan and I/O scan. As the IEC 1131-3 PLC specification does not address the issue of the relationship between these two concepts, it must be assumed that the same level of variety exhibited in current PLC implementations will continue to be exhibited in IEC 1131-3 compliant implementations. A survey of current implementations of ST shows this to be the case.

The two broad categories of timing issues, then, are 1) ensuring that the maximum program scan time is not exceeded, and 2) ensuring that field I/O data is appropriately 'fresh' before making control flow decisions based on it. These are discussed below.

10.1.3.1 Ensuring Maximum Program Scan Time not exceeded

ST programs, like all PLC programs, are executed repetitively in what is called the PLC scan. PLCs

contain a watchdog timer to determine if the maximum allowable time for a program scan has been exceeded, and will typically halt the execution of the system if this time is exceeded. These program halts may be deleterious in a safety-critical system. Depending upon the implementation, this Watchdog Timer may have either a fixed or modifiable value.

Since ST allows interative constructs (i.e. `FOR..TO..BY ... DO ... END_FOR`, `REPEAT ... UNTIL ... END_REPEAT`, and `WHILE ... DO ... END_WHILE`), ST programs must in particular be analyzed to ensure that this maximum PLC program scan execution time is not exceeded for all possible worst-case conditions (i.e., maximum iterations on all loops). Use of these constructs should be minimized to the extent possible.

10.1.3.2 Ensuring Currency of Field I/O Values

The order of program execution should ensure that the variables that are used in calculations or control flow decisions are current values of inputs or results.

The types of timing issues that should be verified in the audit or review of a real-time PLC system depend strongly on factors that are specific to the methods used by various PLC operating systems for scanning the real world I/O. Generally speaking, these methods may be classed into four categories, as follows:

1. *Immediate I/O Update.* PLC has no separate I/O Scan - I/O is updated as required by each program statement.

2. *Asynchronous I/O Update.* PLC I/ O scan occurs asynchronously from PLC Program scan. Both this method and the Immediate I/O Update method allow the value of a real-world input to vary in the course of execution of a single PLC program scan.

3. *Captured Asynchronous.* PLC I/O scan occurs asynchronously from the PLC Program Execution scan, but input and output values are captured in a buffer to eliminate the possibility of variance of input values during a single Program scan.

4. *Synchronous.* PLC I/O scan is fully synchronous with the Program Execution scan.

10.1.3.3 Accounting for Sensor and Actuator Latencies

PLC systems vary widely in the input latency (delay time between when a sensor value changes and when the new status is made available to the PLC Program), and the related output latency (delay time between the command from the PLC program to change the status of an output and the actual change in status of the actuator device). Depending upon the sensor/actuator types used, the type of I/O scan/program scan relationship as mentioned above, and the networking for remote I/O drops, these latencies may range from over 500 msec to less than 2 msec.

Given these differences, both the PLC programmer and program auditor should be explicitly aware of the nature of the PLC system used for a particular application, and that this information should be explicitly stated in the PLC program documentation. Timing issues that may need to be reviewed in PLC systems include:

•The impact of noise on latencies

•The impact of changes in variable state during the execution of a single program scan (on systems where this is possible)

•The variance and upper limits on input and output latencies — and what may be necessary to ensure that upper limits are known.

10.1.3.4 Minimizing the Use of Tasking

The generic guidelines are partially applicable. The tasking model for PLCs as defined by the IEC 1131-3 specification, and hence applicable to ST programs, is designed explicitly to eliminate timing uncertainties, and hence should be utilized in safety-critical systems, with the appropriate precautions. The following are specific guidelines:

•*Associate time-critical programs with PLC tasks:* ST programs, like all PLC programs, are executed repetitively. Once the end of the program is reached, and typically after some system overhead (I/O Updates, System Health status checks, etc.), control is returned to the beginning of the program. This cycle is known as a PLC scan. The response time of the application determined by the periodicity and repeatability of this scan. Association of the ST program to an IEC 1131-3-defined periodic task assigns a particular periodicity and priority to the execution of the ST program. According to the IEC 1131-3 specification, if the ST program is not associated with a periodic task, it will be executed as the lowest priority on the PLC system. Because only tasks with assigned periodicities have deterministic scan times, the response time can not be known for programs not associated with such tasks.

•*Ensure that utilization limits are not exceeded.* PLC utilization should be less than the manufacturer's specification for maximum allowable utilization based on worst case execution times. For example, if an ST program is assigned to a task with a 30 msec periodicity, but can take 30 msec or even longer to execute, there there will be a significant problem. Normally, the PLC will shut down upon repeated scan time overruns. The IEC 1131-3 specification requires that manufacturers of IEC 1131-3 compliant PLC systems provide tools for performing this type of utilization analysis. Such tools should be used to determine the worst case condition, the results checked against actual system performance, and the results documented. This should be accomplished even if the PLC program consists of but a single task.

•*Avoid multiple task periodicities:* If possible, all PLC programs should be assigned to the same task periodicity — particularly in PLCs without support for local variables. There are two reasons for this guideline:

(1) The potential for programming errors in accessing a global data base from tasks of differing periodicities is quite high due to the difficulty of considering all cases of sequencing and worst case timing.

(2) Rising-edge and falling edge detection data can be lost. In many PLC systems, rising and falling edge detection of I/O points is calculated from the current status of the I/O point compared to the saved status of that point on the last I/O scan of the servicing task. So, for example, an input serviced by a 10 msec task will have a rising edge condition for a single 10 msec scan after coming on. If a rising edge instruction on this point is used in a 50 msec task, the probability is 0.2 that the rising edge will not be seen by this instruction in the 50 msec task, as this instruction is executed at 50 msec intervals, and the actual condition is true for only 10 msec. This uncertainty is not appropriate for safety systems.

The decision whether or not to use the IEC 1131-3 multitasking features in a particular PLC program must balance the need for explicitly controllable response times with the potential for programming errors due to overutilization of the CPU and update of a global memory pool at different periodicities.

10.1.3.5 Minimizing the Use of Interrupt Driven Processing

The generic guidelines apply — the use of interrupt handling in ST should be avoided if possible. The IEC 1131-3 specification contains no explicit discussion of interrupt processing, and the ST portion of this specification does not include any keywords or standard functions for handling interrupts. However, an IEC 1131-3 compliant system may implement interrupt handling as an extension to the languages as specified. In general, interrupt processing should be handled through the PLC features supporting periodic tasking, watchdog timers, diagnostics, and fault routines.

10.1.3.6 Avoidance of Self-Modifying Code

The generic guidelines do not apply. ST provides no language facilities that would allow the possibility of self-modifying code. The reader should note that this is not the same as the guideline against on-line modification of code pointed out in the section on instrumentation.

10.2 Robustness

Robustness refers to the capability of the software to survive off-normal or otherwise unanticipated conditions. Since unanticipated events can happen during an accident or excursion, it is vital for a safety system to survive an accident and continue working. This section discusses the topics of exception handling and I/O checking in relation to robustness.

10.2.1 Exception Handling

The IEC 1131-3 specification, and the ST language portions contained within it, make no reference to explicit exception handling constructs. Exception handling in PLCs has traditionally been done via the PLC operating system's health monitoring routines or its fault routines, either manually or automatically. In fact, the IEC 1131-3 specification acknowledges that the nature and type of error-handling procedures are outside of the requirements for compliance with the specification (IEC 1131-3, section 1.5.2) It is anticipated that IEC 1131-3 compliant systems will continue to provide this type of functionality as allowable extensions to the language system. An examination of current IEC 1131-3 compliant PLC programming systems indicates this to be the case.

10.2.1.1 Use of System Health Monitoring

PLC systems provide System Health Monitoring information to the application program, usually in the form of designated internal bits and words located in a status file. Information may include execution times for particular tasks, I/O update information, I/O operating status information, battery state information, and other information used to determine the operating status of a PLC system. The nature, extent, and type of this information is not covered by the IEC 1131-3 specification. Safety programs should make use of the available System Health information as part of the functionality for a particular application.

10.2.1.2 Use of Fault Routines

The IEC 1131-3 Programmable Controller Language Specification does not mandate that a compliant PLC system contain a designated hardware fault routine that will execute upon detection of one of a range of fault conditions, similar in concept to the Fault Routine available in current Allen Bradley PLCs (Allen Bradley, 1991). However, a non-periodic task could be set up to trigger off appropriate internal diagnostic information, to handle particular types of errors (fall-back modes on I/O failures, warnings to operator interfaces, etc). Usage of such a construct should be considered in the design phases of programming a system of this kind.

10.2.1.3 Watchdog Timer

A note in section 2.7.2 of the IEC 1131-3 specification (Tasks) states that "The manufacturer shall provide information to enable the user to determine that all (task) deadlines will be met in a proposed configuration". However, the specification does not mention any form of checks to determine if these deadlines are met at runtime, since this falls into the error-checking category of constructs not covered by the document.

Many PLCs with an IEC 1131-3 compliant multitasking capability implement a watchdog timer for each periodic task. Typically, these systems will ignore a single task overrun and flag that an overrun has occurred, but will shut down the system upon N consecutive task overruns, where the value of N will vary from system to system. Where available, ST programs should monitor the task overrun flag, and react appropriately. It is not appropriate for a safety-critical system of this nature to exhibit regular task overruns—this should be looked for specifically during the check-out phases of development.

10.3 Traceability

Attributes specifically related to traceability include the use of built-in functions and the use of compiled program libraries.

10.3.1 Use of Built-in Functions

The generic guidelines are partially applicable. The IEC 1131-3 specification defines a set of standard Function Blocks callable from any of the 5 specified languages, including Structured Text. These, as well as any others that may be included by a particular implementation of ST, have been developed for a real-time control environment. However, for safety critical applications, it is necessary to extensively test such functions to ensure that they run to successful completion over the entire anticipated input range.

Unlike user-defined functions, built-in functions in IEC 1131-3 may be defined to be overloaded—i.e., they may accept any valid type for input parameters. For instance, the ADD function block may accept integers or floating point variables as inputs or outputs, performing implicit conversions in the process of performing the calculation. Use of this overloading capability should be minimized in safety-critical systems. As noted previously, testing must be performed over the entire input range for each data type used in such systems.

10.3.2 Use of Compiled Libraries

Compiled libraries are libraries of routines written and compiled by an entity other than the development group, in addition to the built-in functions described above. Since there is no consensus yet on anything other than the source representation for ST, there are currently no third party ST compiled libraries available. At the time these do become available (if any) the generic guidelines apply.

10.4 Maintainability

The software maintainability attributes in this section are limited to those affecting safety. These include:

•Readability
•Modular Design
•Functional Cohesiveness
•Malleability
•Portability

10.4.1 Readability

Readability allows software source code to be understood by qualified personnel other than the writer. Readability is very important, as almost all programs are modified or debugged by someone other than the original author at some time during the life of the program. Readability should in large measure be based on project-specific guidelines; however, the following are guidelines specific to ST.

10.4.1.1 Conformance to Indentation Guidelines

ST is a relatively new language, and unlike C, there have not arisen different traditions for indentation styles and the associated controversies. ST programming environments typically incorporate 'smart editors', which enforce a particular indentation style determined by the manufacturers of the system. In the event that such an editing system is not being used on a particular project, it is recommended that the indentation style described below be used. It should be noted that this style is at variance with the style given in the ST programming examples in the IEC 1131-3, especially in the first requirement.

•A maximum of one ST statement per line should be used

•Comments should have the same indentation as the objects described, or should appear on the end of the same line as the statement

•Branching Constructs should be indented

•Looping Blocks should be indented

•Variable declaration blocks should be indented

•Contents of `PROGRAM` or `FUNCTION_BLOCK`s should be indented.

An example of a properly indented and formatted ST Function Block follows.

```
    (* STACK_INT *)

FUNCTION_BLOCK STACK_INT(* Routine for a 128 Integer Stack *)

   VAR_INPUT
      PUSH, POP: BOOL R_EDGE;  (* Basic Stack Operations *)
      RESET:     BOOL;         (* Reset Control          *)
      IN:        INT;          (* Input Integer          *)
      N:         INT;          (* Max depth after Reset  *)
   END_VAR(* INPUT *)

   VAR_OUTPUT
      EMPTY:    BOOL := 1;    (* Stack Empty            *)
      OVRFLO:   BOOL := 0;    (* Stack Overflow         *)
      OUT:      INT  := 0;    (* Top of Stack Data      *)
   END_VAR(* OUTPUT *)

   VAR
      STK:      ARRAY[0..127] OF INT;  (* Internal Stack *)
      NI:       INT := 128;    (* Storage for N on Reset *)
      PTR:      INT := -1;     (* Stack Pointer          *)
   END_VAR(* INTERNAL *)

   (* Function Block Body Starts Here *)

   IF RESET THEN
      (* Actions On Reset *)
      OVRFLO:= 0;
      EMPTY := 1;
      PTR   := -1;
      NI    := LIMIT(MN:=1,IN:=N,MX:=128);
      OUT   := 0;
   ELSIF POP & NOT EMPTY THEN
      (* Actions on Stack Pop *)
      OVRFLO:= 0;
      PTR   := PTR - 1;
      EMPTY := PTR < 0;
      IF EMPTY THEN
         OUT := 0;
      ELSE
         OUT := STK(PTR);
      END_IF; (* EMPTY *)
   ELSIF PUSH & NOT OFLO THEN
      (* Actions on Stack Push *)
      EMPTY  := 0;
      PTR    := PTR + 1;
```

```
        OVRFLO := (PTR = NI);
        IF NOT OVRFLO THEN
           OUT       := IN;
           STK(PTR)  := IN;
        ELSE
           OUT := 0;
        END_IF; (* OVERFLOW *)
     END_IF; (* RESET *)
  END_FUNCTION_BLOCK (* STACK_INT *)
```

10.4.1.2 Descriptive Identifier Names

The generic guidelines apply. It should be noted that the IEC 1131-3 specification mandates a minimum of only six characters of uniqueness for an identifier name, which can represent a challenge for descriptive power. The following are specific guidelines:

•*The names of variables, functions, and function blocks should be descriptive and closely related to the entities they represent.*

•*Avoid use of direct memory locations — use identifiers, not addresses.* ST, as all other languages defined in IEC 1131-3, allows reference to PLC I/O via either direct reference, or via declared identifiers. I/O references are assigned to Identifiers in the variable declaration blocks of programs. As direct I/O or memory references do not have to be declared, and can result in side effects during program execution, a program that uses this available construct can be difficult to understand, debug, and maintain. Use of the Direct Reference facility in ST, or any other IEC 1131-3 language, should therefore be avoided in anything other than assignment of Declared I/O. It should be noted that the IEC 1131-3 specification forbids the use of Direct I/O and memory references in functions and function blocks, thus avoiding a serious safety problem.

•*Avoid short and cryptic names to the maximum extent possible.* Differences between variables with related names should occur early in the name.

10.4.1.3 Comments and Internal Documentation

The generic guidelines apply. Incomplete, outdated, and inconsistent comments impede review and maintenance. Reasonable use of comments is recommended. Minimum comments for internal documentation should include:

•Each program module (program, function, function block) should have a header that describes the purpose of the module, the date of creation, the author, the date of each revision, the author of each revision, and the changes made to each revision. It should also refer to the higher level design specification and explain how the requirements of this specification are met in this module.

•A Function or Function Block should have a header that describes the input and output variables, any referenced or modified global or direct variables, and an explanation of the algorithms used. It should also document other Functions or Function Blocks accessed.

•Key variables should have brief descriptions.

•Comments should be used where critical steps are executed, or where less obvious constructs are being used

•Dated comments with the programmer's initials should be used for any changes after a product is delivered, to enhance traceability.

•The ends of loops and selection statements should be tagged with comments.

10.4.1.4 Limitations on Subprogram Size

The generic guidelines apply. Functions and Function Blocks should be limited in size, depending largely on the project guidelines.

10.4.1.5 Minimize Mixed Language Programming

Unlike other computer language systems, each of the languages defined by the IEC 1131-3 has particular strengths in expressing particular types of algorithms. ST, for example, is a good language for defining mathematical relationships and algorithms, but is generally inferior in readability to LD (Ladder Diagram, the IEC name for Relay Ladder Logic) at expressing complex boolean relationships. SFC, as might be expected from the name Sequential Function Chart, is optimized for representing sequences of operations. PLCs and PLC-like systems that implement multiple languages defined by the IEC 1131-3 specification do so to allow the programmer to take advantage of the fact that each of these languages is targeted for a particular portion of the entire control problem.

Unlike the generic guidelines, therefore, appropriate use of multiple IEC 1131-3 languages in a single program is to be encouraged, with each language used for the types of control for which it is best suited. Sections of code written in ST could be, therefore, mainly concerned with mathematical relationships between variables, as opposed to sequencing or boolean tasks.

10.4.1.6 Minimize Obscure or Subtle Programming Constructs

The usage of obscure or subtle programming constructs should be avoided. ST is a small language, without many constructs that are obscure. The single construct within ST that is most likely to result in obscure programs is the use of directly referenced variables versus declared variables. The use of direct variables should be avoided.

10.4.1.7 Minimize Dispersion of Related Elements

The generic guidelines apply. If related elements of the code are dispersed in a program, this makes it necessary to refer to multiple locations within a source listing in reviewing or modifying the source code.

One cause of dispersion in IEC 1131-3 systems is related to the use of multiple languages, as different languages may be used in different functions or function blocks. The convenience of having multiple languages available with different strengths and weaknesses should be balanced against this requirement to limit dispersion of related elements.

10.4.1.8 Minimize Use of Literals

Literals, (e.g. actual numbers in the source code) are much more difficult to identify than names to which a constant value is assigned at the beginning of the module. ST, as an IEC 1131-3 member language, supports the VAR CONSTANT declaration, which defines variables that can be read from, but not written to, by the program. These CONSTANT variables should be used in preference to literals.

10.4.2 Abstraction

This principle depends on the following specific base attributes:

- Modularity
- Information Hiding
- Minimization of the use of global variables
- Minimization of the complexity of the interface and defining allowable operations.

Although ST was not defined with object-oriented attributes, there are approaches described in the following sections that can provide the benefits of abstraction. These are described in the following subsections.

10.4.2.1 Modularity

The IEC 1131-3 definitions of PLC languages, including ST, support block structured and modular programming concepts via the function and function block definitions. Any ST program should be programmed as a set of well defined cohesive functions/function blocks, each dedicated to a specific problem.

10.4.2.2 Information Hiding

The generic guidelines partially apply. IEC 1131-3 languages support the use of variables limited in scope to the current function or function block, that cannot be accessed from any other routine in the PLC. These variables may be used to support the concepts of information hiding, and indeed should be used in this fashion.

10.4.2.3 Minimization of the Use of Global Variables

The generic guidelines apply where local variables are supported in the PLC ST implementation. There are two sources of global variables in IEC 1131-3 defined languages: declared global variables and directly represented variables. Because of the strong potential for unintended side effects, it is desirable to limit the use of either of these types of global variables in safety related programs. As the potential for unintended side effects is particularly strong with directly represented variables, which directly access a PLC memory or I/O point, their use is strongly discouraged.

10.4.2.4 Minimization of Interface Complexity

The generic guidelines apply. Interfaces are a frequent cause of software failures. Complex interfaces are difficult to review and maintain, and are therefore not desirable in safety-related programs. The following are specific guidelines:

•*Limit the number of parameters.* The number of parameters to an ST function or Function Block should be minimized. Large numbers of parameters can make interfacing complex and increase the risk of variable inversion..

•*Use Structured Variables.* When there are large numbers of parameters and some of those parameters are related, they should be defined in an ST `struct`, and the structure passed as a parameter.

•*Use the capabilities of IEC 1131-3 development environments.* The IEC 1131-3 specification defines both textual and graphics modes for function declarations. Because of this, many implementations of the IEC 1131-3 language suite allow a programmer or reviewer to switch between viewing/editing the two representations. Even though ST uses the textual format, use of the capabilities of the development environment, where available, to view/document the function declaration both textually and graphically will increase the clarity and readability of the program.

10.4.3 Functional cohesiveness

Cohesiveness is the manner and degree to which the tasks performed by a single software module

are related to each other. Functional cohesiveness refers to a clear correspondence between the functions of a program and the structure of its components.

Functions and function blocks in ST should have one clearly discernable purpose and be given a name that reflects that purpose. The same principle of a single purpose function should also be applied to variables.

10.4.4 Malleability

Malleability is the ability of a software system to accommodate changes in functional requirements. Malleability extends data abstraction with the motivation of isolating areas of potential changes.

Avoiding the use of directly represented variables was discussed in section 10.4.2.3. Malleability is one more reason why their use should also be avoided in this regard, as the I/O mapping of a PLC system is one aspect subject to change between implementations.

10.4.5 Portability

Portability is the ease with which a system or component can be transferred from one hardware or software environment to another. From the perspective of safety, the benefit of portability is the adherence to standard programming constructs that yield predictable and consistent results across different operating platforms.

The IEC 1131-3 standard allows the implementation of both subsets and supersets of the languages described therein to be described as compliant. Furthermore, there are some areas, like the format and syntax of directly represented variables, that are subject to widely varying interpretation by different vendors of PLC systems. This situation makes portability in ST programs problematic, although less so than is the case for Relay Ladder Logic. Avoidance of ST extensions where possible, and avoidance of the use of directly represented variables outside of variable declarations are the primary tasks involved in maximizing the portability of an ST program.

Since ST is extensible, as can the other IEC 1131-3 languages, it should be noted that the portability of any program is limited by the target hardware, and the source program adherence to the IEC 1131-3 programming standard.

References

International Electrotechnical Commission (IEC), *Programmable Controllers Programming Languages*, IEC Standard 1131, Part 3, 1993.

Allen Bradley, *PLC-5 Programming Software – Programming*, Publication 6200-6.4.7, November 1991.

11 Function Block Diagrams

This chapter discusses the applicability of the generic language attributes to the PLC Function Block Diagram Language (FBD), as defined by the IEC 1131-3 PLC language specification (IEC, 1993)[25]. FBD is a two-dimensional graphic language that can appear similar to the AND-OR diagrams used in hardware design. There is also an ASCII-character only notation for FBD as well, used for illustrative purposes within the IEC 1131-3 specification, although most FBD programming packages today use a graphical presentation. FBD may be distinguished from Ladder Diagram (LD) in that in FBD, the lines connecting various Function Blocks may represent data of any of the available types (boolean, word, float, etc.) whereas in LD, the connecting lines represent strictly boolean relationships. FBD programs, like other PLC programs, are executed repetitively in a PLC scan, and operate on a fixed PLC memory map/I/O base. An overview of FBD is given in Appendix A.

11.1 Reliability

Reliability of software is based upon the capability of a software component to perform its programmed tasks under stated conditions for a specified period of time. Reliability depends upon the runtime predictability of Memory Utilization, Control Flow, and Timing. FBD-specific guidelines based on these concepts follow.

11.1.1 Predictability of Memory Utilization

The key element in predicting memory utilization is to avoid the use of dynamic memory allocation. The FBD language, like ST and Ladder Logic, contains no keywords or functions that allow dynamic memory allocation to take place. The memory management scheme is fixed and deterministic, resulting in a predictable memory management scheme defined prior to runtime.

11.1.2 Predictability of Control Flow

Predictability of Control Flow is the capability to determine easily and unambiguously what path (i.e., what set of branches and in what order) the program will execute under specified conditions.

---

[25]The majority of FBD implementations follow the IEC 1131-3 FBD specification and are derived from it. FBD precursors to the IEC 1131-3 standardization are explicitly outside the scope of this document -- while they influenced the IEC specification, they are regarded as separate languages for the purposes of this report.

11.1.2.1 Maximizing Structure

The generic guidelines apply. Use of GOTO or equivalent execution control statements, which result in an unstructured shift of execution from one branch of a program to another, is difficult to trace and understand. FBD does support both conditional and unconditional jumps, as shown in the following example. FBD Function Blocks should avoid such statements in safety-critical programs where possible.

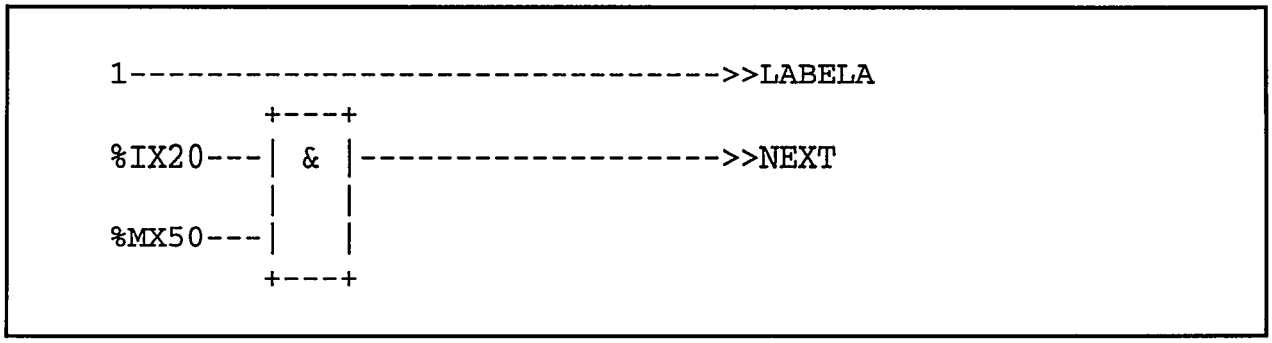


Figure 11-1 Examples of Unconditional and Conditional Jumps

The above example shows an unconditional jump to Label LABELA, unconditional because the boolean value of 1 is used as the jump condition. The second FBD construct shows a jump to the label NEXT upon the condition that input %IX20 and internal bit %MX50 are both true. The notation used is the ASCII notation of the IEC 1131-3 specification.

11.1.2.2 Minimizing Control Flow Complexity

The generic guidelines apply. Complex control flow makes the program difficult to understand and maintain and is a source of unpredictable control. The control flow in FBD is graphically expressed, making it fairly easy to the follow the flow of FBD statements in simple programs. However, the two-dimensional unconstrained nature of FBD, plus the usage of graphic elements for both Boolean and other data types, make it possible to use FBD in a manner which makes the control flow difficult or impossible to understand.

The following specific guidelines apply:

•*Decomposition.* The FBD program should be decomposed into cohesive Function Blocks. This is especially important for FBD, as even moderately sized FBD program constructs can become difficult to follow, due to the high degree of interconnectivity allowed.

•*Nesting Level Limits.* Care should be taken that function block nesting levels are not excessive. Note that for FBD, in order to keep code sizes small and control flow easy to follow and understand,

this requirement is in conflict with the requirement to decompose the FBD programs above. Balance between these two requirements should be sought, on a per-project basis. Decomposition (control flow) should have precedence over code sizing issues however.

11.1.2.3 Initialization of Variables Before Use

The generic guidelines apply to internal and retentive variables but not to variables associated with field inputs. Such input variables may be associated with field inputs, in which case their initial values take on the state of these devices and should not be initialized by an FBD program prior to use. Internal and retentive vairables should be initialized on a warm start as well.

Internal variables, or variables associated with output devices, have explicit default values in accordance with the IEC 1131-3 specification. However, variables of this type should be explicitly assigned initial values when declared, for purposes of clarity. Declared CONSTANT variables must be assigned values at declaration time.

The IEC 1131-3 specification defines RETENTIVE variables as those which will hold their values over a PLC power cycle. These values should also be initialized by the program to their correct cold start values, but it should be noted that the initial values of retentive variables during a warm start condition will be their values at the time of PLC shutdown, and not the initial values as shown in the program. This should be noted in the documentation.

Most IEC 1131-3 based FBD programming systems have the variable declaration portions as a separate dedicated editor included with the development system.

11.1.2.4 Single Entry and Exit Points in Subprograms

The generic subprogram types defined by IEC 1131-3 for FBD (and all other languages in the specification) are the *Function*, which returns a single value, and the *Function Block*, which returns one or more values. These may have but a single entry point when written in FBD. FBD allows a Function or Function Block to have multiple exit points, by use of the RETURN construct, however. This should be avoided wherever possible.

11.1.2.5 Minimization of Interface Ambiguities

The generic guidelines apply as indicated below. Interface errors account for a large portion of coding errors. An example of such errors is reversing the order of arguments when calling a Function or Function Block. It should be noted that FBD's graphic presentation makes these kinds of interface ambiguities unlikely, as each parameter to a function or function block is presented as attached to a line with a label of the parameter name. The following are specific guidelines:

•*Test the validity of input arguments at the beginning of a routine, and test the validity of the results before exiting* Such testing is important for avoiding errors that can compromise the integrity of the system. Range checking inside a Function or Function Block is preferred.

11.1.2.6 Use of Data Typing

The generic guidelines apply. The following measures should be taken to reduce data typing errors:

•*Minimize the use of Type Conversions, and eliminate implicit type conversions.* The FBD specification allows the use of overloaded built-in functions/function blocks - that is, functions that will operate on input data elements of various types within a generic type designator. For instance, an overloaded addition function on generic type ANY_NUM can operate on data of types LREAL (64 bit floating point), REAL (32 bit floating point) , DINT (32 bit integer), INT (16 bit integer) , and SINT (8 bit integer). Usage of these overloaded function blocks with mixed input/output data types should be avoided.

•*Do not use overloading.* IEC 1131-3 does not allow user-created overloaded functions or function blocks, so this is not an issue.

•*Avoid the use of mixed-mode operations.* Operations using multiple data types should be avoided. If such operations are necessary, they should be clearly identified and described using prominent comments in the source code. Explicit use of type conversion functions should be made, if practical, to make the designer's intention clear.

•*Use a single data type in evaluations and relational operations.* Expressions involving arithmetic operations or relational operations should have either a single data type or the proper set of data types for which conversion difficulties are minimized.

•*Avoid Mixing Signed and Unsigned variables.* Mixing signed and unsigned variables in arithmetic and logical operations raises safety concerns and should be avoided in safety systems. Explicit use of FBD type conversion functions should be made, if practical, to make the designer's intentions clear.

11.1.2.7 Precision and Accuracy

The generic guidelines apply. Safety software must provide adequate precision and accuracy for the intended application. At the same time, the software must also tolerate the inconsistencies from operations on floating point numbers. The following are specific guidelines for FBD:

•*Use the LREAL Data Type for floating point, where available.* The LREAL (64 bit floating point) should be used for floating point calculations in safety systems, when available. The REAL Data type should be avoided because it may not provide adequate precision and accuracy.

•*Account for floating point properties in relational operations.* The equality comparisons on floating point numbers should be avoided in safety systems, since the machine representation of floating point numbers may lack precision and may have a small residue error. Inequality comparisons should be utilized and equality comparisons should be avoided on floating point numbers.

The following example illustrates the potential problems:

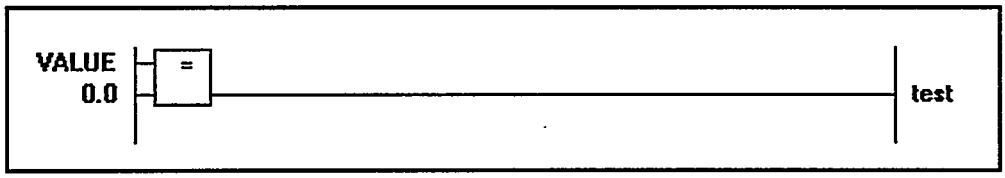


Figure 11-2 Inappropriate Comparison with Floating Point Zero

The comparison of the FLOAT named VALUE to 0.0 in the above example is likely to be false because of rounding errors, even if the value is expected to be zero. The condition should be modified as follows:

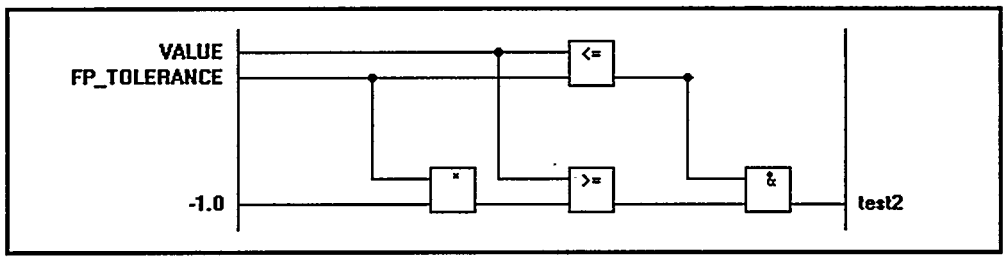


Figure 11-3 Safe Method for Comparison with Floating Point Zero

In this example, if VALUE is less than the Floating Point Tolerance (stored as a CONSTANT FLOAT), and if it is greater than the Negative Floating Point Tolerance (calculated here by multiplying the tolerance by -1.0) the comparison is TRUE, and Boolean variable `test2` will be true.

•*Account for truncation in integer operations* Integer arithmetic operations can generate truncation and rounding errors more often than floating-point arithmetic. Integer truncation errors are most serious as a result of division. In FBD, as well as the other IEC 1131-3 languages, the results of integer divisions are always truncated toward zero. Truncation errors can cause safety concerns when the results with truncation are used in comparisons and conditions for control decisions. Therefore, a rounding-off technique should be utilized. As FBD conversions from LREAL to the various integer types are automatically rounded to the nearest integer, a typical method might be to perform the division calculation with LREALs, and then convert the result to an INT.

•*Ensure that arithmetic conversion produces a result that fits in the assigned memory locations.* When it is necessary to use one of the FBD type conversion functions, care must be taken to ensure that the converted value can be represented in the shorter data type. For example, attempting to convert a REAL value of 12,341.56 to an SINT (8 bit signed integer) would result in an error.

11.1.2.8 Order of Precedence of Arithmetic, Logical, and Functional Operators

The generic guidelines apply. The FBD order of precedence is determined by the nature of the graphic presentation itself. FBD calculations should be carefully ordered to complete the correct calculations.

With all IEC 1131-3 languages except SFC, a function block may be written in any of the available languages, and may be called from any of the available languages. This is a fundamental aspect of these types of systems. This report deals strictly with the FBD portions of such a mixed-mode program.

In the instance of calling a function block with code written in ST from an FBD diagram, for example, the order of precedence issues regarding which element of the FBD Diagram is executed in what order are, as stated above, determined by the nature of graphic presentation. ST code that implements the function of a particular block cannot change this, and the requirements for this code are covered in the ST section of this report.

11.1.2.9 Separating Assignment from Evaluation

FBD does not allow assignment inside an evaluation expression. The generic guidelines are thus not applicable.

11.1.2.10 Proper Handling of Program Instrumentation

The generic guidelines apply to the development environment rather than to the FBD program itself. No application support is necessary for instrumentation in FBD programs. PLCs generally provide a development environment that allows debugging during real-time program execution, and IEC 1131-3 requires that such support facilities be present. However, the specification allows for variability, and the exact features of this debugging environment, as well as its effect on real-time response, depend on the make and model of PLC and its associated programming software. Some of the tools often included are on-line monitoring of program variables, collection of time histograms, insertion of break points, and single-step program execution. Some PLC systems allow break points during the I/O process.

The following specific guidelines apply to the use of this environment:

•*Do not modify running programs.* Most PLCs also provide a facility that allows the modification of the PLC program while it is being executed. However, the safety consequences of misuse of this feature can be severe. Not only can a programmer accidentally introduce errors into a running PLC program by using this feature, but the added communications load and system overhead associated with the program change transfer itself can cause problems with the real time response of the system.

•*Remove all development environment-related execution control commands.* Execution control commands for debugging, including breakpoints, single step branches, and other features should be removed prior to downloading the operational version of the software to the PLC. Record of an explicit check for this step should be made and verified during the review process.

•*Remove all development environment-related internal monitoring commands:* Internal program monitoring commands, such as watchpoints, snapshots, and data collection for histograms should be removed prior to downloading the operational version of the software to the PLC. Record of an explicit check for this step should be made and verified during the review process.

11.1.2.11 Control of Class Library Size

FBD does not allow the programmer to overload operators or functions, therefore the generic guidelines are not applicable. However, built-in functions and function blocks provided by the PLC manufacturer may be overloaded. Source code and documentation are usually not provided with such functions, and thus, extensive testing is necessary to verify correct and continuous operation over the anticipated input domain for each data type used in such blocks.

11.1.2.12 Minimizing Dynamic Binding

Dynamic Binding is not a feature supported by IEC 1131-3 languages. The generic guidelines are not applicable.

11.1.2.13 Control of Operator Overloading

The generic guidelines are not applicable. FBD does not allow the programmer to overload operators or functions. However, built-in functions and function blocks provided by the PLC manufacturer may be overloaded. Source code and documentation are usually not provided with such functions, and thus, extensive testing is necessary to verify correct and continuous operation over the anticipated input domain for each data type used in such blocks.

11.1.3 Predictability of Timing

Control flow timing issues in PLC programs stem from the nature of the PLC program scan and I/O scan. The IEC 1131-3 PLC specification does not address the issue of the relationship between these two concepts. It must therfore be assumed that the same level of variety exhibited in current PLC implementations will continue in IEC 1131-3 compliant implementations. A survey of current implementations of FBD shows this to be the case.

The two broad categories of timing issues, then, are 1) ensuring the maximum program scan time is not exceeded, and 2) ensuring that field I/O data is appropriately 'fresh' before making control flow decisions based on it. These are discussed below.

11.1.3.1 Ensuring Maximum Program Scan Time not exceeded

FBD programs, like all PLC programs, are executed repetitively in what is called the PLC scan. PLCs contain a watchdog timer to determine if the maximum allowable time for a program scan has been exceeded, and will typically halt the execution of the system if this time is exceeded. These program halts may be deleterious in a safety-critical system. Depending upon the implementation, this Watchdog Timer may have either a fixed or modifiable value.

11.1.3.2 Ensuring Currency of Field I/O Values

The order of program execution should ensure that the variables that are used in calculations or control flow decisions are current values of inputs or results.

The types of timing issues that should be verified in the audit or review of a real-time PLC system depend strongly on factors that are specific to the methods used by various PLC operating systems for scanning the real world I/O. Generally speaking, these methods may be classed into four categories, as follows:

1. *Immediate I/O Update.* PLC has no separate I/O Scan - I/O is updated as required by each program statement.

2. *Asynchronous I/O Update.* PLC I/ O scan occurs asynchronously from PLC Program scan. Both this method and the Immediate I/O Update method allow the value of a real-world input to vary in the course of execution of a single PLC program scan.

3. *Captured Asynchronous.* PLC I/O scan occurs asynchronously from the PLC Program Execution scan, but input and output values are captured in a buffer to eliminate the possibility of variance of input values during a single Program scan.

4. *Synchronous.* PLC I/O scan is fully synchronous with the Program Execution scan.

11.1.3.3 Accounting for Sensor and Actuator Latencies

PLC systems vary widely in the input latency (delay time between when a sensor value changes and when the new status is made available to the PLC Program), and the related output latency (delay time between the command from the PLC program to change the status of an output and the actual change in status of the actuator device). Depending upon the sensor/actuator types used, the type of I/O scan/program scan relationship as mentioned above, and the networking for remote I/O drops, these latencies may range from over 500 msec to less than 2 msec.

Given these differences, both the PLC programmer and program auditor should be explicitly aware of the nature of the PLC system used for a particular application, and this information should be explicitly stated in the PLC program documentation. Timing issues that may need to be reviewed in PLC systems include:

•The impact of noise on latencies

•The impact of changes in variable state during the execution of a single program scan (on systems where this is possible)

•The variance and upper limits on input and output latencies — and what may be necessary to determine the upper limits.

11.1.3.4 Minimizing the Use of Tasking

The generic guidelines are partially applicable. The IEC 1131-3 tasking model for PLCs is designed explicitly to eliminate timing uncertainties, and hence should be utilized in safety-critical systems, with the appropriate precautions. The following are specific guidelines

•*Associate time-critical programs with PLC tasks.* FBD programs, like all PLC programs, are executed repetitively. Once the end of the program is reached, and typically after some system overhead (I/O Updates, System Health status checks, etc.), control is returned to the beginning of the program. This cycle is known as a PLC scan. The response time of the application determined by the periodicity and repeatability of this scan. Association of the FBD program to an IEC 1131-3-defined periodic task assigns a particular periodicity and priority to the execution of the FBD program. According to the IEC 1131-3 specification, if the FBD program is not associated with a periodic task, it will be executed as the lowest priority on the PLC system. Because only tasks with assigned periodicities have deterministic scan times, the response time can not be known for programs not associated with such tasks.

•*Ensure that utilization limits are not exceeded.* PLC utilization should be less than the manufacturer's specification for maximum allowable utilization based on worst case execution times.

For example, if an FBD program is assigned to a task with a 30 msec periodicity, but can take 30 msec or even longer to execute, there there will be a significant problem. Normally, the PLC will shut down upon repeated scan time overruns. The IEC 1131-3 specification requires that manufacturers of IEC 1131-3 compliant PLC systems provide tools for performing this type of utilization analysis. Such tools should be used to determine the worst case condition, the results checked against actual system performance, and the results documented. This should be accomplished even if the PLC program consists of but a single task.

•*Avoid multiple task periodicities:* If possible, all PLC programs should be assigned to the same task periodicity — particularly in PLCs without support for local variables. There are two reasons for this guideline:

(1) The potential for programming errors in accessing a global data base from tasks of differing periodicities is quite high due to the difficulty of considering all cases of sequencing and worst case timing.

(2) Rising-edge and falling edge detection data can be lost. In many PLC systems, rising and falling edge detection of I/O points is calculated from the current status of the I/O point compared to the saved status of that point on the last I/O scan of the servicing task. So, for example, an input serviced by a 10 msec task will have a rising edge condition for a single 10 msec scan after coming on. If a rising edge instruction on this point is used in a 50 msec task, the probability is 0.2 that the rising edge will not be seen by this instruction in the 50 msec task, as this instruction is executed at 50 msec intervals, and the actual condition is true for only 10 msec. This uncertainty is not appropriate for safety systems.

The decision whether or not to use the IEC 1131-3 multitasking features in a particular PLC program must balance the need for explicitly controllable response times with the potential for programming errors due to over utilization of the CPU and update of a global memory pool at different periodicities.

11.1.3.5 Minimizing the Use of Interrupt Driven Processing

The generic guidelines apply — the use of interrupt handling in FBD should be avoided as possible. The IEC 1131-3 specification contains no explicit discussion of interrupt processing, and the FBD portion of this specification does not include any keywords or standard functions for handling interrupts. However, an IEC 1131-3 compliant system may implement interrupt handling as an extension to the languages as specified. In general, interrupt processing should be handled through the PLC features supporting periodic tasking, watchdog timers, diagnostics, and fault routines.

11.1.3.6 Avoidance of Self-Modifying Code

The generic guidelines do not apply. FBD provides no language facilities that would allow the possibility of self-modifying code. The reader should note that this is not the same as the guideline against on-line modification of code pointed out in the section on instrumentation.

11.2 Robustness

Robustness refers to the capability of the software to survive off-normal or otherwise unanticipated conditions. Since unanticipated events can happen during an accident or excursion, it is vital for a safety system to survive an accident and continue working. This section discusses the guidelines on exception handling and I/O checking.

11.2.1 Exception Handling

The IEC 1131-3 specification, and the FBD language portions contained within it, make no reference to explicit exception handling constructs. Exception handling in PLCs has traditionally been done via the PLC operating system health monitoring routines, or via Fault Routines triggered, either manually or automatically. In fact, the IEC 1131-3 specification acknowledges that the nature and type of error-handling procedures are outside of the requirements for compliance with the specification (IEC 1131-3, section 1.5.2). It is anticipated that IEC 1131-3 compliant systems will continue to provide this type of functionality as allowable extensions to the language system. An examination of current IEC 1131-3 compliant PLC programming systems indicates this to be the case.

11.2.1.1 System Health Monitoring

PLC systems provide System Health Monitoring information to the application program, usually in the form of designated internal bits and words. Information may include execution times for particular tasks, I/O update information, I/O operating status information, battery state information, and other status information used to determine the proper operating status of a PLC system. Unfortunately, the nature, extent, and type of this information is not covered by the IEC 1131-3 specification. However, the safety program should make use of such information (as available) as part of its health checking and exception processing

11.2.1.2 Fault Routines

The IEC 1131-3 Programmable Controller Language Specification does not mandate that a compliant PLC system contain a designated hardware fault routine that will execute upon detection of one of a range of fault conditions, similar in concept to the Fault Routine available in current Allen Bradley PLCs (Allen Bradley, 1991). However, a non-periodic task could be set up to trigger appropriate internal diagnostic information to handle particular types of errors (fall-back modes on I/O failures, warnings to operator interfaces, etc). Usage of such a construct should be considered in the design phases of programming for a system of this kind.

11.2.1.3 Watchdog Timer

A note in section 2.7.2 (Tasks) of the IEC 1131-3 specification states that "The manufacturer shall provide information to enable the user to determine that all (task) deadlines will be met in a proposed configuration". However, the specification does not mention any form of checks that these deadlines are met at runtime, since this falls into the error-checking category of constructs not covered by the document.

Many PLC and PLC-like systems with 1131-3 compliant multitasking capability implement a watchdog timer for each periodic task. Typically, these systems will ignore a single task overrun and flag that an overrun has occurred, but will shut down the system upon *N* consecutive task overruns, where the value of *N* will vary from system to system. Where available, FBD programs should monitor the task overrun flag, and react appropriately. It is not appropriate for a safety-critical system of this nature to exhibit regular task overruns — this should be looked for specifically during the check-out phases of development.

11.3 Traceability

Attributes specifically related to traceability include the use of built-in functions and the use of compiled program libraries.

11.3.1 Use of Built-in Functions

The IEC 1131-3 specification defines a set of standard Function Blocks callable from any of the five specified languages, including Function Block Diagram. These, as well as any others that may be included by a particular implementation of FBD, have been explicitly developed and tested for a real-time control environment. However, for safety critical applications, it is necessary to test such functions extensively to ensure that they operate successfully over the entire range.

Unlike user-defined functions, built-in functions in IEC 1131-3 may be defined to be overloaded — i.e., they may accept any valid type for input parameters. For instance, the ADD function block may accept integers or floating point variables as inputs or outputs, performing implicit conversions in the process of performing the calculation. Use of this overloading capability should be minimized in safety-critical systems.

11.3.2 Use of Compiled Libraries

Compiled libraries are libraries of routines written and compiled by an entity other than the development group, in addition to the built-in functions described above. Since there is currently no consensus on anything other than the source representation for FBD, there are currently no third party FBD compiled libraries available. At the time these do become available (if any) the generic guidelines apply.

11.4 Maintainability

The software maintainability attributes in this section are limited to those affecting safety. These include:

•Readability
•Modular Design
•Functional Cohesiveness
•Malleability
•Portability

11.4.1 Readability

Readability allows software source code to be understood by qualified personnel other than the writer. Readability is an important characteristic, as almost all programs are modified or debugged by someone other than the original author at some time during the life of the program. Readability should in large measure be based on project-specific guidelines; however, the following are some guidelines specific to FBD.

11.4.1.1 Layout of FBD Diagrams

The issue of FBD diagram layout is the analog of text-based language indenting issues. FBD is a free-form two dimensional graphically oriented language, without many of the constraints of a Ladder Diagram. Layout of FBD diagrams is a major issue in regard to readability, even more so than in LD programming.

Care should be taken when writing FBD programs to ensure that the layout is clean and well-ordered, so that connections between the various sections are clear. Individual FBD diagrams should be limited in size, as a single large interconnected FBD diagram can be very difficult to follow. Aggressive use of function blocks should be considered to keep FBD diagrams small and clean.

The larger the FBD diagram, the more difficult it is to read and understand. The primary modularization mechanism in the IEC 1131-3 languages is the function block. Therefore, function blocks should be used aggressively, e.g., modularization should be used aggressively in FBD programming, in order to keep individual diagrams readable.

The programmer should strive to avoid long, convoluted, connections between elements of an FBD diagram, as these are difficult to follow. Inputs and outputs to and from a particular function block in a diagram should be as close to that function block as possible, consistent with maintaining a clear presentation. Connections from one block to another should run left to right, and where possible, looping back right to left should be avoided. Points where lines cross but are not connected should be avoided as well, to the extent possible. Constructs where outputs of an FBD are surrounded by other function blocks and connections should be avoided wherever possible. Some examples of these are given below.

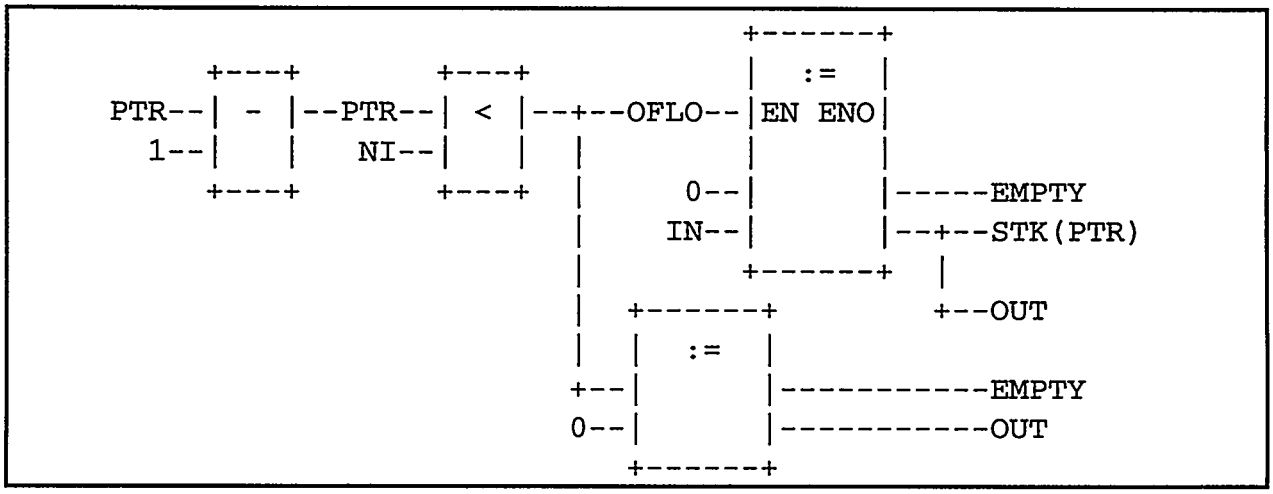


Figure 11-4 Example of Inputs Obscured by Surrounding Blocks

In the above diagram, written in the ASCII style of the IEC 1131-3 specification, the 0 and IN inputs to the EN ENO Function Block are completely surrounded by other FBD constructs. The following diagram, constructed with a commercially available FBD programming package, illustrates looping back of connections, and connection crossovers.

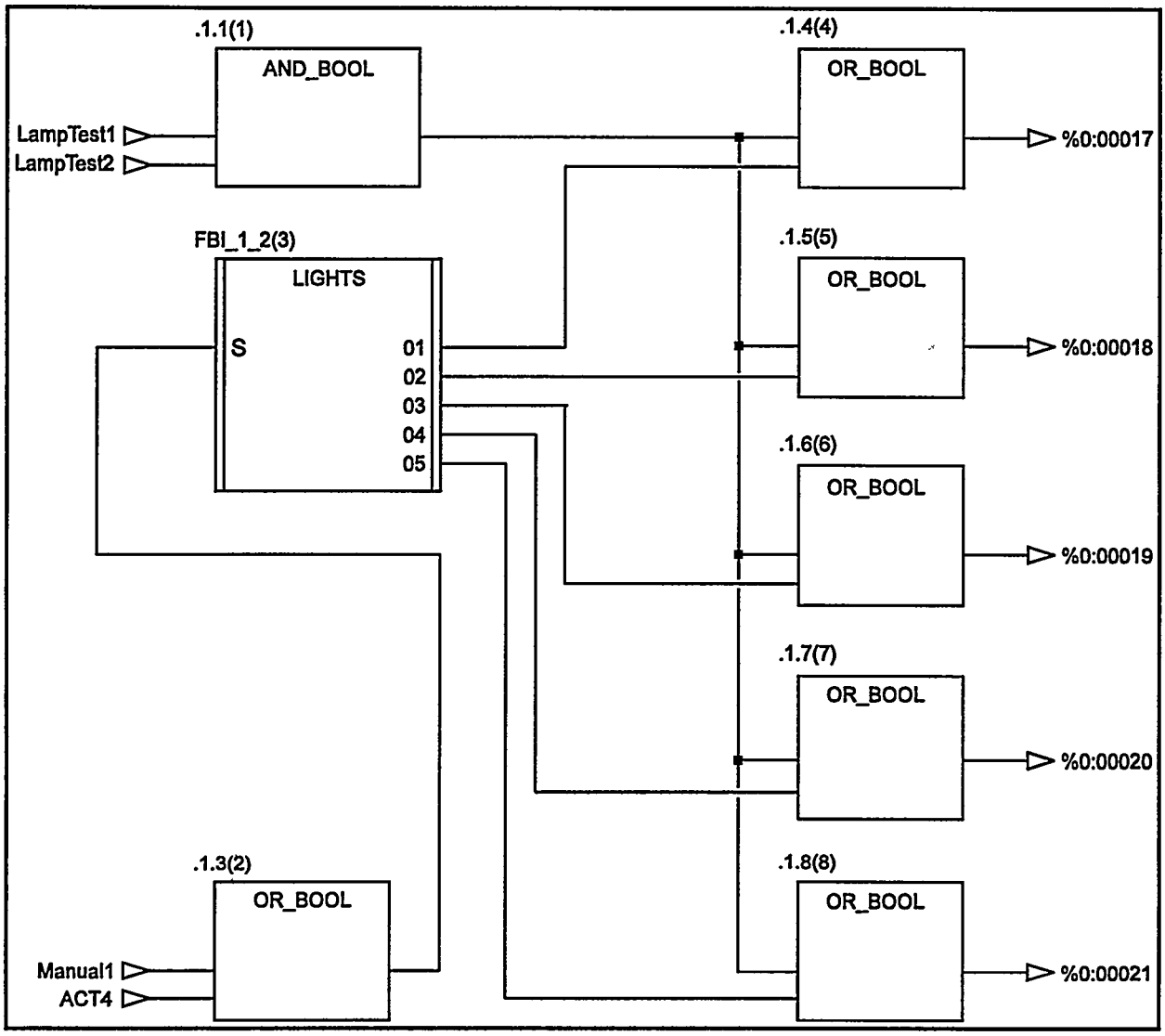


Figure 11-5 Example of Inappropriate FBD Layout

It should be noted that strict compliance with the above guidelines for an FBD layout may result in the inability to accomplish a desired function in FBD – the above should be regarded as goals for layout, and not rigid standards. In instances where it is necessary to use a less-than-clear FBD layout to accomplish a particular task, there should be additional amplifying documentation explaining the purpose and operation of the FBD diagram affected.

11.4.1.2 Descriptive Identifier Names

The generic guidelines apply. The names of variables, functions, and function blocks should be descriptive and closely related to the entities they represent. Short and cryptic names should be avoided to the maximum extent possible. Differences between variables with related names should occur early in the name. The following specific guideline applies:

•*Avoid the use of Direct PLC I/O or Memory references.* FBD, like all of the other languages defined in IEC 1131-3, allows reference to PLC I/O via either direct reference or via declared identifiers. I/O references are assigned to Identifiers in the variable declaration blocks of programs. As direct I/O or memory references do not have to be declared, and can result in side effects during program execution, a program that uses this construct can be difficult to understand, debug, and maintain. Use of the Direct Reference facility in FBD, or any other IEC 1131-3 language, should therefore be avoided in anything other than assignment of Declared I/O. It should be noted that the IEC 1131-3 specification forbids the use of Direct I/O and memory references in functions and function blocks, thus avoiding a serious safety problem.

> It should be noted that the IEC 1131-3 specification mandates a minimum of only six characters of uniqueness for an identifier name, which can represent a challenge for descriptive power.

11.4.1.3 Comments and Internal Documentation

The generic guidelines apply. Incomplete, outdated, and inconsistent comments impede review and maintenance. Reasonable use of comments is recommended. Minimum comments for internal documentation should include:

•Each program module (program, function, function block) should have a header that describes the purpose of the module, the date of creation, the author, the date of each revision, the author of each revision, and the changes made to each revision. It should also refer to the higher level design specification and explain how the requirements of this specification are met in this module.

•A Function or Function Block should have a header that describes the input and output variables, any referenced or modified global or direct variables, and an explanation of the algorithms used. It should also document other Functions or Function Blocks accessed.

•Key variables should have brief descriptions.

•Comments should be used where critical steps are executed.

•Dated comments with the programmer's initials should be used for any changes after a product is delivered, to enhance traceability.

•Additional comments should be included for FBD sections that are in violation of any of the previous sections layout guidelines.

11.4.1.4 Limitations on Subprogram Size

The generic guidelines apply. Functions and Function Blocks should be limited in size, depending largely on the project guidelines.

11.4.1.5 Minimize Mixed Language Programming

The generic guidelines do not apply. Appropriate use of multiple IEC 1131-3 languages in a single program is to be encouraged, with each language used for the types of control for which it is best suited.

Unlike other computer language systems, each of the languages defined by the IEC 1131-3 has particular strengths in expressing particular types of algorithms. ST, for example, is a good language for defining mathematical relationships and algorithms, but is generally inferior in readability to Relay Ladder Logic (called Ladder Diagram by the IEC) at expressing complex boolean relationships. SFC, as might be expected from the name Sequential Function Chart, is optimized for representing sequences of operations. FBD is similar to SFC, except that it is more 'free form' than structured. This makes it a good choice for more process or asynchronous applications, but it can also be used as an enhancement for all other languages. PLCs and PLC-like systems that implement multiple languages defined by the IEC 1131-3 specification do so to allow the programmer to take advantage of the fact that each of these languages is targeted for a particular portion of the entire control problem.

11.4.1.6 Minimize Obscure or Subtle Programming Constructs

The generic guidelines apply. The usage of obscure or subtle programming constructs should be avoided. Programming techniques that rely on side effects of using directly represented variables are an example of one such potentially dangerous construct.

11.4.1.7 Minimize Dispersion of Related Elements

The generic guidelines apply. If related elements of the code are dispersed in a program, this makes it necessary to refer to multiple locations within a source listing in reviewing or modifying the source code.

One cause of dispersion in IEC 1131-3 systems is related to the use of multiple languages, as different languages may be used in different functions or function blocks. The convenience of having multiple languages available with different strengths and weaknesses should be balanced against this requirement to limit dispersion of related elements.

11.4.1.8 Minimize Use of Literals

Literals, (e.g. an actual number in the source code) are much more difficult to identify than names to which a constant value is assigned at the beginning of the module. FBD, as an IEC 1131-3 member language, supports the VAR CONSTANT declaration, which defines variables that can be read from, but not written to, by the program. These CONSTANT variables should be used in preference to literals.

11.4.2 Abstraction

This principle depends on the following specific base attributes:

- Modularity
- Information Hiding
- Minimization of the use of global variables
- Minimization of the complexity of the interface and defining allowable operations

The IEC 1131-3 languages, including FBD, are not defined with object-oriented attributes the way C++ is, for example. However, there are tools available in FBD that can provide some of the benefits of abstraction.

11.4.2.1 Modularity

The IEC 1131-3 definitions of PLC languages, including FBD, support block structured and modular programming concepts via the function and function block definitions. Any FBD program should be programmed as a set of well defined cohesive functions/function blocks, each dedicated to a specific problem. Additional discussion of modularity as related to readability is discussed in Section 11.4.1.1.

11.4.2.2 Information Hiding

The generic guidelines apply. IEC 1131-3 languages support the use of variables limited in scope to the current function or function block, that cannot be accessed from any other routine in the PLC. These variables may be used to support the concepts of information hiding, and indeed should be used in this fashion.

11.4.2.3 Minimization of the Use of Global Variables

The generic guidelines apply where local variables are supported. There are two sources of global variables in IEC 1131-3 defined languages: declared global variables and directly represented variables. Because of the strong potential for unintended side effects, it is desirable to limit the use of either of these types of global variables in safety related programs. The potential for unintended

side effects is particularly strong with directly represented variables, which directly access a PLC memory or I/O point. Their use should therefore be strongly discouraged. The IEC 1131-3 standard does mandate support for the concepts of local and global variables, but this is not found in all PLC implementations.

11.4.2.4 Minimization of Interface Complexity

The generic guidelines apply. Interfaces are a frequent cause of software failures. Complex interfaces are difficult to review and maintain, and are therefore not desirable in safety-related programs. The following are specific guidelines:

•*Limit the number of parameters.* The number of parameters to an FBD function or Function Block should be minimized. Large numbers of parameters can make interfacing complex as well as creating a confusing visual image.

•*Use Structured Variables.* When there are large numbers of parameters and some of those parameters are related, they should be defined in an IEC 1131-3 `struct`, and the structure passed as a parameter.

•*Use of the capabilities of IEC 1131-3 development environments.* The IEC 1131-3 specification defines both textual and graphics modes for function declarations. Because of this, many implementations of the IEC 1131-3 language suite allow a programmer or reviewer to switch between viewing/editing the two representations. Even though FBD uses the graphic format, use of the capabilities of the development environment, where available, to view/document the function declaration both textually and graphically will increase the clarity and readability of the program.

11.4.3 Functional cohesiveness

Cohesiveness is the manner and degree to which the tasks performed by a single software module are related to each other. Functional cohesiveness refers to a clear correspondence between the functions of a program and the structure of its components.

Functions and function blocks in FBD should have one clearly discernable purpose and be given a name that reflects that purpose. The same principle of a single purpose function should also be applied to variables.

11.4.4 Malleability

Malleability is the ability of a software system to accommodate changes in functional requirements.

Malleability extends data abstraction with the motivation of isolating areas of potential changes.

The use of directly represented variables was discussed in both section 10.4.2.3 and 11.4.2.3. Decreased malleability is yet another reason why their use should also be avoided; the I/O mapping of a PLC system is one likely to change between implementations.

11.4.5 Portability

Portability is the ease with which a system or component can be transferred from one hardware or software environment to another. From the perspective of safety, the benefits of portability are the adherence to standard programming constructs that yield predictable and consistent results across different operating platforms.

The IEC 1131-3 standard allows the implementation of both subsets and supersets of the languages described therein to be described as compliant. Furthermore, there are some areas, like the format and syntax of Directly Represented Variables, that are subject to widely varying interpretation by different vendors of PLC systems. This situation makes portability in FBD programs problematic, although less so than is the case for Relay Ladder Logic. Avoidance of FBD language extensions is desired where possible, and avoidance of the use of directly represented variables outside of variable declarations are the primary tasks involved in maximizing the portability of an FBD program.

Since FBD is extensible, as are the other IEC 1131 languages, it should be noted that the portability of any program is limited by the target hardware and the source program adherence to the IEC 1131 programming standard.

References

International Electrotechnical Commission (IEC), *Programmable Controllers Programming Languages*, IEC Standard 1131, Part 3, 1993.

Allen Bradley, *PLC-5 Programming Software – Programming*, Publication 6200-6.4.7, November 1991.

APPENDIX A. Language Descriptions

This Appendix contains brief descriptions of the languages, run-time environments, and programming platforms[26] that are widely used in the industrial computing environment but are less known in the larger software development community. The intention of this appendix is to provide an introduction and overview of the issues. References at the end of the Appendix section provide more detailed information.

Section A.1 discusses Programmable Logic Controllers (PLCs) and general issues associated with the associated IEC 1131-3 programming languages. Section A.2 discusses PLC ladder logic, their most widely used programming language. Section A.3 discusses IEC 1131 Sequential Function Charts (SFCs), the main significance of which is to allow Ladder Logic to be combined with other languages recognized by the IEC 1131 standard. Section A.4 discusses PLC Structured Text, and Section A.5 discusses Function Block Diagrams. Section A.6 discusses PL/M and some of the issues associated with Intel RMX, the operating system that supports that programming language for real-time applications.

[26]As will be discussed in this Appendix, it is sometimes difficult to distinguish between the language, development environment, and run-time environment.

A.1 Overview of PLCs

A Programmable Logic Controller[27] (PLC) is an industrial computer specialized for real time applications. PLCs are integrated systems containing a processor, power supply, input modules, output modules and special purpose modules. Input modules interface with plant equipment and convert the field signals to logic levels for the processor to read. The processor uses these inputs to perform control functions based on application software. Output modules transmit the signals via an interface with the plant equipment. In addition there are special modules for communication with other computers, specialized dedicated functions, and conventional high-level language co-processors. Ladder logic will be used in the examples for the purpose of exposition.

A.1.1 Software Development for PLCs

PLCs have unique functions for process control including sequential logic, Proportional plus Integral plus Derivative (PID) control loops, and numerical operation. Programs are usually written on a Personal Computer using a specialized programming language (most commonly one of the four IEC 1131-3 languages described in this appendix). At present, most program development tools are proprietary and must be procured from the manufacturer of the PLC.

The diagram in Figure A-1 graphically depicts the use of PLC software development system. The hardware includes a Personal Computer (PC), communications link, and the target PLC. The PC-resident programming environment is composed of a shell that enables the programmer to develop the application software using functions supported by the processor hardware and firmware. Functions provided by the shell include:

- Programmer interface (editor)
- File manager to store and retrieve programs and data
- Communication interface with the PLC to download/upload programs
- Documentation and reporting tool
- On-line monitoring of application program.

The application software itself is contained in a "binary" file executable on the PLC (a binary file for these languages actually contains the source code, which may not consist of normal text as in the case of general purpose languages). This file is either edited on the PC (development process), or downloaded to the PLC processor to run (execution process).

After the application program is written, it is downloaded to the PLC memory for execution. The PLC also provides software packages for operator interface (HMI) and supervisory control and data acquisition (SCADA), to be used on engineering stations interfacing the PLC.

---

[27]The European designation for a PLC is a Programmable Controller, or PC

Once loaded into the PLC memory, the application software is executed by the PLC whenever it is in the "RUN" mode.

The structure of the binary file is specified by the PLC manufacturer. It can be viewed as a database file that defines the exact state of the PLC program and data. This model of the binary file is useful for the discussion of the programming "shell" given below.

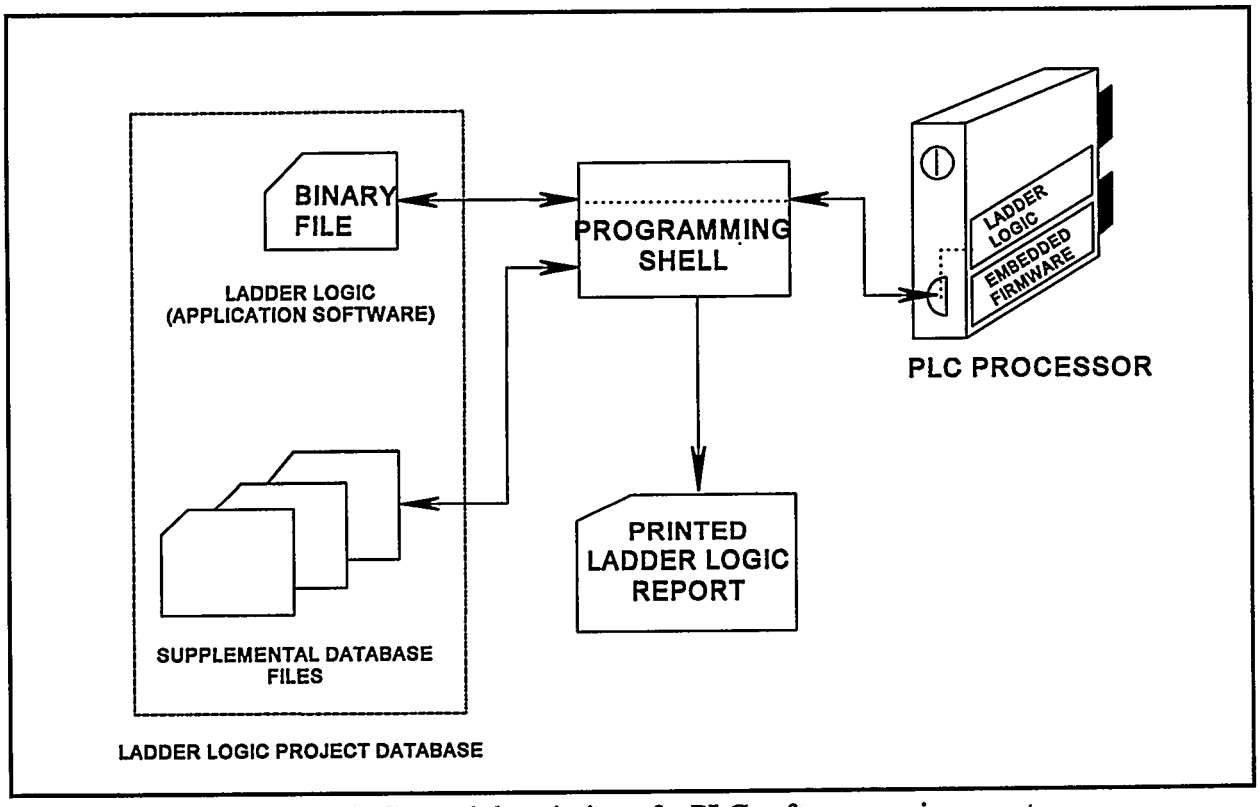


Figure A-1 General description of a PLC software environment.

A.1.2 Runtime Environment

The PLC runtime environment is firmware which provides the operating system services and library functions associated with the PLC. In the RUN mode, the PLC firmware runs as a real-time executive which processes the (Ladder Logic) instructions that have been loaded into the program RAM area. The program runs in a continuous loop which consists of the following major phases:

- Input read and output write scan
- Housekeeping
- Program scan (logic solve).

These are described in the following subsections and depicted in Figure A-2.

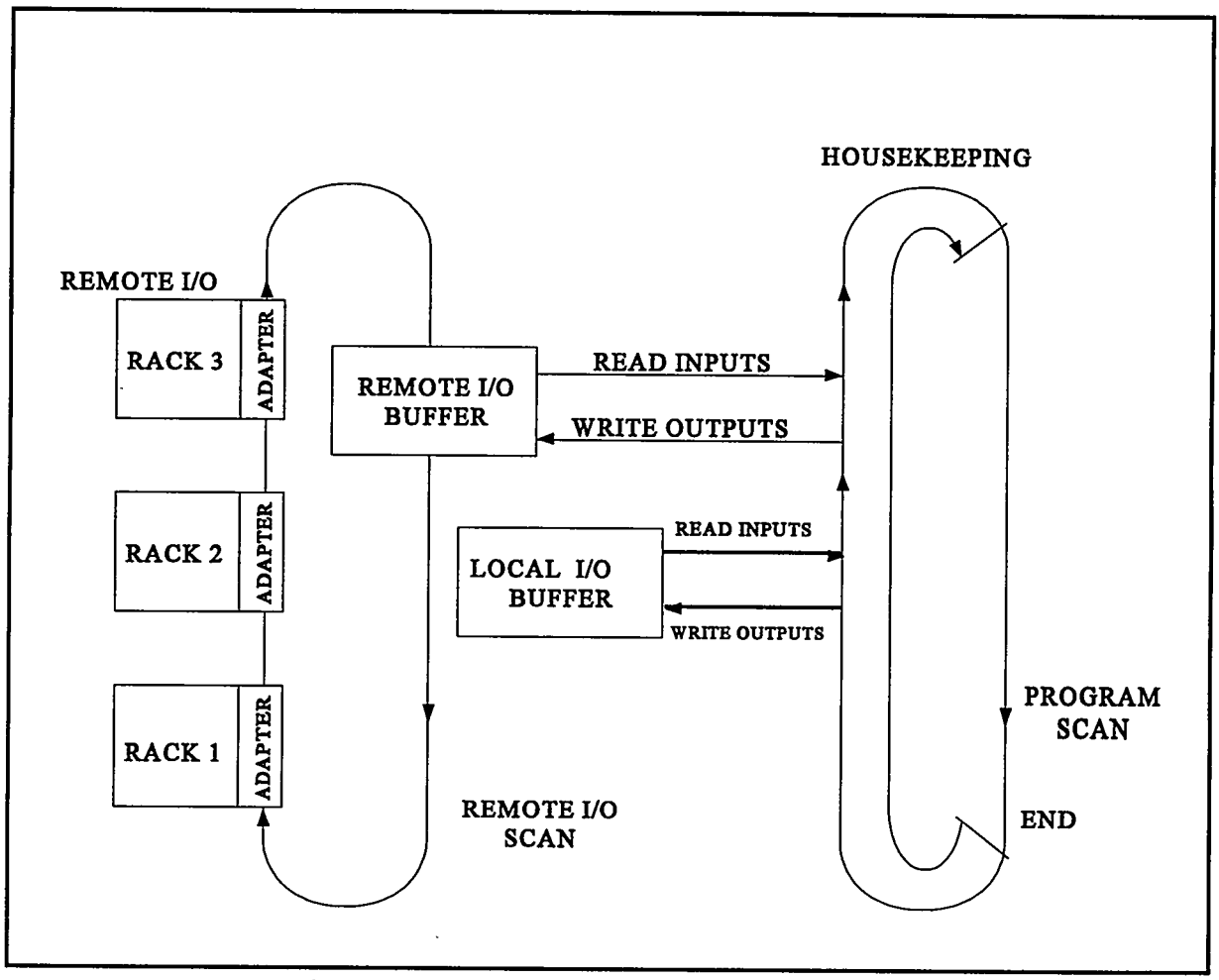


Figure A-2 Real time execution of PLC program.

A.1.2.1 Input Read and Output Write Scan

During the input/output (I/O) scan, the processor updates its internal input and output buffers with data being read from or written to local or remote I/O devices. Local I/O devices are the input and output cards residing in the same physical chassis as the PLC processor. Remote I/O devices reside external to this chassis and communicate with the processor's Peer Communications Interface port[28].

I/O data for input and output cards used in the application are maintained in input and output image tables. Typically, the PLC will organize the I/O image tables. This means that the inputs which are present will read into an area in memory. The program will write into another area of memory which is used to represent the outputs. It can be said that the input image table is representative of 'how

[28]In some PLCS, remote I/O devices communicate over a remote I/O link, not the Peer Communications Interface port which is reserved for inter-processor communications.

the inputs are perceived', and the output image table is 'the desired state' of the outputs. These tables are accessible to the Ladder Logic program as data files. During the I/O scan, data read from input cards are placed in appropriate locations in the input image table. At the same time, output data written to the output image table by the Ladder Logic are transferred to the appropriate output cards[29].

A.1.2.2 Housekeeping

Following the I/O scan, the PLC performs what is referred to as "housekeeping." This portion of the program cycle is used by the real-time executive to maintain and update its own internal state.

A.1.2.3 Program Scan

The program scan is the portion of the overall cycle where Ladder Logic instructions of the user's application software are executed. Here, the embedded firmware program operates on the portions of memory (RAM) that have been loaded previously with the application software from the binary file.

Program files contain the actual instructions to be executed. Data files are used to maintain program variables and other data structures required by the logic. It is the responsibility of the firmware program to properly decode and execute instructions in the program files. The program must also properly update the contents of the data files based on these instructions.

Detailed information about the specific Allen Bradley PLC firmware selected for description in this report can be found in the references (Allen Bradley, 1991a).

A.1.3 IEC 1131-3 Programming Languages

IEC 1131, Part 3, specifies the semantics and syntax of a unified suite of five programming languages for PLCs. These languages can be grouped into two categories: textual and graphical. Graphical languages are based upon graphical representation, that is, lines, boxes and text to represent specific relations among inputs and outputs. Appropriate quantities flow along lines between elements according to well defined rules. There are three graphical programming languages: *Ladder Logic, Sequential Function Charts,* and *Functional Block Diagrams.* Ladder logic is the most common of PLC languages and is discussed in the following section of this appendix. Sequential function charts can be used as a simple language, but their most important

[29]In some PLCs, the output image table data is written to the outputs all at once, and this occurs AFTER the completion of a full program scan.

function is to integrate modules written in other languages into a single higher level program. These are discussed in Section A.3 of this appendix. Function Block Diagrams uses block diagrams to interconnect the function blocks and are described in Section A.5 of the appendix as well as in the main body of the report.

Textual languages consist of a defined set of characters, rules for combining characters with one another to form words or other expressions, and the assignment of meaning to some of the words or expressions. There are two textual languages defined in the standard: *Instruction List (IL)* and *Structured Text (ST).* IL is a very low-level language, and may be considered as a standard Assembly Language for PLCs. Because the focus of this document and the appendix in on high level languages, IL not discussed. Structured text is a textual programming language using assignment, sub-program control, and selection and iteration statements to represent the application program for a PLC. ST, as distinguished from IL, is the high-level text-based language for PLCs. Much of its syntax is derived from Pascal. ST is discussed in Section A.4 of this appendix as well as in the main body of the report.

The models of execution, program organization, and variable handling of all IEC 1131-3 languages are based on a common hierarchical architecture consisting of *Configurations, Resources, Tasks,* and *Programs.*

Configurations are the highest level at which Global variables and Directly Represented Variables may be shared and accessed. A *Configuration* may often correspond to a single PLC unit, but certain types of PLC Network Architectures as well as multi-processor PLCs also meet this definition. A Configuration is composed of one or more *Resources.* Each *Resource* corresponds to a *signal processing function,* its associated *man-machine interface functions,* and *sensor-actuator interface functions.* A single-processor stand-alone PLC Configuration would have but a single Resource. A Configuration composed of a dozen processors capable of sharing the defined global variables and directly represented variables, on the other hand, would have 12 *Resources* associated with it.

Each *Resource* may have Global Variables (which are limited in scope to that *Resource*), zero or more defined *Tasks*, and *Programs* associated with those *Tasks. Tasks* may be defined as *periodic*, in which case they are defined with a specified periodicity, or as *non-periodic*, in which case they are executed upon the detection of the rising edge of a boolean variable. *Tasks* may also be assigned an execution priority. *Tasks* may also be scheduled pre-emptively, or non-preemptively. A *Program* not assigned to a *Task* will execute repetitively at the lowest priority level.

Programs in the IEC 1131-3 architecture begin with a variable declaration section, followed by the program statements themselves. Programs may contain calls to *Functions*, which return a single value, or *Function Blocks*, which return one or more values. Each *Program*, *Function*, or *Function Block* is written in one of the five IEC 1131-3 defined languages. Multi-language programming is accomplished by calling a *Function* or *Function Block* written in one language from a *Program*, *Function*, or *Function Block* written in another.

Variables in IEC 1131-3 languages may be either *Symbolic Variables*, or *Directly Represented*

Variables. Directly Represented Variables provide a standard nomenclature for direct access to specific addresses of the I/O and internal memory map of the PLC. All *Directly Represented Variables* begin with a '%' character, followed by a location prefix, a size prefix, and then a sequence of numbers to indicate the actual location. Some examples of these and their meanings:

%QX75	Output (Q) Bit (X) number 75
%IW215	Input (I) Word (W) number 215
%MW48	Internal (M) Word (W) number 48
%I5.8.3	Input (I) Bit (Implied) in Rack 5, Slot 8, Point 3

The precise meaning of the hierarchy of location numbers is not defined, so it is possible to have constructs like the following, taken from an actual PLC architecture:

%MW3.23.8.12.2.4,

which corresponds in this particular case to, from right to left, the 4th Internal 16 Bit Integer Word located in Module subsection 2 of module 12 of Rack 8, of the unit at drop 23 of Fieldbus Network 3. *Directly Represented Variables* do not have to be declared. Their use is legal only in *Programs* and *Configurations.*

Symbolic Variables do need to be declared. In the case where *Symbolic Variables* refer to actual input and output points, the declaration assigns them to these points by associating them with the appropriate *Directly Represented Variables. Symbolic Variables* that do not refer to I/O points need not be assigned a *Directly Represented Variable* - the IEC 1131-3 language system will assign these an address at compile time

A.2 PLC Ladder Logic

Ladder Logic is an instruction set to provide services of real time, I/O, user interface, and similar services. These services are associated with the special requirements of the PLC applications domain. Because Ladder Logic is targeted toward special applications, it provides features that are compatible with real-time control application requirements. These features, when used correctly and appropriately can contribute to the safe operation of the program.

The origin of Ladder Logic is the Relay Ladder Logic notation which was first introduced to represent combinations of contacts and coils of relays using specific notation. These combinations implemented logical functions (e.g., AND or OR). The introduction of PLCs transformed Ladder Logic from a hardware design notation to a high level language, specialized for process and logic control. The Ladder Logic language, in the case of the PLC, is not the traditional limited Ladder Logic implemented with relays, but an advanced language supported by the numerical capabilities of the processor, while the Ladder Logic notation serves only a graphical user interface. Ladder Logic supports all types of programming structures from advanced subroutines, parameter passing, loops, mathematical functions, proportional plus integral plus derivative (PID) controllers, I/O calls, timers, and any other features of a high-level language. Although much changed from their original

purpose and implementation, current forms of Ladder Logic are still similar to relay logic, allowing electrical engineering personnel who have traditionally have been in charge of factory automation to review and understand the code. This is an important advantage throughout the development process.

Ladder Logic is not a formally defined programming language. Each manufacturer has its own variation of Ladder Logic. In addition, many of the features associated with programming the PLC are not features of Ladder Logic itself, but the programming environment, the "shell," and the firmware mentioned above. The variety of ladder logic implementations is due to the strong coupling between software and hardware dictated by the requirements of the industrial control applications domain.

A.2.1 Elements of Ladder Logic

Ladder Logic programs consist of the following types of elements (IEC, 1993):

•*Power rails:* Ladder Logic networks are delimited on the left and right by vertical lines known as left and right power rails, respectively. The right power rail may be explicit or implied.

•*Link elements and states*: Links indicate power flow in the rungs of the Ladder Logic diagram. A link element may be horizontal or vertical. A horizontal link transmits the state of the element to its immediate left to the element to its immediate right. The state of an element can be either ON or OFF. A vertical link intersects with one or more horizontal links on each side and its state is the inclusive OR of the states of the horizontal links on its left. This state is transmitted to all horizontal links attached to the vertical link on its right.

•*Contacts*: A contact is an element which imparts a state to the horizontal link on its right side equal to the AND of the state of the horizontal link on its left side with an appropriate function. A contact does not modify the value of the associated Boolean variable. There are four types of contacts as described in Table A-1.

•*Coils*: A coil copies the state of the link on its left to the link on its right without modification, and stores an appropriate function of the state or transition of the left link into the associated Boolean variable. There are nine types of coils as described in Table A-2.

•*Functions and function blocks*: A function is a program unit which, when executed, yields exactly one result. A function block may yield more than one result. Internal variables of a function or function block are not accessible to users of the function. In Ladder Logic, at least one Boolean input and one Boolean output is shown for each function block to allow power to flow through the block.

Table A-1 Contacts

Static Contacts	
--| |--	Normally open contact A normally open contact is one for which the state of its left link is copied to the right link only if the associated Boolean variable is ON.
--|/|--	Normally closed contact A normally closed contact is one for which the state of its left link is copied to the right link only if the associated Boolean variable is OFF.
Transition-Sensing Contacts	
--|P|--	Positive transition-sensing contact A positive transition-sensing contact is one for which the state of the right link is ON only if a transition from OFF to ON in the associated Boolean variable is sensed when the left link is ON.
--|N|--	Negative transition-sensing contact A negative-transition sensing contact is one for which the state of the right link is ON only if a transition from ON to OFF in the associated Boolean variable is sensed when the left link is ON.

Table A-2 Coils

Momentary Coils		
1	*** --( )--	Regular Coil The state of the left link is copied to the associated Boolean variable and to the right link.
2	*** --(/)--	Negated coil The state of the left link is copied to the right link. The inverse of the state of the right link is copied to the associated Boolean variable.
Latched Coils		
3	*** --(S)--	SET (latch) coil The associated Boolean variable is set to the ON state when the left link is in the ON state, and remains set until reset by a RESET coil.
4	*** --(R)--	RESET (unlatch) coil The associated Boolean variable is set to the OFF state when the left link is in the ON state, and remains reset until set by a SET coil.
Retentive Coils*		
5	*** --(M)--	Retentive (memory) coil
6	*** --(SM)--	SET retentive (memory) coil
7	*** --(RM)--	RESET retentive (memory) coil
Transition-Sensing Coils		
8	*** --(P)--	Positive transition-sensing coil The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from OFF to ON is sensed. The state of the left link is always copied to the right link.
9	*** --(N)--	Negative transition-sensing coil The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from ON to OFF is sensed. The state of the left link is always copied to the right link.

* The action of Coils 5, 6, and 7 is identical to that of Coils 1, 3, and 4, respectively, except that the associated Boolean variable is automatically declared to be in retentive memory without the use of the VAR RETAIN declaration.

A.2.2 PLC Ladder Logic Example

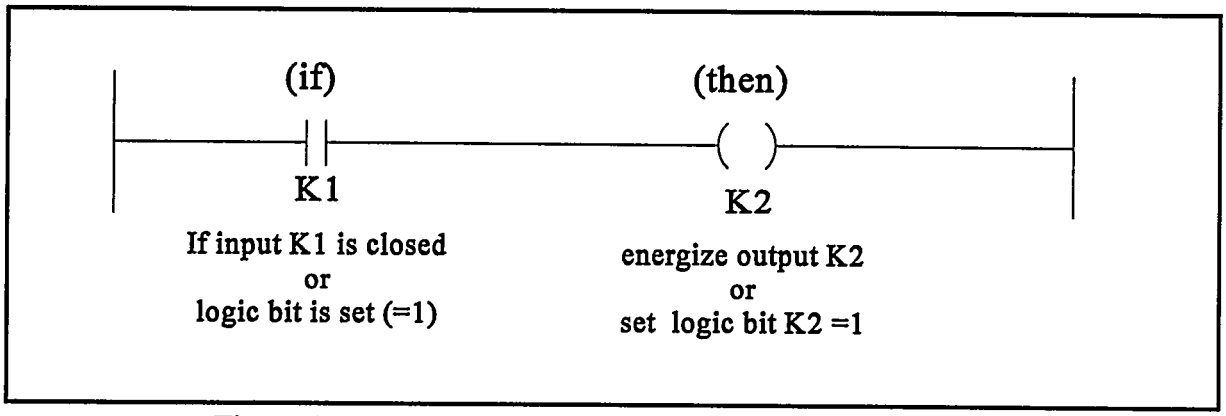


Figure A-3 Ladder logic "rung" with IF/THEN configuration.

Ladder Logic language provides a unique representation for computer programs. In ladder logic each line of code is graphically displayed as a "rung" of a ladder. The top rung instructions are performed first and then each consecutive rung instructions are performed in their respective sequence. As shown in Figure A-3, each rung consists of an IF(input)/THEN(output) decision. The left half of the rung contains a condition that must be true for any output instruction(s) on the right half of the rung to be performed. If the left side of the rung does not contain a condition, the output instruction on the right side is performed continuously.

A problem with ladder logic program structure is the potential for unintended behavior. This can be shown even in the simple example above using the distinction between retentive and non-retentive output instructions. A non-retentive output will reset or turn off. A retentive output will remain in its last state. Although Logic rungs are logic elements and need to be logically true in order to execute the output (or outputs) on that rung, should the rung NOT be logically true, then the output could still perform an action if the output is retentive.

Figure A-4 is another example which presents the implementation of two-out-of-three (2oo3) voting in Ladder Logic. The first "rung", numbered 0, implements the 2oo3 voting in Ladder Logic. Any two of the tree inputs being ON will close two contacts in one of the three parallel paths and energize the coil labeled ACTUAL_INPUT

The value of ACTUAL_INPUT is defined by:

```
ACTUAL_INPUT    =    (INPUT_1*INPUT_2)    +    (INPUT_2*INPUT_3)    +
(INPUT_1*INPUT_3), where
```

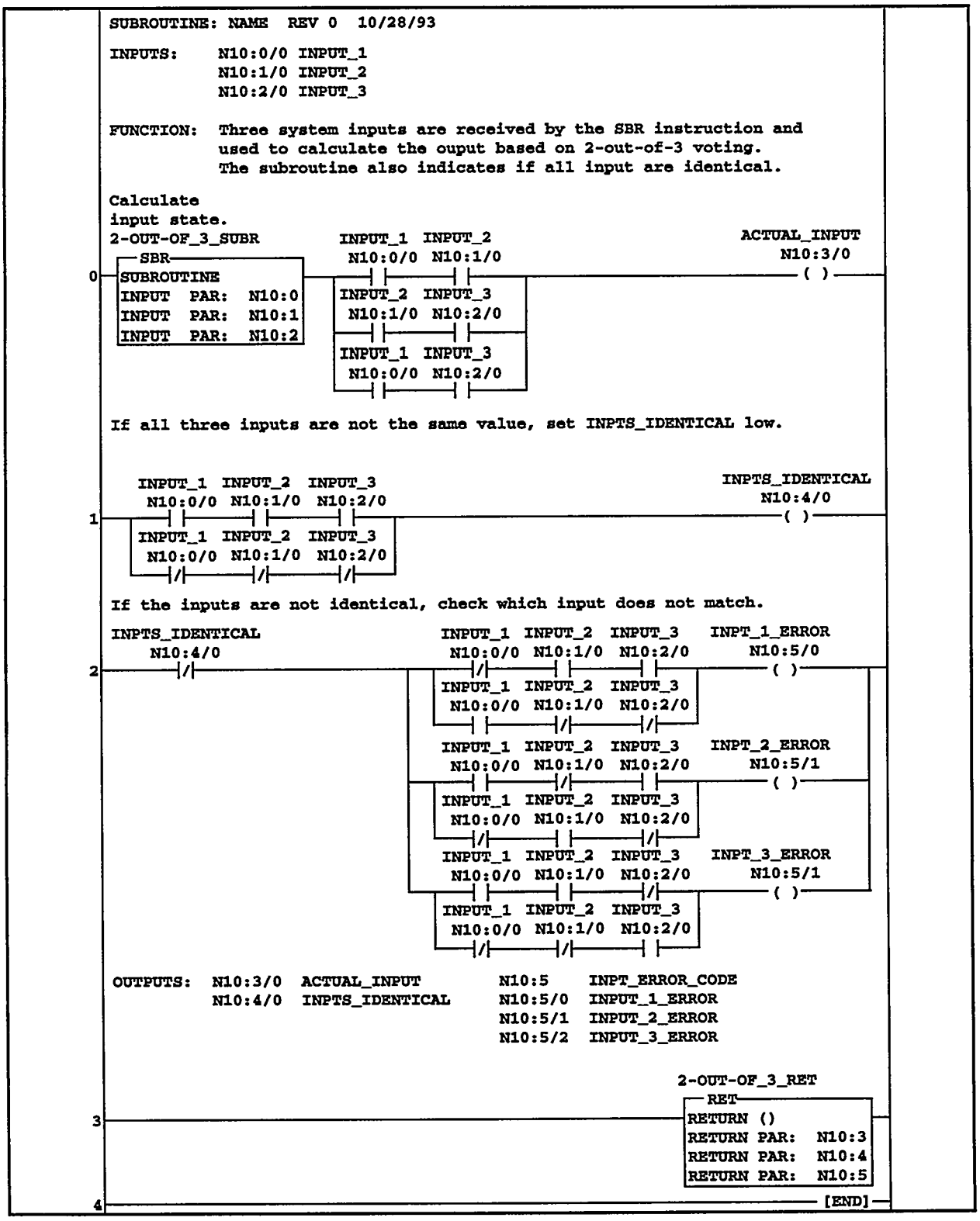


Figure A-4 Example of Ladder Logic.

* = AND operand
+ = OR operand

Rungs 1 and 2 of the subroutine are for annunciation only. The coil in rung 1 is energized if the three inputs are either all ON or all OFF. Rung 2 identifies the input which differs from the other two if all three are not identical. Note that all six permutations of the three inputs are present in rung 2. Rung 3 simply outputs the results generated by the previous rungs.

A.2.3 General Description - Ladder Logic Programming Shell

As noted above, the ladder logic application development environment, or programming "shell" provides functions for the development of the Ladder Logic application software. This shell features are a key factor in the development, testing, and verification of ladder logic programs and include:

- Ladder Logic editing
- On-line communication with PLC processor for:
  - Uploading/downloading Ladder Logic files to processor's memory
  - On-line Ladder Logic editing of program in processor's memory
  - Real-time monitoring of PLC status for debugging
- Generation of printed Ladder Logic reports.

A development platform is used to run the Ladder Logic programming shell and maintain the Ladder Logic files. (In most cases the platform used is an IBM PC/AT compatible). The shell communicates from the development platform to the PLC via a specialized hardware communications link. At no time does any shell software run on the target PLC processor.

Editing of the Ladder Logic may be performed in either an OFF-LINE or an ON-LINE mode. In both cases, the shell software converts the binary Ladder Logic information into a graphic screen display that may be modified by the user. Changes made to the Ladder Logic in the OFF-LINE mode are saved in the binary file. In the ON-LINE mode, changes are made directly to the PLC program/data memory via a live communications link. (Changes made in this manner must subsequently be uploaded from the PLC if they are to be saved in the binary file for configuration management purposes.)

The programming shell software maintains a number of supplemental files in addition to the binary file to form a complete Ladder Logic project database. These files primarily contain symbolic and comment information used strictly to aid the user in the development process. They have no impact on the data structures contained in the binary file or the PLC memory.

The on-line communication capability of the programming shell is required to move Ladder Logic (application software binary file) information between the PLC processor and the development platform, where the user interface resides. As mentioned above, this feature can be used to edit Ladder Logic[30]. It can also be used to download a Ladder Logic program residing in the binary file to the PLC or to upload a binary from the PLC to file.

Run-time debugging is another function performed using the on-line communication feature. Various "windows" into the PLC memory can be set up to view memory contents updated dynamically as the processor is running the application software. A "histogram" function, which can record the changes to a particular memory location over time to a log file, is also available.

Generation of printed Ladder Logic reports is the final key function of the programming shell software that is required for the development of the Ladder Logic application software. The printed report is the output of a conversion from the binary Ladder Logic data files to a human-readable text format. The accuracy of this conversion is critical because it provides the only written documentation of the application as it resides within the PLC.

A.2.4 Ladder Logic Modularization

Some Ladder Logic implementations provide the feature of subroutines. These are Ladder Logic programs that can be called by another program. When a subroutine is called, control is transferred to the subroutine, until it encounters a RETURN command[31], which transfers control to the next rung. Each subroutine is stored in a different file. With each subroutine it is possible to associate unique files of local variables[32]. Subroutines can also access all the global variables defined in the program. Figure A-5 shows the mechanism of calling a subroutine in Ladder Logic. In this example the "main" program in file #2 calls a subroutine in file #10.

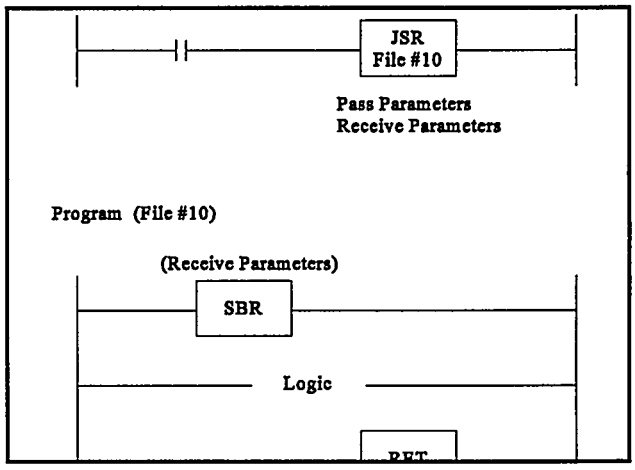


Figure A-5 Subroutine calling in Ladder Logic.

[30]This may possible in the PLC memory directly. However, some PLC's require that a copy of the program on the hard disk or operator interface computer be identical to the PLC's memory. Changes are made to the disk or offline copy. Once completed, the shell software interacts with the PLC, gaps memory and inserts the new/changed rung.

[31]Or END OF PROGRAM statement

[32]Not all PLCs support local variables

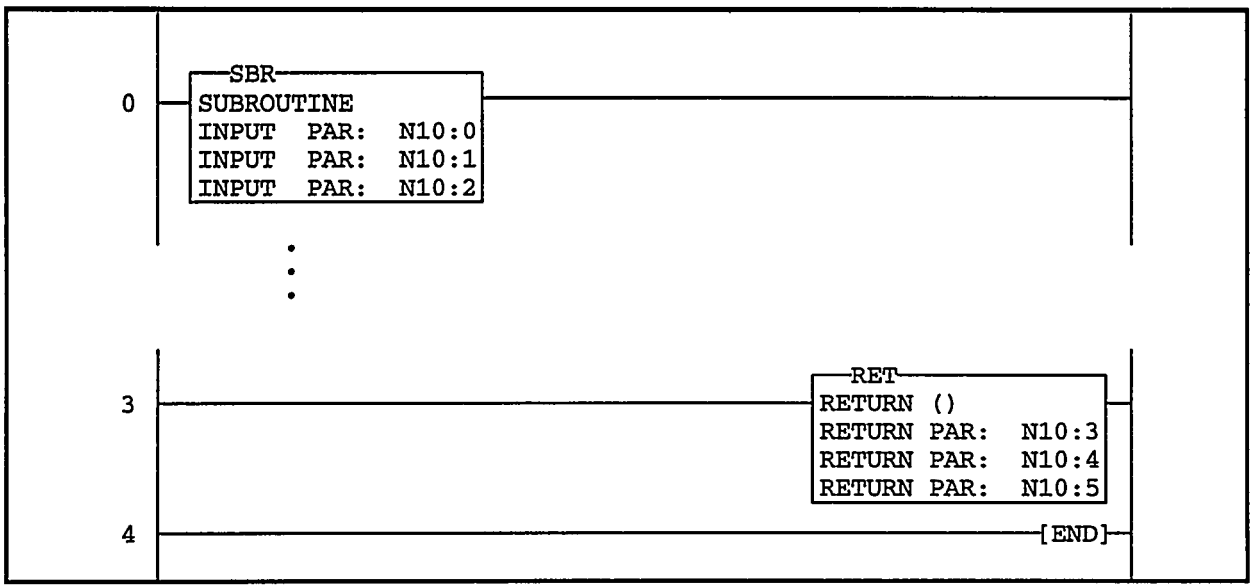


Figure A-6 Subroutine interface (parameter passing).

The subroutine call specifies which parameters shall be passed to the subroutine, and which parameters are returned by the subroutine. For example, Figure A-6 shows a subroutine that accepts three input parameters words (N10:0, N10:1, and N10:2), and returns three output parameters words (N10:3, N10:4, and N10:5).

The calling instruction, shown in Figure A-7, passes the parameter stored in N9:2 to N10:0, N9:3 to N10:1, and N9:5 to N10:2. The subroutine returns parameter N10:3 to N9:11, N10:4 to N9:13, and N10:5 to N9:14.

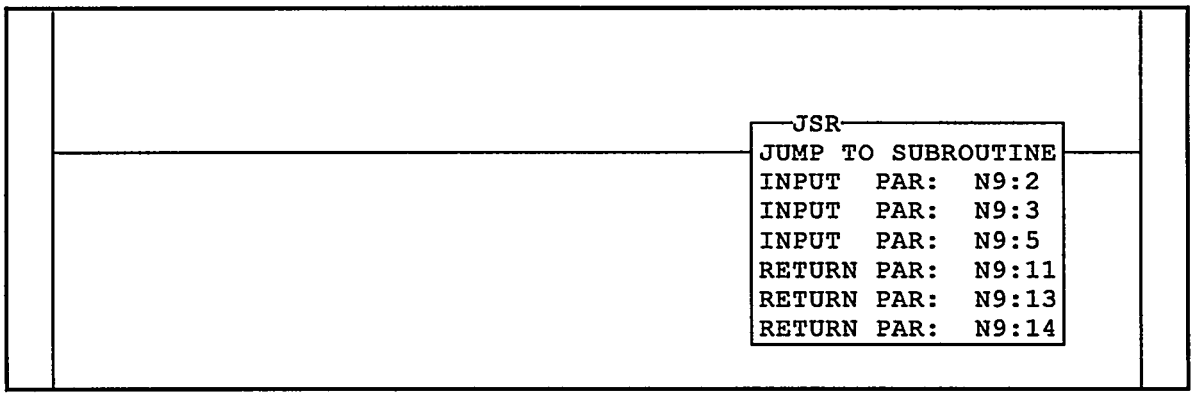


Figure A-7 Subroutine call interface (parameter passing).

A.3 Sequential Function Charts

SFC is not a language but a structuring tool for the organization of programs. SFC elements provide

a means of partitioning a program into a set of "steps" and "transitions." Each step is associated with a set of operations that are performed while this step is active. Under certain conditions a transition becomes active, the current step is not executed anymore, and another step is executed. The SFC helps to modularize programs that can be broken into exclusive steps, each step executed under different conditions.

In sequential function chart programs, steps and transitions are arranged in series and parallel paths, and they are numbered with the file numbers that contain their ladder logic. The programmable controller scans the logic of a step repeatedly until its transition logic goes true. Then the program scan moves to the next step or steps, and the previously active step is turned off.

At a high level of abstraction, without considering the detail, SFCs are similar to subroutines. Benefits of using SFC for programming PLCs include:

•SFC, as a dedicated sequencing language, has a closer cognitive fit than any of the other PLC languages to the types of sequencing operations commonly performed by PLCs. This makes reading and writing SFC programs much simpler than programs written solely in Ladder Logic.

•As the machine sequence is represented directly by the SFC program, both machine and programming problems are typically much easier to find and correct.

•As inactive SFC steps and transitions are not scanned by the PLC program, the program scan time of the PLC is typically greatly reduced.

The actual code executed in a Step or Transition can be written in Ladder Logic, IL, ST, Function Block, or, in some PLCs, SFC. In the case where a Step's actions are written in SFC, that step is referred to as a macro-step. The necessity for specifying the actions of a Step and a Transition's condition in a language other than SFC is why SFC is sometimes described as a meta-language. SFC is, however, a sequencing language in its own right.

An example sequential function chart program is shown in Figure A-8 (Hughes, 1989) to explain the symbols used in a typical program. The top of the program contains a start block to define the beginning of the program. The next block is the initial step, where the programmable controller starts function chart execution and returns to this step from the end of program unless directed otherwise by the program logic. This block is identified by a double-sided box.

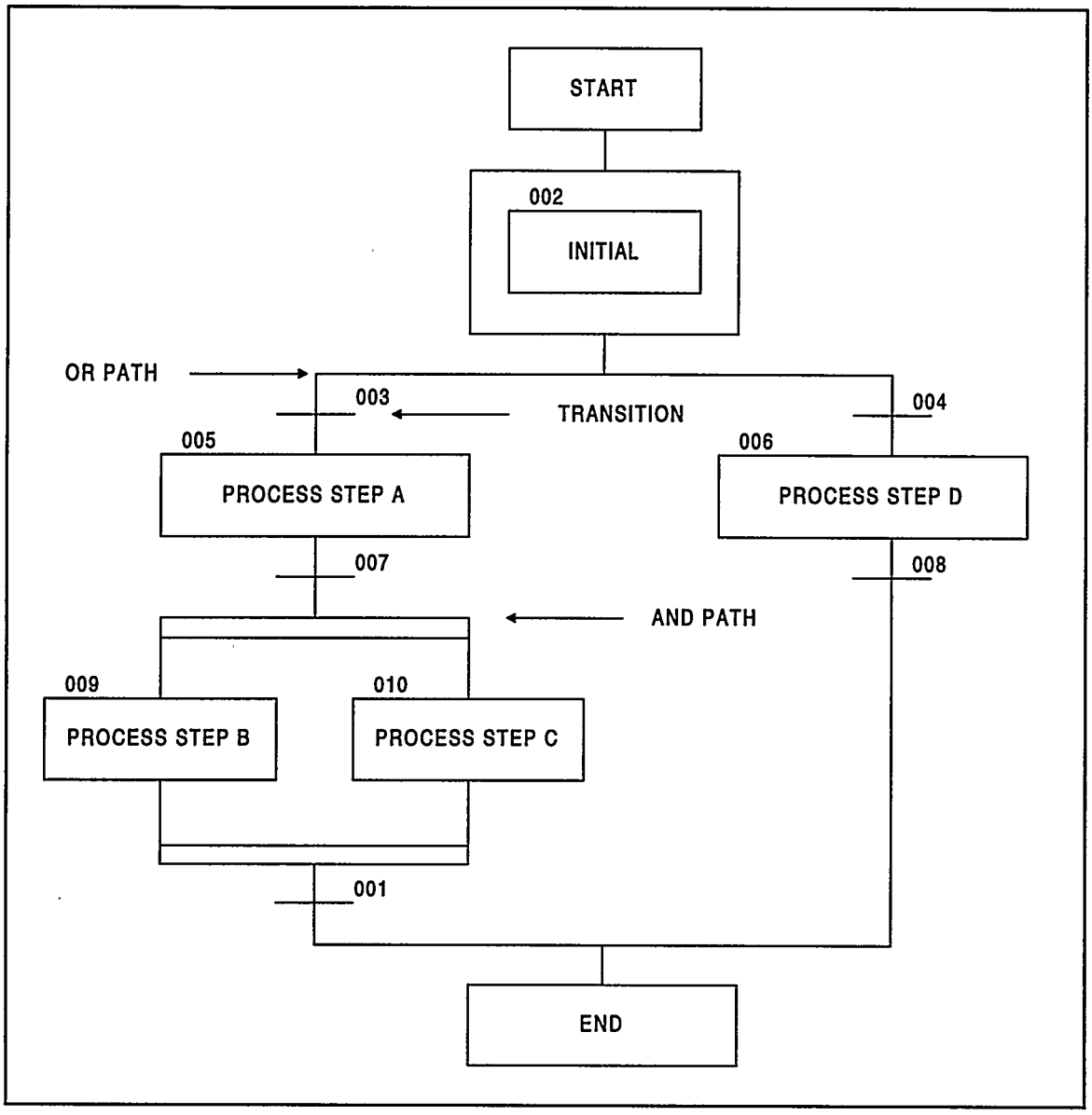


Figure A-8 Example of Sequential Function Chart

As noted above, the step block is the function chart's basic unit and contains ladder logic for each independent stage of the process or machine operation. It is identified by a single-sided box.

The transition is the logic condition that the processor checks after completing the active step. When the transition logic is true, the step preceding the transition is disabled, and the step following it becomes active. The transition is normally a single logic rung, identified by a short horizontal line below its corresponding stop (see Figure A-8).

The OR (divergence of sequence) path is identified by a single horizontal line at the beginning and end of a logic zone. The processor selects one of several parallel paths depending on which transition goes true first. Normally, the number of parallel paths is limited to seven.

The AND (simultaneous sequence) path is identified by a horizontal double line at the beginning and end of a zone. The processor can normally execute up to seven paths at the same time.

The following sections provide additional details on each of these constructs.

A.3.1 SFC Steps

A step represents a situation in which the behavior of a program follows a set of rules defined by the associated actions of the step. A step can be either active or inactive. An active step is executed (scanned). An inactive step is not executed (not scanned). At any given moment a program might have more than one active step. A step can be seen as a subroutine that is called when the active condition occurs. The call to the subroutine is avoided when the inactive condition occurs. At any given time the state of the program is defined by the response of the active steps to their respective inputs.

Each step is identified by a label and has a program that invokes the actions performed by the step. Steps are represented as boxes containing an identifying number. A Step must always be followed by a transition.

A.3.2 SFC Transitions

A Transition represents the condition whereby control passes from steps preceding the transition, to one or more successor steps. When a transition is true it causes the exit from the preceding step and entry into the following step. The transition is represented by a horizontal line across the vertical link. Each transition has an associated transition condition which is the result of the evaluation of a boolean expression. The IEC standard states that it shall be an error if any side effect (such as the assignment of a value to a variable other than the transition name) occurs during the evaluation of a transition condition. Most PLC SFC implementations expressly prevent this from occurring, however, even if a particular implementation allows side effects in transition expression execution, this type of programming construct should be strictly avoided. Every Transition must be followed by a Step.

A.3.3 SFC Actions

Zero or more actions can be associated with each step. A step that has zero actions shall be considered as having a WAIT function, that is, waiting for a successor transition condition to become true.

A.3.4 SFC Control Structures

The control structures used in Sequential Function Charts include Divergence of Sequence Selection, Simultaneous Sequences, and Directed Links.

- *Divergence of Sequence Selection:* A Divergence of Sequence is described by the case where a single Step has multiple alternate Transition conditions and associated sequences following it. When the Step is active, all of these Transition conditions are scanned by the PLC. The first of these to become true 'selects' the single sequence that will be followed subsequently.

- *Simultaneous Sequences:* Simultaneous Sequences are used when a number of parallel machine sequences need to be started and stopped simultaneously. A Simultaneous Sequence is represented by a double horizontal line following a Transition, and followed by several Steps. The number of Steps allowed to follow a Simultaneous Sequence is an implementation dependent issue. Different implementations of SFC will scan the active steps in a Simultaneous Sequence in different orders. Therefore, it is considered poor programming practice to have the proper operation of a Simultaneous Sequence depend upon the order of processing of active steps in these sequences within a single scan. The PLC program auditor should explicitly check for this. Simultaneous Sequences are used when parallel processes need to be explicitly synchronized at their beginning and their ending. Where asynchronous sequences that do not require this kind of synchronization are desired, they should be coded as independent SFC Charts.

- *Directed Link:* The Directed Link is used to move control from one portion of a SFC program to another. It has two forms. The first, is as a continuous line with arrows indicating the direction of control flow. The second form uses a 'goto' arrow and an associated 'label' in place of the continuous line. In this form, each goto has a single unique label to receive the control flow. Directed Links cannot be used to jump into, out of, or between paths of a simultaneous sequence. Each SFC program must contain at least one Directed Link, to return control flow back to the designated Initial Step.

Figure A-9 is an example of an SFC program that uses Divergence of Sequence Selection, Simultaneous Sequences, and Directed Links. This example concerns the operation of a traffic light, with normal operation during high traffic hours, and blinking lights after midnight. The Divergence of Sequence Selection selects between these two modes based on the time of day: a daytime mode with the familiar Red, Green, Yellow sequence, and a late night mode where blinking red lights are substituted for this sequence. The Simultaneous Sequence is used to start the East/West sequence of lights at the same time as the North/South sequence, and to ensure that they end at the same time, so that the action of these two sequences remains synchronized (very important for traffic control applications). The directed link is shown leading from the last process step back to the beginning of the application.

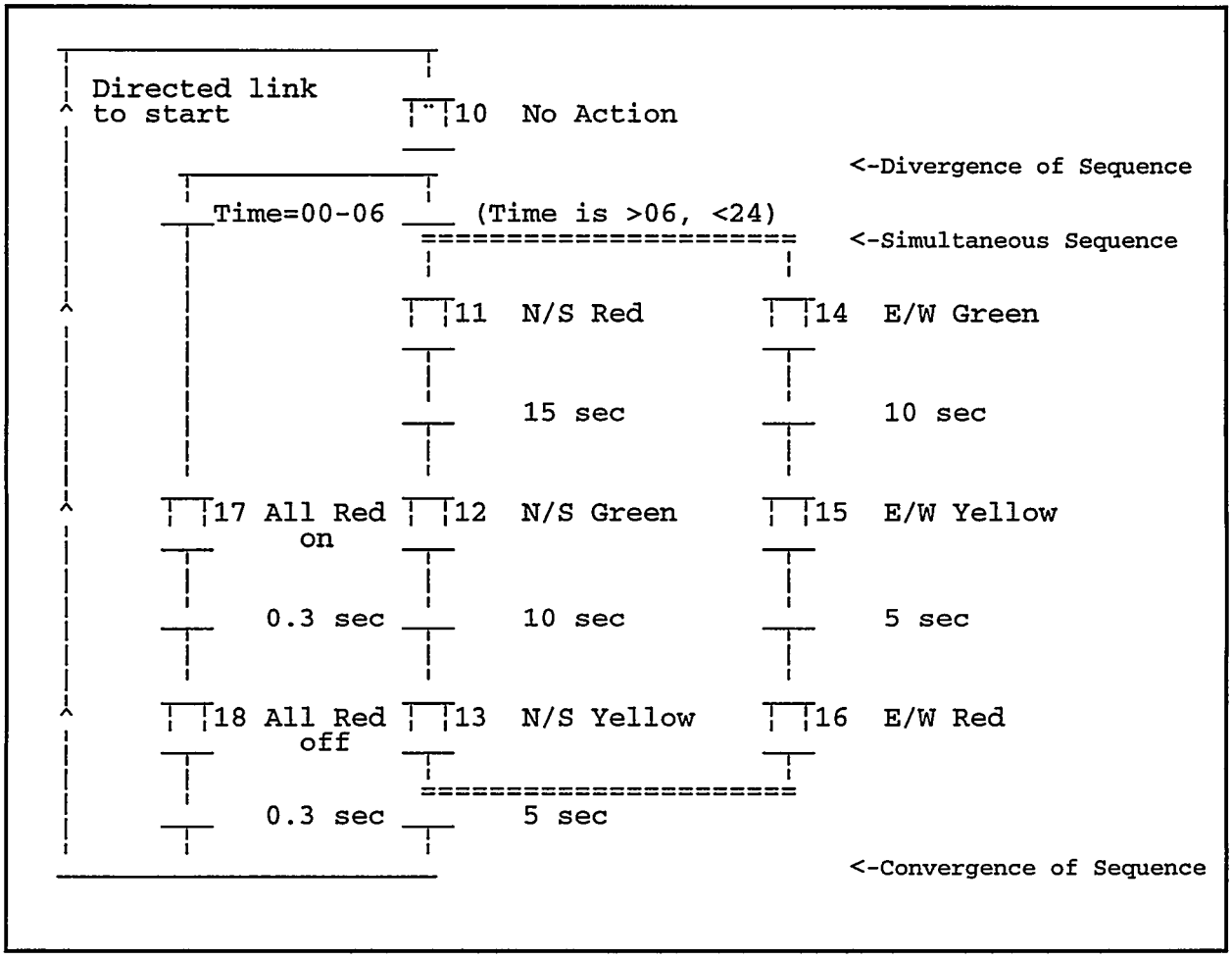


Figure A-9 Sequential Chart for Traffic Light

A Divergence of Sequence selection occurs under Step 10, as indicated by a single horizontal line. The two Transitions involved in the Divergence of Sequence Selection are programmed to be mutually exclusive, by the time of day. The Sequence including Steps 11 through 16 is an example of simultaneous sequences, with one sequence for the East/West lights (Steps 14, 15,and 16), and one sequence for the North/South Lights (Steps 11, 12, and 13). This Simultaneous Sequence is indicated by a double horizontal line at both the beginning and ending of the construct, as specified in IEC standards 1131-3 and 848.

The transition under Step 13 is a 5 second timer, which begins activation when ALL of the final steps in the preceding Simultaneous Sequence (in this case, 13 and 16) are active. This is standard behavior for the transition condition immediately following the end of a Simultaneous Sequence - the transition is only executed when ALL of the prior Steps are active.

Steps 17 and 18, which are only active between midnight and 6:00 AM because of the preceding divergence of sequence selection, blink red lights on the North/South sides, and yellow lights on the

East/West side for 0.3 seconds apiece. Finally, the directed link returns control of program flow to the initial Step, Step 10. As one of the transitions under Step 10 will *always* be true, the small amount of time spent in Step 10 does not affect proper operation of the traffic light.

A.4 Structured Text Language

ST (Structured Text) is a high-level, block structured, text-based language. It is a small language, with only about 40 keywords, and 10 different types of statements. Several PLC manufacturers provided high-level text-based languages prior to the advent of IEC 1131-3. These languages have influenced the design of ST. However, languages of this type have not been as pervasive as Ladder Logic. As a result there are fewer ST implementations, and all are generally based on the IEC 1131-3 standard. ST shares its execution model, program organization model, and variable handling model with the other four IEC 1131-3 languages which were described in the first section of this appendix.

An example of an ST *Configuration* showing the above-described features is given in the IEC 1131-3 specification. This *Configuration* is reproduced here, for purposes of illustration:

```
CONFIGURATION CELL_1

      VAR_GLOBAL
            W: UINT;
      END_VAR

      RESOURCE STATION_1 ON PROCESSOR_TYPE_1

            VAR_GLOBAL
                  Z1: BYTE;
            END_VAR

            TASK SLOW_1(INTERVAL := t#20ms, PRIORITY := 2);
            TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1);

            PROGRAM P1 WITH SLOW_1:
                  F(X1 := %IX1.1);

            PROGRAM P2 :
                  G(OUT1 =>W,
                                    FB1 WITH SLOW_1,
                                    FB2 WITH FAST_1);
      END_RESOURCE

      RESOURCE STATION_2 OF PROCESSOR_TYPE_2

            VAR_GLOBAL
                  Z2:               BOOL;
                  AT %QW5:    INT;
            END_VAR

            TASK PER_2(INTERVAL := #t50ms, PRIORITY := 2);
```

```
            TASK INT_2(SINGLE   := Z2,      PRIORITY := 1);

            PROGRAM P3 WITH PER_2:
                  F(X1 :=Z2, X2:= W);

            PROGRAM P4 WITH INT_2:
                  H(HOUT => %QW5, FB1 WITH PER_2);

      END_RESOURCE

      VAR_ACCESS
            ABLE    : STATION_1.%IX1.1      : BOOL READ_ONLY;
            BAKER   : STATION_1.P1.X2       : UINT READ_WRITE;
            CHARLIE: STATION_1.Z1           : BYTE;
            DOG     : W                     : UINT READ_ONLY;
      END_VAR

END_CONFIGURATION
```

This *Configuration*, CELL_1, consists of two *Resources*, STATION_1 and STATION_2. Each of these Resources has two *Tasks* associated with it, and two *Programs* as well. Note that some *Function Blocks* within these programs are associated with different Tasks. This Configuration also defines four Global Variables - W with a scope of the entire Configuration, Z1 with a scope limited to Resource STATION_1, and variables Z2 and %QW5 (A Directly Referenced Word Output Variable) with a scope limited to Resource STATION_2. Finally, this Configuration ends with a VAR_ACCESS section, which assigns Configuration-wide scoped global variables to their direct I/O addresses.

Following the Variable Declaration part, an ST program consists of a number of statements, containing valid ST expressions. Expressions yield single values, and are composed of operators and operands. Operands may be literal values, variables as described above, or *Function* invocations. The operators available in ST are shown in the following table, in order of highest precedence to lowest:

Table A-3 List of ST Operators (in descending order of precedence)

Operation	Symbol
Parenthesization	( )
Function Invocation	Identifier(argument list)
Exponentiation	**
Negation	-
Complement	NOT
Multiply	*
Divide	/
Modulo	MOD
Add	+
Subtract	-
Comparisons	<,>, <=, >=
Equality	=
Inequality	<>
Boolean AND	&
Boolean AND	AND
Boolean Exclusive OR	XOR
Boolean OR	OR

The ST Languages contain the following statement types. All ST IEC 1131-3 Function Block Diagrams statements (other than comments) are terminated with a semicolon.

Comments: Comments are placed between '(*' and '*)' symbols, like so:

```
(* This is a comment *)
```

Assignment: Assignment allows the transfer of the value of a valid ST Expression to a variable. ':=' is used as the assignment operator. Some examples:

```
A := B;
C := C+1;
D := SIN(X) + 12 ;
```

Function Block Invocation:

Function Blocks are invoked by the name of the *Function Block*, followed by a parenthesized list of input assignments. The input assignments need not be in any particular order, and need not be complete. If an input value is not given in *Function Block* invocation, the previous value of that input, or initial value if this is the first call to this *Function Block*, is used.

Outputs from the *Function Block* are given addresses of the form `function_block_name.output_name`. As an example, consider a Function Block named `FUN`, with inputs `IN1`, `IN2`, and `IN3`, and outputs `OUT1`, `OUT2`, and `OUT3`.

The following would be legal uses of this *Function Block*:

```
FUN(IN1:= 1, IN2:=B, IN3:= %IW2.3);
FOO := FUN.OUT1;
BAR := FUN.OUT2;

FUN(IN2:=FOO, IN1:= BAR);
BAZ := FUN.OUT1 * 5;
```

Return:

Used to return from a *Function* or *Function Block*.

Selection:

ST has two kinds of selection statements: the `IF..THEN..ELSE..END_IF` statements, and the `CASE..OF` statements. Complete examples follow:

```
IF FOO > MAX THEN
      MAX := FOO;
ELSIF (FOO  < MAX) AND  (FOO > MIN) THEN
      MID := FOO;
ELSE
      MIN := FOO;
END_IF;
CASE BAR OF
      1,5:  BAZ := FRED;
      2  :   BAZ := GEORGE;
ELSE
      BAZ := BAR;
END_CASE;
```

Iteration:

The `FOR..DO`, `WHILE..DO`, and `REPEAT..UNTIL` statements make up the available iteration statements available in ST. The `EXIT` statement is used to leave a loop prior to satisfaction of the tested condition. Short examples of the syntax of these follow:

```
FOR I := 1 TO 100 BY 2 DO
      IF WORDS[I] = 'KEY' THEN
            J := I;
            EXIT;
      END_IF;
END_FOR;

J := 1;
WHILE J <= 100 DO
      J := FUN(J) +J;
END_WHILE;

K := 1;
REPEAT
      K := K+2;
      UNTIL K>100
END_REPEAT;
```

A.5 Function Block Diagram Language

FBD is a free-form graphic language, designed to look very much like a circuit diagram, or logic diagram commonly in use by hardware engineers. The definition of FBD's format and appearance is actually specified by IEC 617, part 12. The IEC 1131-3 specification formalized the IEC 617 notation as a programming language versus a documentation standard.

The IEC 1131-3 specification is based on implementations from Germany. Implementations of FBD tend not to differ as much from one another as existing implementations of Ladder Logic Diagrams. As an IEC 1131-3 specified language, FBD shares the IEC 1131-3 Architecture with the other languages described in that specification.

Like many graphically-oriented languages for industrial programming, FBD programs consist of a number of boxes, referred to as Function Blocks, connected by lines. Each Function Block, as seen in Figure A-10, has inputs on the left side, outputs on the right side, and some notation indicating its function. Inputs and outputs may or may not have an associated label. Some sample Function Blocks are shown here, in the ASCII-derived notation used by the IEC 1131-3 specification.

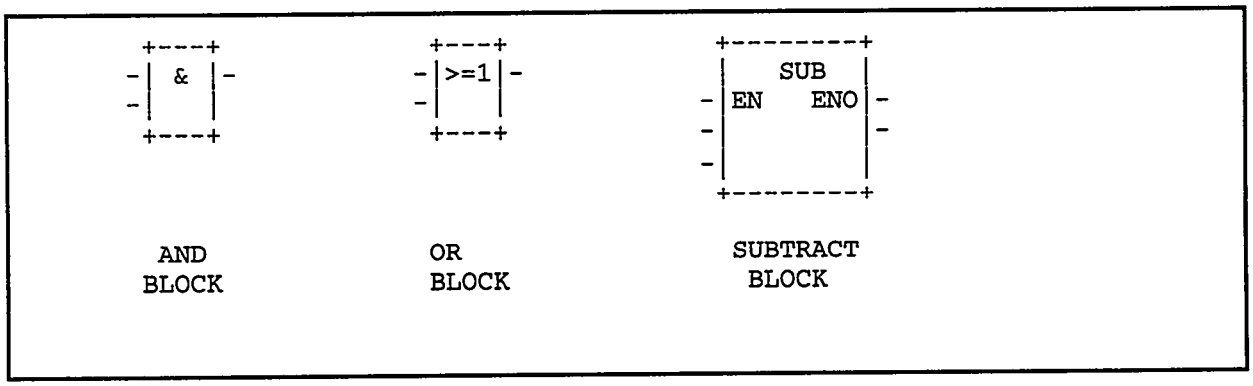


Figure A-10 Examples of Function Blocks in IEC 1131-3 ASCII Notation

The lines in FBD denote Signal Flow, versus the Power Flow concept in Ladder Diagram, or the Activity Flow concept denoted by the lines in an SFC diagram. Unlike Ladder Diagram, the Signal Flow represented by a line in FBD may denote any of the available data types supported in a particular PLC system - a line may be a Boolean value, an Integer or a Real. Also in contrast to Ladder Diagram, where Power Flow proceeds from left to right across the diagram, lines indicative of Signal Flow in an FBD program may proceed in any direction as long as outputs of one block are connected to inputs of the next. An example of an FBD program that uses several data types is shown in Figure A-11.

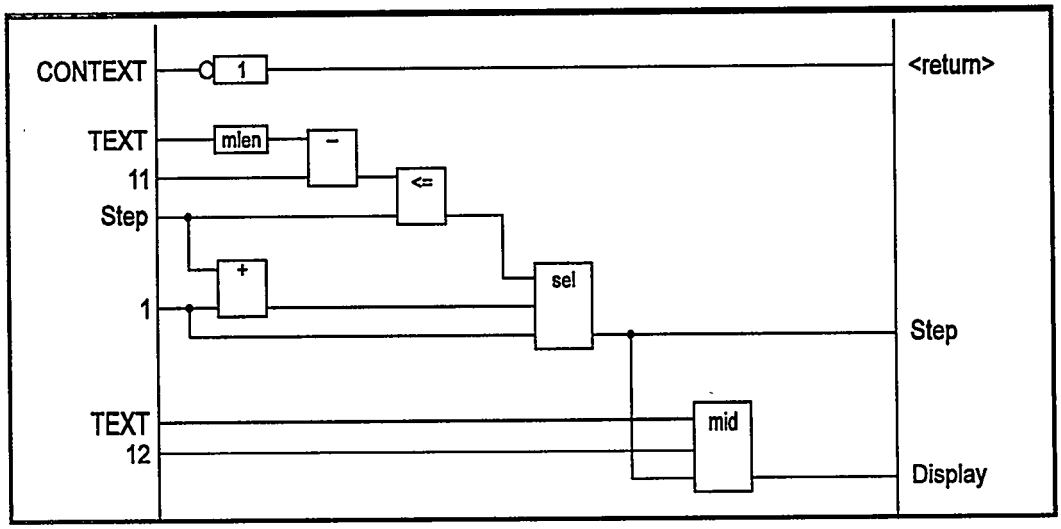


Figure A-11 Example of FBD Data Types

In this example, CONTEXT has been defined as a BOOLEAN input, TEXT is a Message (character string) Input, Step is an Integer, and Display is a Message Output. This program fragment selects characters from the TEXT variable and places them into the Display variable in accordance with the value of the Step input variable, if and only if the CONTEXT input variable is TRUE.

Looking at the example, it can be seen that Signal Flow lines may branch when it is required to transport a value to several places. Combining of values, of course, must be done in a function block; however, there is no FBD equivalent of the Ladder Diagram 'wired-or' construct. It can also be seen that crossing of Signal Flow lines without affecting the signals is allowed, as is the case just to the left of the 'mid' function block in the above example. Branches are distinguished from simple crossings by the presence of a dot at the site of the branch, as would be expected.

Execution order in an FBD program is determined by data flow. The Function blocks that have all of their inputs available first are executed first. Then the function blocks that have their input parameters determined by the function blocks just executed are executed, and so on. Large, highly interconnected FBD diagrams can thus have non-obvious execution orders associated with them. Some FBD editing systems provide tools to assist with the visualization of execution order.

Generally, any function or function block in an IEC 1131-3 compliant programming system is accessible in FBD. This means that all of the standard functions defined in the IEC specification, any user-defined functions and function blocks, and any extended functions and functions blocks provided by a particular implementation may be used in an FBD program.

A.6 PL/M

This appendix section discusses the PL/M programming language. The first section describes its history, the second describes how executable code is generated, the third section contains a top-level description of the language itself, and the final section provides additional language-specific recommendations on the project level.

A.6.1 Language History

The *Programming Language for Microcomputers* (PL/M) was introduced in 1976 by the Intel Corporation. It was introduced to provide a higher-level language for their 8-bit, 8080 microprocessor. PL/M was modeled after IBM's popular PL/1 structured programming language. At that time BASIC and FORTRAN IV were the dominant popular higher-level languages. PL/M was the first block-structured language available for microcomputers and encouraged the use of structured programming techniques developed and promoted by the IBM Corporation, and others such as E. Dijkstra and C.A.R Hoare.

Intel developed the PL/M Language, compiler, linker, and simulator as a proprietary language to promote the sales of their silicon products. Unlike other languages, the standards were kept within Intel, and they did not seek standard certification from ANSI or the IEEE.

PL/M has been an evolutionary language for Intel. It has evolved with and in support of the Intel microprocessor product line. The following table is a partial list of PL/M compilers and their status.

Table A-4 PL/M Compilers

Processor	Compiler	Status
8080/8085/Z80	Intel PL/M-80	In public domain
	BSO Tasking 80/PL	Available
8051/8052	Intel PL/M-51	Discontinued
	BSO Tasking 8051 PL/M	Available
8096/80196	Intel PL/M-96	Discontinued
8086/80186	Intel PL/M-86	Discontinued 3/94
	BSO Tasking 80/PL	Available
80286	Intel PL/M-286	Discontinued 3/94
80386/80486	Intel PL/M-386	Discontinued 12/94

The PL/M languages differ considerably among target microprocessors. These differences are so great that they create a safety issue. This is addressed in Section A.6.4.2 below. Information on the PL/M languages may be obtained from documents shown in the references at the end of this chapter. In general, documents made available by the Intel Literature Department provide the most comprehensive information on the structure and syntax of the language (Intel, 1980; Intel, 1984; Intel, 1985a; Intel, 1985b; Intel, 1988; Intel, 1992).

In the late 1980s and early 1990s, "C", Ada, and other more advanced languages became popular. Market pressures, in conjunction with the popularity of the new languages, gradually caused Intel to phase out and diminish support for the PL/M language set. The user should be aware of this trend when choosing to make long-term plans to use and support products with PL/M. Recommendations and guidelines are discussed in the sections below.

A.6.2 Generation of Executable PL/M Programs

PL/M compilers translate source code into relocatable object modules. These modules can then be combined with other modules coded in PL/M, assembly language, or other higher-level languages. The compilers provide listing outputs, error messages, and a number of compiler controls that aid in developing and debugging programs.

To complete a software program, the object modules developed are combined with any necessary support libraries using a Linkage Editor program such as LINK86, BND286, or BND386. The resulting program is still in relocatable format and requires one last step to make it executable.

A Locate program transforms the relocatable program module into a program with absolute addresses. This locator program properly divides the program into EPROM / ROM sections and into RAM data memory sections. The locator also assigns the system STACK address. After the program modules are combined and located, the program can be debugged using an in-circuit emulator

system (such as ICE-386), or a software debugger such as DB86 or DB386. PL/M is a data-typed language. The compiler does data-type compatibility checking during compilation to help detect logic errors in programs.

A.6.3 Language Overview

PL/M accesses the hardware features of the target microprocessor more easily than C. This section briefly discusses program structure, data types, addressing mechanisms, and interrupt structures. Authoritative descriptions of the language are shown in the reference list for this Appendix.

A.6.3.1 PL/M Program Structure

PL/M is a block structured language. Every statement in a PL/M program is a part of at least one block. A block is a well-defined group of statements that begin with a DO statement or a PROCEDURE declaration statement, and end with an END statement.

Every PL/M program consists of one or more modules, each containing one or more DO-END blocks. Each program module must begin with a labeled DO statement and end with an END statement. A module DO-END block can contain other nested DO-END blocks; however, it cannot itself be contained or nested inside of another block.

Between the DO and END statements there are other PL/M statements that constitute program logic. These PL/M statements are said to be a part of the corresponding DO-END block that surrounds it. DO-END blocks can be nested inside each other within the module block.

The second type of block in a PL/M program is the Procedure Definition Block. This block begins with a procedure definition statement (PROCEDURE) and ends with an END statement. Like the DO-END block, other PL/M statements can be placed within the procedure block to form procedure program logic.

A unique aspect of PL/M is that procedure blocks can be nested. This feature allows PL/M to keep support procedures local and hidden from other procedures and code blocks outside the containing procedure.

A.6.3.2 Data Types

In the original PL/M-80 compiler, only two data types were defined: BYTE (unsigned 8-bit), and ADDRESS (a 16-bit unsigned value). ADDRESS values were store in high-byte, low-byte reverse order according to the 8080 architecture. These data types corresponded with the register data width of the basic 8080/8085 microprocessor.

PL/M-86 has additional data types of WORD, INTEGER, REAL, and POINTER. Data type ADDRESS became synonymous with WORD for compatibility, and use of WORD was encouraged over ADDRESS.

PL/M-286 and 386 further added new data types including OFFSET and SELECTOR. WORD was a 16-bit number in PL/M-286, but became a 32-bit number in PL/M-386. Data-type mapping compiler controls of $WORD16 and $WORD32 were introduced in an attempt to provide some basic data-type compatibility for the 80x86 processor family.

A.6.3.3 Addressing Mechanisms

In the 80x86 processors and above, keeping the segmented address mechanisms hidden from the user is a problem. In using pointers, the user has to deal with SELECTORS and OFFSETS and their differences between processors. Use of these addressing mechanisms requires detailed attention, as they all differ among the 8086, 80286, and the 80386.

A.6.3.4 Interrupt Structures, I/O Schemes, and Flags

PL/M-86 defines methods for setting up interrupt vectors for the 8086. PL/M-286/386 defines an Interrupt Descriptor Table for handling interrupts in 80286/80386 applications. In general, the handling of interrupts are not transparent and compatible. They must be given specialized attention for each processor. I/O schemes also differ according to processor family – some are confined to 8 bits only, while others allow multi-byte I/O.

Hardware flags such as SIGN, CARRY, and numerous others are also hardware dependent. They often contain entities which vary in width from 8-bits to 32-bits. Bit assignments for like flags are not necessarily found in the same order between processors.

A.6.4 General Guidelines for Using PL/M

PL/M is a language that has experienced a decline in use and popularity in the industry over the past few years. Part of the reason for the decline may be the proprietary nature of the language; it is not supported by any outside standards committees such as ANSI or the IEEE.

A.6.4.1 An Almost Obsolete Language

The PL/M language is no longer popular, and this fact should be weighed carefully by organizations desiring to use or continue using this language. Although some third-party vendors may continue supporting PL/M into the future as part of their product line, no vendor focuses on providing PL/M as its prime or flagship product.

On December 31, 1994, Intel discontinued PL/M-386. This was the last of the PL/M products in its software development product line. The PL/M-86/286 product had already been discontinued in March 1994. And, although no final date was provided, PL/M-51 has apparently been discontinued for some time. Intel's oldest PL/M product, PL/M-80 for the 8080/8085, has been placed in the public domain. Copies are available for download from the Intel BBS electronic bulletin board system[33] Intel offers information and support to customers on a PL/M to "C" source-code converter program to facilitate the conversion.

A.6.4.2 Target Platform Specific Language

Unlike ANSI standard languages such as "C," the syntax and semantics of certain areas of PL/M vary according to the processor. PL/M can be grouped into families that have similar attributes. For example, PL/M-80, PL/M-86, and PL/M-96 are somewhat similar; and an upgrade chapter is provided in the Intel manuals. PL/M-286 and PL/M-386 are also very similar. However, PL/M-51 for the 8051/8052 microcontroller family is very different from other versions of PL/M.

All of the PL/M languages have the same control structure elements. The major areas in which PL/M compilers differ are: *data types, addressing mechanisms, interrupt structures, I/O schemes,* and *hardware flags.*

A.6.4.3 Third Party Support

At this time, only two companies have been identified that support the PL/M language. Only one of them offers a PL/M compiler. The second company has a line of language translation programs, one of which is a PL/M to "C" translator.

Boston Systems Office Tasking, also known as BSO, has two PL/M compilers.[34] One compiler is described as supporting all derivative processors of the 8051 family. It is hosted on MSDOS /PCDOS, micro-VAX, VMS/Ultrix, HP9000, SUN-3, and the Apollo 3000 systems. The compiler is claimed to be fully compatible with the Intel PL/M-51 language definition. The differences between the BSO/Tasking compiler and the Intel PL/M-51 compiler, in the Company's words, are:

- BSO/Tasking PL/M-51 allows bit-structures to be AT'ed at byte variables in bit-addressable memory.
- BSO/Tasking PL/M-51 supports floating point arithmetic.

---

[33] Intel Embedded Control Systems Electronic Bulletin Board, (916)356-3605.

[34] Boston Systems Office/Tasking, 333 Elm Street, Dedham MA 02026, (617)320-9400.

- Compiler controls and invocation commands are mostly different.
- Object modules are not generated.
- Error messages are different.
- Compiler limits generally equal or exceed those of Intel PL/M-51.

Program language conversion support for PL/M is provided by Micro-Processor Services, Inc[35]. The company advertises that it provides a full line of PL/M to "C" converter programs for PL/M-51, PL/M-80, PL/M-86, PL/M-96, PL/M-286, and PL/M-386. The firm also provides translation and conversion services on a per-source-code line basis for those not wishing to purchase and operate the conversion programs.

A.6.5 New Project Guidelines and Recommendations

The guidelines below should be followed.

- Ensure an adequate supply of PL/M language tools exist.
- Archive or store additional tools to last the expected duration of the system or product.
- Search for and become acquainted with companies, individuals, or consultants who can provide continued support for the PL/M language.
- Prepare for a migration path to an alternate language.

A.6.4.5 Existing Project Guidelines and Recommendations

For those individuals or groups that must supply maintenance and upgrade support for products already containing PL/M source code, the above guidelines still apply. In addition, project software developers and project leaders should make appropriate management individuals aware of the long-term trend for PL/M.

As the supply of software tools and talent diminish, individuals and organizations should actively participate and support special interest groups and Internet discussion groups to continuously seek out and maintain contact with those still familiar with PL/M. Many participants of the Internet newsgroup: **comp.arch.embedded** have shown interest in discussions on the PL/M language. This group apparently includes a number of individuals who have used the language in the past.

---

[35] Microprocessor Services Inc., 92 Stone Hurst Lane, Dix Hills, NY 11746, (516)499-4461

Conversely, the most popular and prominent consultants newsgroup, **alt.computer.consultants,** has shown no activity, interest, or response to news thread topics on the PL/M language.

BSO Tasking is one organization whose software product offering does provide for a graceful transition. The first example in Chapter 7 of the main volume shows a good migration path for software developed for the 8051. At the top level, PL/M modules can be migrated to "C" and still comfortably fit with the complete BSO development tool line. A similar migration path also exists for Intel PL/M-51 to Intel C-51.

References

Allen Bradley, *PLC-5 Programming Software - Programming*, Publication 6200-6.4.7, November, 1991.

Allen Bradley, *PLC-5 Programming Software - Software Testing and Maintenance*, Publication 6200-6.4.10 November, 1991.

ICOM PLC-5 Ladder Logic, User's Manual.

ANSI/IEEE 729-1983, Glossary of Software Engineering Terminology, Institute of Electrical and Electronic Engineers, 1983

Hughes, T.A., *Programmable Controllers*, Instrument Society of America, Research Triangle Park, NC, 1989

International Electrotechnical Commission (IEC), *Programmable Controllers General Information,* IEC Standard 1131, Part 1, 1992. (Available in the U.S. from the American National Standards Institute, New York.)

International Electrotechnical Commission (IEC), *Programmable Controllers Programming Languages*, IEC Standard 1131, Part 3, 1993. (Available in the U.S. from the American National Standards Institute, New York.)

Intel Corporation, *PL/M Programming Manual*, 9800268B, Chandler, Arizona, 1977.

Intel Corporation, *PL/M-86 Programming Manual*, 9800466-02B, Chandler, Arizona, 1980.

Intel Corporation, *8086 Software Tool Box, Volume II,* 122310-001, Chandler, Arizona, 1984.

Intel Corporation, *PL/M-86 User's Guide*, 121636-004, Chandler, Arizona, 1985.

Intel Corporation, *8086 Software Tool Box*, 122203-002, Chandler, Arizona, 1985.

Intel Corporation, *PL/M-96 User's Guide for DOS Systems*, 481644-001, Chandler, Arizona, 1988.

Intel Corporation, *PL/M Programmer's Guide*, 452161-002, Chandler, Arizona, 1990.

Intel Corporation, *PL/M-386 Programmer's Guide*, 611052-001, Chandler, Arizona, 1992.

Appendix B. Summary of Language Guidelines

This Appendix contains tabular summaries of the language guidelines for the languages discussed in the main body of the report. In addition to the summary, a relative weighting for the guideline is provided. These weightings are general and may change based on the specifics of each project.

Generic (Language Independent) Attributes

Generic (Language Independent) Attributes

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
2.1.1.1 Dynamic Memory Allocation	High	Minimize dynamic memory allocation.	Use of dynamic memory can cause crashes.	Release allocated memory as soon as possible.
2.1.1.2 Memory Paging and Swapping	High	Minimize memory paging and swapping.	Memory paging and swapping can cause significant delays in response time.	Not Applicable.
2.1.2.1 Structure	Medium	Avoid *goto*'s.	*goto*'s make execution time behavior difficult to fully predict as well as introducing uncertainty into control flow.	Clearly document, justify, and test.
2.1.2.2 Control Flow Complexity	High	Minimize control flow complexity.	Excess complexity makes it difficult to predict the program flow and impedes review and maintenance.	Project guidelines should set specific limits on nesting levels.
2.1.2.3 Initialization of Variables	High	Initialize variables before use.	Uninitialized variables can cause anomalous behavior.	Not Applicable.
2.1.2.4 Single Entry and Exit Points	Medium	Use single entry and exit points in subprograms.	Multiple entries and exits introduce control flow uncertainties.	Document secondary entry and exit points.
2.1.2.5 Interface Ambiguities	Medium	Minimize interface ambiguities.	Interface errors account for many coding errors.	Not Applicable.
2.1.2.6 Data Typing	High	Use data typing.	Data typing prevents misuse of data.	Not Applicable.
2.1.2.7 Precision and Accuracy	High	Provide adequate precision and accuracy.	Correct results needed in safety critical calculations.	Not Applicable.
2.1.2.8 Order of Precedence	Medium	Use parentheses rather than default order of precedence.	Incorrect precedence assumptions cause errors; source code open to misinterpretation.	Use other forms to enhance readability if parentheses are excessive.

Generic (Language Independent) Attributes

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
2.1.2.9 Side Effects	Medium	Avoid functions or procedures with side effects.	To avoid unplanned dependencies and bugs.	Not Applicable.
2.1.2.10 Separating Assignment from evaluation	Medium	Separate assignments from evaluation statements.	Incorporation of assignments into evaluation statements can cause unanticipated side effects.	Not Applicable.
2.1.2.11 Program Instrumentation	Medium	Minimize run-time perturbations.	These practices improve checkout and verification of code.	Intrusive instrumentation is sometimes necessary for problem resolution. Remove instrumentation and perform regression testing.
		Maintain visibility of instrumentation in run-time source code.		
		Conform to software instrumentation guidelines.		
2.1.2.12 Library Size	Medium	Control class library size.	Large class libraries are unmanageable and have performance penalties.	Not Applicable.
2.1.2.13 Dynamic Binding	High	Minimize dynamic binding.	Dynamic binding causes unpredictability in name/class association and reduces run-time predictability.	Justify dynamic binding.
2.1.2.14 Operator Overloading	Medium	Control operator overloading.	Operator overloading is problematic for predictability.	Sometimes acceptable for achieving uniformity across different data types.

Generic (Language Independent) Attributes

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
2.1.3.1 Tasking	High	Minimize the use of tasking.	Timing uncertainties, sequence on execution uncertainties and vulnerability to race conditions and deadlocks may result.	Tasking requires compelling justification.
2.1.3.2 Interrupt Driven Processing	High	Minimize the use of interrupt driven processing.	Interrupts lead to non-deterministic response times.	Minimize processing for handling interrupts. Return to primary program control as soon as possible.
2.2.1.1 Internal Diversity	Medium	When internal diversity is used, all interface versions must be identical.	Internal diversity minimizes the possibility of design or implementation-related failure.	Deviation from common interfaces should be documented and justified.
2.2.1.2 External Diversity	Medium	External diversity should be implemented in a disciplined manner.	External diversity minimizes the possibility of design or implementation-related failure.	Not Applicable.
2.2.2.1 Local Handling of Exceptions	High	Handle exceptions locally.	Local exception handling helps isolate problems more easily and more accurately.	If not possible, thorough testing and analysis to verify behavior during exception handling is required.
2.2.2.2 Maintain External Control Flow	High	Preserve external control flow by handling the exception in the responsible module.	Interruption of control flow creates uncertainty in execution.	If not possible, thorough testing and analysis to verify behavior during exception handling is required.

Generic (Language Independent) Attributes

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
2.2.2.3 Uniform Exception Handling	High	Use general and defined exceptions, conform to specific project guidelines on exceptions and uniform placement.	Undisciplined use of exception handling can result in inconsistent processing of the same exception condition in different parts of the code.	Not Applicable.
2.2.3.1 Input Data Checking	High	Check input data validity.	Checks reduce the probability of crashes and incorrect results.	May not be applicable if input can be "trusted."
2.2.3.2 Output Data Checking	High	Check output data validity.	Checks reduce the probability of crashes and incorrect results.	May not be necessary if downstream input checking performed.
2.3.1 Built-In Functions	Low	Control the use of built-in functions through project specific guidelines.	Built-in functions have unknown internal structure, limitations, precision and exception handling.	Conduct thorough testing and error tracking.
2.3.2 Compiled Libraries	Low	Control the use of precompiled libraries.	Precompiled libraries have unknown internal structure, limitations, precision and exception handling.	Conduct thorough testing and error tracking.
2.4.1.1 Indentation Guidelines	Medium	Conform to indentation guidelines.	Indentation guidelines improve readability and maintainability.	Not Applicable.
2.4.1.2 Descriptive Identifier Names	Medium	Use descriptive identifier names.	Descriptive identifier names improve readability and maintainability.	Not Applicable.
2.4.1.3 Comments and Internal Documentation	Medium	Conform to comment guidelines.	Necessary to verify conformance to requirements, code inspections and maintenance.	Not Applicable.

Generic (Language Independent) Attributes

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
2.4.1.4 Subprogram Size	Medium	Limit subprogram size in accordance with project coding standards.	Facilitate review and maintenance.	Justify larger sizes. Provide with additional documentation and comments.
2.4.1.5 Mixed Language Programming	Medium	Minimize mixed language programming.	Mixed language programming is hard to read and maintain.	Separate "foreign" code so that readability is enhanced.
2.4.1.6 Obscure or Subtle Programming Constructs	High	Minimize obscure and subtle programming constructs.	Obscure coding presents problems in review and maintenance and raises safety concerns.	When it cannot be avoided, use comments to minimize the impact of obscure or subtle code.
2.4.1.7 Dispersion of Related Elements	Medium	Minimize the dispersion of related elements.	Dispersed elements necessitate multiple accesses to review or maintain code and are therefore susceptible to errors.	Provide clear references, rationale and overall source code organization.
2.4.1.8 Use of Literals	Medium	Minimize the use of literals.	The use of symbolic constants enhances code reliability and consistency.	Associate comment with each literal to facilitate search/replace.
2.4.2.1 Global Variables	Medium	Minimize the use of globals.	Globals have the potential for undesired side effects.	Clearly identify global variables.
2.4.2.2 Complexity of Interfaces	Medium	Minimize the complexity of interfaces.	Complex interfaces are a frequent cause of software failures.	Closely inspect and clearly identify interfaces.
2.4.3.1 Single Purpose Function and Procedure	Medium	Use single purpose functions and procedures.	Functional cohesion facilitates review and maintenance.	Clearly identify and rationalize groupings of functions.
2.4.3.2 Single Purpose Variables	Medium	Use each variable for a single purpose only.	To facilitate review and maintenance.	Not Applicable.

Generic (Language Independent) Attributes

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
2.4.4.1 Isolation of Alterable Functions	Medium	Isolate alterable functions.	Isolation of alterable functions facilitates review and maintenance.	Clearly comment alterable sections.
2.4.5.1Isolation of Non-Standard Constructs	Medium	Isolate implementation dependent constructs.	Simplifies porting to changed hardware configurations.	Not Applicable.

Ada83

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
3.1.1.1 Dynamic Memory Allocation	High	Dynamic use of memory should be strongly discouraged.	Dynamic memory can cause the heap to grow too large and crash the system.	Release allocated memory as soon as possible. Utilize the length clause feature to pre-allocate dynamic memory pools. Put *STORAGE_ERROR* exception handlers in program units which allocate dynamic memory.
		Avoid dynamically created tasks.	Dynamic tasks use up unknown and potentially large amounts of dynamic memory. Memory allocated to dynamic tasks cannot be explicitly deallocated.	Not applicable.
		Minimize use of unconstrained types.	Due to their impact on memory allocation.	Use with caution and justify.
		Avoid recursion.	Recursion uses up unknown and potentially large amounts of dynamic memory.	Put exception handlers for *STORAGE_ERROR* in recursive subprograms.
		Do not instantiate generic units during runtime.	Generic units are not desirable in any safety-significant software.	Not Applicable.
		Minimize use of large composite objects.	Large objects can cause stack overflows.	Not Applicable.
		Use **length** clauses if dynamic memory allocation is necessary.	To reserve memory in advance.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Provide handlers for the predefined exception **STORAGE_ERROR** if dynamic memory allocation is necessary.	To provide a graceful recovery from memory exhaustion.	Not Applicable.
		Explicitly deallocate dynamic memory.	The run-time executive's garbage collector should not be relied upon.	Not applicable.
		Do not assign values of dynamically allocated access objects to other access objects.	They may point to invalid addresses if the original memory is deallocated.	Not applicable.
3.1.1.2 Memory Paging and Swapping	High	No Ada specific guideline, see the generic guideline.	Not Applicable.	Not Applicable
3.1.2.1 Structure	Medium	Do not use *goto*'s.	The use of *goto*'s clouds the code structure and should be avoided.	Clearly document, justify, test.
		No more than one *exit* statements for a loop.	More than one or two exit conditions from a loop indicate lack of cohesion, i.e., more than one purpose for loop.	Clearly document, justify.
		Minimize *return* statements.	Multiple *return* statements can confuse meaning of subprogram.	Clearly document and justify.
3.1.2.2 Control Flow Complexity	High	Minimize control flow complexity.	Excess complexity makes it difficult to predict the program flow and impedes review and maintenance.	Project guidelines should set specific limits on nesting levels.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Use *if .. elsif* instead of nested *if* statements.	Reduces complexity and nesting levels.	Not applicable.
		Use *case* statements in preference to *if .. elsif* statements whenever possible.	Reduces complexity and nesting levels.	Use *if* statements if only two branches or if control path not dependent upon discrete value.
		When using *case*, also use *when others* to catch unplanned or unknown alternatives.	*when others* traps unplanned and unknown alternatives.	Clearly document, justify, test.
3.1.2.3 Initialization of Variables	High	Initialize all variables.	Ada compilers will not generally initialize variables, therefore the contents are undefined.	The access type is always initialized to *null*.
		If initialization is via function call, perform initialization in program unit body, not declaration section.	Body of function may not have been elaborated when declaration section of program unit is being elaborated; *PROGRAM_ERROR* exception could be raised.	*Elaborate* pragma can be used to ensure body of function elaborated before declaration section.
		Do not initialize very large objects via aggregates.	Could cause system crash in some implementations.	Not applicable.
3.1.2.4 Single Entry and Exit Points	Medium	One normal (non-exception) *entry* and *exit* per subprogram.	Single *entry* and *return* points are easier to understand, test, and less expensive to design, build, and maintain than multiple entries and returns.	Document secondary entry and exit points.
		Limit the number of exception entry/exit points.	To avoid complicating the control flow.	Clearly document each entry/exit point.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid multiple task entry points.	To minimize program complexity.	Not Applicable.
3.1.2.5 Interface Ambiguities	Medium	Minimize interface ambiguities.	Interface errors account for many coding errors.	Not Applicable.
		Explicitly specify the modes of parameters.	Aids understandability for those who do not know default mode.	Not Applicable.
		Restrict the use of *in out* mode.	Reduces ambiguity; makes plain the intention regarding changes in parameter. Results in safer coding; objects passed into subprograms and meant to be unchanged cannot unintentionally be changed.	Objects of limited type cannot be *out* mode.
		Use named parameters for calling functions and procedures.	Using named parameters for calling functions and procedures, improves readability and reliability.	Not Applicable.
		Use target type instead of access type when data of the target type only is to be processed.	Reduces ambiguity; makes plain which data, pointer or target data, is to be processed. Results in safer coding; objects of a target type passed into subprograms and meant to be unchanged cannot unintentionally be changed.	Clearly justify, document, and test.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid aliased parameters.	When aliased parameters are used, results from subprograms frequently incorrect and implementation dependent.	Clearly document, justify, and test any use.
3.1.2.6 Data Typing	High	Take advantage of Ada's strong typing.	Strong typing catches range errors as well as typing errors, making safer code.	Not Applicable.
		Do not suppress Ada's run-time constraint checks.	Strong typing conducts run-time range checks of parameters entering procedures and functions and of copy operations to variables.	Clearly document and justify any deviation.
		Define scalar data types with the narrowest possible limits.	Enhances early detection of out-of-range data values.	Not Applicable.
		Use care in scalar subexpressions in Ada 83 implementations.	Some implementations check intermediate values for out-of-range conditions.	Use implementations that do not have this problem.
		Minimize the use of type conversions.	Type conversions subvert the benefits of strong typing.	Justify and document.
		Avoid use of unchecked conversions.	This may lead to assigning illegal values to objects.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Limit the use of access type data.	Access data is harder to verify and maintain.	Limit direction to situations where there are no other reasonable alternatives, performing validation on indirectly accessed data prior to setting or use to ensure the correctness of the accessed data.
		Avoid declaring variables in package specifications.	To increase data abstraction and reduce coupling.	Not Applicable.
3.1.2.7 Precision and Accuracy	High	Do relational tests on real values with <= and >= rather than <, >, =, and /=.	The values of fixed point and floating point numbers only approximate the specific numbers. The operations on these numbers are also approximations. Therefore, proper precision and accuracy are necessary for critical systems.	Not Applicable.
		Use Ada attributes in comparisons and checking for small values for real numbers.		
		Test carefully using Ada attributes around special values such as 0.0.		
3.1.2.8 Order of Precedence	Medium	Use parentheses or other mechanisms for ensuring that the order of evaluation of operations is explicitly stated.	The default order of precedence should not be depended on if any misinterpretation can be made.	Use other forms to enhance readability if parentheses are excessive.
		Use parentheses to separate operations of different precedence.	Less chance of misinterpretation.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Use parentheses or other mechanisms for ensuring that the order of evaluation of operands is correct.	If expressions contain functions with side effects that affect each other, results of expressions will be implementation dependent.	Avoid functions with side effects.
3.1.2.9 Side Effects	Medium	Verify that functions do not have side effects.	Side effects can lead to problems with unplanned dependencies and can cause bugs that are very hard to find.	Not Applicable.
3.1.2.10 Separating Assignment from Evaluation	Medium	Separate assignments from evaluations.	To avoid side effects.	Not Applicable.
3.1.2.11 Program Instrumentation	Medium	No Ada specific guideline, see the generic guideline.	Not Applicable.	Not Applicable.
3.1.2.12 Library Size	Medium	No Ada specific guideline, see the generic guideline.	Not Applicable.	Not Applicable.
3.1.2.13 Dynamic Binding	High	Minimize dynamic binding.	Dynamic binding causes unpredictability in name/class associations and reduces run-time predictability.	All cases where dynamic binding is required should be justified.
3.1.2.14 Operator Overloading	Medium	Guidance on use of operator overloading should be included in a project specific standards manual and coding should comply with this standard.	Operator overloading can be problematic from the perspective of predictability because it is unclear how a compiler would bind code for different polymorphic code.	Sometimes acceptable for achieving uniformity across different data types.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid operator overloading when the inherent precedence of the operator is different from that desired.	Misinterpreting operator precedence could lead to incorrect results for expressions.	Use parentheses to override inherited precedence. Document usage.
		Preserve conventional meaning of overloaded operators.	Failure to preserve conventional meaning results in confusing code.	Not Applicable.
3.1.3.1 Tasking	High	Minimize the use of tasking.	Timing uncertainties, sequence of execution uncertainties, and vulnerability to race conditions and deadlocks.	Tasking requires compelling justification.
		Avoid using the *abort* statement.	If a task is aborted, then all tasks dependent on it are aborted. Furthermore subprograms and blocks which were called by it will also be aborted. If the task was suspended, the abort will cause it to appear to have been completed. Delays are canceled by aborts.	Aborts require compelling justification.
		Avoid dynamic tasking.	Dynamic tasking complicates the predictability of the run-time behavior of the program.	Thoroughly justify dynamic tasking and thoroughly test that all problems it can cause are handled.
		Use *delay*.only for waiting, not synchronization.	*delay* sets a **minimum** time delay.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Minimize the number of *accept* and *select* statements per *task*. Minimize the number of *accept* statements per *entry*.	These guidelines are motivated by the reduction of conceptual complexity and the need to control the task body size. Additionally, a large number of *accept* and *select* statements carries with it a large amount of inter-task communication and overhead.	Not applicable.
		Avoid conditional entry calls. Avoid selective waits with else parts. Avoid timed entry calls. Avoid selective waits with delay alternatives.	Use of these constructs always poses a risk of race conditions. Their use in loops, particularly with poorly chosen task priorities, can have the effect of busy waiting. Also these constructs are very much implementation dependent.	Justify any usage. Thoroughly test for occurrence of race conditions.
		Minimize the use of the *PRIORITY* pragma.	Ada tasking is based on preemption and requires that tasks be synchronized only by features of the language. The scheduling algorithm is not defined by the language and may vary from time slice to preemptive priority. Some implementations provide several choices that a user may select for the application.	In real-time systems it is often necessary to tightly control the tasking algorithm in order to obtain the required performance. This may require non-preemptive tasking. Program such tasking in Ada.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
3.1.3.2 Interrupt Driven Processing	High	Declare interrupt values using named constants and isolate them from other declaration clauses.	Interrupts lead to non-deterministic response times. Interrupt entries are implementation dependent features that may not be supported.	Not Applicable.
		Isolate interrupt receiving tasks into implementation dependent package bodies.		
		Pass interrupts to tasks via normal entries.		
		Do not use task entry points for interrupt processing.		
3.1.3.3 Runtime Environment	Medium	Characterize timing for the Ada Runtima Environment (RTE).	The RTE is delivered by vendors and needs to be tested to ensure that it is deterministic, functionally correct, and satisfies timing requirements.	Not Applicable.
3.1.3.4 Automatic Memory Management	Medium	Avoid automatic memory management.	Automatic garbage collection if a source of timing uncertainty.	Not Applicable.
3.2.1 Software Diversity	Medium	No Ada specific guideline, see the generic guideline.	Not Applicable	Not Applicable
3.2.2.1 Local Handling of Exceptions	High	Minimize propagation of exceptions.	It may obscure program logic.	Justify and document clearly.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Localize handling of predefined exceptions.	System failures can be avoided if exceptions are handled locally.	If not possible, use thorough testing and analysis to verify behavior during exception handling.
3.2.2.2 External Flow Control	High	Declare exceptions to be handled in calling program units alongside declaration of called unit.	Allows calling program unit to choose what action to take upon raising of exception.	Not Applicable
3.2.2.3 Uniform Exception Handling	High	Clearly express and document exception handling.	To clarify control flow.	Not Applicable.
		Handle predefined exceptions.	To catch unexpected error conditions.	Not Applicable.
		Do not raise predefined exceptions explicitly.	To avoid unanticipated behavior.	Not Applicable.
		Handle all program-defined exceptions.	It is not good practice to ignore exceptions in safety critical systems; they can be propagated to the Real Time Executive and cause the system to come down.	Not Applicable
		Use exception handling only for abnormal events.	Exceptions increase control flow complexity and should not be used where inappropriate.	Not Applicable.
		Minimize side effects.	Eliminates side effects in case of exceptions.	May increase time and memory requirements by unallowable amounts.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid compiler vendor specific exceptions.	Inhibits portability and understandability.	Justify and document any use of compiler specific exceptions.
		Use and flag **other** in exception handler definitions.	All conditions in exception handling must be well defined.	Not Applicable.
3.2.3 Input and Output Data Checking	High	Check input data.	Input data should be regarded as untrustworthy until proven otherwise.	May not be applicable if input can be "trusted". Output checking may not be necessary if downstream input checking is performed.
3.3.1 Built-In Functions	Low	Ada has few built-in functions. Therefore no Ada specific guideline. See the generic guideline.	Not Applicable	Not Applicable
3.3.2 Compiled Libraries	Low	Avoid built-in libraries.	Libraries prevent the programmer from knowing the accuracies, limitations, robustness, and exception handling of the built-in functions.	Thorough testing and error tracking.
3.3.4 Traceability between Source and Compiled Code	High	Maintain traceability between source code and compiled code.	To avoid configuration management problems.	If separate compilation is needed, use the **with** clause and appropriate tools.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
3.3.5 Generic Units	Medium	Minimize the use generic units.	Generic units obscure traceability.	If using generic units, - Instantiate only during initialization - Avoid global variables - Document the restrictions on parameters.
3.4.1.1 Indentation Guidelines	Medium	Indentation improves readability and allows the reader to see the structure of the program.	This is especially useful for finding the beginnings and ends of data structures and control flow structures.	Not Applicable.
		Indent and align nested control structures, continuation lines, and embedded program units consistently. Also, distinguish between indentations for nesting and for continuation lines.	Improves readability.	Not Applicable.
3.4.1.2 Descriptive Identifier Names	Medium	Follow project-specific guidelines on naming.	Not Applicable.	Not Applicable.
		Separate words in compound names with underscores.	This will improve reliability because the reader will be able to more easily read the names.	Not Applicable.
		Use underscore "_" to promote readability on numbers.	This will improve reliability because the reader will be able to more easily read the numbers for verification.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Use abbreviations with care.	Avoid jargon, use the names given by the application.	Not Applicable.
3.4.1.3 Comments and Internal Documentation	Medium	Source code shall be supplemented with Ada comments that explain the code.	Comments clarify code and help code maintenance.	Not Applicable.
		Use comments to relate code to higher level design.	Comments should clarify data structure, not repeat what source code states.	Not Applicable.
		Use blank lines to delineate related statements.	Improves understandability.	Not Applicable.
		Avoid use of escapes from language restrictions.	Justified on individual basis elsewhere.	Not Applicable.
		Indicate the scope of renaming.	To improve reviewability.	Not Applicable.
		Provide comments on exception handling.	To improve flow control.	Not Applicable.
		Provide comments on dynamic memory allocation.	Dynamic memory allocation is discouraged and need proper justification and documentation.	Not Applicable.
		Provide comments on tasking.	To provide traceability.	Not Applicable.
3.4.1.4 Subprogram Size	Medium	No Ada specific guideline, see the generic guideline.	Not Applicable.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
3.4.1.5 Mixed Language Programming	Medium	Avoid machine code inserts.	There is no requirement on how machine code should be implemented. It is possible that two different vendors' syntax would be different for an identical target and, certainly, differences in lower-level details such as register conventions would hinder portability.	If machine code inserts must be used to meet a project requirement, recognize the portability decreasing effects and isolate and highlight their use.
		Minimize interfaces with other languages.	The problems with employing *pragma INTERFACE* are complex. These problems include pragma syntax differences, conventions for linking/binding Ada to other languages, and mapping Ada variables to foreign language variables, among others.	It is often necessary to interact with other languages, if only an assembly language to reach certain hardware features. In these cases, clearly document the requirements and limitations of the interface and *pragma INTERFACE* usage.
		Isolate and clearly document machine language inserts.		
		Isolate all subprograms employing *pragma INTERFACE* to an implementation-dependent (interface) package.		
3.4.1.6 Obscure or Subtle Programming Constructs	High	No Ada specific guideline, see the generic guideline.	Not Applicable.	Not Applicable.
3.4.1.7 Dispersion of Related Elements	Medium	No Ada specific guideline, see the generic guideline.	Not Applicable.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
3.4.1.8 Use of Literals	Medium	Use symbolic constants instead of literals.	It is far easier to change one value set at the beginning of a source code file than it is to guarantee that all literals associated with such a parameter have been changed completely and correctly throughout all relevant source code files.	Not Applicable.
		Use Ada attributes instead of literals.	Improves portability. Removes need to change source code whenever a value obtainable by Ada attribute occurs.	Not Applicable.
3.4.1.9 Renaming	Medium	Avoid obscuration when renaming.	Do not use renaming if obscuration will occur.	Use only one level of renaming. Have project specific rules and conventions for renaming. Keep a registry of renamed identifiers for a project.
3.4.1.10 Bitmaps	Medium	Use representation clauses for bit mapping.	To facilitate reviews and reduce the possibility of coding errors.	Not Applicable.
3.4.2.1 Global Variables	Medium	Minimize the use of global variables.	Global variables obscure the passage of data between the inner and outer subprograms.	Clearly identify globals.
3.4.2.2 Complexity of Interfaces	Medium	No Ada language specific guidelines, see the generic guidelines.	Not Applicable	Closely inspect and clearly identify interfaces.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
3.4.2.3 Avoid coupling	Medium	Avoid declaring variables in library package specifications.	Low coupling should be a goal because 1) the fewer the number of connections between modules, the less chance of a failure in one module to propagate; 2) the fewer the number of connections between modules, the less chance a change in one module will cause problems in another and therefore increasing reusability; and 3) the fewer the number of connections between modules, the easier the learning curve is for the programmer to learn about the other modules.	Not Applicable.
3.4.3 Functional Cohesiveness	Medium	Every subprogram should have one clearly discernible purpose.	Every subprogram should have one clearly discernable purpose with input and output parameters related to that purpose.	Not Applicable.
3.4.4 Malleability	Medium	No Ada specific guideline, see the generic guideline.	Not Applicable.	Not Applicable.
3.4.5 Portability	Medium	Do not use busy loops to suspend execution.	The timing of a loop cannot be determined when the code is ported to a different compiler, different machine, or even different operating system.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Validate assumptions about the implementation of language feature when specific implementation is not guaranteed or specified. Do not assume a correlation between SYSTEM.TICK and package CALENDAR or type DURATION.	Although such a correlation may exist, it is not required to exist.	Not Applicable.
		Avoid the use of package SYSTEM constants except in attempting to generalize other machine dependent constructs.	Since the values in this package are implementation provided, unexpected effects can result from their use.	Do not use package SYSTEM constants to parametrize other implementation dependent features and access collection size.
		Use only pragmas and attributes defined by the Ada Standard.	The Ada LRM permits an implementor to add pragmas and attributes to exploit a particular hardware architecture or software environment. These are obviously even more implementation specific and therefore less portable than are an implementor's interpretation of the predefined pragmas and attributes.	Some implementation dependent features are gaining wide acceptance in the Ada community to help alleviate inherent inefficiencies in some Ada features.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid the direct invocation of or implementation dependence upon an underlying host operating system or Ada run-time support system.	Features of an implementation not specified in the Ada LRM will usually differ between implementations.	In real-time embedded systems, often it is not possible to avoid making calls to low-level support system facilities. Isolate the uses of these facilities.
		Minimize and isolate the use of the predefined package *LOW_LEVEL_IO*.	*LOW_LEVEL_IO* is intended to support direct interaction with physical devices that are usually unique to a given host or target environment. In addition, the data types provided to the procedures are implementation defined. This allows vendors to define different interfaces to an identical device.	Those portions of an application that must deal with this level of I/O, e.g., device drivers and real-time components dealing with discretes, are inherently non-portable.
		Restrict and isolate variables of the type **SYSTEM.ADDRESS**.	These variables are hardware-specific.	Not Applicable.

C and C++

C and C++

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.1.1.1 Dynamic Memory Allocation	High	Avoid dynamic memory allocation.	Dynamic memory allocation could cause unpredictable memory utilization and system failure.	Release allocated memory as soon as possible.
4.1.1.2 Memory Paging and Swapping	High	No C or C++ specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
4.1.1.3 Parameter Passing	Medium	Limit the number and size of parameters passed to routines.	Parameter passing to functions takes stack memory and can cause unpredictable stack memory utilization.	Pass large structures by pointer or reference. Use class definitions for related parameters.
4.1.1.4 Recursion	Medium	Minimize recursive function calls.	Recursive function calls can cause unpredictable stack memory utilization and stack overflow.	Ensure finite recursion.
4.1.1.5 Boundary Checking	Medium	Utilize functions with boundary checking.	Automatic boundary checking is not strong in C and C++.	Not Applicable.
	Medium	Do not use gets. Preferred to write user specified function.	Gets does not have adequate limit checks. Writing own routine allows better error handling.	Use fgets with caution.
4.1.1.6 Memory Block Move	Medium	Use memmove, not memcpy.	Avoid problems with memory overlap.	Not Applicable.
4.1.1.7 Memory at Power Up	High	Examine memory at power up.	Ensure correct functioning of memory.	Not Applicable.
4.1.1.8 Wrapping of built-in functions	Medium	Wrap built-in functions to include error checking.	Most built-in functions do not include safety features.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.1..1.9 Proper Array Indexing	Medium	Ensure that array indices are in the range 0 to *n-1*.	Index origins differ ammong languages.	Not Applicable.
4.1.2.1 Structure	Medium	*goto* shall be eliminated in safety systems.	The instruction *goto* is considered unstructured code.	Clearly document and justify.
		Functions with a nature similar to *goto*, such as setjmp and longjmp, shall be eliminated.	These two functions can jump from one subroutine location to another subroutine and make programs unstructured.	Clearly document and justify.
4.1.2.2 Control Flow Complexity	High	*switch .. case* shall be used to replace multiple *if . . else if . . . else if . . else* if possible.	Complicated control flow makes the program difficult to understand and maintain and is the source of unpredictable control.	Clearly document and justify.
		When utilizing *if . . . else*, the code block shall be bounded by brackets.	Brackets avoid mismatches between *if* and *else*.	Clearly document and justify.
		When utilizing *switch .. case*, *default* shall be explicitly defined.	*default* sometimes represents an error condition and shall be examined carefully.	Clearly document and justify.
		Check for dead code.	Unreachable code causes confusion.	Not Applicable.
4.1.2.3 Initialization of Variables and pointers	High	Reinitialize automatic variables.	Variables with automatic scope will contain "garbage" before explicit initialization and between function calls.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Initialize global variables in separate routines.	To ensure that variables are properly set at warm reboot.	Not Applicable.
		Initialize global variables only once.	Global variables may or may not be initialized by compiler.	Not Applicable.
		Do not use pointers to variables outside their scope.	Variables may contain garbage outside their scope.	Not Applicable.
		Initialize pointers.	Using an uninitialized pointer can overwrite the memory pointed by the pointer.	Not Applicable.
		Ensure presence of indirection operator for pointers.	Compiler may not catch type mismatches.	Not Applicable.
		Use the ~ operator to initialize to all 1's.	To be compatible with all word sizes.	Not Applicable.
4.1.2.4 Single Entry and Exit Points	Medium	Avoid multiple **returns**.	Multiple **returns** cause uncertainties similar to **gotos**.	Document and justify secondary exit points.
		Avoid **setjmp** and **longjmp**.	These commands jump outside function boundaries and deviate from the normal control flow.	Use only for exception handling; document and justify.
		Avoid pointers to functions.	Pointer to functions cannot be initialized and may point to non-executable code.	Document and justify.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Restrict the use of **throw** and **catch.**	Though preferable to **setjmp** and **longjmp,** these are relatively new features of C++ and may not be stable.	Validate compiler implementation.
4.1.2.5 Minimizing Interface Ambiguities	Medium	Prototype functions and procedures.	Avoids changing the order of arguments in C++.	Not Applicable.
		Avoid functions with an indefinite number of arguments.	These functions are difficult to verify.	Not Applicable.
		Alternate data types.	Avoids changing the order arguments.	Not Applicable.
		Avoid variable argument lists.	Using default arguments is preferable	Not Applicable.
		Ensure consistency of variable types with the function prototype.	To avoid unintended type conversions or casts.	Not Applicable.
		Test inputs and outputs.	Avoids changing the order of arguments.	Not Applicable
		Use byte alignment for small systems or if the CPU allows it.	Byte alignment saves resources and makes it easy to examine files.	Not applicable.
		Do not pass expressions as parameters to subroutines and macros.	Minimizes the complexity of the interface.	Not Applicable

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Eliminate increment ++ and decrement -- operators in parameter passing to subroutines or macros.	Increment ++ and decrement -- can create some undefined expressions when they are used as parameters, thus raising safety concerns.	Not Applicable
		Use bit masks instead of bit fields.	Bit fields are implementation dependent.	Not Applicable.
4.1.2.6 Data Typing	High	Limit the use of implementation dependent types.	To increase portability.	Not Applicable
		Minimize type conversions, and eliminate implicit or automated type conversions. A pointer shall not be casted to a different type of pointer.	These practices reduce strong typing and can cause safety problems.	Use explicit casting.
		Avoid use of mixed-mode operations.	Mixed-mode operations reduce strong typing and can cause safety problems.	If necessary, they shall be clearly identified and described using prominent comments in the source code.
		Use a single data type in evaluations and relational expressions.	Enhances strong typing and can avoid safety problems.	Not Applicable.
		Avoid the use of **typedef**'s for unsized arrays.	This feature is badly supported and error-prone.	
		Avoid multiple declarations of the same identifier with several types.	This may be a source of confusion.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid mixing signed and unsigned variables.	This raises safety concerns.	Use explicit casts.
		Limit the use of indirect addressing.	Strongly typed array indices and pointers reduce the possibility of referencing invalid locations.	Not Applicable.
		Avoid using the same identifier for different types.	May result in undefined behavior.	Not Applicable.
4.1.2.7 Precision and Accuracy	High	Provide adequate precision and accuracy for the intended safety application.	This practice preserves the integrity of the algorithms.	Not Applicable
		Use double precision.	To ensure adequate precision.	Not Applicable.
		Account for floating point properties in relational operations.	The equality comparison is unreliable for floating point.	Not Applicable.
		Account for truncation integer operations.	Truncation in division of negative numbers is implementation dependent.	Not Applicable.
		Account for optimization.	Subexpressions may be moved or eliminated by optimizing compilers.	Not Applicable.
		Ensure that arithmetic results are representable by the destination type.	Conversion to shorter types may have unpredictable results.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.1.2.8 Order of Precedence	Medium	Use parentheses rather than default order of precedence in macros and bitwise and relational operations.	Avoid hard to find computational errors.	Use other forms to enhance readability if parentheses are excessive.
		Ensure that values in an expression do not depend on the order of evaluation.	The order of evaluation is implementation dependent.	Not Applicable.
4.1.2.9 Side Effects	High	Generic guidelines apply.	Not Applicable.	Not Applicable.
4.1.2.10 Separating Assignment from Evaluation	Medium	Separate assignments from evaluation statements.	Mixing assignments with evaluation statements causes side effects.	Not Applicable.
4.1.2.11 Program Instrumentation	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
4.1.2.12 Library Size	Medium	Control class library size.	A system becomes unmanageable or has large performance penalties if it has too many classes and objects.	Not Applicable.
		Avoid multiple inheritance.	Multiple inheritance may cause ambiguities and maintenance problems.	Not Applicable.
4.1.2.13 Dynamic Binding	High	Minimize dynamic binding.	The unpredictability of the name/class association reduces the predictability of the run-time behavior and it hampers debugging, understanding, and tracing.	All cases where dynamic binding is required should be justified.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.1.2.14 Operator Overloading	Medium	The meaning of an overloaded operator should be natural, not clever.	Operator overloading can obscure predictability .	Sometimes acceptable for achieving uniformity across different data types.
		Keep operator precedence by parentheses, not by default order.	Operator overloading can obscure predictability .	Sometimes acceptable for achieving uniformity across different data types.
		Explicitly define class operators and declare them private.	Built-in definitions may not remain consistent between implementations.	Not Applicable.
		Ensure consistency of class operators.	Built-in definitions may not remain consistent between implementations.	Not Applicable.
4.1.2.15 Compiler Warnings	Medium	Enable and heed compiler warnings.	Any warning may be a potential safety concern.	Not Applicable.
4.1.3.1 Tasking	High	Minimize tasking	C and C++ do not support multi-tasking. Their standard library functions may not be re-entrant. Using those functions in tasking environments can generate unspecified results.	Tasking requires compelling justification.
4.1.3.2 Interrupt Driven Processing	High	Minimize interrupt driven processing.	Interrupt driven processing leads to non-deterministic maximum response times.	If used, the code and processing time within the interrupt shall be minimized.
		Limit function calls within interrupt service routines.	To reduce control flow complexity.	Only re-entrant functions should be called by interrupt service routines.

C and C++

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.2.1 Software Diversity	Medium	No C or C++ specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
4.2.2.1 Local Handling of Exceptions	High	Handle exceptions locally.	System failures can be avoided if exceptions are handled locally.	If not possible, use thorough testing and analysis to verify behavior during exception handling.
4.2.2.2 External Flow Control	High	Preserve control flow external to the module responsible for the exception.	Safety is enhanced by preservation of control flow external to the module responsible for the exception.	If not possible, use thorough testing and analysis to verify behavior during exception handling.
4.2.2.3 Uniformity of Exception Handling	High	Rely on signals and traps instead of operating system features.	To avoid non-portable vendor-specific features.	Not Applicable.
		Use **throw** and **catch** in C++ instead of **setjmp** and **longjmp**.	**setjmp** and **longjmp** are difficult to recover from.	Not Applicable.
4.2.3 Input and Output Data Checking	High	Perform run-time checks on input data. Check pointers before use.	Accidental data corruption in one module can have serious consequences on subsequent processing if allowed to propagate to other modules.	May not be applicable if input can be "trusted". May not be necessary if downstream input checking is performed.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.3.1 Built-In Functions	Low	Minimize the use of built-in functions.	Requirements for developing those built-in functions, exception handling, and the characteristic of those functions may not be the same as the ones in the safety systems. The number of built-in functions may vary from one compiler to another.	Thorough testing, and error tracking.
4.3.2 Compiled Libraries	Low	Minimize the use of compiled libraries.	Concerns in 4.3.1 Built-In Functions apply. Functions with same names but different characteristics among vendors raises portability concerns.	- Ensure that names in externally developed libraries are distinct. - Document all cases of dynamic binding. - Ensure that development and runtime shared libraries are identical.
4.3.3 Utilizing Control Tools	Medium	Use version control tools.	Avoids errors due to interfacing.	Not Applicable
4.4.1.1 Indentation Guidelines	Medium	Conform to indentation guidelines.	Code for safety systems shall be reviewed by peers or supervisors. The readability is essential for such reviews.	Not Applicable
4.4.1.2 Descriptive Identifier Names	Medium	No C or C++ specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
4.4.1.3 Comments and Internal Documentation	Medium	Conform to comment and documentation guidelines.	Incomplete, outdated, and inconsistent comments impede review and maintenance.	Not Applicable

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.4.1.4 Limiting Subprogram Size	Medium	Project guidelines are required on subprogram size.	Large subprogram units are hard to read and maintain.	Justify larger size and provide additional documentation and comments.
4.4.1.5 Minimizing Mixed Language Programming	Medium	Minimize mixed language programming.	Mixed language programming presents difficulties for reviewers and maintainers and is therefore a safety concern.	When this practice cannot be avoided, minimize difficulties by placing the "foreign" language code adjacent to the dominant language routine with which it interfaces.
4.4.1.6 Minimizing Obscure or Subtle Programming Constructs	High	Minimize obscure or subtle programming constructs.	Such coding practices present problems in review, and maintenance and hence, are safety concerns.	When it cannot be avoided, use comments to minimize the impact of obscure or subtle code.
		Avoid the use of the ? : operator.	This operator makes the code more difficult to read.	Not Applicable.
		Use table-driven alternatives when appropriate.	To create code which is easier to review and maintain.	Not Applicable.
		Avoid using default parameters to combine functions.	This will make code difficult to maintain.	Not Applicable.
		Avoid complex expressions inside a condition.	This will make the code more error-prone.	Not Applicable.
		Maximize the use the scope resolution operator.	To avoid ambiguities.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid pointers to members.	They unnecessarily complicate the code.	Use virtual functions.
		Use the **virtual** keyword wherever necessary.	To avoid unintended calls to member functions.	Not Applicable.
4.4.1.7 Minimizing the Dispersion of Related Elements	Medium	Place **#include** directive at the beginning of programs.	To make it easier to trace dependencies.	Clearly document and justify.
		Place all external function prototypes in close proximity.	To make it easier to update the code.	Clearly document and justify.
		Segregate base from derived classes.	To avoid unintended changes to the class hierarchy.	Not Applicable.
4.4.1.8 Minimizing the Use of Literals	Medium	Safety systems shall utilize **const** variables or **#define** instead of literals.	Literals are much more difficult to find during modification and maintenance and can cause safety problems.	Associate comment with each literal to facilitate search/replace.
		Use parentheses to avoid expansion problems on **#defines**.	Corrects improper expansion of **#defines**.	Not Applicable.
		Enumeration constants are preferred to **#defines** in sequences of several integer numbers.	It is easier to modify code when a new number needs to be inserted in a sequence.	Not Applicable

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
4.4.2.1 Minimize the use of Global Variables	Medium	Minimize the use of global variables.	This avoids side effects.	- Keep global variables and associated functions in the same file. - Declare global variables in one header file. - Initialize global variables in one place.
4.4.2.2 Minimize the Complexity of Interfaces	Medium	Limit the number of parameters Use structures or classes Avoid expressions in parameter lists	Complex interfaces are difficult to review and maintain and can cause safety problems.	Closely inspect and clearly identify interfaces.
4.4.3 Functional Cohesiveness	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
4.4.4 Malleability	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
4.4.5.1 Avoid Implementation dependent types	Medium	Avoid the use of implementation dependent types such as **int**.	To ensure portability among platforms.	Not Applicable.
4.4.5.2 Avoid Reserved Words	Medium	Avoid using reserved words, including standard library function names and names starting with underscores.	The misuse of reserved words can lead to serious problems.	Not Applicable
4.4.5.3 Minimize Hardware Dependencies	Medium	Define hardware-dependent addresses symbolically.	To ensure portability among different platforms.	Not Applicable

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Use the **volatile** attribute for data items that are mapped to hardware.		
		Avoid use of bit fields.		
		Do not measure time by counting clock cycles.		

PLC Ladder Logic

PLC Ladder Logic

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
5.1.1 Dynamic Memory Allocation	High	No PLC specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
5.1.2.1 Structure	Medium	Use engineering judgement with *gotos*.	Use *gotos* only if the structure is clear although structured programming is preferred.	Clearly document, justify, test.
		Use watchdog timers or scan counters with backward jumps.	PLCs do not limit directions, leading to timer faults.	
		Ensure that initialization has occurred before the jumps.	Otherwise data words can be left uninitialized.	
5.1.2.2 Control Flow Complexity	High	Reduce complex logic by breaking into cohesive subunits and limiting nesting levels.	Very simple structure is easy to understand and predict real-time behavior.	Not Applicable
5.1.2.3 Initialization of Variables	High	Audit all relevant variables that are initialized.	There exists no explicit assignment in Ladder Logic.	Not Applicable
5.1.2.4 Single Entry and Exit Points	Medium	Single returns only. Project guidelines strictly limit multiple returns.	Ladder Logic only supports single entry points, but allows multiple returns.	Document all multiple returns.
5.1.2.5 Interface Ambiguities	Medium	Verify that interfaces are well defined and documented.	Ladder Logic does not support interface checking, only type checking.	Not Applicable
5.1.2.6 Data Typing	High	Ensure that data table properly accounts for variable types.	PLCs do not support range checking or strong data typing.	Not Applicable

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Ensure that type conversion will not result in an error. Develop project-specific guidelines.		
5.1.2.7 Precision and Accuracy	High	Verify that the processor and language support the floating point accuracy needed.	Not Applicable.	Not Applicable.
5.1.2.8 Order of Precedence	Not Applicable	Order of precedence does not exist.	Not Applicable.	Not Applicable.
5.1.2.9 Side Effects	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
5.1.2.10 Separating Assignment from Evaluation	Not Applicable	Assignment is usually separate from evaluation in PLCs.	Not Applicable.	If it is not possible to separate the two: - Use buffer variables or output coils - Develop project-speciific guidelines.
5.1.2.11 Program Instrumentation	Not Applicable	No application level support is needed.	This is provided by the PLC environment.	Not Applicable.
5.1.2.12 Library Size	Not Applicable	Classes and objects are not supported.	Not Applicable.	Not Applicable.
5.1.2.13 Dynamic Binding	Not Applicable	No run-time binding permitted.	Not Applicable.	Not Applicable.

PLC Ladder Logic

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
5.1.2.14 Operator Overloading	Not Applicable	Overloading and polymorphism are not supported.	Not Applicable.	Not Applicable.
5.1.3.1 Tasking	Not Applicable	Tasking is usually not supported on PLCs	Not Applicable.	If tasking is supported: - Account for processing capacity - Account for concurrrent access to global variables.
5.1.3.2 Interrupt Driven Processing	High	If interrupts are used, show that all timing and safety function requirements are met. Account for interrupts in critical response times.	Interrupts are not widely supported. PLCs and Ladder Logic use deterministic polling.	Not Applicable.
5.1.3.3 Synchronization	High	Avoid synchronization.	Race conditions and deadlocks are very hard to predict.	When synchronization is required, select the best platform that supports it.
5.1.3.4 Self Modifying Code	High	Avoid self changing code.	On-line program changes are not permanent.	Not Applicable.
5.2.1 Functional Diversity	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
5.2.2.1 System Health Monitoring	Medium	Ensure completeness, correctness and observability of parameters.	Need to ensure adequacy of monitoring.	Not Applicable.

PLC Ladder Logic

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
5.2.2.2 Fault Routines and Shutdown Behavior	Medium	Ensure completeness, correctness and observability of parameters.	Need to ensure adequacy of failure handling.	Not Applicable.
5.2.2.3 Watch-Dog Timer	High	Initialize when needed. Ensure adequate fault routine. Use external timer when needed.	Need to ensure adequacy of failure handling.	Not Applicable.
5.2.3 Error Containment	High	See guidelines on data types and parity bits.	No explicit Ladder Logic capabilities.	Not Applicable.
5.3.1 Built-In Functions	Low	No PLC specific guidelines, see generic guidelines.	Not Applicable	Not Applicable.
5.3.2 Compiled Libraries	Not Applicable	Compiled libraries are not supported.	Not Applicable	Not Applicable.
5.4.1.1 Notation	Medium	Use standard notation.	Required by PLC.	Not Applicable.
5.4.1.2 Conformance to Indentation Guidelines	Medium	No PLC specific guidelines, see generic guidelines.	Not Applicable.	Not Applicable.
5.4.1.3 Descriptive Identifier Names	Medium	No PLC specific guidelines, see generic guidelines.	Not Applicable	Not Applicable.
5.4.1.4 Comments and Internal Documentation	Medium	Document the hierarchy of subroutines and the flow of data and information among subroutines.	These two items are important to understand the system and enable independent review.	Not Applicable.

PLC Ladder Logic

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
5.4.1.5 Subprogram Size	Medium	Limit subroutines to 10 to 50 rungs.	Small programs represent a large screen area, which makes debugging and review cumbersome.	Not Applicable.
5.4.1.6 Mixed Language Programming	Not Applicable	Ladder Logic does not support "foreign" languages.	Not Applicable.	Not Applicable.
5.4.1.7 Obscure or Subtle Programming Constructs	High	No PLC specific guidelines, see generic guidelines.	Not Applicable.	Not Applicable.
5.4.1.8 Dispersion of Related Elements	Not Applicable	Dispersion is not possible in Ladder Logic.	Not Applicable.	Not Applicable.
5.4.1.9 Use of Literals	Medium	Use symbolic constants instead of literals.	To protect literals, and control the uniformity of the value.	Not Applicable.
5.4.2.1 Modularity	Medium	Use subroutines if available.	Changes are easier.	Not applicable.
5.4.2.2 Information Hiding	Medium	See Appendix A.	See Appendix A.	Not Applicable.
5.4.2.3 Global Variables	Medium	No PLC specific guidelines, see generic guidelines.	Not Applicable.	Not Applicable.
5.4.2.4 Complexity of Interfaces	Medium	No PLC specific guidelines, see generic guidelines.	Not Applicable.	Not Applicable.
5.4.3 Functional Cohesiveness	Medium	No PLC specific guidelines, see generic guidelines.	Not Applicable.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
5.4.4 Malleability	Medium	No PLC specific guidelines, see generic guidelines.	Not Applicable.	Not Applicable.
5.4.5 Portability	Medium	No PLC specific guidelines, see generic guidelines.	Not Applicable.	Not Applicable.
5.5 Security	High	Use locks and passwords.	To prevent unauthorized access.	Not Applicable.

IEC 1131 Sequential Function Charts

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
6.1.1 Memory Utilization	N/A	SFCs do not allocate memory.	Not applicable.	Not applicable.
6.1.2.1 Structure	Medium	Avoid the use of *goto*'s	Use of *goto* statements that result in an unstructured shift of execution are difficult to trace and understand.	Clearly document, justify, and test.
6.1.2.2 Control Flow Complexity	High	Minimize control flow complexity	Excess complexity makes it difficult to predict the program flow and impedes review and maintenance.	Not applicable.
6.1.2.3 Initialization of Variables	High	Account for initialization as part of the program design. Account for initialization of process steps and transitions.	To ensure that variables are properly initialized by both the SFC and the underlying language.	Not Applicable.
6.1.2.4 Single Entry and Exit Points	Medium	Single entry and exit points are enforced by the SFC grammar.	Multiple entries and exits introduce control flow uncertainties	Not Applicable.
6.1.2.5 Interface Ambiguities	Medium	Latch all bits which need to stay on between steps.	Non-retentive bits are reset during the post-scan.	Not Applicable.
6.1.2.6 Use of Data Typing	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.1.2.7 Precision and Accuracy	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
6.1.2.8 Order of Precedence	Medium	All transitions in a divergence of selection sequence should be mutually exclusive.	To avoid ambiguities when multiple transitions are evaluated as true simultaneously.	Not Applicable.
6.1.2.9 Avoiding Side Effects	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
6.1.2.10 Separating Assignment from Evaluation	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.1.2.11 Program Instrumentation	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.1.2.12 Library Size	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.1.2.13 Dynamic Binding	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.1.2.14 Operator Overloading	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.1.3.1 Use of Tasking	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.1.3.2 Interrupt Driven Processing	High	Demonstrate that system/software can meet all requirements under most demanding conditions of interrupt occurrence.	To satisfy safety requirements.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
6.1.3.3 Divergence of Sequence	High	Define Mutually exclusive transition conditions.	To explicitly exclude the possibility of multiple transitions in such a stucture being evaluated as true simultaneously.	Not Applicable.
		Ensure convergence of sequence following divergence of sequence.	Predictability of control flow.	Not Applicable.
		Account for limits on the number of transitions.	Portability and predictability of control flow.	Not Applicable.
6.1.3.4 Simultaneous Sequences	High	Avoid dependence on execution order.	Portability and predictability of control flow.	Not Applicable.
		Use simultaneous sequences only where synchronization is required.	Predictability of control flow.	Not Applicable.
6.1.3.5 Post Scan Timing	High	Do not set timers in a transition.	The processor does not postscan transition files.	Not Applicable.
6.2.1 Transparency of Diversity	Medium	Account for the safety impact of the order of execution of diverse steps.	To avoid unintended outcomes.	Not Applicable.
		Account for all local and global variables necessary to support replicated processing in transition files.	To ensure that no variable in transition files are uninitialized or overwritten.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
6.2.2 Exception Handling	High	Use *GOTO* or *JMP* to handle interruption of flow control with care.	These commands are not intended for interrupt processing.	Not Applicable.
		Ensure that two events, transitions, and exception handling do not conflict with each other.	To satisfy safety requirements.	Not Applicable.
		Ensure the safety of exception handling during a process step.	To satisfy safety requirements.	Not Applicable.
		Ensure the safety of exception handling during a transition.	To satisfy safety requirements.	Not Applicable.
		Ensure the safety of restart after an exception.	To satisfy safety requirements.	Not Applicable.
6.2.3 Input and Output Checking	N/A	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.3.1 Use of Built-In Functions	High	Minimize the use of built-in functions.	The requirements and definitions may not be the same for different platforms.	Verify the exact performance of steps and transitions under normal and abnormal conditions
6.3.2 Use of Compiled Libraries	Low	Generic guidelines apply.	Not Applicable.	Not Applicable
6.4.1.1 Indentation Guidelines	Medium	Not applicable to SFCs.	Not Applicable.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
6.4.1.2 Descriptive Identifier Names	High	Generic guidelines apply.	Not Applicable.	Not Applicable
6.4.1.3 Comments and Internal Documentation	High	Provide clear and unambiguous descriptions of steps.	Incomplete, outdated, and inconsistent comments impede review and maintenance.	Not Applicable.
		Provide clear and unambiguous descriptions of interfaces.		
		Provide clear and unambiguous descriptions of transitions.		
6.4.1.4 Limitations on Subprogram Size	High	Enforce through external administrative procedures.	Large subprograms are hard to review and maintain.	Justify larger size and provide additional documentation and comments.
6.4.1.5 Mixed Language Programming	Medium	Use SFC for sequencing.	SFC notation is clearer than Ladder Logic.	Not Applicable.
		Do not use SFC for interlocking or evaluation of logical relationships.	SFC is not suited for this purpose.	Not Applicable.
		Do not use SFC for mathematical operations or evaluation of mathematical relationships.	Structures Text is more suitable for this purpose.	Not Applicable.

IEC 1131 Sequential Function Charts

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
6.4.1.6 Obscure or Subtle Language Constructs	Medium	Avoid nesting of subroutines within an SFC step.	The assumption is that an SFC step is one subroutine.	Not Applicable.
		Do not use SFC constructs which are not related to sequencing.	Sequencing is the main purpose of SFC.	Not Applicable.
		Avoid backward directed links in parallel paths.	This makes SFC programs difficult to maintain.	Not Applicable.
6.4.1.7 Dispersion of Related Elements	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
6.4.1.8 Use of Literals	Medium	Not applicable to SFCs.	Not Applicable.	Not Applicable.
6.4.1.9 Use of Macro-Steps	Low	Follow project guidelines in the use of macro-steps.	There is a potential for misuse of macro-steps.	Not Applicable.
6.4.2.1 Use of Global Variables	Medium	Use local variables for internal operations if supported by the language.	Use of global variables may have unanticipated side effects.	Justify and clearly identify global variables.
6.4.2.2 Complexity of Interfaces	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.
6.4.3 Functional Cohesiveness	Low	Each step should have one clearly discernible purpose related to the time in which it is executed.	To enhance reviewability and maintainability.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
6.4.4 Malleability	Low	Segregate constants from what is expected to be changed.	To improve and clarify interfaces.	Macro-steps must be used with care (see 6.4.1.9).
6.4.5 Portability	Medium	Only IEC 1131 compliant SFCs should be used.	The SFC will not be portable otherwise.	Not Applicable.

Pascal

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
7.1.1.1 Dynamic Memory Allocation	High	Dynamic use of memory should be strongly discouraged.	If the program heap grows too large while it is running, then the computer will crash.	Release allocated memory as soon as possible.
7.1.1.2 Memory Paging and Swapping	High	No Pascal specific guideline, see the generic guideline.	Not Applicable	Not Applicable
7.1.1.3 Avoiding Recursion	High	Do not use recursion.	Recursion uses stacks and can use up available memory in the heap.	Release allocated memory as soon as possible.
7.1.1.4 Use of Handles with Pointers	High	If pointers must be used, use handles whenever possible.	Handles allow memory management to recapture and compact free memory.	Release allocated memory as soon as possible.
7.1.1.5 Use of Direct Memory Access	Medium	Do not use direct memory access under Windows in Turbo Pascal.	Although Turbo Pascal permits access to memory directly, this is not a safe practice under Windows.	Release allocated memory as soon as possible.
7.1.2.1 Maximizing Structure	Medium	Minimize *gotos*.	The use of *goto* clouds the structure of the code and therefore should be avoided.	Clearly document and justify.
		Use *else if* whenever possible.	The use of *else if* where appropriate helps to avoid program structure and logical errors.	Clearly document and justify.
7.1.2.2 Control Flow Complexity	High	If exits from within loops can not be avoided, use *gotos*.	In Pascal the loops can be labeled in order to clarify the meaning of multiple loops and the code structure.	Project guidelines should set specific limits on nesting levels.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
7.1.2.3 Initialization of Variables	High	Initialize all variables.	Variables should be initialized to some known value before using them.	Not Applicable
7.1.2.4 Single Entry and Exit Points	Medium	One *return* per subprogram.	Single exit points from procedures and functions are easier to understand, test, and less expensive to design, build and maintain than multiple entries and exits.	Document secondary exit pointer if used.
7.1.2.5 Interface Ambiguities	Medium	Avoid use of function or procedure parameters which depend on the order of evaluation.	Do not expect the evaluation of function or procedure parameters to occur in any particular order, since this is compiler implementation dependent.	Not Applicable
7.1.2.6 Data Typing	High	The limits on data types should not be excessive.	It forces unnecessary errors to be generated by unanticipated but not unsafe computational inaccuracies.	Not Applicable
		Minimize the use of implicit type conversions.	Use of type conversions is strongly discouraged by most authors.	Not Applicable
		Limit the use of indirection (pointers).	Pointers are a form of dynamic memory and should be avoided.	Not Applicable

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
7.1.2.7 Precision and Accuracy	High	Precision and accuracy issues include the meaning and use of fixed point and floating point numbers, round off errors, type declarations and digital accuracy, and portability.	Precision and accuracy must be sufficient to assure proper functioning of algorithms.	Not Applicable
7.1.2.8 Order of Precedence	Medium	Use parentheses for ensuring that the order of evaluation of operations is explicitly stated.	The default order of precedence such as left to right with multiplication and addition should not be depended on.	Use other forms to enhance readability if parentheses are excessive.
		Do not depend on the order of evaluation.	As permitted by the Pascal standards, operands of an expression are frequently evaluated differently from the left to right order in which they are written.	Not Applicable.
		Use care in multiple condition flow statements.	The order of evaluation cannot be guaranteed.	Not Applicable.
7.1.2.9 Functions or Procedures with Side Effects	Medium	Verify that functions do not have side effects.	Side effects can lead to problems with unplanned dependencies and can cause bugs that are very hard to find.	Use other forms to enhance readability if parentheses are excessive.
7.1.2.10 Separating Assignment from Evaluation	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.1.2.11 Program Instrumentation	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
7.1.2.12 Library Size	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.1.2.13 Dynamic Binding	High	Dynamic binding should be avoided if possible.	Dynamic binding use the memory heap and is therefore are susceptible to problems.	All cases where dynamic binding is required should be justified.
7.1.2.14 Operator Overloading	Not Applicable	Pascal does not support operator overloading.	Not Applicable	Not Applicable
7.1.3.1 Tasking	Not Applicable	Pascal does not support tasking.	Not Applicable	Not Applicable
7.1.3.2 Interrupt Driven Processing	High	Isolate interrupt receiving tasks into implementation dependent packages. Pass the interrupt to the main tasks via a normal entry.	Interrupt entries are implementation dependent features that may not be supported.	Interrupt isolated entries can increase the interrupt latency time. Where this is unacceptable, the interrupt entries must be proliferated.
7.2.1 Transparency of Functional Diversity	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.2.2 Exception Handling	Not Applicable	Standard Pascal does not support exception handling.	Not Applicable	Not Applicable
7.2.3 Input and Output Data Checking	High	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.3.1 Built-In Functions	Low	The project should control which functions are available for project work.	Pascal functions are portable to other compilers; the Turbo Pascal functions are not portable to other compilers.	Through testing and error checking.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
7.3.2 Compiled Libraries	Low	Avoid the use of compiled libraries.	Libraries prevent the programmer from knowing the accuracies, limitations, robustness, and error handling of the built-in functions.	Thorough testing and error checking.
7.4.1.1 Indentation Guidelines	Medium	Conform to indentation guidelines	Indentation improves readability and allows the reader to see the structure of the program.	Not Applicable
7.4.1.2 Descriptive Identifier Names	Medium	Choose names that are self-documenting as possible. Separate words in compound names with underscores.	These improve readability.	Not Applicable
7.4.1.3 Comments and Internal Documentation	Medium	Source code shall be supplemented with Pascal comments that explain the code.	This improves readability.	Not Applicable
7.4.1.4 Subprogram Size	Medium	No Pascal specific guidelines, see generic guidelines	Not Applicable	Not Applicable
7.4.1.5 Mixed Language Programming	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.4.1.6 Obscure or Subtle Programming Constructs	High	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.4.1.7 Dispersion of Related Elements	Medium	Minimize dispersion of related elements. Use compilation units to group related elements.	When elements are disperse throughout the code, it is very hard to check, validate, and maintain the code.	Provide clear reference, rationale, overall source code organization.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
7.4.1.8 Use of Literals	Medium	Use symbolic constants instead of literals.	Hard coded numeric constants decrease readability and complicates maintainability	Associate comment with each literal to facilitate search/replace.
7.4.2.1 Global Variables	Medium	Minimize the use of global variables.	Global variables obscure the passage of data between the inner and outer subprograms. Variables should be kept local to the routines which set and use them.	If coupling is required between modules, make those dependencies visible and document to avoid problems.
7.4.2.2 Complexity of Interfaces	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.4.3 Malleability	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.4.4 Functional Cohesiveness	Medium	No Pascal specific guidelines, see generic guidelines.	Not Applicable	Not Applicable
7.4.5 Portability	Medium	Avoid to use of the *mod* operator.	Not all compilers follow the Standard in this respect. Therefore use caution when porting *mod*.	Not Applicable

PL/M

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
8.1.1.1 Dynamic Memory Allocation	High	Minimize dynamic memory allocation	Use of dynamic memory can cause crashes	Release allocated memory as soon as possible.
8.1.1.2 Memory Paging and Swapping	High	Minimize memory paging and swapping	Memory paging and swapping can cause significant delays in response time.	
8.1.1.3 Memory Bank Switching and Shadow Memory	Medium	Avoid hardware bank-switching.	Bank switching is a source of unreliability.	Clearly document, justify, and test.
8.1.2.1 Maximizing Structure	Medium	Eliminate *goto*'s.	The instruction *goto* is considered unstructured code.	Clearly document and justify.
8.1.2.2 Minimizing Control Flow Complexity	High	Generic guidelines apply.	Not Applicable.	Not Applicable.
8.1.2.3 Initializing Variables Before Use	Medium	Initialize all variables.	Uninitialized variables can be a source of latent software bugs.	Not Applicable.
8.1.2.4 Single Entry and Exit Points in Subprograms	Medium	Use single entry and exit points.	Multiply entry and exit points introduce uncertainties in control flow.	Document and justify secondary entry and exit points.
8.1.2.5 Minimizing Interface Ambiguities	Medium	Use procedure CALL templates and Cut and Paste to avoid argument list errors.	Interface errors account for many coding errors.	Not Applicable.
8.1.2.6 Data Typing	High	Use data typing	Data typing prevents misuse of data; contains errors	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
8.1.2.7 Precision and Accuracy	High	Account for different hardware.	Correct results needed in safety critical calculations	Not Applicable.
		Account for optimization in floating point computations.	Compilers may rearrange or delete subexpressions.	Not Applicable.
		Verify numeric precision in ported code.	Different platforms may have different precision limitations.	Not Applicable.
		Express precision in terms of numeric ranges.	Terms such as **word** are platform dependent.	Not Applicable.
8.1.2.8 Order of Precedence	Medium	Use parentheses rather than default order of precedence	Incorrect precedence assumptions cause errors; source code open to misinterpretation	Use other forms to enhance readability if parentheses are excessive.
8.1.2.9 Side effects	Medium	Generic Guidelines apply.	Not Applicable.	Not Applicable.
8.1.2.10 Separating Assignment from Evaluation	Medium	Separate assignments from evaluation statements	Incorporation of assignments into evaluation statements can cause unanticipated side effects	Not Applicable.
8.1.2.11 Program Instrumentation	Medium	Minimize run-time perturbations Maintain visibility of instrumentation in run-time source code Conform to software instrumentation guidelines	These practices improve checkout and verification of code.	Intrusive instrumentation is sometimes necessary for problem resolution. Remove instrumentation and perform regression testing.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
8.1.2.12 Class Library Size	N/A	Not applicable to PL/M.	Not Applicable.	Not Applicable.
8.1.2.13 Dynamic Binding	High	Eliminate overlay or shadow ROM code.	Difficult to test and debug.	Not Applicable.
8.1.2.14 Operator Overloading	N/A	Not applicable to PL/M.	Not Applicable.	Not Applicable.
8.1.2.15 Compiler Optimization and Hardware Flags	Medium	Account for compiler optimizations in sequence of operations and hardware flags.	Compilers can rearrange or eliminate subexpressions.	Use assembly language for functions that use hardware flags.
8.1.3.1 Use of Tasking	N/A	PL/M does not support concurrent processing.	Not Applicable.	Not Applicable.
8.1.3.2 Interrupt Driven Processing	High	Minimize the use of interrupt driven processing	Interrupts lead to non-deterministic response times.	Minimize processing for handling interrupts. Return to primary program control as soon as possible.
		Interrupt handlers should be as short and simple as possible.	To reduce control flow complexity.	Not Applicable.
		Avoid nested interrupts.	To reduce control flow complexity.	Not Applicable.
		Interrupt handlers should not alter shared data.	To reduce control flow complexity.	Not Applicable.
8.2.1 Software diversity	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
8.2.2 Handling of Exceptions	High	Handle exceptions locally.	Local exception handling helps isolate problems more easily and more accurately.	If not possible, thorough testing and analysis to verify behavior during exception handling is required.
8.2.3 Input and Output Checking	High	Check input and output data.	Accidental data corruption in one module can have serious consequences on subsequent processing if allowed to propagate to other modules.	May not be applicable if input can be "trusted". May not be necessary if downstream input checking is performed.
8.3.1 Built-In Functions	Low	Control the use of built-in functions through project specific guidelines	Built-in functions have unknown internal structure, limitations, precision, exception handling,	Conduct thorough testing and error tracking.
8.3.2 Compiled Libraries	Low	Control the use of compiled libraries	Compiled libraries have unknown internal structure, limitations, precision, exception handling,	Conduct thorough testing and error tracking.
8.4.1.1 Indentation Guidelines	Medium	Conform to indentation guidelines	Indentation guidelines improve readability and maintainability.	Not Applicable
8.4.1.2 Descriptive Identifier Names	Medium	Use descriptive identifier names	Descriptive identifier names improve readability and maintainability.	Not Applicable
8.4.1.3 Comments and Internal Documentation	Medium	Conform to comment guidelines	Necessary to verify conformance to requirements, code inspections, maintenance	Not Applicable
8.4.1.4 Subprogram Size	Medium	Generic guidelines apply.	Not Applicable.	Not Applicable.

PL/M

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
8.4.1.5 Mixed Language Programming	Medium	Minimize mixed language programming.	Mixed language programming is hard to read and hard to maintain.	Isolate second language functions and couple as loosely as possible
8.4.1.6 Obscure or Subtle Programming Constructs	High	Minimize obscure and subtle programming constructs	Obscure coding presents problems in review and maintenance and raises safety concerns.	When it cannot be avoided, use comments to minimize the impact of obscure or subtle code.
8.4.1.7 Dispersion of Related Elements	Medium	Minimize the dispersion of related elements	Dispersed elements necessitate multiple accesses to review or maintain code, and therefore are susceptible to errors.	Provide clear reference, rationale, overall source code organization.
8.4.1.8 Use of Literals	Medium	Minimize the use of Literals.	The use of constants enhances code reliability and consistency.	Associate comment with each literal to facilitate search/replace.
8.4.2.1 Global Variables	Medium	All global variables should be initialized in exactly one place.	To avoid multiple definitions.	Not Applicable.
		All exports from a module should be explicitly global all other explicitly declared static. All importing modules should use the header file only.	To avoid multiple definitions.	Not Applicable.
		Use macros for local variables in emulators, simulators, and debuggers.	To avoid complicating the use of local variables.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
8.4.2.2 Complexity of Interfaces	Medium	Limit the number of arguments used in the calling program.	Large number of arguments can cause confusion and errors in a safety-related program.	Use smaller functions.
		Do not use ambiguous or terse expressions.	Use of meaningless expressions for modes or options can cause confusion to the programmer.	Not Applicable.
		Explicitly state restrictions and limitations.	Lack of clear restrictions and limitations can complicate the interface.	Not Applicable.
8.4.2.3 Use of modules	Medium	Use modules to facilitate data abstraction.	To enhance maintainability by limiting data visibility.	Not Applicable.
8.4.3 Functional Cohesiveness	Medium	Function of a program and structure of its components should have clear correspondence.	To facilitate review and maintenance of the program.	Not Applicable.
8.4.4.1 Isolation of Alterable Functions	Medium	Place functions in DO;-END modules within source code file to which they belong.	Placing alterable function in one file may result in collection of unrelated procedures.	Clearly comment alterable sections.
8.4.4.2 Isolation of Hardware Specific Functions	Medium	Write code for peripheral devices in the form of device drivers.	Calling code will not be impacted by a change in the device driver code.	Not Applicable.
8.4.5 Portability	Medium	PL/M is obsolescent and of limited portability.	Not Applicable.	Plan for migration to another language.

Ada95

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
9.1.1.1 Dynamic Memory Allocation	High	Dynamic use of memory should be strongly discouraged.	Dynamic memory can cause the heap to grow too large and crash the system.	Release allocated memory as soon as possible. Utilize the length clause feature to pre-allocate dynamic memory pools. Put *STORAGE_ERROR* exception handlers in program units which allocate dynamic memory.
		Avoid dynamically created tasks.	Dynamic tasks use up unknown and potentially large amounts of dynamic memory. Memory allocated to dynamic tasks cannot be explicitly deallocated.	Not applicable.
		Avoid recursion.	Recursion uses up unknown and potentially large amounts of dynamic memory.	Put exception handlers for *STORAGE_ERROR* in recursive subprograms.
		Use aliased objects and components subject to restrictions.	Good technique for "safe" pointer algorithms, but can require unpredictable storage if not used properly.	Avoid language facilities that interfere (e.g., discriminants) with the use of aliased objects and components..
		Avoid/restrict the use of access types.	Can lead to unsafe dynamic memory allocation techniques and erroneous storage pool allocation schemes.	Several alternatives, e.g., create array of base type and use index as access variable.
		Avoid user-defined finalization.	Very difficult to analyze statically.	Explicitly invoke operations inherent in user-defined finalization.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Do not instantiate generic units during runtime.	Generic units are not desirable in any safety-significant software.	Not Applicable.
		Minimize use of large composite objects.	Large objects can cause stack overflows.	Not Applicable.
		The **others** clause for aggregates should not be used.	The **others** clause permits size of aggregate objects to be dynamic.	Inspect aggregate implementation to ensure no dynamic storage used.
		Minimize use of unconstrained types.	Due to their impact on memory allocation.	Use with caution and justify.
		Use **length** clauses if dynamic memory allocation is necessary.	To reserve memory in advance.	Not Applicable.
		Provide handlers for the predefined exception STORAGE_ERROR if dynamic memory allocation is necessary.	To provide a graceful recovery from memory exhaustion.	Not Applicable.
		Explicitly deallocate dynamic memory.	The run-time executive's garbage collector should not be relied upon.	Not applicable.
		Do not assign values of dynamically allocated access objects to other access objects.	They may point to invalid addresses if the original memory is deallocated.	Not applicable.
9.1.1.2 Memory Paging and Swapping	High	No Ada specific guideline, see the generic guideline.	See generic guidelines.	See generic guidelines.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
9.1.2.1 Structure	Medium	Do not use **gotos**.	The use of **gotos** clouds the code structure and should be avoided.	Clearly document, justify, test.
		No more than one **exit** statement per loop.	Multiple exit conditions from a loop indicate lack of cohesion, i.e., more than one purpose for loop.	Clearly document, justify.
		Minimize **return** statements.	Multiple **return** statements can confuse meaning of subprogram.	Clearly document and justify.
9.1.2.2 Control Flow Complexity	High	Limit nesting levels.	Excess complexity makes it difficult to predict the program flow and impedes review and maintenance.	Project guidelines should set specific limits on nesting levels.
		Use **if .. elsif** instead of nested **if** statements.	Reduces complexity and nesting levels.	Not applicable.
		Use **case** statements in preference to **if .. elsif** statements whenever possible.	Reduces complexity and nesting levels.	Use **if** statements if only two branches or if control path not dependent upon discrete value.
		When using **case**, also use **when others** to catch unplanned or unknown alternatives.	**when others** traps unplanned and unknown alternatives.	Clearly document, justify, test.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
9.1.2.3 Single Entry and Exit Points	Medium	One normal (non-exception) **`entry`** and **`exit`** per subprogram.	Single **`entry`** and **`return`** points are easier to understand, test, and less expensive to design, build, and maintain than multiple entries and returns.	Document secondary entry and exit points.
		Limit the number of exception entry/exit points.	To avoid complicating the control flow.	Clearly document each entry/exit point.
		Avoid multiple task entry points.	To minimize program complexity.	Not Applicable.
		Show a subprogram body is elaborated before it's called.	To avoid `Program_Error` exception.	**`Pragma`** `Elaborate_Body` can ensure no code executed between spec and body.
		Indefinite return types from functions should not be used.	Anonymous return object poses control flow problem and dynamic memory problem.	Returned object has the tag of the returned type.
		Indefinite formal procedures in parameters in functions should be used with care.	To minimize complexity and avoid need for dynamic memory.	Use constrained return type or procedure returning value as `out` parameter.
9.1.2.4 Program Instrumentation	Medium	No Ada specific guideline, see the generic guideline.	See generic guidelines.	See generic guidelines.
9.1.3.1 Tasking	High	Minimize the use of tasking.	Timing uncertainties, sequence of execution uncertainties, and vulnerability to race conditions and deadlocks.	Tasking requires compelling justification.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid using the **abort** statement.	If a task is aborted, then all tasks dependent on it are aborted. Furthermore subprograms and blocks which were called by it will also be aborted. If the task was suspended, the abort will cause it to appear to have been completed. Delays are canceled by aborts.	**Aborts** require compelling justification.
		Avoid dynamic tasking.	Dynamic tasking complicates the predictability of the run-time behavior of the program.	Thoroughly justify dynamic tasking and thoroughly test that all problems it can cause are handled.
		Use **delay**.only for waiting, not synchronization.	**delay** sets a minimum time delay.	Not Applicable.
		Minimize the number of **accept** and **select** statements per **task**. Minimize the number of **accept** statements per **entry**.	These guidelines are motivated by the reduction of conceptual complexity and the need to control the task body size. Additionally, a large number of **accept** and **select** statements carries with it a large amount of inter-task communication and overhead.	Not applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid conditional entry calls.	Use of these constructs always poses a risk of race conditions. Their use in loops, particularly with poorly chosen task priorities, can have the effect of busy waiting. Also these constructs are very much implementation dependent.	Justify any usage. Thoroughly test for occurrence of race conditions.
		Avoid selective waits with else parts.		
		Avoid timed entry calls.		
		Avoid selective waits with delay alternatives.		
		Minimize the use of the `PRIORITY` **`pragma`**.	Ada tasking is based on preemption and requires that tasks be synchronized only by features of the language. The scheduling algorithm is not defined by the language and may vary from time slice to preemptive priority. Some implementations provide several choices that a user may select for the application.	In real-time systems it is often necessary to tightly control the tasking algorithm in order to obtain the required performance. This may require non-preemptive tasking. Program such tasking in Ada.
9.1.3.2 Interrupt Driven Processing	High	Declare interrupt values using named constants and isolate them from other declaration clauses.	Interrupts lead to non-deterministic response times. Interrupt entries are implementation dependent features that may not be supported.	Not Applicable.
		Isolate interrupt receiving tasks into implementation dependent package bodies.		

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Pass interrupts to tasks via normal entries.		
		Do not use task entry points for interrupt processing.		
9.1.3.3 Runtime Environment	Medium	Characterize timing for the Ada Runtime Environment (RTE).	The RTE is delivered by vendors and needs to be tested to ensure that it is deterministic, functionally correct, and satisfies timing requirements.	Not Applicable.
9.1.4.1 Initialization of Variables	High	Initialize all variables.	Ada compilers will not generally initialize variables, therefore the contents will be undefined.	The access type is always initialized to **`null`**.
		If initialization is via function call, perform initialization in program unit body, not declaration section.	Body of function may not have been elaborated when declaration section of program unit is being elaborated; `PROGRAM_ERROR` exception could be raised.	**`pragma`** `Elaborate` can be used to ensure body of function elaborated before declaration section.
		Do not initialize very large objects via aggregates.	Could cause system crash in some implementations.	Not applicable.
9.1.4.2 Interface Ambiguities	Medium	Minimize interface ambiguities.	Interface errors account for many coding errors.	Not Applicable.
		Explicitly specify the modes of parameters.	Aids understandability for those who do not know default mode.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Restrict the use of `in out` mode.	Reduces ambiguity; makes plain the intention regarding changes in parameter. Results in safer coding; objects passed into subprograms and meant to be unchanged cannot unintentionally be changed.	Objects of limited type cannot be **out** mode.
		Use named parameters associations.	Using named parameters for calling functions and procedures, improves readability and reliability.	Not Applicable.
		Use target data type instead of pointer type when referencing data.	Reduces ambiguity; makes plain which data, pointer or target data, are to be processed. Results in safer coding; objects of a target type passed into subprograms and meant to be unchanged cannot unintentionally be changed.	Clearly justify, document, and test.
		Avoid aliased parameters.	When aliased parameters are used, results from subprograms frequently incorrect and implementation dependent.	Clearly document, justify, and test any use.
		Use representation items/clauses cautiously.	There may be unforeseen consequences of representation choices.	Use representation-specified entities in most constrained way possible and with smallest scope.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Apply special restrictions to use of **pragma** pack.	Can yield erroneous results, e.g., when applied to 'alignment or **aliased**-applied object.	Verify restrictions are adhered to.
		Scrutinize use of representation attributes.	There may be unforeseen consequences.	Use in most constrained way possible and with smallest scope.
		Avoid enumeration clauses.	There may be unforeseen consequences, but may be used in body to interface with environment.	Use in most constrained way possible and with smallest scope; recast value into enumeration-type object after validity check..
		Avoid enumerated types with representation clauses as basis for iterative constructs (e.g., **for**).	Unanticipated effects on memory, timing, and value checking.	Use derived integer type with range including all needed values.
9.1.4.3 Data Typing	High	Take advantage of Ada's strong typing.	Strong typing catches range errors as well as typing errors, making safer code.	Not Applicable.
		Do not suppress Ada's run-time constraint checks.	Strong typing conducts run-time range checks of parameters entering procedures and functions and of copy operations to variables.	Clearly document and justify any deviation.
		Define scalar data types with the narrowest possible limits.	Enhances early detection of out-of-range data values.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Use care in scalar subexpressions ported from Ada 83 implementations.	Some Ada 83 implementations check intermediate values for out-of-range conditions.	Use Ada 95 implementations or use Ada 83 implementations that do not have this problem.
		Minimize the use of type conversions.	Type conversions subvert the benefits of strong typing.	Justify and document.
		Avoid use of unchecked conversions.	This may lead to assigning illegal values to objects.	Not Applicable.
		Limit the use of access type data.	Access data is harder to verify and maintain.	Limit direction to situations where there are no other reasonable alternatives, performing validation on indirectly accessed data prior to setting or use to ensure the correctness of the accessed data.
		Avoid declaring variables in package specifications.	To increase data abstraction and reduce coupling.	Not Applicable.
9.1.4.4 Precision and Accuracy	High	Do relational tests on real values with <= and >= rather than <, >, =, and /=.	The values of fixed point and floating point numbers only approximate the specific numbers. The operations on these numbers are also approximations. Therefore, proper precision and accuracy are necessary for critical systems.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Use Ada attributes in comparisons and checking for small values for real numbers.		
		Test carefully using Ada attributes around special values such as 0.0.		
9.1.4.5 Order of Precedence	Medium	Use parentheses or other mechanisms for ensuring that the order of evaluation of operations is explicitly stated.	The default order of precedence should not be depended on if any misinterpretation can be made.	Use other forms to enhance readability if parentheses are excessive.
		Use parentheses to separate operations of different precedence.	Less chance of misinterpretation.	Not Applicable.
		Use parentheses or other mechanisms for ensuring that the order of evaluation of operands is correct.	If expressions contain functions with side effects that affect each other, results of expressions will be implementation dependent.	Avoid functions with side effects.
		Account for full evaluation in logical expressions.	Avoid subtle logic errors.	Use alternative constructs, e.g., **AND THEN**, if necessary.
9.1.4.6 Side Effects	Medium	Verify that functions do not have side effects.	Side effects can lead to problems with unplanned dependencies and can cause bugs that are very hard to find.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
9.1.4.7 Separating Assignment from Evaluation	Medium	Separate assignments from evaluations.	To avoid side effects.	Not Applicable.
9.1.4.8 Library Size	Medium	Avoid derivations from partial views of a type and avoid extension of tagged type outside its scope (declarative region).	Can allow child package to have multiple simultaneous views.	Procedural (management) control.
9.1.4.9 Dynamic Binding	High	Minimize dynamic binding.	Dynamic binding causes unpredictability in name/class associations and reduces run-time predictability.	All cases where dynamic binding is required should be justified.
9.1.4.10 Operator Overloading	Medium	Guidance on use of operator overloading should be included in a project specific standards manual and coding should comply with this standard.	Operator overloading can be problematic from the perspective of predictability because it is unclear how a compiler would bind code for different polymorphic code.	Sometimes acceptable for achieving uniformity across different data types.
		Avoid operator overloading when the inherent precedence of the operator is different from that desired.	Misinterpreting operator precedence could lead to incorrect results for expressions.	Use parentheses to override inherent precedence. Document usage.
		Preserve conventional meaning of overloaded operators.	Failure to preserve conventional meaning results in confusing code.	Not Applicable.
9.2.1 Software Diversity	Medium	No Ada specific guideline, see the generic guideline.	See generic guidelines.	See generic guidelines.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
9.2.2.1 Local Handling of Exceptions	High	Minimize propagation of exceptions.	It may obscure program logic.	Justify and document clearly.
		Localize handling of predefined exceptions.	System failures can be avoided if exceptions are handled locally.	If not possible, use thorough testing and analysis to verify behavior during exception handling.
9.2.2.2 External Flow Control	High	Declare exceptions to be handled in calling program units alongside declaration of called unit.	Allows calling program unit to choose what action to take upon raising of exception.	Not Applicable
9.2.2.3 Uniform Exception Handling	High	Clearly express and document exception handling.	To clarify control flow.	Not Applicable.
		Handle predefined exceptions.	To catch unexpected error conditions.	Not Applicable.
		Do not raise predefined exceptions explicitly.	To avoid unanticipated behavior.	Not Applicable.
		Handle all program-defined exceptions.	It is not good practice to ignore exceptions in safety critical systems; they can be propagated to the Real Time Executive and cause the system to crash.	Not Applicable
		Use exception handling only for abnormal events.	Exceptions increase control flow complexity and should not be used where inappropriate.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Minimize side effects.	Eliminates side effects in case of exceptions.	May increase time and memory requirements by unallowable amounts.
		Avoid compiler vendor-specific exceptions.	Inhibits portability and understandability.	Justify and document any use of compiler-specific exceptions.
		Use and flag `other` in exception handler definitions.	All conditions in exception handling must be well defined.	Not applicable.
		Demonstrate absence of failure of language-defined checks.	Failure of a check should be considered to be erroneous execution.	Not applicable.
9.2.3 Input and Output Data Checking	High	Check input data.	Input/output data should be regarded as untrustworthy until proven otherwise.	Not applicable. May be augmented with exception handling.
9.2.3.1 Input/Output Data	High	Never disable automatic checks on `in out` and `out` data.	Input/output data should be regarded as untrustworthy until proven otherwise.	Not applicable. May be augmented with exception handling.
9.2.3.2 Packages	High	Restrictions must be placed on the use of packages.	Package use can lead to `elaboration_check` failure or elaboration delay.	Restrictions minimize negative effects.
9.2.3.3 Private Types	High	Private types may be used only if subject to severe limitations.	Private types create two views; these views can cause anomalies.	Limitations minimize negative effects.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
9.3.1 Built-In Functions	Low	Ada has few built-in functions. Therefore no Ada specific guideline. See the generic guideline.	See generic guidelines.	See generic guidelines.
9.3.2 Compiled Libraries	Low	Avoid built-in libraries.	Libraries prevent the programmer from knowing the accuracies, limitations, robustness, and exception handling of the built-in functions.	Thorough testing and error tracking.
9.3.4 Traceability between Source and Compiled Code	High	Maintain traceability between source code and compiled code.	To avoid configuration management problems.	If separate compilation is needed, use the **with** clause and appropriate tools.
9.3.5 Generic Units	Medium	Minimize the use of generic units.	Generic units obscure traceability, testability, and analyzability.	If using generic units, - Instantiate only during initialization - Avoid global variables - Document the restrictions on parameters.
		Do not use shared generics.		
		Type re-export should not be used.		
		Overridden operations should be actual parameters.		
		Instantiate each generic only once per capability.		
9.4.1.1 Indentation Guidelines	Medium	Indentation improves readability and allows the reader to see the structure of the program.	This is especially useful for finding the beginnings and ends of data structures and control flow structures.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Indent and align nested control structures, continuation lines, and embedded program units consistently. Also, distinguish between indentations for nesting and for continuation lines.	Improves readability.	Not Applicable.
9.4.1.2 Descriptive Identifier Names	Medium	Follow project-specific guidelines on naming.	Not Applicable.	Not Applicable.
		Separate words in compound names with underscores.	This will improve reliability because the reader will be able to more easily read the names.	Not Applicable.
		Use underscore "_" to promote readability of numbers.	This will improve reliability because the reader will be able to more easily read the numbers for verification.	Not Applicable.
		Use abbreviations with care.	Avoid jargon, use the names given by the application.	Not Applicable.
9.4.1.3 Comments and Internal Documentation	Medium	Source code shall be supplemented with Ada comments that explain the code.	Comments clarify code and help code maintenance.	Not Applicable.
		Use comments to relate code to higher level design.	Comments should clarify data structure, not repeat what source code states.	Not Applicable.
		Use blank lines to delineate related statements.	Improves understandability.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid use of escapes from language restrictions.	Justified on individual basis elsewhere.	Not Applicable.
		Indicate the scope of renaming.	To improve reviewability.	Not Applicable.
		Provide comments on exception handling.	To improve flow control.	Not Applicable.
		Provide comments on dynamic memory allocation.	Dynamic memory allocation is discouraged and needs proper justification and documentation.	Not Applicable.
		Provide comments on tasking.	To provide traceability.	Not Applicable.
9.4.1.4 Subprogram Size	Medium	No Ada-specific guideline, see the generic guideline.	See generic guidelines.	See generic guidelines.
9.4.1.5 Mixed Language Programming	Medium	Avoid machine code inserts.	There is no requirement on how machine code should be implemented. It is possible that two different vendors' syntax would be different for an identical target and, certainly, differences in lower-level details such as register conventions would hinder portability.	If machine code inserts must be used to meet a project requirement, recognize the portability decreasing effects and isolate and highlight their use.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Minimize interfaces with other languages.	The problems with employing **pragma** INTERFACE are complex. These problems include pragma syntax differences, conventions for linking/binding Ada to other languages, and mapping Ada variables to foreign language variables, among others.	It is often necessary to interact with other languages, if only an assembly language to reach certain hardware features. In these cases, clearly document the requirements and limitations of the interface and **pragma** INTERFACE usage.
		Isolate and clearly document machine language inserts.		
		Isolate all subprograms employing **pragma** INTERFACE to an implementation-dependent (interface) package.		
9.4.1.6 Obscure or Subtle Programming Constructs	High	No Ada-specific guideline, see the generic guideline.	See generic guidelines.	See generic guidelines.
9.4.1.7 Dispersion of Related Elements	Medium	No Ada-specific guideline, see the generic guideline.	See generic guidelines.	See generic guidelines.
9.4.1.8 Use of Literals	Medium	Use symbolic constants instead of literals.	It is far easier to change one value set at the beginning of a source code file than it is to guarantee that all literals associated with such a parameter have been changed completely and correctly throughout all relevant source code files.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Use Ada attributes instead of literals.	Improves portability. Removes need to change source code whenever a value obtainable by Ada attribute occurs.	Not Applicable.
9.4.1.9 Renaming	Medium	Avoid obscuration when renaming.	Do not use renaming if obscuration will occur.	Use only one level of renaming. Have project-specific rules and conventions for renaming. Keep a registry of renamed identifiers for a project.
		Ensure subtype indication in renaming declaration statically matches renamed object subtype.	Constraint specified by the `subtype_indication` of an object renaming is ignored.	Not Applicable.
9.4.1.10 Bitmaps	Medium	Use representation clauses for bit mapping.	To facilitate reviews and reduce the possibility of coding errors.	Not Applicable.
9.4.2.1 Global Variables	Medium	Minimize the use of global variables.	Global variables obscure the passage of data between the inner and outer subprograms.	Clearly identify globals.
9.4.2.2 Complexity of Interfaces	Medium	No Ada language specific guidelines, see the generic guidelines.	Not Applicable	Closely inspect and clearly identify interfaces.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
9.4.2.3 Avoid coupling	Medium	Avoid declaring variables in library package specifications.	Low coupling should be a goal because the fewer the number of connections between modules: 1) the less chance of a failure in one module to propagate, 2) the less chance a change in one module will cause problems in another [This increases reusability] and 3) the easier the learning curve is for the programmer in learning about the other modules.	Not Applicable.
9.4.3 Functional Cohesiveness	Medium	Every subprogram should have one clearly discernible purpose.	Every subprogram should have one clearly discernible purpose with input and output parameters related to that purpose.	Not Applicable.
9.4.4 Malleability	Medium	No Ada-specific guideline, see the generic guideline.	See generic guidelines.	See generic guidelines.
9.4.5 Portability	Medium	Do not use busy loops to suspend execution.	The timing of a loop cannot be depended upon if the code is ported to a different compiler, different machine, or even different operating system.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Validate assumptions about the implementation of language feature when specific implementation is not guaranteed or specified. Do not assume a correlation between SYSTEM.TICK and **package** CALENDAR or **type** DURATION.	Although such a correlation may exist, it is not required to exist.	Not Applicable.
		Avoid the use of **package** SYSTEM constants except in attempting to generalize other machine dependent constructs.	Since the values in this package are implementation-dependent, unexpected effects can result from their use.	Do not use **package** SYSTEM constants to parametrize other implementation-dependent features and access collection size.
		Use only pragmas and attributes defined by the Ada Standard.	The Ada ARM permits an implementor to add pragmas and attributes to exploit a particular hardware architecture or software environment. These are obviously even more implementation-specific and therefore less portable than are an implementor's interpretation of the predefined pragmas and attributes.	Some implementation-dependent features are gaining wide acceptance in the Ada community to help alleviate inherent inefficiencies in some Ada features.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
		Avoid the direct invocation of or implementation dependence upon an underlying host operating system or Ada run-time support system.	Features of an implementation not specified in the Ada ARM will usually differ between implementations.	In real-time embedded systems, often it is not possible to avoid making calls to low-level support system facilities. Isolate the uses of these facilities.
		Minimize and isolate the use of the predefined package LOW_LEVEL_IO.	`LOW_LEVEL_IO` is intended to support direct interaction with physical devices that are usually unique to a given host or target environment. In addition, the data types provided to the procedures are implementation defined. This allows vendors to define different interfaces to an identical device.	Those portions of an application that must deal with this level of I/O, e.g., device drivers and real-time components dealing with discretes, are inherently non-portable.
		Restrict and isolate variables of the **type** `SYSTEM.ADDRESS`.	These variables are hardware-specific.	Not Applicable.

PLC Structured Text

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
10.1.1 Dynamic Memory Allocation	N/A	Not applicable to ST.	ST does not allow dynamic memory allocation.	Not Applicable
10.1.2.1 Structure	N/A	Not applicable to ST.	ST does not have `goto` statements.	Not Applicable
10.1.2.2 Control Flow Complexity	High	Use the case construct instead of multiple `IF...ELSIF`.	Easier to understand and modify.	Not Applicable
		Define default clauses for `CASE` statements	To avoid unanticipated program flow	Not applicable
		Minimize `EXIT` in iterative constructs	To reduce multiple or hard-to-find exits from loops	Mark every `EXIT` with an inline comment
10.1.2.3 Initialization of Variables	High	Initialize internal and retentive variables before use.	Internal variables have default values. Retentive variable keep their values across warm starts. These values may not be those intended.	Not Applicable
10.1.2.4 Single Entry and Exit Points	Medium	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.1.2.5 Interface Ambiguities	Medium	Avoid using Direct PLC I/O or Memory references.	Direct references are not declared and may be difficult to debug.	Not Applicable
10.1.2.6 Data Typing	High	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.1.2.7 Precision and Accuracy	High	Use the `LREAL` data type where available.	The shorter `REAL` data type may not provide adequate precision.	Not Applicable.
10.1.2.8 Order of Precedence	Medium	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
10.1.2.9 Side Effects	Medium	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.1.2.10 Separating Assignment from Evaluation	N/A	ST does not allow assignment inside an evaluation expression.	Not Applicable.	Not Applicable.
10.1.2.11 Program Instrumentation	Medium	Online monitoring of PLC programs should be avoided	It interferes with real-time response, may cause non-reproducible errors.	Not Applicable.
10.1.2.12 Library Size	N/A	Classes and objects are not supported in ST.	Not Applicable.	Not Applicable.
10.1.2.13 Dynamic Binding	N/A	No run-time binding permitted.	Not Applicable.	Not Applicable.
10.1.2.14 Operator Overloading	N/A	Overloading and polymorphism are not supported.	Not Applicable.	Not Applicable.
10.1.3.1 Program Scan Time	High	Ensure that program scan time is not exceeded.	To avoid spurious triggering the watchdog timer.	Not Applicable.
10.1.3.2 Currency of I/O values	High	Ensure that field I/O values are current	Changing I/O values during a program scan may have unintended side effects	Use synchronous I/O scan if available.
10.1.3.3 Sensor and actuator delays	Medium	Account for delays in sensor and actuator signals.	These delays may vary widely.	Not Applicable.
10.1.3.4 Tasking	High	Account for task priorities.	To ensure appropriate real time performance	Not Applicable.
10.1.3.5 Interrupt Driven Processing	High	If interrupts are available, generic guidelines apply.	Interrupts are not widely supported.	See Generic Guidelines.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
10.1.3.6 Self Modifying Code	N/A	ST does not allow self-modifying code.	Not Applicable.	Not Applicable.
10.2.1.1 System Health Monitoring	Medium	Ensure completeness, correctness and observability of parameters.	Need to ensure adequacy of monitoring.	Not Applicable.
10.2.1.2 Fault Routines	Medium	Ensure completeness, correctness and observability of parameters.	Need to ensure adequacy of failure handling.	Not Applicable.
10.2.1.3 Watchdog Timer	High	Initialize when needed. Ensure adequate fault routine. Use external timer when needed.	Need to ensure adequacy of failure handling.	Not Applicable.
10.3.1 Built-In Functions	Low	Use only Function Blocks predefined in the Standard.	Not Applicable.	Not Applicable.
10.3.2 Compiled Libraries	Low	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.1.1 Conformance to Indentation Guidelines	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.1.2 Descriptive Identifier Names	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.1.3 Comments and Internal Documentation	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.1.5 Subprogram Size	Medium	Generic Guidelines apply.	See Generic Guidelines.	Not Applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
10.4.1.6 Mixed Language Programming	N/A	Use of other IEC 1131 languages in ST is encouraged.	These languages were designed for interchangeable	Clearly document and justify.
10.4.1.7 Obscure or Subtle Programming Constructs	High	The use of direct variables should be avoided.	This is the only construct in ST that may result in obscure programs.	Not Applicable.
10.4.1.8 Dispersion of Related Elements	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.1.9 Use of Literals	Medium	Use symbolic constants instead of literals.	To protect literals, and control the uniformity of the value.	Not Applicable.
10.4.2.1 Modularity	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.2.2 Information Hiding	Medium	Use variables limited in scope.	This is the extent of information hiding available in ST.	Not applicable.
10.4.2.3 Global Variables	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.2.4 Complexity of Interfaces	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.3 Functional Cohesiveness	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
10.4.4 Malleability	Medium	Avoid Direct variables. Avoid direct I/O references.	These constructs are difficult to change safely.	Clearly document and justify.
10.4.5 Portability	Medium	Avoid the use of ST extensions.	Vendor-specific extensions will make programs non-portable.	Not applicable.

Function Block Diagrams

FBD

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
11.1.1 Dynamic Memory Allocation	N/A	Not applicable to FBD.	FBD does not allow dynamic memory allocation.	Not applicable
11.1.2.1 Structure	N/A	Generic Guidelines apply.		
11.1.2.2 Control Flow Complexity	High	Decompose the program into cohesive function blocks.	These two guidelines are in conflict; a balance should be struck on a case-by-case basis.	Clearly document and justify.
		Limit function block nesting levels.		
11.1.2.3 Initialization of Variables	High	Initialize internal and retentive variables before use.	Internal variables have default values. Retentive variable keep their values across warm starts. These values may not be those intended.	Not applicable
11.1.2.4 Single Entry and Exit Points	Medium	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.1.2.5 Interface Ambiguities	Medium	Avoid using Direct PLC I/O or Memory references.	Direct references are not declared and may be difficult to debug.	Not applicable
11.1.2.6 Data Typing	High	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.1.2.7 Precision and Accuracy	High	Use the LREAL data type where available.	The shorter REAL data type may not provide adequate precision.	Not applicable.
11.1.2.8 Order of Precedence	Medium	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.1.2.9 Side Effects	Medium	Generic guidelines apply.	See Generic Guidelines.	See Generic Guidelines.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
11.1.2.10 Separating Assignment from Evaluation	N/A	FBD does not allow assignment inside an evaluation expression.	Not applicable.	Not applicable.
11.1.2.11 Program Instrumentation	Medium	Online monitoring of PLC programs should be avoided	It interferes with real-time response, may cause non-reproducible errors.	Not applicable.
11.1.2.12 Library Size	N/A	Classes and objects are not supported in FBD.	Not applicable.	Not applicable.
11.1.2.13 Dynamic Binding	N/A	No run-time binding permitted.	Not applicable.	Not applicable.
11.1.2.14 Operator Overloading	N/A	Overloading and polymorphism are not supported.	Not applicable.	Not applicable.
11.1.3.1 Tasking	High	Account for task priorities.	To ensure appropriate real time performance	Not applicable.
11.1.3.2 Interrupt Driven Processing	High	If interrupts are available, generic guidelines apply.	Interrupts are not widely supported.	See Generic Guidelines.
11.1.3.3 Program Scan Time	High	Ensure that program scan time is not exceeded.	To avoid spurious triggering the watchdog timer.	Not applicable.
11.1.3.4 Currency of I/O values	High	Ensure that field I/O values are current	Changing I/O values during a program scan may have unintended side effects	Use synchronous I/O scan if available.
11.1.3.5 Sensor and actuator delays	Medium	Account for delays in sensor and actuator signals.	These delays may vary widely.	Not applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
11.1.3.6 Self Modifying Code	N/A	FBD does not allow self-modifying code.	Not applicable.	Not applicable.
11.2.1.1 System Health Monitoring	Medium	Ensure completeness, correctness and observability of parameters.	Need to ensure adequacy of monitoring.	Not applicable.
11.2.1.2 Fault Routines	Medium	Set up a non-periodic task for internal diagnostics.	Need to ensure adequacy of failure handling.	Not applicable.
11.2.1.3 Watchdog Timer	High	Monitor task overrun flag. Minimize potential runtime task overruns.	Need to ensure task deadlines are met.	Not applicable.
11.3.1 Built-In Functions	Low	Use only Function Blocks predefined in the Standard.	Not applicable.	Not applicable.
11.3.2 Compiled Libraries	Low	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.1.1 Layout of FBD diagrams	Medium	- Use Function Blocks aggressively - Keep inputs and outputs to each function block close - Avoid right-to-left connections - Avoid crossing connections	To make FBD diagrams more readable and maintainable.	Not applicable.
11.4.1.2 Descriptive Identifier Names	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.1.3 Comments and Internal Documentation	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
11.4.1.4 Subprogram Size	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.1.5 Mixed Language Programming	Medium	Use of other IEC 1131 languages in FBD is encouraged.	These languages were designed for interchangeable	Clearly document and justify.
11.4.1.6 Obscure or Subtle Programming Constructs	High	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.1.7 Dispersion of Related Elements	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.1.8 Use of Literals	Medium	Use symbolic constants instead of literals.	To protect literals, and control the uniformity of the value.	Not applicable.
11.4.2.1 Modularity	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.2.2 Information Hiding	Medium	Use variables limited in scope.	This is the extent of information hiding available in FBD.	Not applicable.
11.4.2.3 Global Variables	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.2.4 Complexity of Interfaces	Medium	Limit the number of parameters.	To simplify the interface and the visual image.	Not applicable.
		Use structured variables.	To limit the number of parameters.	Not applicable.
		Use the capabilities of the IEC 1131-3 development environment.	Using both the textual and graphic modes will increase the clarity and readability of the problem.	Not applicable.

Generic Characteristics	Significance	Guideline	Rationale	Mitigation
11.4.3 Functional Cohesiveness	Medium	Generic Guidelines apply.	See Generic Guidelines.	See Generic Guidelines.
11.4.4 Malleability	Medium	Avoid Direct variables. Avoid direct I/O references.	These constructs are difficult to change safely.	Clearly document and justify.
11.4.5 Portability	Medium	Avoid the use of FBD extensions.	Vendor-specific extensions will make programs non-portable.	Not applicable.

Appendix C Glossary

The definitions in this glossary were derived from the following sources:

- ANSI/MIL-STD-1815A, *Reference Manual for the Ada Language*,, American National Standards Institute/U.S. Department of Defense, 1983.
- ANSI/IEEE 729-1983, *Glossary of Software Engineering Terminology,* Institute of Electrical and Electronic Engineers, 1983
- Allen-Bradley, *PLC-5 Programming Software - Programming*, Publication 6200-6.4.7 November, 1991
- Digital Equipment Corporation, *Programming in VAX-11 C,* Publication AA-L370A-TE, Maynard, MA, May, 1982.

accept statement

In Ada, See entry.

access type

In Ada, a value that designates an object created by an allocator. The designated object can be read and updated via the value of the access type. The definition of an access type specifies the type of the objects designated by values of the access type. If uninitialized, a value of an access type (an access value) is a null value. See also collection.

actual parameter

See *parameter*.

aggregate

In Ada, the evaluation of an aggregate yields a value of a composite type. The value is specified by giving the value of each of the components. Either positional association or named association may be used to indicate which value is associated with which component.

allocator

In Ada, an allocator creates an object and returns a new access value which designates the object.

arithmetic operator

An operator that performs an arithmetic operation. Examples include the unary minus (-), multiplication (*), division (/), addition (+) and subtraction (-).

array

An aggregate data type consisting of subscripted elements of the same type. Elements of an array can have one of the fundamental types or can be structures, unions, or other arrays (to form multidimensional arrays).

array type

A value of an array type consists of components which are all of the same subtype (and hence, of the same type). Each component is uniquely distinguished by an index (for a one-dimensional array) or by a sequence of indices (for a multidimensional array). Each index must be a value of a discrete type and must lie in the correct index range.

assignment

Assignment is the operation that replaces the current value of a variable by a new value. An assignment statement specifies a variable on the left, and on the right, an expression whose value is to be the new value of the variable.

assignment expression

In C/C++,an expression of the form:

E1 *asgnop* E2

where El must be an lValue, *asgnop* is an assignment operator, and E2 is an expression. The type of an assignment expression is that of its left operand. The value of an assignment expression is that of the left operand after the assignment has taken place. If the operator is of the form "*op*=", then the operation El *op* (E2) is performed, and the result is assigned to the object referred to by El; El is evaluated only once.

asterisk (*)

In C/C++, as a unary operator, treats its operand as an address and results in the contents of that address. As a binary operator, multiplies two operands, performing the usual arithmetic conversions. As an assignment operator (*=), multiplies an expression by the value of the object referred to by the left operand, and assigns the product to the object.

attribute

In Ada, the evaluation of an attribute yields a predefined characteristic of a named entity; some attributes are functions.

binary operator

An operator that is placed between two operands. The binary operators include arithmetic operators, shift operators, relational operators, equality operators, bitwise operators (AND, OR, and XOR), logical connectives, and the comma operator, in that order of precedence. All binary operators group from left to right. (Note: C has no operator for exponentiation.)

bitwise operator

In C/C++, an operator that performs a bitwise logical operation on two operands, which must be integral. The usual arithmetic conversions are performed. Both operands are evaluated. All bitwise operators are associative, and expressions using them may be rearranged. The set comprises, in order of precedence, the single ampersand ([&] bitwise AND), the circumflex ([^] bitwise exclusive OR), and the single vertical bar ([|] bitwise inclusive OR).

block

See compound statement.

block statement

A block statement is a single statement that may contain a sequence of statements. It may also include a declarative part, and exception handlers; their effects are local to the block statement.

body

A body defines the execution of a subprogram, package, or task. A body stub is a form of body that indicates that this execution is defined in a separately compiled subunit.

cast

In C/C++, an expression preceded by a cast operator of the form "(typename)". The cast operator forces the conversion of the evaluated expression to the given type. The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction. The cast operator has the same precedence as the other unary operators. See also *type conversion.*

character

(1) A member of the ASCII character set.
(2) An object of the C data type char -- that is, a byte. (An object of type char always represents a single character, not a string.)
(3) A constant of type char, consisting of up to four ASCII characters enclosed in single quotes (', not "). See also *string.*

cohesiveness

The manner and degree to which the tasks performed by a single software module are related to one another.

collection

In Ada, a collection is the entire set of objects created by evaluation of allocators for an access type.

comma operator

In C/C++, an operator used to separate two expressions:E1, E2

The expressions El and E2 are evaluated left to right, and the value of El is discarded. The type and value of the comma expression are those of E2.

comment

In C/C++, a sequence of characters introduced by the pair /* and terminated by */. Comments are ignored during compilation. They may not be nested.

In C++, in addition to /* ... */, a sequence of characters starting with // and ending with a newline. In Ada, a sequence of characters starting with -- and ending with a newline.

compilation unit

A compilation unit is the declaration or the body of a program unit, presented for compilation as an independent text. It is optionally preceded by a context clause, naming other compilation units upon which it depends by means of one more with clauses.

component

In Ada, a component is a value that is a part of a larger value, or an object that is part of a larger object.

composite type

In Ada, a composite type is one whose values have components. There are two kinds of composite type: array types and record types. Records are called structures in C/C++.

compound statement

A compound statement consisting of valid C/C++ statements enclosed in braces ({ }). Compound statements can also include declarations. The scope of these variables is local to the block.

conditional operator

The C/C++ operator (? :), which is used in conditional expressions of the form:

E1 ? E2 : E3

where El, E2, and E3 are expressions. El is evaluated, and if it is nonzero, the result is the value of E2; otherwise, the result is the value of E3. Only one of E2 and E3 is evaluated.

constant

A primary expression whose value does not change. A constant may be literal or symbolic.

constant expression

An expression involving only constants. Constant expressions are evaluated at compile time and may therefore be used wherever a constant is valid.

constraint

In Ada, a constraint determines a subset of the values of a type. A value in that subset satisfies the constraint.

context clause

See *compilation unit.*

conversion

The changing of a value from one data type to another. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; that type is also the type of the assignment expression. In C/C++, conversions are also performed when arguments are passed to functions: **char** and **short** become **int; unsigned char** and **unsigned short** become **unsigned int; float** becomes **double.** Conversions can also be forced by means of a *cast* (see). Conversions are performed on operands in arithmetic expressions by the usual arithmetic conversions.

data definition

The syntax that both declares the data type of an object and reserves its storage. For variables that are internal to a function, the data definition is the same as the declaration. For external variables, the data definition is external to any function (an external data definition).

declaration

A statement that defines the characteristics (such as data type) of one or more variables.

declarative part

In Ada, a declarative part is a sequence of declarations. It may also contain related information such as subprogram bodies and representation clauses.

derived type

In Ada, a derived type is a type whose operations and values are replicas of those of an existing type. The existing type is called the parent type of the derived type.

designate

In Ada, See *access type*, *task*.

direct visibility

See *visibility*.

discrete type

A discrete type is a type which has an ordered set of distinct values. The discrete types are the enumeration and integer types. Discrete types are used for indexing and iteration, and for choices in case statements and record variants. Discrete types are also called sets in Pascal.

discriminant

In Ada, a discriminant is a distinguished component of an object or value of a record type. The subtypes of other components, or even their presence or absence, may depend on the value of the discriminant.

discriminant constraint

In Ada, a discriminant constraint on a record type or private type specifies a value for each discriminant of the type.

diversity

The realization of the same function by different means.

elaboration

In Ada, the elaboration of a declaration is the process by which the declaration achieves its effect (such as creating an object); this process occurs during program execution.

entry

In Ada, an entry is used for communication between tasks. Externally, an entry is called just as a subprogram is called; its internal behavior is specified by one or more accept statements specifying the actions to be performed when the entry is called.

enumerated type

An enumerated type is a discrete type whose values are represented by enumeration literals which are given explicitly in the type declaration. These enumeration literals are either identifiers or character literals.

equality operator

In C/C++, one of the operators == (equal to) or != (not equal to). They are analogous to the relational operators, but at the next lower level of precedence. In Ada, these are = and /= respectively.

evaluation

The evaluation of an expression is the process by which the value of the expression is computed. This process occurs during program execution.

exception

An exception is an error situation which may arise during program execution. To raise an exception is to abandon normal program execution so as to signal that the error has taken place. An exception handler is a portion of program text specifying a response to the exception. Execution of such a program text is called handling the exception.

expanded name

In Ada, an expanded name denotes an entity which is declared immediately within some construct. An expanded name has the form of a selected component: the prefix denotes the construct (a program unit; or a block, loop, or accept statement); the selector is the simple name of the entity.

exponentiation operator

The C language does not provide an exponentiation operator. In Ada, it is **.

expression

An expression (i.e., series of tokens) that the compiler can use to produce a value. Expressions have one or more operands and, usually, one or more operators. (An identifier with no operator is an expression that yields a value directly.) Operands are either identifiers

(such as variable names) or other expressions, which are sometimes called subexpressions. See also *operator.*

external variable

A variable that is defined externally to any function. External variables provide a means other than argument passing for exchanging data between the functions that comprise a C/C++ program.

fixed point type

See *real type.*

floating point type

See *real type.*

formal parameter

See *parameter.*

function

The primary unit from which C programs are constructed. A function definition begins with a name and argument list, which are followed by the declarations of the arguments (if any) and the body of the function enclosed in braces ({ }). The function body consists of the declarations of any local variables and the set of statements that perform its action. Functions need not return a value to the caller. All C functions are external; that is, a function may not contain another function. See also *function call.*

function call

A primary expression followed by parentheses. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function. In C, any previously undeclared identifier followed immediately by parentheses is contextually declared as a function returning **int.** A function may call itself recursively.

fundamental type

In C/C++, the set of arithmetic data types plus pointers. The fundamental types in C/C++ comprise those data types that can be represented naturally on a particular machine; usually, this means integers and floating-point numbers of various machine dependent sizes, and machine addresses.

generic unit

In Ada, a generic unit is a template either for a set of subprograms or for a set of packages. A subprogram or package created using the template is called an instance of the generic unit. A generic instantiation is the kind of declaration that creates an instance. A generic unit is written as a subprogram or package but with the specification prefixed by a generic formal part which may declare generic formal parameters. A generic formal parameter is either a type, a subprogram, or an object. A generic unit is one of the kinds of program unit.

handler

See *exception.*

identifier

A sequence of letters and digits used as the name of an entity. In C/C++, the first 31 of an identifier must be unique. In Ada, the length is limited to that of a line in the source code. The underscore (_) is considered a letter in this context. The first character of an identifier must be a letter. Upper- and lowercase letters specify different identifiers in VAX-11 C. Note, however, that all external names are converted to uppercase to be consistent with VAX/VMS.

index

See array type.

index constraint

An index constraint for an array type specifies the lower and upper bounds for each index range of the array type in Ada.

indexed component

An indexed component denotes a component in an array. It is a form of name containing expressions which specify the values of the indices of the array component. An indexed component may also denote an entry in a family of entries.

initializer

The part of a declaration that gives the initial value(s) for the preceding declarator. In C/C++, an initializer consists of an equal sign (=) followed by either a single expression or a comma-separated list of one or more expressions in braces.

instance

An object created according to a given definition. See also *generic unit.*

integer type

An integer type is a discrete type whose values represent all integer numbers within a specific range in Ada or Pascal.

integral type

In C/C++, one of the data types char or int (all sizes, signed or unsigned).

keyword

A word (series of characters) that is reserved by the language and cannot be used as an identifier. Keywords identify statements, storage classes, data types, and the like.

label

A label is the target of a *goto* statement.

lexical element

A lexical element is an identifier, a literal, a delimiter, or a comment.

limited type

In Ada, a limited type is a type for which neither assignment nor the predefined comparison for equality is implicitly declared. All task types are limited. A private type can be defined to be limited. An equality operator can be explicitly declared for a limited type.

literal

A literal represents a value literally, that is, by means of letters and other characters. A literal is either a numeric literal, an enumeration literal, a character literal, or a string literal.

logical expression

An expression made up of two or more operands separated by logical connectives. Each operand must be of a fundamental type or must be a pointer or other address expression. Operands do not have to be of the same type. In C/C++, logical expressions always return 1 or 0 (type **int**) to indicate a true or false value, respectively. Logical expressions are always evaluated from left to right, and the evaluation stops as soon as the result is known.

lvalue

In C/C++, an lvalue is an expression which can be assigned to. An lvalue is required on the left-hand side of an assignment operator (hence its name) and as the operand of certain other operators, such as the increment (++) and decrement (--) operators. A variable name is an example of an lvalue, since its address can be taken (with &), and values can be assigned to it. A constant is an example of an expression that is not an lvalue.

macro

Used primarily in C/C++, a text substitution that is defined with the **#define** preprocessor control line and includes a list of "parameters." The parameters in the **#define** control line are replaced at compile time with the corresponding arguments from a macro reference encountered in the source text.

mode

In Ada, see parameter.

model number

In Ada, a model number is an exactly representable value of a real type. Operations of a real type are defined in terms of operations on the model numbers of the type. The properties of the model numbers and of their operations are the minimal properties preserved by all implementations of the real type.

multiplicative operator

An operator that performs multiplication (*), division (/), or modulo arithmetic. It performs the usual arithmetic conversions on its operands. The modulo operator (% in C/C++, **MOD** in Ada) yields the remainder of the division of the first operand by the second.

name

A name is a construct that stands for an entity: it is said that the name denotes the entity, and that the entity is the meaning of the name. See also declaration, prefix.

named association

A named association specifies the association of an item with one or more positions in a list, by naming the positions.

Programmable Logic Controller (PLC)

A special purpose computer having a central processing unit (CPU), power supply, programming panel, inputs and outputs. A PLC also provides the capability to support remote Input/Output, special purpose Input/Output, Input/Output housing, connection cables, and communication boards.

object

One of the basic elements that the language can manipulate — that is, the elements to which operators can be applied. In objects include data (such as integers, real numbers, or characters), data structures (arrays, structures, unions), and other user-defined data types.

operation

An operation is an elementary action associated with one or more types. It is either implicitly declared by the declaration of the type, or it is a subprogram that has a parameter or result of the type.

operator

An operator is an operation which has one or two operands. A unary operator is written before an operand; a binary operator is written between two operands. This notation is a special kind of function call. An operator can be declared as a function. Many operators are implicitly declared by the declaration of a type (for example, most type declarations imply the declaration of the equality operator for values of the type).

overloading

An identifier can have several alternative meanings at a given point in the program text: this property is called overloading. For example, an overloaded enumeration literal can be an identifier that appears in the definitions of two or more enumeration types. The effective meaning of an overloaded identifier is determined by the context. Subprograms, aggregates, allocators, and string literals can also be overloaded.

package

In Ada, a package specifies a group of logically related entities, such as types, objects of those types, and subprograms with parameters of those types. It is written as a package declaration and a package body. The package declaration has a visible part, containing the declarations of all entities that can be explicitly used outside the package. It may also have a private part containing structural details that complete the specification of the visible entities, but which are irrelevant to the user of the package. The package body contains implementations of subprograms (and possibly tasks as other packages) that have been specified in the package declaration. A package is one of the kinds of program unit.

parameter

A variable declared in an external function definition, between the function name and the body of the function. In Ada, the *mode* of a parameter, i.e. whether it is an input, an output, or both, is indicated in the functions call.

parent type

See derived type.

pointer

In C/C++, a variable that contains the address of another variable or function. A pointer is declared with the unary asterisk operator. Called **access** type in Ada.

portability

The ease with which a system or component can be transferred from one hardware or software environment to another.

pragma

A pragma is a specific kind of compiler directive.

prefix

A prefix is used as the first part of certain kinds of name. A prefix is either a function call or a name.

preprocessor directives

Lines of text in a C/C++ source file that change the order or manner of subsequent compilation. The control lines are a previous **#define**), **#include** (for inclusion of external source text), **#line** (to specify a line number to the compiler),**#module** (to specify a module name to the linker), and **#if**, **#ifdef**, **#ifndef**, **#else**, and **#endif** (to conditionalize the compilation of the program).

primary expression

An expression that contains only a primary-expression operator, or no operator. Primary expressions include previously declared identifiers, constants, strings, function calls, subscripted expressions, and references to structure or union members.

primary-expression operator

A C/C++ operator that qualifies a primary expression. The set of such operators consists of paired brackets (to enclose a single subscript), paired parentheses (to enclose an argument list or to change the associativity of operators), a period (to qualify a structure or union name with the name of a member), and an arrow (to qualify a structure or union member with a pointer or other address-valued expression).

private part

See package (Ada specific term)

private type

A private type is a type whose structure and set of values are clearly defined, but not directly available to the user of the type. A private type is known only by its discriminants (if any) and by the set of operations defined for it. A private type and its applicable operations are defined in the visible part of a package, or in a generic formal part. Assignment, equality, and inequality are also defined for private types, unless the private type is limited (Ada specific term)

procedure

See *subprogram.*

program

A program is composed of a number of compilation units, one of which is a subprogram called the main program. Execution of the program consists of execution of the main program, which may invoke subprograms declared in the other compilation units of the program.

program unit

In Ada, a program unit is any one of a generic unit, package, subprogram, or task unit.

qualified expression

A qualified expression is an expression preceded by an indication of its type or subtype. Such qualification is used if, in its absence, the expression would be ambiguous (for example as a consequence of overloading).

raising an exception

See *exception.*

range

In Ada, a range is a contiguous set of values of a scalar type. A range is specified by giving the lower and upper bounds for the values. A value in the range is said to belong to the range.

range constraint

A range constraint of a type specifies a range, and thereby determines the subset of the values of the type that belong to the range.

real type

A real type is a type whose values represent approximations to the real numbers. There are two kinds of real type: fixed point types are specified by absolute error bound; floating point types are specified by a relative error bound expressed as a number of significant decimal digits.

record type

In Ada, a value of a record type consists of components which are usually of different types or subtypes. For each component of a record value or record object, the definition of the record type specifies an identifier that uniquely determines the component within the record.

recursion

The process in which a software module calls itself.

relational operator

One of the operators <, >, <=, or >=. In C/C++, the result (type **int**) is 1 or 0, indicating a true or false relation, respectively. The usual arithmetic conversions are performed on the two operands. Relational operators group from left to right.

reliability

The ability of a system or component to perform its required functions under stated conditions for a specified period of time.

renaming declaration

A renaming declaration declares another name for an entity.

rendezvous

rendezvous is the interaction that occurs between two parallel tasks when one task has called an entry of the other task, and a corresponding accept statement is being executed by the other task on behalf of the calling task.

representation clause

In Ada, a representation clause directs the compiler in the selection of the mapping of a type, an object, or a task onto features of the underlying machine that executes a program. In some cases, representation clauses completely specify the mapping; in other cases, they provide criteria for choosing a mapping.

robustness

The capability of the software to survive off-normal or other unanticipated conditions, or the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

satisfy

See *constraint, subtype.*

scalar

A single object (as opposed to *aggregate),* that is, an object or value of a scalar type does not have components. A scalar type is either a discrete type or a real type. The values of a scalar type are ordered.

scope

The portion of a program in which a particular name has meaning. The scope of names declared in external definitions extends from the point of the definition's occurrence to the end of the compilation unit in which it appears. The scope of the names of function parameters is the function itself. The scope of names declared in any block (that is, after the brace beginning any compound statement) is restricted to that block. Names declared in a block supersede any other declaration of the name, including external definitions, for the extent of that block. In C/C++, **struct, union, typedef,** and **enum** tags are identifiers that are subject to the same scope rules as other identifiers. Member names in structure or union references are not subject to the same scope rules (see *uniqueness).* The scope of a *label* is the entire function containing the label.

selected component

In Ada, a selected component is a name consisting of a prefix and of an identifier called the *selector.* Selected components are used to denote record components, entries, and objects designated by access values; they are also used as expanded names.

selector

See *selected component.*

simple name

See *declaration, name.*

shift operator

In C/C++, one of the binary operators < < or > >. Both operands must have integral types. The value of E1<<E2 is El (interpreted as a bit pattern) left-shifted by E2 bits. The value of El >>E2 is El right shifted by E2 bits.

statement

A statement specifies one or more actions to be performed during the execution of a program. Statements include expression statements (an expression followed by a semicolon in most languages), null statements (the semicolon by itself), compound statements (blocks), and an assortment of statements identified by keywords.

storage class

The attribute that, with its type, specifies C's interpretation of an identifier. The storage class determines the location and lifetime of an identifier's storage. Examples are static, external, and **auto.**

string

(1) An array of characters

(2) A constant consisting of a series of ASCII characters enclosed in quotation marks. Such a constant is declared implicitly as an array of char, initialized with the given characters, and terminated by a NULL character (ASCII 0, C escape sequence \0).

structure

In C/C++, an aggregate type consisting of a sequence of named members. Each member may have any type. A structure member may also consist of a specified number of bits, called a field.

subcomponent

A subcomponent is either a component or a component of another subcomponent.

subprogram

In Ada, a subprogram is either a procedure or a function. A procedure specifies a sequence of actions and is invoked by a procedure call statement. A function specifies a sequence of actions and also returns a value called the result, and so a function call is an expression. A subprogram is written as a subprogram declaration, which specifies its name, formal parameters, and (for a function) its result; and a subprogram body which specifies the sequence of actions. The subprogram call specifies the actual parameters that are to be associated with the formal parameters. A subprogram is one of the kinds of program unit.

subtype

A subtype of a type characterizes a subset of the values of the type. The subset is determined by a constraint on the type. Each value in the set of values of a subtype belongs to the subtype and satisfies the constraint determining the subtype.

subunit

See *body*.

symbolic constant

In C/C++, an identifier assigned a constant value by a **#define** directive. A symbolic constant may be used wherever a literal is valid.

task

In Ada, a task operates in parallel with other parts of the program. It is written as a task specification (which specifies the name of the task and the names and formal parameters of its entries), and a task body which defines its execution. A task unit is one of the kinds of program unit. A task type is a type that permits the subsequent declaration of any number of similar tasks of the type. A value of a task type is said to designate a task.

tokens

The fundamental elements making up the text of a C program. Tokens are identifiers, keywords, constants, strings, operators, and other separators. White space (such as spaces, tabs, newlines, and comments) is ignored except where it is necessary to separate tokens.

type

A type characterizes both a set of values, and a set of operations applicable to those values. A type definition is a language construct that defines a type. A particular type is dependent on the language used (e.g. in Ada a type is either an access type, an array type, a private type, a record type, a scalar type, or a task type).

type name

In essence, the declaration of an object of a given type that omits the name of the object.

unary operator

An operator that takes a single operand. In C/C++, some unary operators can be either prefix or postfix. The set includes the asterisk (indirection), ampersand (address of), minus (arithmetic unary minus), exclamation (logical negation), tilde (one's complement), double plus (increment), double minus (decrement), cast (force type conversion), and sizeof (yields size, in bytes, of its operand).

union

In C/C++, an aggregate type which can be considered a structure all of whose members begin at offset 0 from the base and whose size is sufficient to contain any of its members.

uniqueness

A property of the names used for certain structure and union members. A name is unique if either of these conditions is true:

- The name is used only once.
- It is used in two or more different structures (or unions), but each use denotes a member at the same offset from the base and of the same data type.

The significance of uniqueness is that a unique member name can be used to refer to a structure in which the member name was not declared (although a warning message is issued).

use clause

In Ada, a use clause achieves direct visibility of declarations that appear in the visible parts of named packages.

variable

An identifier used as the name of an object (see object).

variant part

A variant part of a record specifies alternative record components, depending on a discriminant of the record. Each value of the discriminant establishes a particular alternative of the variant part.

visibility

At a given point in a program text, the declaration of an entity with a certain identifier is said to be visible if the entity is an acceptable meaning for an occurrence at that point of the identifier. The declaration is visible by selection at the place of the selector in a selected component or at the place of the name in a named association. Otherwise, the declaration is directly visible, that is, if the identifier alone has that meaning.

visible part

See *package.*

with clause

See *compilation unit.*

Appendix D Relationship of Generic Attributes to Other Work

This Appendix compares the attributes defined in Chapter 2 to relevant standards and published research in software safety and quality. As such, it supports the technical basis of the work through the third item defined in Chapter 1 ("A substantive body of knowledge exists and the preponderance of the evidence supports a technical conclusion"). Sections D.1 and D.2 show the relationship among these criteria and IEEE Std 603 and IEC Publication 880, respectively. Section D.3 shows the relationship to IEEE Std 7-4.3.2-1993. Section D.4 compares the attributes to a widely cited software quality framework developed by the U.S. Air Force Rome Laboratory. Finally, section D.5 shows how the work of other researchers in high integrity and safety related software corresponds to the attributes.

D.1 IEEE Standard 603

IEEE Std 603-1991, "IEEE Standard Criteria for Safety Systems for Nuclear Power Generating Stations", is a significant standard for system level safety. In its earlier (1980) version, this standard represents one of the foundations of assessing the safety of I&C systems in general; the 1991 version added items pertinent to digital systems. Currently, the NRC uses Regulatory Guide 1.152, "Criteria for Programmable Digital Computer System Software in Safety-Related Systems of Nuclear Power Plants" and ANSI-ANS-7-4.3.2-1982, "Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations" for guidance when performing reviews of digital systems. The 1993 version of 7-4.3.2, makes that standard a "daughter" to IEEE Std. 603-1991. Thus, the safety criteria defined in Section 5 of IEEE Std. 603 are a basis for assessing digital systems.

Table D-1 compares the top level generic attributes relates to the safety issues identified in IEEE Std. 603-1991. Since each column contains at least one entry, this demonstrates that the top level attributes described in Chapter 2 pertain to safety issues. Because lower level attributes are traceable to the top level attributes shown in the table, all the generic attributes identified in this report can be associated with safety relevant criteria. The detailed entries in the table show that the generic attributes described in Chapter 2 address all safety criteria 603 except the following inapplicable criteria:

- *Equipment qualification:* This is a hardware issue with minor effects on system software.
- *Information displays:* This is a requirements and design issue.
- *Auxiliary features:* This is a requirements and design issue.
- *Multi-unit stations:* This is a requirements and design issue.
- *Human factors considerations:* This is primarily a design issue.

Table D-1 Comparison of Generic Attributes with IEEE Std-603-1991 Criteria

IEEE 603 Criterion	Top Level Generic Attributes				Remarks
	Relia-bility	Robust-ness	Trace-ability	Maintain-ability	
5.1 Single failure	all	all			see note 1
5.2 Completion of protective action	2.1.2	2.2.2, 2.2.3			Control flow, exception handling, error containment
5.3 Quality			all	2.4.1, 2.4.2	Readability, data abstraction
5.5 System Integrity	2.1.3	2.2.3			Timing, error containment
5.6 Independence		2.2.1, 2.2.3			Diversity, error containment
5.7 Test and Calibration			all	see note 2	Instrumentation, data abstraction cohesiveness, malleability
5.9 Control of access				2.4.4	Malleability
5.10 Repair				all	
5.11 Identification			all	2.4.4	Malleability
5.15 Reliability	all				

Notes:

(1) Software can cause single point failures when (a) the program crashes on encountering an unusual data value or control state, and (b) the program returns wrong results under unusual conditions. Safety concerns arising from (a) are minimized when memory utilization, control flow, and timing are predictable as discussed in Section 2.1. Concerns arising from (b) are minimized by controlled of software diversity and exception handling, and by error containment all of which are discussed in Section 2.2

(2) Software testing in the operational environment is required only after changes are made (software does not deteriorate with use). The cited attributes permit isolation of areas affected by changes, and thus permit focusing the test effort on these areas. The presence of the attributes facilitates assessment of the completeness of test and enhances safety.

D.2 IEC Publication 880

Paragraph 5.2 of IEC 880 contains the essential requirements for languages, translators, and other tools. Additional guidance on these subjects (not mandatory) is provided in Appendix D of IEC 880. Table D-2 summarizes relevant provisions from these two sections of the document and shows how they are related to the generic attributes identified in the previous section. Appendix D guidance is denoted by an asterisk (*), and only the priority 1 (highest priority) recommendations are shown. The notation in this table is identical to that used in Table D-1. The following provisions of IEC Document 880 are not addressed by the attributes identified in Chapter 2 of this report:

•*Problem-oriented languages are preferred to machine-oriented ones:* The selection of a development language is the responsibility of the I&C vendor and is not within the scope of an NRC audit.

•*Automated test tools should be available* and *The use of automated tools is recommended:* These are development process issues which are not related to the specific language in which the safety software has been written.

Similar to the preceding subsection, the presence of an entry in each narrow column signifies that the corresponding top level attribute has been found relevant to safety in the IEC document. The presence of an entry for each row in at least one of the narrow columns indicates that the Chapter 2 attributes cover the IEC 880 document concerns.

Table D-2 Relationship between Top Level Generic Attributes and IEC 880 Recommendations

IEC 880 Provision	Top Level Generic Attributes				Remarks
	Reliability	Robustness	Traceability	Maintainability	
A thoroughly tested translator shall exist and be used			all		
The language shall be unambiguously defined. Features with respect to which there may be ambiguities shall not be used.			all	2.4.2, 2.4..5	Data abstraction, portability
The language and its translator should not preclude the use of error-limiting constructs		2.2.2, 2.2..3			Exception handling, error containment
The language and its translator should not preclude the use of Translation-time type checking	2.1.2.6				Data typing
The language and its translator should not preclude the use of Run-time type and array bound-checking, and parameter checking	2.1.2.6				Data typing
Where auxiliary system programs (documentation aids) are used, they should be thoroughly tested.*			all		
The recommendations of Appendix B (structured design, etc.) should be supported*	2.1.2.1			2.4.1	Structure, readability
Run-time exceptions should be raised for exceeding array boundaries, exceeding a declared range, and passing parameters of the wrong type.*	2.1.2.6	2.2.2, 2.2.3			Data typing, exception handling, error containment
The range of each variable should be determinable at translation time.*	2.1.2.6				Data typing
During expression evaluation, no external assignments should be allowed within the scope of the expression.*	2.1.2.9				Separating assignment and evaluation

D.3 IEEE Std 7-4.3.2 1993, Appendix F

Appendix F of IEEE Std-7.4.3.2 (IEEE, 1993) lists items of concern in the identification and resolution of abnormal conditions and events. Most of these concerns relate to requirements, system-level design, hardware design, and software design, and are therefore not within the scope of this document. However, Section F.2.3.5 identifies abnormal conditions and events related to computer code. Table D-3 shows how the attributes support the concerns of Appendix F.

Table D-3 Support Provided by Attributes of Chapter 2 to Items of Concern in ACES Analysis of IEEE 7-4.3.2

Items of Concern in IEEE 7-4.3.2	Support from Attributes of Chapter 2
Evaluate equations, algorithms, and control logic for potential problems, including forgotten cases or steps, duplicate logic, neglect of extreme conditions, unnecessary functions, misinterpretation, missing condition tests, wrong variable checked, incorrect iteration of loop, etc.	Predictability of control flow (2.1.2) and readability (2.4.1)
Confirm correctness of algorithms, accuracy, precision, discontinuities, out of range conditions, breakpoint, erroneous inputs, etc.	Precision and accuracy (2.1.2.7) and all base attributes under robustness (2.2)
Evaluate the data structure and usage in the code to provide adequate confidence that the data items are defined and used properly	Data typing (2.1.2.6)
Provide adequate interface compatibility of software modules with each other and with external hardware and software	Data abstraction (2.4.2) and most base attributes under predictability of control flow (2.1.2)
Provide adequate confidence that the software operates within the constraints imposed upon it by the requirements, design, and the target computer	All base attributes under reliability (2.1)
Examine non-critical code to provide adequate confidence that it does not adversely affect the function of critical software. As a general rule, safety software should be isolated from non-safety software. The intent is to prove that this isolation is complete	Data abstraction (2.4.2)
Provide adequate confidence that the results of coding activities are within timing and sizing constraints	Predictability of memory utilization (2.1.1) and predictability of timing (2.1.3)

Appendix F does not distinguish between system design, software design, and language issues. Therefore a one-to-one correspondence with the attributes defined in Chapter 2 of this report cannot be established. However, that at least one attribute can be associated with each of the concerns indicates that no major area has been overlooked in the generation of the attributes.

D.4. Rome Laboratory Software Quality Framework

The list of Software Quality Factors generated by the Rome Laboratory metrics framework (Bowen, 1985; Wigle, 1985) is widely used, has been continuously updated (Murine, 1994), and is the basis for software metrics evaluation by a consortium that includes large system integrator and defense organizations. It is not restricted to software quality factors that affect safety, and thus its principal value for this study is to serve as a check that the safety oriented selection of attributes in Chapter 2 has not overlooked anything from this broader context that might be relevant to safety. The top level 13 factors have been stable over the last ten years. The relation of these factors to the Chapter 2 attributes is shown in Table D-4.

Table D-4 Chapter 2 Attributes and Factors in the USAF Rome Laboratory Framework

Rome Laboratory Quality Factor	Top Level Generic Attributes				Remarks
	Reliability	Robustness	Traceability	Maintainability	
Reliability	all				
Survivability		all			
Correctness	2.1.2.7		all		Precision and accuracy
Maintainability				all	
Verifiability			all		
Expandability				2.4.3, 2.4.4	Cohesiveness, malleability
Flexibility				2.4.4	Malleability
Portability				2.4.5	Portability (adherence to standards)
Efficiency					Not a safety issue; sufficient resources must be provided by design
Integrity					Defined as access protection; not a language issue
Usability					Defined as not needing training; not a language issue
Interoperability					May conflict with separation requirements of IEEE Std. 603; not a language issue
Reusability					A design rather than a language issue

D.5 Other Published Research

Significant research on relevance of software attributes to system safety has been published by (Leveson, 1992; Turner, 1992; Bullock, 1980; Cuthill, 1993; Andersen, 1984; Petersen, 1984). These references were selected because they span a fairly long time frame (1980 to 1992), are oriented to nuclear safety, and originate from diverse sources (academia, a U.S. national laboratory, a European standards organization, and the U.S. NIST). As in Section 3.4, the references contain a mix of design and implementation issues, with considerable emphasis on the former. Subject to the restrictions imposed by this mismatch, Table D-5 shows that the issues raised in these references have not been overlooked in the attribute structure identified in the Chapter 2.

Table D-5 also demonstrates differences between the approach taken in the generic attributes developed in this work versus that of previous researchers. For example, this report regards quality as a complex attribute including elements of reliability, readability, traceability, and portability (i.e., adherence to standards). Because other researchers were considering a broader range of issues in the system design and development process, they included issues such as fail-safe operation, minimizing critical data and code, and testability. On the other hand, there are areas where there is a close correspondence between this work and others. Attributes which directly correspond include reliability, maintainability, error containment, and diversity.

Table D-5 Relationship between Generic Attributes and Safety Concerns or Criteria Identified by Other Researchers

Author	Criterion or Concern	Corresponding Attributes from Chapter 2
Leveson	Isolation and protection	Robustness (2.2), particularly error containment (2.2.3)
	Minimizing unsafe failure modes	None (design level issues)
	Fail safe design	
	Minimizing safety critical code and data	
Bullock	Accuracy	Precision and accuracy (2.1.2.7), use of compiled libraries (2.3.2), readability (2.4.1)
	Completeness	*(both accuracy and completeness are partially design issues)*
	Understandability	Readability (2.4.1), cohesiveness (2.4.3)
	Maintainability	Maintainability (2.4)
	Testability	Reliability (2.1) maintainability (2.4) *(primarily a design issue)*
	Reliability	Reliability (2.1)
	Comments	Comments (2.4.1.3)
	Modularity	Data abstraction (2.4.2), cohesiveness (2.4.3)
Cuthill	Modularity: Separated execution sequences with limited interaction;	Data abstraction (2.4.2), cohesiveness (2.4.3)
	Functional diversity: Provably separate execution sequences;	Functional diversity (2.2.1)
	Traceability	Traceability (2.3)
	Removal of ambiguity	Reliability (2.1)
Andersen and Petersen	High reliability	Reliability (2.1)
	Safeguards against handling errors:	Exception handling (2.2.2)
	Safeguards against intended misuse:	None *(design issue)*
	Fault Correction, Fail to Safe, Fail to Operational:	Diversity (2.2.1), exception handling (2.2.2)

References

Andersen, O. and P.G. Petersen, *Standards and regulations for software approval and certification,* ElektronikCentralen Report ECR 154 (Denmark), 1984.

Bowen, T.P. and G.B. Wigle and J.T. Tsai, "Specification of Software Quality Attributes" Report, 3 Vols. RADC-TR-85-37, available from NTIS, 1985.

Bullock, J.B., briefing charts contained in Working Group Report on Software Reliability Verification and Validation, *IEEE/NRC Working Conference on Advanced Electrotechnology Applications to Nuclear Power Plants,* IEEE Cat. No.TH0073-7, January, 1980.

Campbell, D. and V. Castellano and O. Cole, et. al., *Ada/6000 Tool Set,* O.C. Systems, Fairfax, Virginia, 1994.

Chillarege, R., "Orthogonal Defect Classification," *IEEE Trans. SW Engineering*, 1992.

Cuthill, B., "Applicability of Object Oriented Design Methods and C++ to Safety Critical Systems," *Proceedings of the Digital System Reliability and Nuclear Safety Workshop,* NUREG CP-0136, NIST SP 500-216, 1993.

Dahl, O.J. and E.W. Dijkstra and C.A.R. Hoare, *Structured Programming,* Academic Press, London and New York, 1972.

Gottfried, R.and D. Naiditch, *Using Ada in Trusted Systems, Proc. of COMPASS 93,* May, 1993, National Institute of Standards and Technology, Washington, DC, 1993.

Henderson, J., "Low level programming," in *Software Engineer's Reference Book,* J.D. McDermid, ed., CRC Press, Inc., 1993.

Institute of Electrical and Electronic Engineering, IEEE Std 100-1977, *IEEE Standard Dictionary of Electrical and Electronic Terms.*

Institute of Electrical and Electronic Engineers, Nuclear Power Engineering Committee, IEEE Std. 603-1991, *IEEE Standard for Nuclear Power Generating Stations.*

Institute of Electrical and Electronic Engineers, IEEE-Std-7 -4.3.2-1993, *IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Station.*

International Electrotechnical Commission (IEC), "Software for Computers in the Safety Systems of Nuclear Power Stations," Standard 880-1986.

Kopetz, H., "Real-time systems," in *Software Engineer's Reference Book,* J.D. McDermid, ed., CRC Press, Inc., Cleveland, Ohio, 1993.

Leveson, N.G. and C.S. Turner, *An Investigation of the Therac-25 Accidents,* University of California, Irvine Technical Report 92-108, Irvine, California, 1992.

Liao, Y., "Requirements for Directed Automatic Instrumentation Generation for Program Monitoring and Measuring," in *IEEE Trans. SW Engineering*, 1991.

McGarry, F., "The Impacts of Software Engineering," briefing presented to the NRC Advisory Committee on Reactor Safeguards (ACRS), August 21, 1992.

Meek, B.L., "Early High-Level languages," in *Software Engineer's Reference Book,* J.D. McDermid, ed., CRC Press, Inc., 1993.

Murine, G.E., "Rome Laboratory Framework Implementation Guidebook", RL-TR-94-149, USAF Rome Laboratory, March 1994.

Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM,* Vol. 15, No. 12, 1972.

Parnas, D.L. and A.J. van Schowen and S.P. Kwan, "Evaluation of Safety Critical Software," *Communications of the ACM,* Vol. 33, No. 6, p. 636, June, 1990.

Royce, W.W., written comments in *Proceedings of the Digital Systems Reliability and Nuclear Safety Workshop,* NUREG/CP-0136, NIST SP 500-216, 1993.

Smith, D.J. and K.B. Wood, *Engineering Quality Software: A review of Current Practices, Standards, and Guidelines Including New Methods and Development Tools.* New York: Elsevier Applied Sciences, 1989.

Thayer, R., "Software Reliability Study," Rome Air Development Center report RADC TR 76-238, March, 1976.

U.S. Department of Defense, "Weapon System Software Development," MIL-Std-1679 (Navy), 1978.

U.S. Department of Defense, DoD-Std-2167A, *Software Development Standard,* Appendix C, 1986.

U.S. Department of Defense, DoD Std 2167A, *Software Development Standard,* Appendix D, Washington, D.C.

Witt, B.I. and F.T. Baker and W.W. Merritt, *Software Architecture and Design.* Van Nostrand Reinhold, New York, 1994.

Appendix E Backgrounds of Subject Matter Experts and Reviewers

This Appendix provides brief descriptions of the backgrounds and qualifications of the 18 subject matter experts who provided substantial technical input or reviews to this document. As the Principal Investigator, Dr. Herbert Hecht's background appears first. All other participants are listed in alphabetical order.

1 Herbert Hecht,

Program Manager
PLC Subject Matter Expert

Ph. D.,	University of California, Los Angeles, 1967
MEE	Polytechnic Institute of Brooklyn, 1949
BEE	College of the City of New York, BEE, 1944

Dr. Herbert Hecht has been involved with software issues associated with critical real time control systems since his work on the Titan II Intercontinental Ballistic Missile guidance system in the middle 1960's. Dr. Hecht's involvement in safety systems for nuclear power plants began in 1980 with his participation in the conference on the application of advanced electrotechnology application to nuclear power plants. Since that time, he has participated in audits of digital safety systems including the Arkansas Nuclear One Core Protection Calculator System, the South Texas Utilities Qualified Parameter Display System, and several others. In addition, he has been the Principal Investigator of NRC sponsored research on verification and validation guidelines for high integrity systems (NUREG/CR-6293) and on earlier work on high integrity systems (NUREG/CR-6113). Dr. Hecht 's background in safety critical application of programmable logic controllers includes design and development of ladder logic code for short range ground transportation systems. The first application of this software was the Big Thunder Mountain ride at Disneyland; additional applications are at Houston Intercontinental and other airport people mover systems. His previous experience includes work on ladder logic diagrams in aircraft avionics and flight control systems while at Sperry Corporation.

2 Derek Decker

Task 3 Reviewer for PLC Ladder Logic and IEC 1131 Sequential Functional Charts

Mr. Decker is an expert in PLC programming and I&C systems integration. As a PLC application engineer for Texas Instruments, he installed and programmed PLCs in a wide variety of systems, both for in-plant use and for customers. Examples include wave solder machines and blood sterilization controls. At Telemecanique, Inc., he was Technical Services Manager serving as the

recognized expert in North American on the usage of advanced PLCs—the first to use what is now recognized as the IEC 1131-3 PLC language suite. Using these PLCs, he developed applications for manufacturing and food processing. He also prepared application notes and courses on the Telemecanique product line. He has written articles for *Control Engineering* and *PLC Insider.*

3 Stephen Graff

Task 2 Subject Matter Expert for Ada, Pascal

M.S., Systems and Control Theory, UCLA, 1973
B.S., Electrical Engineering, University of Maryland, 1969

Steve Graff has experience in real time aerospace computer applications ranging from fighter aircraft to the Galileo and Ulysses space probes. Mr. Graff supported research in software complexity metrics and developed routines to analyze complex metrics such as Tsai. and dataflow. His experience in high integrity systems includes the Oceanic Advanced Automation System (AOAS), the F14 and F15 fighter avionics controls (in Ada), real time spacecraft ground control systems. and classified applications. Mr. Graff's experience in Pascal includes teaching at the university senior and post graduate level where he was responsible for creating and evaluating programming for classic computer science problems such as linked lists, trees, graphs, and double linked lists. Mr. Graff's expertise in other languages provides the project with additional depth and also provides a better perspective from which to judge the relative strengths and weaknesses of PL/1 and Pascal.

4 William Green

M.S., Astronomy, San Diego State University, 1975
B.A., Astronomy, University of Minnesota, 1965

Mr. Green has a total of 22 years' experience as a programmer, with 10 years' experience in Ada software development. He has designed, coded, modified, and tested programs on Defense Satellite Program, MILSTAR and other projects in Ada and FORTRAN. This work involved real time ground-based satellite attitude control. In addition to his real-time background, Mr. Green has written Ada syntactical and lexical analysis programs in support of the development of software tools for the measurement of software complexity metrics. Mr. Green also has an extensive background in system analysis and software testing in satellite ground support subsystems in Ada and other languages. This work includes writing, analysis, and criticism of requirements and design documents, design plans, test plans, and test procedures.

5 Myron Hecht,

Assistant Project Manager, Report Editor
Task 1 Pascal Subject Matter Expert

M.S.,	University of California, Los Angeles, Nuclear Engineering, 1976
M.B.A.,	University of California, Los Angeles, Computers & Information Systems, 1982
B.S.,	University of California, Los Angeles, Chemistry (Cum Laude), 1975

Myron Hecht has 20 years' software development experience in real time, scientific, and high integrity software programming. He has previously worked on NRC-sponsored research on design and verification and validation guidelines for high integrity software. He has ten years' experience supporting the FAA in air traffic control software development in 6 different computer languages (including Pascal). In this capacity, he analyzed more than ten thousand failure reports and identified trends and software development practices which negatively affect stability and reliability. Previously, he directed software development for a fault tolerant distributed control system implemented in C for the EBR II site. He has performed several studies analyzing software fault distributions on the basis of error reports generated by NASA/JPL and large aircraft development organizations. He has also investigated the improvement of software complexity metrics to predict software failure densities in Ada avionics software as part of a Phase I SBIR (now in Phase II for the U.S. Air Force). In earlier work, he developed and demonstrated the feasibility of fault-tree based design methodologies for fault tolerant software (SIFT and FTMP). Mr. Hecht received his graduate training in nuclear engineering and began his career in nuclear nonproliferation and environmental analyses. In that capacity, he has programmed extensively in PL/1, Pascal, and FORTRAN.

6 Michael Justice

Task 3 Reviewer for PL/M

B.S., Computer / Electrical Engineering, University of Illinois, 1975

Michael Justice has worked in industrial automation and process control for Amoco Oil, Intel, Wizdom Systems, and Synergetic. His experience includes real-time control software, communications, real-time operating systems, and hardware device drivers. His experience in PL/M includes:

- Support of the language as a software specialist and consulting engineer at Intel,
- Heading the development of a successful line of PC-based PLCs implemented in PL/M,

- Serving as a technical consultant in his current position as Vice President at Synergetic Micro Systems, an engineering services firm serving major electronic and industrial manufacturers in the Mid-west.

Experience with other languages includes C, and Assembly Languages on single board computers (SBC) and microprocessors in all major buses (PC, STD, VME, MULTIBUS) and manufacturers (Intel, NEC, Motorola).

7 Shlomo Koch

Task 2 PLC Ladder Logic and Sequential Functional Chart Subject Matter Expert

Ph.D.	Electrical Engineering Rensselaer Polytechnic Institute (RPI), Troy, NY. 1992.
M.Sc.	Electrical Engineering Technion - Israel Institute of Technology, 1978.
B.Sc.	Electrical Engineering Technion - Israel Institute of Technology, 1973.

Dr. Koch has extensive experience developing safety critical applications for PLC systems. During the last six years, he was responsible for the development and implementation of computer-based systems for safety-related applications in the nuclear industry including:

•a PLC-based load sequencer for Northern States Power (NSP), Prairie Island,

•a containment isolation status system for Tennessee Valley Authority (TVA), Browns Ferry, and

•a study for implementing a PLC-based reactor protection system for NSP.

Two of these systems are now licensed and operational. Dr. Koch has also worked on safety critical applications outside the nuclear industry, such as the pharmaceutical industry and medical devices that are regulated by the FDA that requires product validation, and the chemical and petrochemical industry that is regulated by OSHA that requires shutdown systems. Prior to obtaining his Ph.D, Dr. Koch developed safety critical systems for 9 real time microprocessor-based defense systems. He is well versed in hazard analysis using Mil-Std-882B and MOD-56. Dr. Koch's experience with PLCs extends beyond safety critical systems. He has designed and programming of PLC-based systems for the local industry that includes paper mills, water treatment, machinery control, drive control, and sequencing logic, Dr. Koch has obtained national recognition through his numerous publications and standards activities. He is a member of the IEEE 7-4.3.2 standard committee, "application criteria for programmable digital computer systems in safety systems of nuclear power generating stations". He is also a member of the ISA SP84 standard committee on "application of PES in safety systems for the process industry". He is a regular participant and speaker at EPRI, NUSMG, IEEE, ACM and other technical conferences on nuclear I&C system digital upgrades, software V&V and regulatory requirements.

8 James Leivo

Task 1 Nuclear Systems Consultant

B.S. Electrical Engineering, Carnegie Mellon University, 1966

James Leivo is a registered Professional Engineer with over 25 years' experience in the nuclear power industry and related areas. His past work includes technical direction of the design and retrofit of I&C and computer systems and project management while employed at Westinghouse, NUS Corporation, and Los Alamos Technical Associates. Mr Leivo has served as a Consultant to NRC Instrumentation and Control Systems Branch, performed technical/ safety evaluations of computer-based reactor protection and safety instrumentation systems for advanced LWR designs and operating LWR upgrades. For nuclear utilities, Mr. Leivo has provided consulting services for independent assessment of safety and non-safety related I&C systems, electrical systems, and computer systems. This work has included thread audits and hazard analyses of safety and non-safety systems being retrofit into nuclear power plants.

9 Don Lin

Task 2 C/C++ Subject Matter Expert

Ph. D. Computer Engineering, University of Michigan, Ann Arbor, MI, 1988
M.S.E. Computer, Information, and Control Engineering, University of Michigan Ann Arbor, MI, 1985
B. S. Electrical Engineering Beijing Normal University, Beijing, 1982

Dr. Lin's experience in high integrity software comes from his extensive experience in implantable medical devices, medical instrumentation design, and patient care devices. He has both developed software and managed software development teams for these devices using C, C++ and Assembly language. He also has experience in the testing and certification requirements of high integrity software through the premarket licensing process of the FDA. As part of his work on medical instrumentation, Dr. Lin has developed expertise in high performance computer system design, digital signal processing and pattern recognition, real and protected mode programming, software version control, clinical trial and data collection. Dr. Lin has also developed printer drivers and barcode readers. The integrity of these devices are of importance in medicine because of the life critical decisions which are made on the basis of printed output. Dr. Lin's abilities have been recognized by numerous awards in both his native country (China) and in the U.S., He holds two patents for medical instrumentation. Dr. Lin's familiarity with C and C++ comes from his work on a variety of operating systems, microprocessors, and compilers. His knowledge of potential problems and pitfalls comes from the extensive testing required for his software and devices in the medical field.

10 Kamran Ossia

Task 3 Reviewer for C and C++

Ph.D.	Electrical Engineering, University of Toronto, 1989
M.A.Sc.	Electrical Engineering, University of Toronto, 1983
B.Sc.	Electrical Engineering, Arya-Mehr (currently Sharif) University of Technology, Tehran, 1979

Kamran Ossia has extensive experience in software development for scientific and nuclear applications. For his Ph.D. dissertation he developed a Matlab package for digital control system design and analysis. From 1989 to 1995 he was with Atomic Energy of Canada as control system designer, safety system analyst, and senior design engineer where took part in design and verification of reactor shutdown system software, control room user interface design, feasibility study of multiplexing signals inside the reactor building, analysis of nuclear reactor shutdown systems, updating of nuclear reactor simulation programs, design of a reactor regulating system on a distributed control system, and optimization of the flux detector layout for nuclear reactors. Dr. Ossia has collaborated in publications on stability analysis and control of mechanical systems, including missiles, rotating beams, gyros and columns.

11 Jeremy Pollard

Task 3 Reviewer for PLC Ladder Logic and IEC 1131 Sequential Function Charts

Jeremy Pollard is the author of a monthly newsletter on PLC programming, and has been responsible for the teaching of more than 1000 individuals on Allen Bradley equipment. He established a large Allen Bradley training center in Toronto for that leading manufacturer of controllers. He has assisted other organizations such as Flexis and TopDOC in developing PC-based PLC control systems, and developed the control algorithms and supervised implementation of a control system at Corning Glass Works. In addition to his instruction and consulting on behalf of Allen Bradley, Mr. Pollard has initiated PLC training at a local college which resulted in a significant increase in student attendance and revenue. Mr. Pollard publishes regularly in *Control Engineering* magazine, and has published in other trade journals as well.

12 Bo Sanden

Task 3 Reviewer for Ada

Ph.D.,	Computer Science, Royal Institute of Technology, Stockholm, 1978
M.S and B.S.,	Engineering Physics (combined), Lund Institute of Technology, Sweden, 1970

Dr. Sanden is an Associate Professor, ISSE Department, George Mason University. His Research areas are in concurrency, use of Ada, course work and thesis direction in Ada, real time software design, program design, compiler design, software engineering. Prior to entering University faculty positions, he was technical project manager of a distributed transaction processing system. This high integrity system included transaction scheduling, recovery, restart mechanisms constructed by Dr. Sanden. In other language work Dr. Sanden was analyst, designer, and assembly programming consultant on a high performance JSP compiler. Dr. Sanden's recognition in software development includes being appointed to develop the curriculum for the newly established Masters program in Software Systems Engineering at George Mason University. He is the author of 23 referred papers and books. One of these books on Ada is now being used as a text at GMU and many other universities. He has authored 4 papers other refereed publications on Ada. His dissertation research was on restarting of real time systems

13 Eltefaat Shokri

Task 3 Ada and C/C++ Reviewer

Ph.D	Electrical & Computer Engineering, University of California, Irvine, 1993
M.S.	Computer Science, Sharif University of Technology, Tehran, 1983
B.S.	(*cum laude*) in Computer Science, Meshad University, Meshad (Iran), 1980

Eltefaat Shokri has expertise in distributed object-oriented real time systems. Prior to performing his dissertation research in this area, Dr. Shokri was a lecturer in computer languages at Meshad University. While engaged in post-doctoral research at the University of California, Irvine, he developed DREAM, a real-time, object-oriented kernel for fault tolerant distributed systems in C and C++. Dr. Shokri is now developing a library of reusable software components for distributed systems implemented in Ada-95 for the U.S. Air Force Rome Laboratory, and is also developing a library to support adaptive fault tolerance for extended space missions for NASA/JPL.

14 Arthur Sorkin

Ph.D.,	Computer Science, University of California Los Angeles, 1977
Ph.D.,	Computer Science, University of California San Diego, 1971

Task 3 C/C++, Pascal, and PL/M reviewer

Dr. Arthur Sorkin has extensive experience in writing compilers and instruction in multiple computer languages. He was the Pascal compiler manager at Gould and author of language reference manual. He was Project manager, PLM/S86 cross compilation system for IBM 370s, and designed and implemented the syntax and semantic checker and error recovery routines for that compiler. He managed the compiler and utility group for the Vitesse mini-supercomputer company, and was responsible for porting assembler, loader, and debugger to AIX on IBM 370 mainframes. Dr. Sorkin was recipient of an IBM Doctoral fellowship and was appointed Visiting Associate Professor, U.C. Davis; joint appointment with Lawrence Livermore National Laboratory. Dr. Sorkin's work in high integrity systems includes performed research in network computer security at Lawrence Livermore. He also automated portions of a clinical laboratory automation system, and developed Antisubmarine warfare software. He is the author of 12 refereed publications.

15 Ann Tai

Ph.D.,	Computer Science, University of California, Los Angeles, 1992
M.S.,	Computer Science, University of California, Los Angeles, 1986
B.S.,	Mathematics/Computer Science, University of California, Los Angeles, 1984

Task 2 Subject Matter Expert for C/C++

Dr. Tai has experience programming in real time systems for C. In addition, she has performed research in verification and validation and modeling for dependable systems. Dr. Tai participated in the NUREG CR 6113 preliminary language study which involved analyzing Ada, C, C++, and PL/M and developing performance benchmarks. Her other work includes reliability modeling, performability modeling (the integration of reliability, fault tolerance, and performance), and has developed high integrity software in C under SoHaR's SBIR contract with the U.S. Department of Energy for hierarchical distributed fault tolerant reactor control. Dr. Tai also developed of a methodology for verification of critical software based on the integration of functional testing, structural testing, and fault trees. Implemented tool written in Pascal. She participated in earlier NRC sponsored work on Development of guidelines for development and licensing of software used in Class 1E reactor safety systems. Dr. Tai previously employed at JPL where she programmed and analyzed the Realtime Weather Processor. She was also on the Computer Science Faculty of the University of Texas at Dallas for one year.

16 K.S. Tso

Task 3 Review for Ada

Ph.D.	Computer Science, University of California, Los Angeles, March 1987,
M.S.	Electronic Engineering. Philips International Institute, the Netherlands, June 1981,
B.S.	Electronics The Chinese University of Hong Kong, June 1979,

Dr. Tso has more than 16 years' experience programming real time and high integrity software in C, Ada and Assembly. He is currently working on two high integrity R&D projects: a fault tolerant robotic control system which will have a recovery time of less than 40 msec. The initial application of the controller will be a spaceborne inspection system which continuously scans the outside of a large spacecraft for meteorite and other damage. The second project is the development of Ada fault tolerant software components. This contract was awarded in the competitive Small Business Innovative Research program. Dr. Tso successfully developed reusable software components which could be integrated on a network of UNIX workstations to create a fault tolerant radar processing application. Continued work including fault injection testing, validation, and documentation is now in progress under a Phase II SBIR contract. Dr. Tso has developed extensive language expertise through earlier projects with SoHaR in which he created parsers for the C and Ada programming languages. These parsers were the bases of tools used to create conditional tables, which serve as test specifications for high integrity software, and for the analysis of Ada source code to analyze metrics such as Halstead, McCabe, and modified metrics to account for the real-time multitasking properties of Ada (this work was done jointly with M. Hecht). In earlier work on fault tolerance, Dr. Tso developed the DEDIX test bed which was used for evaluation of multiversion software fault tolerance. MVS fault tolerance includes the development of the same application using diverse languages but a single specification. Prior to engaging in research on fault tolerance, Dr. Tso performed research in networking, and worked as an engineer at an electronics firm in Hong Kong.

17 Douglas Wendelboe

Task 2 Subject Matter Expert for PL/M

B.S., Electrical Engineering, Pennsylvania State University, 1972

Douglas Wendelboe is active in the design of microprocessor-based products and instrumentation. He has worked on all major Intel microcontrollers and microprocessors, and has also developed systems on the Motorola 68HC05 and 68HC11 families. He has developed software in PL/M, C, C++, and Assembler. Significant real-time control software projects include medical pacing systems analyzers, in-circuit emulators, meat packing weighing systems, injection mold temperature controllers, blood analyzers, mine shovel weighing and monitoring systems, vehicle inventory systems, and immunology software cartridges. Mr. Wendelboe has also been involved in hardware design and test system development. Prior to founding his own company in 1981, Mr. Wendelboe was employed at Microchip Technology, Kroy Inc., IBM, Honeywell, and Unisys.