सत्यमेव जयते

भारत सरकार
GOVERNMENT OF INDIA
भाभा परमाणु अनुसंधान केन्द्र
BHABHA ATOMIC RESEARCH CENTRE

PROGRAMMING GUIDELINES FOR COMPUTER SYSTEMS OF NPPs

by
R. M. Suresh babu and U. Mahapatra
Control Instrumentation Division

1999

BARC/1999/E/021

BARC/1999/E/021

GOVERNMENT OF INDIA
ATOMIC ENERGY COMMISSION

PROGRAMMING GUIDELINES FOR COMPUTER SYSTEMS OF NPPs

by
R.M. Suresh babu and U. Mahapatra
Control Instrumentation Division

BHABHA ATOMIC RESEARCH CENTRE
MUMBAI, INDIA
1999

BIBLIOGRAPHIC DESCRIPTION SHEET FOR TECHNICAL REPORT
(as per IS : 9400 - 1980)

01	*Security classification :*	Unclassified
02	*Distribution :*	External
03	*Report status :*	New
04	*Series :*	BARC External
05	*Report type :*	Technical Report
06	*Report No. :*	BARC/1999/E/021
07	*Part No. or Volume No. :*	
08	*Contract No. :*	
10	*Title and subtitle :*	Programming guidelines for computer systems of NPPs
11	*Collation :*	30.p.
13	*Project No. :*	
20	*Personal author(s) :*	R.M. Suresh babu; U. Mahapatra
21	*Affiliation of author(s) :*	Control Instrumentation Division, Bhabha Atomic Research Centre, Mumbai
22	*Corporate author(s) :*	Bhabha Atomic Research Centre, Mumbai - 400 085
23	*Originating unit :*	Control Instrumentation Division, BARC, Mumbai
24	*Sponsor(s) Name :* *Type :*	Department of Atomic Energy Government

Contd... (ii)

30	*Date of submission :*	August 1999
31	*Publication/Issue date :*	September 1999
40	*Publisher/Distributor :*	Head, Library and Information Services Division, Bhabha Atomic Research Centre, Mumbai
42	*Form of distribution :*	Hard copy
50	*Language of text :*	English
51	*Language of summary :*	English
52	*No. of references :*	
53	*Gives data on :*	
60	*Abstract :* Software quality is assured by systematic development and adherence to established standards. All national and international software quality standards have made it mandatory for the software development organisation to produce programming guidelines as part of software documentation. This document contains a set of programming guidelines for detailed design and coding phases of software development cycle. These guidelines help to improve software quality by increasing visibility, verifiability, testability and maintainability. This can be used organisation-wide for various computer systems being developed for our NPPs. This also serves as a guide for reviewers.	
70	*Keywords/Descriptors :* NUCLEAR POWER PLANTS; ALGORITHMS; QUALITY ASSURANCE; VALIDATION; RECOMMENDATIONS; FLOWSHEETS; COMPUTER CODES; DOCUMENTATION; DATA; RELIABILITY	
71	*INIS Subject Category* :	E2200
99	*Supplementary elements :*	

Abstract:

Software quality is assured by systematic development and adherence to established standards. All national and international software quality standards have made it mandatory for the software development organisation to produce *programming guidelines* as part of software documentation. This document contains a set of programming guidelines for detailed design and coding phases of software development cycle. These guidelines help to improve software quality by increasing visibility, verifiability, testability and maintainability. This can be used organisation-wide for various computer systems beings developed for our NPPs. This also serves as a guide for reviewers.

Keywords:
Software quality, visibility, software life cycle, detailed design, software coding

Foreword

Software quality is defined as the degree to which it possesses the desired combination of attributes. Two of the most important attributes are reliability and correctness. Reliability determines the number of failures per unit time. Correctness determines whether the software does all the specified functions *and does not do* any unspecified functions. While reliability and correctness are the attributes we are most concerned about, we should also carefully examine other important attributes such as visibility, verifiability, testability, and maintainability. Of these, visibility and verifiability are very crucial as they determine our ability to verify correctness of a given software; hence the trustworthiness of the software.

At present, formal software verification methods have very limited application. Hence, the software is usually verified by semi-formal or informal methods. In the absence of formal methods, a software difficult to understand and comprehend could lead to a lot of 'holes' in its verification process. Besides, this could make the software difficult to test and maintain.

This guideline is suggested with an intention to increase visibility, verifiability, testability, and maintainability of software being developed for NPP control applications. It puts forth a set of general and specific guidelines for detailed design and coding of software. Compilation references are from various international standards, guides, and publications including that of IAEA, ESA, IEC, and IEEE. Suggestions from some of the groups in BARC and NPC have also been included.

This guidelines would also help reviewers and software quality assurance personnel to assess the software under review.

It is hoped that organisation-wide usage of this guidelines could result in uniformity in software coding style, which would encourage sharing and reuse of high-quality software.

[illegible]

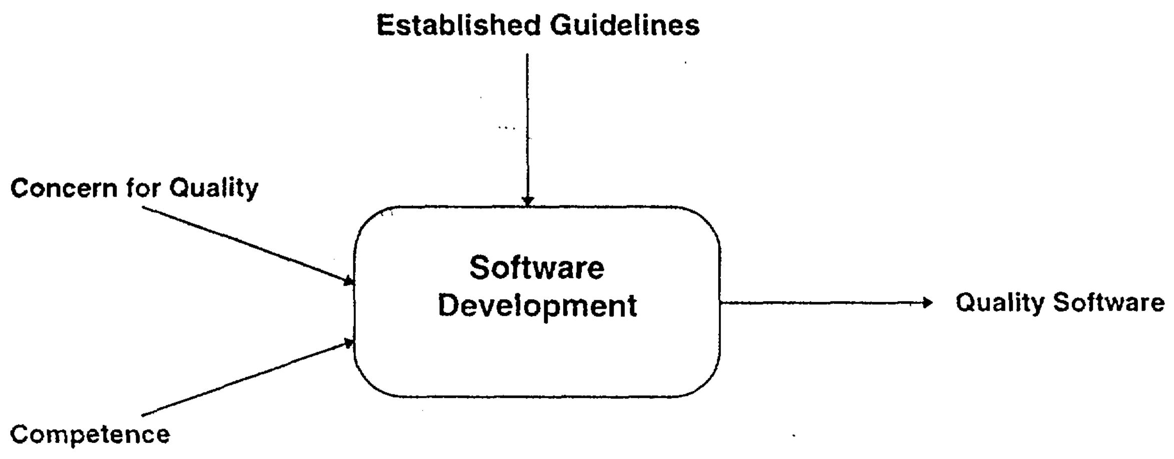
G.P. Srivastava
Head, Control Instrumentation Division

CONTENTS

Programming Guidelines

for

Computer Systems of NPPs



1. Introduction

1.1 Scope

This document has been prepared for software of computer systems used in NPPs. These guidelines are primarily intended for safety-critical and safety-related systems of NPPs. For non-safety systems some of the recommendations may be relaxed.
This document may be used by software developers during detailed design and coding phases. This may also be used in process specifications during software requirements analysis[10].
Technical reviewers may use this document to assess quality of detailed design and source code of a software.

1.2 Purpose

Software coding is basically an art. This allows a high level of freedom in the way programs are written. This unrestricted freedom often makes the software very difficult for another person to understand.
Software engineering involves methods that make the software easy to understand and comprehend; thus making the software verification & validation (V&V) more effective. Only an effective V&V process can *ensure* software correctness. This is especially crucial for highly dependable software such as that used in NPPs.
This means that specific recommendations and guidelines shall restrict the coding process - and in general, the whole software development - with a view to increase software quality factors such as, visibility, comprehensibility, testability, etc. These guidelines shall provide directions for uniform naming conventions and commentary, procedure headings, secure programming language subsets, etc.
All national and international standards have made it mandatory for the software development organisation to produce *programming guidelines* as part of software documentation. This includes general software standards such as ESA[1] and IEEE[2], and safety guides for NPP computer systems, such as IAEA[5] and AERB[4].
The purpose of this document is to establish a set of programming guidelines, which this organisation can use for various computer systems beings developed for our NPPs. This would serve as one of the mandatory documents where the project shall comply with ESA, IEEE, IAEA, or AERB standards/guides, as mentioned above.
In addition to the benefit described above, organisation-wide usage of this document would result in uniformity in software coding style and would encourage sharing and reuse of *high-quality* software.
Section-5 of this document contains description of a program design language (PDL) that can be used in detailed design of software. This section is not a mandatory part of the guidelines; however organisation-wide usage of the PDL would be beneficial as explained above.

1.3 Acronyms

OS	: Operating System

OOD : Object Oriented Design
PDL : Program Design Language
SADT : Structured Analysis and Design Technique

1.4 Definitions

Module: A module consists of (i) a data(or object) and associated interface procedures that operate on the data; or (ii) a collection of procedures associated with a specific function.
Package: Package is a collection of related software modules. A package may have different forms: executable program, library, DLL, etc.
Procedure: Lowest level program entity: a sub-routine.

1.5 References

1. ESA-PSS-05-11 (Issue 1 July '93) - "The software quality assurance plan", ESA, France.
2. IEEE Std. 983-1986 - "IEEE Guide Software Quality Assurance Planning", IEEE press.
3. IEC-880-1986, "Software for computers in safety systems of NPPs".
4. AERB-SG-D25 Draft R0-Mar'1998, "Design Guide on Computer Based Systems".
5. 50-SG-D8,"IAEA Safety Guide for Computer Based Systems Important to Safety".
6. IAEA Tech. Report#367-1994,"Software Important to Safety in Nuclear Power Plants".
7. IEEE Std. 990-1987, "IEEE Recommended Practice for Ada As a Program Design Language".
8. R.S Pressman, "Software Engineering", McGraw-Hill, New York, 1992.
9. "Flow charts versus PDL", Comm. of ACM, 26, 1980.
10. V.A Downes S.J Goldsack, "Programming Embedded Systems with Ada", Prentice-Hall , 1982.
11. DJ Hately, IA Pirbhai, "Strategies For Real-time System Specification", Dorset House Publishing, New York,1987.
12. ESA-PSS-05-05 (Issue 1 May'92) - "Guide to the Software Detailed Design and Production Phase", ESA, France.
13. B.W. Kernigham, "Programming Style: Examples and Counter Examples", Comput. Surveys, 6, 4, Dec'74.
14. N. Wirth, "The composition of a Well-structured Program", Comput. Surveys, 6,4, Dec'74.
15. M. Yohe, "Overview of Programming Practices", Comput. Surveys, 6,4, Dec'74.
16. "Dependability of Critical Computer Systems", Vol.2, EWICS, Elsevier Applied Science.
17. "Methodology for Review of On-line Computer Based Systems of Kaiga-1&2 and RAPP-3&4, Dec'97, Issued by WG-16, PDSC-Kaiga1&2/RAPP3&4.
18. "Developing Highly Reliable Software", IEEE Micro, 17,5, Sept'97.
19. ML Shooman, "Software Engineering: Design, Reliability, and Management".

1.6 The Guidelines: Construction

Inputs to this guideline have come from all the documents, technical papers, and standards listed in section 1.5. Additionally, our own long experience in developing safety-critical software - some of the results of which can be found in [18] - has enabled us to understand many subtle points of software coding. These guidelines have been derived with a view to produce programs that are transparent, easy to comprehend and understand, and easy to verify.
Since no specific programming language is assumed in this guideline, it is independent of the software development methodology/environment. Some languages directly support some of the

desired features specified in the guidelines, while in other cases the programmer would have to make conscious effort to comply with the guidelines.

1.7 The Guidelines: Usage

The guideline is broadly categorised into two: general guidelines and specific guidelines. Each of these is separately given for detailed design and coding.
General guidelines provide general aspects of programming; this provides a frame work and tries to point out good and bad programming practices. This covers all aspects of coding; the programmer should understand and imbibe the philosophy and try to follow the same. Reviewer may find it not easy to assess compliance of a given program with this part of the guidelines since these give only directions and, in general, there will always be some deviations.
Specific guidelines focus on specific design/coding issues including correct usage of control and data structures. It also provides specific recommendations for naming, code layout, headers, etc. *Recommendations are mandatory practices that shall be followed as per this guideline.* Reviewer may use these to mark specific deviations from this guideline.
Section 5 contains recommended Program Design Language to be used in detailed design. This is not mandatory part of this guideline.
In the guidelines the word *shall* identify mandatory material and the word *should* and *may* identify desirable/advisory material.
If this guideline is used, it is mandatory to document all the deviations, if any, from that specified in this document with reason for the deviations.
At last, it must be mentioned, that it is not *enough* to follow these guidelines to produce good software. Since the coding is an art, the programmer should develop own styles that imbibe the spirit of these guidelines.

2. General Guidelines

2.1 General Guidelines for Detailed Design

1. Modules and procedures shall have functional cohesion, i.e. each module/procedure should correspond to a unique function.
2. The interfaces between modules should be simple and uniform throughout the system.
3. It shall be very clear as to which parameters are only used, only defined, or modified.

4. The detailed design shall use a clearly specified program design language (PDL) and it shall be consistently used throughout detailed design.
   Hint: One may use any programming language as PDL, or a specifically defined PDL as the one given in this document.
5. The PDL should be amenable to different implementation languages.

6. Detailed design should show major control structures/control flows/data structures abstracting out the trivial and well-understood ones. It should not be a replication of the final coded software.
   Reason: The objective of detailed design is to show the procedural aspects of the modules, and the data structures; and not details of how to implement these.
   Example: Copying a string or a block is a well-understood construct. Instead of a loop showing copying of each byte, it may be simply written, "*copy the string/block*".
7. Local variables should be used only when it is necessary.
   Reason: In general, usage of local variables implies *going into details* as given in (6) above.

8. Design data correctly and judiciously to reduce program complexity.
   Reason: Many a time the program becomes complex because of poorly designed data.
9. Data hiding should be used throughout.
10. In case of OOD, multiple inheritance shall be used only when it is absolutely necessary.

11. Modules, procedures, data structures, variables and all such entities shall have meaningful names reflecting their purpose.

2.2 General Guidelines for Coding

1. At the lowest level, individual statement and small group of statements should be expressed so they read clearly.
2. In general, an expression should not exceed one line.
3. Expression may be parenthesised to avoid ambiguity.
4. If mixed *types* of variables/constants are used in a statement, explicit type-casting shall be used.
5. Bulky computational expressions of array indexes shall be avoided.

6. The code structure should read clearly. Use a few well-defined control primitives: condition (IF-THEN-ELSE), loop (DO-WHILE), CASE, and grouping (procedures). GOTO's shall not be used.
7. Macros and procedures should be used so that program structure stands out very clearly.
8. Nested macros shall be avoided.
9. Recursive control structures should be avoided. Use them only if data-structure is recursive or if it improves program clarity.

10. Data hiding should be used in all modules and procedures.
11. All variables shall be initialised before use.

12. Programming tricks shall be avoided.
13. Library functions should be used wherever possible; rather than developing *everything*.
14. Dynamic variables should be avoided.

15. Make programs clear and correct, and then if needed make them faster. If algorithm is already the 'best', *measure* the program and improve only the critical part and leave the rest alone.
16. *Example*: If a procedure contains iteration (say, 100 times), a small change within the loop may significantly reduce the procedure execution time.
17. Clarity shall not be sacrificed for small gains in efficiency.
18. As far as possible optimisation should be left to the compiler.
19. Run-time dependency on input parameters shall be avoided; where not possible, it shall be documented.

20. The program should be robust: it should defend itself against bad external data.
21. Effort should be made to anticipate possible changes, and to make program flexible enough to accommodate such changes easily.
    Example: If size of an array holding a name is set to a constant at each instances of its declaration, it would necessitate changes in all these places when names become longer. This may be done easily if the array size is parametric or is declared as a pseudo.

3. Specific Guidelines for Detailed Design

3.1 Naming

Refer to the naming convention in section 4, specific guidelines for coding.

3.2 Verification Information

1. Each procedure or interface function of a module should have pre-conditions, post-conditions, and any other assertions.
   Reason: Pre-condition shall be satisfied to execute a procedure. Post-condition should be satisfied if the pre-condition was satisfied *and* the procedure executed *correctly*. Assertions are logical conditions that should always be true at a given point of program execution. These details make the verification and testing easier.

3.3 Program Design Language (PDL)

Given below are a few characteristics of a PDL that is used for detailed design. For a detailed description refer [7].

1. The PDL should support more than one design methodology. The PDL should support abstraction, decomposition (of program and data), information hiding, and stepwise refinement.
   Reason: No single methodology is known to be superior for all applications and for all development organisation.
2. The PDL should support algorithm design using structured control constructs.
3. The PDL should support data structure definition, scope, identification of constraints on the values and allowable operations, and modification of data objects.
4. The PDL should support ordering of component execution including initiation, termination and execution of sequential and parallel process.
5. The PDL should support data definition dependency and data flow
6. The PDL should support definition of possible error conditions and their consequences
7. The PDL should support definition of asynchronous interrupts
8. The PDL should support structured and unstructured comments. Structured comments are those used to provide more information about the design. Unstructured comments are those used where formal structures are not required (e.g. a trivial copy operation).

Recommendation:
The PDL as described in section 5 is recommended for detailed design; this recommendation is mandatory as per this guideline.

3.4 OS calls

1. Where OS calls are used, they should be clearly identified
2. The OS calls should be generic and should be independent, as far as possible, of a specific OS.

4. Specific Guidelines for Coding

4.1 Programming Language

1. Standard language constructs shall be used throughout.
2. Non-standard language features usage shall be restricted to only a few modules. Non-standard usage shall be clearly marked in the procedure/module heading and reason for their use may be explained in the commentary.
   Reason: Non-standard language features considerably reduces portability of programs.
3. Insecure features of the language shall be avoided or at least restricted, identified and verified.
   Reason: Insecure features are those which are not clearly defined in the language definition; whose implementation makes it impossible to detect violations of language rules by analysis of program text, or by the compiler.
   Example: Many languages do not define order of evaluation of parameters in a call statement. Hence, a program that depends on a particular order of parameter evaluation may malfunction.
4. If possible assembly-level language should not be used. Assembly-level langauge may be used:
   a) when certain processing cannot be done in higher-level language
   b) when parts of the code do not meet timing requirements even for the best algorithm and with compiler optimisation (see 2.2)
   c) when it is mandated by the standards/guidelines/practices adopted for the project
5. In case of assembly-level language:
   a) branching instruction using address substitution shall be avoided. Branch tables should have constant values.
   b) multiple substitution or multiple indexing within same machine instruction shall be avoided.
   c) macros shall be used with same number of parameters always
   d) labels shall be referred by name rather than by absolute address

4.2 Modules and Procedures

1. Modules and procedures shall have functional cohesion, i.e. each module/procedure should correspond to a unique function.
2. The interfaces between modules should be simple and uniform throughout the system.
3. Procedures should have minimum number of parameters.
   Reason: Intents to reduce coupling between procedures.
4. It shall be very clear as to which parameters are only used, only defined, or modified.
5. A procedure shall have only one entry and one exit. Multiple exits may be used only in error handling.
6. Procedures should communicate exclusively via parameters (usage of global variables should be restricted).
7. Types of all parameters of a procedure and the value returned by the procedure shall be explicitly defined.

8. *Types matching* of formal and actual parameters in the *procedure call* shall be ensured. Explicit type-casting shall be used where necessary.
9. A procedure should not contain more than 50 executable statements.

4.3 Code layout

1. Each file should have the following order of the code:
   - file header (see 4.10)
   - data file formats (see 4.10), if any
   - list of include files (general followed by specific)
   - declarations of external procedures/data, if any
   - pseudos and macros, if any
   - definitions of derived types (enumerated types, structures, etc.)
   - definitions of global data, if any
   - module header (see 4.10), module data and procedures, in that order, of each module

2. As far as possible, a file should contain only one module.
3. Within procedures, the variable declarations shall be clearly separated from program statements by a blank line.
4. Declarations shall be close to the blocks to which they apply.
5. Indentation shall be used to indicate logical depth of a statement or a block of sequential statements. The greater the logical depth the further it should be indented. The difference in indentation should be atleast 3 columns.
   Hint: Statements that begin and end constructs such as BEGIN..END, IF..ELSEIF..ELSE..ENDIF should be aligned vertically. For examples see section A.1.6 in appendix-A.
6. Indentation should be used judiciously.
   Reason: Too many levels may prove too hard to read as none at all.

4.4 Naming

1. Names of files, modules, procedures, and variables shall be precise indication of meaning/usage of that object.
2. Programs file and associated header file (e.g. .h file) shall have same name.
3. Visually similar names and unnatural spelling should be avoided. If acronyms or short names are used, their use should be standardised.
   Reason: Similar names would cause typographical errors, which are very difficult to detect.
4. Whenever possible, module, procedure, and variable names should be same as that used in detailed design.
5. The number of significant characters of identifier names used by the compiler/assembler shall be understood, documented, and the naming convention shall comply with this. Specifically, this shall be done for assembly-level programs, where identifier length is very restricted.
   Example: Usually assemblers have very limited length for identifiers, say 8 characters. If names of some of the variables are more than 8, the assembler would use only the first 8 characters. If first 8 characters of two identifiers match, their use in the program becomes ambiguous.

Recommendation:
The following naming convention is recommended:

- all names of derived types (enumerated, subrange, structures, unions etc.) and constants shall have all uppercase letters
- for others, each component of a name shall have the first letter in uppercase (except for cases described below) and rest in lower case
- procedures and global variables shall have the first letter in upper case and all local variables shall have the first letter in lower case
- if there are a very large number of procedures associated with an object (say, windows manager) a short form of the object (say, Wm) may be prepended to the procedure name.

Example: DAYS is enumerated type, PopStack is a procedure, Stack is global variable, and stackPointer is local variable. WmChangeSize, WmGetEvent, etc. are all associated with Windows Manager.

4.5 Labelling

Labels are mostly used in assembly-level programs.

1. Labels shall use lexicographic suffix. The labelling scheme may use gaps between consecutive labels.
   Reason: Lexicographic scheme implies an order making it easier to track the program. Leaving gaps allows for making inevitable corrections without relabeling others
   Example: In lexicographic scheme label SQT10 always precedes SQT20.
2. Guidelines 4.4 (1), (3), and (5) (naming) shall be applied for labelling also.

4.6 Declarations and definitions

1. Data types of variables shall be explicitly declared, whether or not this is required by the language processor; i.e. implicit type definition shall be avoided.
   Example: In 'C' it is not necessary to declare type of a variable; by default it is taken as integer.
2. Each variable should have a single purpose and it should be stated in the commentary.
   Hint: Using the same variables for different purposes makes it difficult to understand program.
3. Possible values of variables should also be stated in the commentary.
4. Where size of a data (e.g. arrays) may change, use pseudos to define the size.

4.7 Global Variables

1. Global variable usage shall be very restricted.
2. Only a single procedure/task should update (write) a global variable while all others should only use (read-only) the global variable.
3. As far as possible, global variable access should use uniform and standard resource handling procedures; or should use information hiding and interface routines.

4.8 Control Structures

1. In *condition* constructs (IF-THEN-ELSE):
   a) the test-condition and action should be simple enough to write one per line.
   b) if a condition is tested, do something; i.e. avoid NULL statement after IF or ELSE
   c) in nested IF statements the test-conditions shall be mutually exclusive.
   d) the action shall be close to the test-condition.

e) avoid bushy tree.

f) "*ELSE BREAK*" and "*ELSE RETURN*" should be avoided.

Reason: Condition construct consists of a test-condition and an action as in "*IF (test-condition) THEN <action-1> ELSE <action-2>;*" This may be nested as in case of "*ELSE IF*". Bushy tree is a long nested "*IF-ELSEIF-ELSE*" statements. NULL statement after *IF* or *ELSE* is difficult to understand. "*ELSE BREAK*" forces one to iterate back. Mutually non-exclusive test-conditions in a nested statement would make the logic unsafe and ambiguous.

2. Floating point numbers shall be properly checked for equality.

Reason: Due to truncation errors, floating point numbers can only be checked for equality within a band.

3. In *loop* constructs:

a) multiple branches out of the loop shall be avoided(except in error handlers)

b) branches into loops shall be avoided

c) loops should only be used with constant maximum loop variables

d) loop count shall be updated only at one place

Reason: (c) above guards against run-time problems and array bound violations. It helps to determine path/loop count and makes test-case generation easier and effective.

4. In *switch* constructs:

a) *case* conditions for the branches shall be mutually exclusive and shall be an exhaustive list of all possibilities.

b) "*default case/branch*" should be reserved for failure handling.

5. Inverse Boolean expressions should be avoided.

6. Dynamic instruction changes shall be avoided.

4.9 Inputs/Outputs

1. Input format and sequence should be in the natural form and order from user's point of view.
2. The outputs should be capable of standing alone with proper headings or labels.
3. Output heading should have the program identification with version, purpose, problem identification, and summary of input data used.
4. Each output item should be positively identified and cluttering should be avoided. Logical group of data should be separated by blank lines or page breaks. Indenting, spacing, and other formatting practices may be used.
5. In case of file input/outputs reference to data file format (see 4.10) shall be given.

4.10 Code Documentation

4.10.1 File Heading

File header shall contain the following in the given order:

- *Package Name:* Name of the software package. Specify package name. Specify version for the main/start-up file that generate executable program, or if this file is associated with only a version (e.g. on different OS or machine platforms)
- *Modules:* Names of the modules contained in this file
- *Related Files:* Other files related to the software package (if there are more than one file, the first file should be the main/start-up file)
- *Function:* Brief description about the software package functions (not the module functions)

- *Usage:* (applicable only for the main/start-up file that generate executable program)Description of the command line to invoke this program with details of flags, switches, and other parameters of the command. If there are different packages, this should be done for each package.
- *Packages:* (applicable only if this program can be packaged in different forms - e.g. executable, object, library, DLL, etc.) Names of different packages and instructions to generate these packages (e.g. compiler directives, macro definitions, etc.)
- *Program Generation:*(applicable only for the main/start-up file that generate executable program) reference to program generation file (e.g. makefile) or specific directive on how to link files.
- *OS:* Operating system dependency for program generation and execution
- *Compiler:* Compiler used with version and any specific compiler directive
- *Portability:* Specify portability problems related to usage of non-standard language features, OS, hardware platform etc.
- *Limitations, Warning:* Any specific limitations of the software or care in using the software
- *Author:* Name of the original author and organisation
- *Creation Date:* Date of first edit.

4.10.2 Data File Formats

Data files are those used as inputs/outputs of a program.
Data files can be abstracted as consisting of a number of records of identical format. Each record, in turn, consists of one or more fields. A field may be hierarchical; i.e., it may consist of lower-level fields. The lowest level field is of a primitive type such as integer, string, etc.
Example: A pay-roll database file consists of a number of records, as many as the number of employees. Each record contains employee information and would consist of a number of fields such as name, date of birth, etc. The name field may be of type string and date field may of type integer.
The following shall be specified for each data file:

- *File Name*: specify whether a fixed filename is used or the file name is given by the user
- *File Type*: Input/ Output/ Input&Output; ASCIIText/ Binary/ etc.
- *Records*: Structure of a record and number of records. Structure shall show the fields that constitute a record. Constant record length is recommended. If records are of variable length, specify delimiters of record.
- *Fields*: Structure of fields showing the full hierarchy in case if lower-level fields exist. For the lowest-level field specify the type, length, and range, if any. Constant field length is recommended. If a field is of variable length, specify delimiters of the field.

4.10.3 Module Heading

Module heading should contain following in the given order:

- *Name*: Module name and configuration item identifier, if any
- *Function:* Brief description about functions of the module
- *Data:* Name(s) of data and their purpose, if any, associated with the module
- *Procedures:* Name and brief description of each procedure of the module
- *Component Name:* Associated component name(s) in Detailed Design Document
- *Limitations:* Limitations, if any, of the algorithm or module implementation
- *Author:* Name of the original author

- *Last Change:* date of last modification and associated version number, if any
- *Change History:* Date, version number(if any), and reason(brief), for each change

4.10.4 Procedure Heading

Procedure heading should contain following in the given order:

- *Name*: Procedure name
- *Function*: Detailed description about the procedure function(s).
- *Sample Call:* Sample of a procedure call with formal parameters and types of all parameters and values returned.
- *Inputs*: Descriptions about each parameter used (read-only)
- *Outputs*: Description about each parameter defined (write-only) and modified (read-write)
- *Returns:* Values returned including that in error conditions
- *Uses*: (if applicable) Names of global variables used (read-only)
- *Modifies*: (if applicable) Names of global variables modified (read-write, write-only)
- *Pre- and Post- Conditions:* specify here or refer to SDD for the condition to be satisfied before executing this procedure and the condition after the procedure execution.
- *Calls*: Names of procedures called
- *Called by*: Names of procedures that call this
- *Last Change:* Date of last change

4.10.5 Comments

1. Comments should be well thought out and designed to convey understanding of the program.
2. Do not reproduce the code in comments.
3. Trivial comments should be avoided; comments should explain difficult and unusual parts of code and illuminate subtle points.
4. Code and comments shall be consistent; comments should be written at the same time the code is written.
5. Comment should not normally interrupt a logical block; comments should be used to separate logical blocks of program
6. There should be good visual separation between code and comments. Where comments are put in the same line as code, comments should always start from a specific column. Comments in separate lines may start beyond beginning of code line from a specific column.

4.11 Using Library (OS calls, run-time library...)

1. Library calls should be clearly identified (e.g. by commentary).
2. Use the correct type (if required use type casting) of parameters in library call.
3. Enable automatic type checking by the compiler; include all the necessary header files.
4. All the value(s), if any, returned by the library function shall always be used; use them correctly; especially check for values returned in error conditions.
5. In case of dynamic memory created by the library call, release the memory after use.

4.12 Using Non-standard features of language

1. As far as possible, non-standard features should be avoided.

2. Non-standard calls should be clearly identified (e.g. list of all such features).
3. These must be limited to very few modules.

4.13 Debug and Diagnostic Codes

1. Codes used for debugging and diagnostics shall be clearly identified and it shall be clear whether these would be executed or not.
2. It shall be possible to exclude these codes with minimum changes in the program. Compiler directives should be used for this purpose.

4.14 Error Checking and Handling

1. Run-time assertions should be used wherever possible.
2. Counters other techniques may be used to check program execution in correct order.

5. Program Design Language

Program Design Language or PDL is used to specify the procedural aspects of the software, mainly in the detailed design phase of the software development. In the past, flow charts were used to show the procedural aspects. Currently PDL is preferred because of its several advantages[9].
PDL uses vocabulary of a natural language (e.g. English) and overall syntax of a programming language (e.g. Pascal). The natural language narrative, embedded directly into the syntactical structure, is used to specify simple operations (e.g. "test flag x"). In contrast, the programming language syntax specifies the overall structure and control flow of the design. The characteristics of a PDL are:

- a fixed syntax of keywords for control and data structures
- a free syntax of natural language that describe processing features
- support for more than one design methodology(e.g. OOD, SADT) [7]

The IEEE standard[7] has a detailed list of desirable characteristics for using Ada as a PDL. The PDL described in this section complies with the IEEE standard.

5.1 CA-PDL

CA-PDL is a PDL based on a mixture of *C* and *Ada* languages. *C* is used as it is most popular and is used in most embedded systems currently being developed in this organisation. However, it has several drawbacks, the most important being: (a) it is cryptic and is not as expressive as Pascal or Ada, (b) it does not directly support concurrent programs (most embedded systems including that used in NPPs use concurrent programs). CA-PDL uses some features of Ada to overcome these deficiencies of *C*. The Ada features used in this section are taken from [9].
The following symbols are used in the PDL specification.

Symbol	**Meaning**
::=	composed of
<x>	x is a symbol
"y"	y is a keyword or a literal
m[]n	repetition, m to n; absence of m indicates value 1 and that of n indefinite
{}	optional
|	OR
+	AND
//	comment

The CA-PDL is specified in the following sections.

5.1.1 Types, Variables, Constants

Constant
<constant_declaration> ::= "CONSTANT" <scalar_type><constant_name>"="<constant_value> ";"

Example:

    CONSTANT FLOAT a=10.0;

Variable

<variable_declaration> ::= <type_name> {"*"} <variable_name> {"="<initial_value>} ";"

<type_name> can be name of any type specified below. Optionally * can be used to indicate that <variable_name> is a pointer to a memory containing the given data type. To select the data using pointer use:

<data_selection_using_pointer> ::= "*"<pointer_variable_name>   // for scalar and array variable
<data_selection_using_pointer> ::= <pointer_variable_name>"->"field[.field] // for record,union

Example:

```
int a=5;
int *b; .....      c = *b;
record_type *z; ........   x = z->filed_1
```

Scalar

<scalar_type> ::= "FLOAT" | "INT" {"RANGE ("<low>..<high>")"} | "CHAR" | "BOOL" |
    "STRING" {("<low>..<high>")"}
// a STRING constant is put between quotes, e.g. "name"
// in case of STRING <low>, <high> indicate the maximum number of characters in the string

Enumerated types

<enum_type_declaration> ::= " TYPE" <type_name> "IS (" [<item>,] ") ;"
Example:

    TYPE COLOUR IS (RED, GREEN);

Derived types

<derived_type_declaration> ::= "TYPE" <type_name> "IS" <basic_type> "RANGE"
    <low>".."<high>";"
<basic_type> ::= <scalar_type> | <enumered_type>
Example:

    TYPE person_id IS INT RANGE 1..10;

Composite types - Arrays

<array_type_declaration> ::= "TYPE" <array-type> "IS ARRAY (" [{<scalar_type> "RANGE"}
    <first>..<last> | "<>",] ") OF" <basic_type>";"
// <first>, <last> are first and last index of the array, more than one range for multidimensional array
// for integer type name and RANGE keyword may be omitted. "<>" is used to indicate that the actual array size is kept open and is fixed at the time of variable definition.
// A variable can be directly defined as an array by omitting the type declaration part above
Example:

```
TYPE names IS ARRAY (1..10) OF STRING;
TYPE names IS ARRAY (<>) OF STRING;
ARRAY (1..10) OF CHAR xyz;
```

An element of an array can be selected by:

```
<array_element_selection> ::= <variable_name>"[" <index> "]"
```

<index> must be of the same type as <scalar_type> used in array type declaration.

Composite types - Records

```
<record_type_declaration> ::= "TYPE" <record_type> "IS RECORD" [<field_type> <field>";"]
        "END RECORD;"
```

Example:

```
        TYPE date IS RECORD
                INT RANGE 1..31 day;
                month_type moth;
        END RECORD;
```

Composite types - Variant Records or Unions

```
<union_type_declaration> ::= "TYPE" <union_type> "(" <variant_type> <variant> ") IS UNION"
        [<field_type> <field>";" | <variant_field> ";"]
        "END UNION;"
<variant_filed> ::= "SWITCH" <variant> ["CASE" <variant_value> <field_type> <field>";"]
        "END SWITCH;"
```

An element of a union can be selected by:

```
<union_element_selection> ::= <variable_name>.<varient_value>[.<field>]
```

<filed> may be expanded depending on <field_type>

Example:

```
        TYPE message (message_class class) IS UNION
                STRING(1..100) data;
                SWITCH class
                        CASE ALARM
                                INT urgency;
                        CASE BROADCAST
                                duration timeout;
                END SWITCH;
        END UNION;
```

5.1.2 Expressions

```
<assignment_operator> ::=    "=" | <any_operator_defined_below>"="
// assignment after an operator implies using left operand for the operation and assignment-as in
'C'
<arithmetic_operator> ::= "**" | "*" | "/" | "%" | "REM" | "+" | "-"
// ** - exponentiation, % - mod, REM - remainder
<logical_operator> ::= "!" | "==" | "!=" | "<" | ">" | "<=" | ">=" | "IN" | "&&" | "||" | "^^"
// !-not, && - AND, || - OR, ^^ - XOR, IN - checks membership on a subtype
```

Example:

```
        TYPE day IS (MONDAY, TUESDAY);
        day today;
```

    IF (today IN day) .....

<binary_operator> ::= "&" | "|" | "^" | "~"
// binary operator is a bitwise operation; ^ - XOR, ~ - NOT

<shift_operator> ::= "<<" <no_of_times> | ">>" <no_of_times>
// << - left shift, >> - right shift

<increment_decrement> ::= "++" | "--"<variable_name> | <variable_name>"++" | "--"
// if operator is used before the variable name, the operation is done prior to using the variable, otherwise the variable is used and then the operation is done.

<aggregate_values> ::= "(" [<item>"," | <index_or_recordfieldname> "=>" <item_value>"," |
    "OTHERS =>" <item_value>,] ")"
// this is used for setting arrays and records elements. In the second method, each index or element is named and assigned. OTHERS indicates all elements other than that initialized.
Example:

    ARRAY (day) OF INT hours_worked=(FRIDAY=>7, OTHERS => 8);

// each statement is termined by a semicolon ";"

5.1.3 Program Control Statements

<condition> ::= "IF" <boolean_expression> "THEN" <statements> {["ELSEIF" <statements>]}
    {"ELSE" <statements> } "ENDIF;"
Example:

```
IF (save command)
        save file;
ELSEIF (load command)
        read file;
ELSE
        display error;
ENDIF;
```

<for_loop> ::= "FOR (" <initial_state> ";" <iteration_condition> ";" <next_iteration_state> ")"
    [<statement> | "BREAK" " | "CONTINUE"] "ENDFOR;"
<while_loop> :: "WHILE (" <boolean_expression> ")" [<statement> | "BREAK" " | "CONTINUE"] "ENDWHILE;"
<repeat_loop> :: "REPEAT [<statement> | "BREAK" " | "CONTINUE"]
    "UNTIL (" <boolean_expression> ");"
// BREAK exits from the loop and CONTINUE does the next iteration; just as in *C* language

<switch_statement> ::= "SWITCH" <scalar_expression> [["CASE" <value> | "DEFAULT"] ":" [<statement> | BREAK]] "ENDSWITCH;"
// only one DEFAULT statement is allowed in a SWITCH construct

5.1.4 Procedure

Procedure is a named program unit. There are no *functions* as in Ada. A procedure may return a value of type scalar. Unless explicitly specified, a procedure returns an integer.

```
<procedure_declaration> ::= {"VOID" | <return_type>} "PROCEDURE" <procedure_name> "("
        [<parameters>] ") IS"
        END <procedure_name>";"
<procedure_definition> ::= {"VOID" | <return_type>} "PROCEDURE" <procedure_name> "("
        [<parameters>] ") IS"        <declarations> "BEGIN" <statements>
        {"EXCEPTION" <exception_handler>}
        END <procedure_name>";"
<parameter> ::= ["IN" | "OUT" | "INOUT"] <type_name> <formal_parameter_name>
<exception_handler> ::= ["WHEN" [<exception_condition> | OTHERS]| <statements> ]
```

VOID indicates that procedure does not return any value, otherwise <return_type> indicates the type of the value returned. <declarations> are for local variables of the procedure.
IN indicates that the parameter is read-only, it cannot be changed by the procedure. OUT indicates values returned to the called program. INOUT indicates that the values passed on may be modified.
When <exception_condition> occurs while execution of the procedure, the control is immediately transferred to the statements given under that. OTHERS defines any other exceptions that specifically defined. The <exception_condition> may be system defined (such as NUMERIC_ERROR, OVERFLOW, etc.) or it may be user defined by the statement (in variable/type declaration section):

```
"EXCEPTION" <exception_condition>";"
```

Within the scope of the above definition (as in the case of variables, e.g. within this procedure), an exception can be raised by:

```
"RAISE" <exception_condition>
```

A system wide exception condition may be used by a global definition as in case of a global type definition.

Scope rule:
All variables declared in the declarative part of a *procedure* have scope upto the end of the *procedure*. Any variable declared outside a *procedure* is a global variable and has scope covering all *procedures*.

Example:
```
VOID PROCEDURE BODY cook (IN raw, OUT cooked) IS
        EXCEPTION OVEN_FAIL;
BEGIN
        light_oven();
        IF (oven has failed)
                RAISE OVEN_FAIL;
        ENDIF
        // do cooking
EXCEPTION
        WHEN OVEN_FAIL
                try_on_hotplate(raw, cooked);
        WHEN OTHERS
                cooked = BAD;
END cook;
```

5.1.5 Package

Encapsulation and information hiding are the most important concepts of modularity. A package is an encapsulated program entity (or a module). It consists one or more procedures providing a set of logically related functions (e.g. a FIFO queue). A package clearly separates external interfaces from the internal implementation. Internal details are hidden from the external programs. The package as defined here is taken from Ada language[9].

```
<package_declaration> ::= "PACKAGE" <package_name> "IS" <visible_declarations>
                          "END" <package_name>";"
<package_definition> ::= "PACKAGE BODY " <package_name> "IS" <hidden_declarations>
                          <procedures_or_tasks>     "END" <package_name>";"
<package_inclusion> ::= "USE" <package_name>";"
```

<visible_declarations> may consists of type declarations and procedure/task declarations showing the interface. <hidden_declarations> consists of variable and types invisible to external programs. Scope of variables in this part are all the procedures of the package. <procedures_or_tasks> consists of internal procedures as well as procedures and tasks declared in the visible part of the package.

Example:

```
PACKAGE job_queue IS
        VOID PROCEDURE put (IN job);    // these are the interfaces to external_programs
        VOID PROCEDURE get(OUT job);
END job_queue

PACKAGE BODY job_queue IS
  ARRAY(1..10) OF INT store;
  INT count;

  VOID PROCEDURE BODY put (IN job)
  BEGIN
        store[count++] = job;
END put;

  VOID PROCEDURE BODY get(OUT job);
  BEGIN
        job = store[--count];
   END get;
END job_queue
```

5.1.6 Parallelism and Tasks

Parallelism or concurrency is supported by means of program entities called "tasks". Tasks are independently executing concurrent processes. The tasks interact with each other through "task synchronisation" and "inter-task communication". The task execution, synchronisation and communication, all are handled by a specific multi-tasking OS/Kernel on which the software is developed. As the concurrency features given in this section are very general, it may be very different from that of a specific OS. In detailed design it is often necessary to use specific OS calls. Hence, it is not mandatory that the software designer should use the features given here for specifying concurrency. However these abstract concurrency features may be used:

- if the design is to be OS independent or if the underlying OS is not yet identified
- in *process specifications* (which are implementation independent) during software requirements analysis[11]

The features given here are mostly taken from Ada language as given in [9]. However, it may be noted that while an *Ada task* can be defined within a program unit only, here a task can be defined wherever a *procedure* can be defined.

Tasks:
A task definition is similar to a procedure definition.

```
<task_declaration> ::= "TASK" <task_name> "IS" [<entry_declaration>]
        "END " <task_name> ";"
<entry_declaration> ::= "ENTRY" <entry_name> "(" <parameter_list> ");"
<task_definition> ::= "TASK BODY" <task_name> "IS" <declarations & statements>
        "END " <task_name> ";"
```

Entry points of a task are the means of task synchronisation and inter-task communication. A task synchronises with another task by calling the entry point of the called task through "accept" call (see below). The <parameter_list> is same as that of *procedure*. The calling task can pass data by parameter list and the called task can return values to the calling task through modifiable parameter (with OUT keyword) as in the case of a *procedure* (see example below).

Task Execution:

```
"ACTIVATE" <task_name> ";"
```

Normally, a task terminates only when it exits (RETURN or END). A task can also be terminated from outside by the following call.

```
"KILL" <task_name> ";"
```

Task synchronisation and communication:
A task can send signal to another task by an entry call:

```
<entry_call> ::= <task_name>"."<entry_name> "(" <actual_parameters> ");"
```

The called task accepts the signal by an accept call:

```
<accept_call> ::= "ACCEPT" <entry_name> "(" <parameters_list> ")" <statements>
```

"END" <entry_name>

Entry selection and guarded selection:
A called task can select any of the possible waiting task for synchronisation. Besides, each of these alternative acceptance call can be guarded by a condition (e.g. don't accept a call to load a queue which is full). These are done by *selection* (for details refer [9]):

<select_construct> ::= "SELECT" [{"WHEN" <guard_condition>} "ACCEPT <entry_name>
<statements> "END" <entry_name> 0["OR"]]
{"ELSE" <statements> | "OR DELAY" <time_in_seconds> <statements>} "END SELECT"

This selects one of the *accept* calls. Which call would be accepted is indeterministic. However, Ada specifies that none of the accept calls would be *starve*. The *when* clause is optional. If it is present, the *accept* is executed only if <guard_condition> is TRUE. If it is not present, *accept* is always executed. *OR* clause separates the alternate *accept* statements. If there are no pending accept calls, the select statement would indefinitely wait. This may be undesirable for some application. For this purpose the *else* clause can be used, the code under which is executed if there are no waiting calls and execution will continue with the statement following *select*. In case a timeout is to incorporated in the select, *delay* clause can be used which sets up a delay of the specified time. After the specified time has elapsed, the statements following *delay* is executed. Only one of the statements, *else* or *delay*, can be present in a select statement. The example below illustrates the concept.
Example:

```
(a)     SELECT
                WHEN condition_1
                        ACCEPT entry_1
                                .....
        OR
                WHEN condition_2
                        ACCEPT entry_2
                                .....
        OR
                        ACCEPT entry_3
                                .....
        ELSE
                ...alternate actions
        END SELECT
(b)     SELECT
                WHEN condition_1
                        ACCEPT entry_1
                                .....
        OR
                DELAY 0.1                   .. 100 ms timeout/delay
                        .. timeout action
        END SELECT
```

A select statement can also be used to timeout the wait for an accept call of a calling task. Only one call is allowed in the selection and an *else* or *or delay* clause can be used.
Example:

```
(a)   SELECT                           (b)   SELECT
              provider.read(...);                    provider.read(...);
      OR                                     ELSE
              DELAY 0.1                              .. alternate actions
              .. alternate actions
      END SELECT                             END SELECT
```

Example: part of a multitasking software
This example, taken from [8], illustrates all the concurrency constructs defined in this section. It is part of a file handling program that takes care of concurrrent file read/write.

```
PACKAGE BODY file_handling IS
// global & type declarations
TASK buffering IS
        ENTRY read_transaction_item (OUT transaction_item x) // transaction_item is a data type
        ENTRY write_transaction_item (IN transaction_item x)
END buffering

// buffering task body
TASK BODY buffering IS
CONSTANT MAX_BUF= 100;
ARRAY (1..MAX_BUF) OF transaction_item buffer;
INT inpoint=1, outpoint=1, count=MAX_BUF;

BEGIN
WHILE (always)
        SELECT
                WHEN count > 0       // stop when buffer full
                        ACCEPT write_transaction_item (IN transaction_item x)
                                buffer[inpoint] = x; // insert in circular buffer
                        END write_transaction_item;
                        inpoint = (inpoint + 1) % MAX_BUFF;
                        count--;
        OR
                WHEN count < MAX_BUFF
                        ACCEPT read_transaction_item (OUT transaction_item x)
                        x = buffer[outpoint];
                        END read_transaction_item;
                        outpoint = (outpoint + 1) % MAX_BUFF;
                        count++;
        END SELECT;
END WHILE;
END buffering;

// procedure called by tasks intending to write to files
PROCEDURE BODY write (IN transaction_file t, IN unit_record u)
```

```
transaction_item i={t,u};
BEGIN
        buffering.write_transaction_item (I);
END write
// procedure called by tasks intending to read files
PROCEDURE BODY read (OUT transaction_item t)
BEGIN
        buffering.read_transaction_item (t);
END read
END file_handling
```