General Style and Coding Standards for Software Projects

Preliminary Version

AUTHOR(s):	APPROVED:	Revised:
Standards Group	SEPG	

Table Of Contents

1. INTRODUCTION

1.1 PURPOSE

The goal of these guidelines is to create uniform coding habits among software personnel in the engineering department so that reading, checking, and maintaining code written by different persons becomes easier. The intent of these standards is to define a natural style and consistency, yet leave to the authors of the engineering department source code, the freedom to practice their craft without unnecessary burden.

When a project adheres to common standards many good things happen:

- Programmers can go into any code and figure out what's going on, so maintainability, readability, and reusability are increased. Code walk throughs become less painful.
- New people can get up to speed quickly.
- People new to a language are spared the need to develop a personal style and defend it to death.
- People new to a language are spared making the same mistakes over and over again, so reliability is increased.
- People make fewer mistakes in consistent environments.
- Idiosyncratic styles and college-learned behaviors are replaced with an emphasis on business concerns - high productivity, maintainability, shared authorship, etc.

Experience over many projects points to the conclusion that coding standards help the project to run smoothly. They aren't necessary for success, but they help. Most arguments against a particular standard come from the ego. Few decisions in a reasonable standard really can be said to be technically deficient, just matters of taste. So, in the interests of establishing the engineering department as a showcase software development environment, be flexible, control the ego a bit, and remember any project is a team effort.

A mixed coding style is harder to maintain than a bad coding style. So it's important to apply a consistent coding style across a project. When maintaining code, it's better to conform to the style of the existing code rather than blindly follow this document or your own coding style.

Since a very large portion of project scope is after-delivery maintenance or enhancement, coding standards reduce the cost of a project by easing the learning or re-learning task when code needs to be addressed by people other than the author, or by the author after a long absence. Coding standards help ensure that the author need not be present for the maintenance and enhancement phase.

1.2 Scope

This document describes general software coding standards for code written in any text based programming language (including high-level languages like C, C++, Basic, Visual Basic, and assembler languages). Many of the guidelines described in this document can be directly applied to programming practices in graphical based languages (such as PLC, graphical, and visual languages). This will be used as the base document for several language specific coding standard documents. Each language specific coding standard will be written to expand on these concepts with specific examples, and define additional guidelines unique to that language.

For each project, this document will be used in conjunction with language and project specific coding standards that, in total define a complete set of coding standards. A description of the **general**, **language specific**, and **project specific** coding standards is provided below:

1.3 Coding Standard Documents:

Each project shall adopt a set of coding standards consisting of three parts:

- **General Coding Standard**, described in this document
- **Language specific coding standards** for each language used, described in separate appendices to this document. These language standards shall supplement, rather than override, the General Coding standards as much as possible.
- **Project Coding Standards**. These standards shall be based on the coding standards in this document and on the coding standards for the given language(s). The project coding standards should supplement, rather than override, the General Coding standards and the language coding standards. Where conflicts between documents exist, the project standard shall be considered correct. Sweeping per-project customizations of the standards are discouraged, so that code can be reused from one project to another with minimal change.

1.4 Other Related Project Documents

The "Life cycle" and "Configuration Management" policy standards define a set of documents required for each project, along with a process for coordinating and maintaining them. Documents referred to in this report include:

- Software Detailed Design Document (SDDD)
- Configuration Management (CM)

1.5 Terms Used In This Document

- The term "**program unit**" (or sometimes simply "unit") means a single function, procedure, subroutine or, in the case of various languages, an include file, a package, a task, a Pascal unit, etc.
- A "**function**" is a program unit whose primary purpose is to return a value.
- A "**procedure**" is a program unit which does not return a value (except via output parameters).
- A "**subroutine**" is any function or procedure.

- An "**identifier**" is the generic term referring to a name for any constant, variable, or program unit.
- A "**module**" is a collection of "units" that work on a common domain.

1.6 References

a. General Coding Standards, Author (s) unknown
b. Steve McConnell, Code Complete, Microsoft Press 1996 (ISBN 1-55615-484-4)
c. Writing Solid Code, Steve McConnell, Microsoft Press 1996 (ISBN 1-55615-551-4)
d. A Modest Software Standard, Jack G. Ganssle, March 1996 Embedded Systems Programming
e. Recommended C Style and Coding Standards, Indian Hill C Style and Coding Standards Updated by authors from Bell Labs, (http://www.cs.huji.ac.il/papers/cstyle/cstyle.html)
f. C coding Guidelines, Real Time Enterprises, Inc. Rev 7, 6/24/96
g. C++ Guidelines (companion document to C coding guidelines), Real Time Enterprises, Inc., Rev 0 11/8/1995
h. C style and Coding Standards for the SDM Project, ROUGH DRAFT, July 3, 1995 (http://www-c8.lanl.gov/sdm/DevelopmentEnv/SDM_C_Style_Guide.hyml
i. Software Design and Coding Standards for C++, Authors Unknown., 7/7/1994
j. C Programming Standards and Guidelines, Internal Document 8/10/93, Software Warehouse index # 100
k. C++ Style Guide, Internal Document, 6/22/90, Software Warehouse index # 201
l. Using Visual Basic for Applications (Appendix D- Style Guide for Professional Quality Code), QUE, (ISBN 1-56529-725-3)
m. Speaking the Language of the PM API, Part 4 (Overview of Hungarian Notation), PC Magazine March 14, 1989)
n. Programming Integrated Solutions with Microsoft Office, Appendix B, Visual Basic Variable Naming and Coding Standards.

1.7 Our Limited Lifetime Warranty

If these standards - when used as directed - fail to perform as expected, they can be edited and adapted to changing environments, applications, business emphasis, and an ever-evolving industry. The *spirit* of this document, not it's rules, should dictate the place of standards and consistency within and across projects.

1.8 The Emotional Topic of Coding Standards

Please be patient with these coding standards until they become natural... it is only then that an honest opinion as to correctness or utility can be formed. They need not impede the feeling of craftsmanship that comes with writing software. Consider the common good. Embrace the decisions of the group.

2. Project Dependent Standards

The standards and guidelines described in this document were selected on the basis of common coding practices of people within our group and from many language specific programming standard documents collected throughout the company and the Internet. They can't be expected to be complete or optimal for each project and for each language. Individual projects may wish to establish additional standards beyond those given here and the language specific documents. Keep in mind that sweeping per-project customizations of the standards are discouraged in order to make it more likely that code throughout a project and across projects adopt similar styles.

This is a list of coding practices that should be standardized for each project, and may require additional specification or clarification beyond those detailed in the standards documents.

1. Naming conventions:
    - What additional naming conventions should be followed. In particular, systematic prefix conventions for functional grouping of global data and also for names of structures, objects, and other data types may be useful.

2. Project specific contents of module and subroutine headers

3. File Organization:
    - What kind of Include file organization is appropriate for the projects data hierarchy
    - Directory structure
    - Location of Make Files

      (note: "Actions taken before compilation or assembly is performed should be the directory in which the source code resides, unless otherwise specified.)

4. Specifications for Error Handling:
    - specifications for detecting and handling of errors
    - specifications for checking boundary conditions for parameters passed to subroutines

5. Revision and Version Control: configuration of archives, projects, revision numbering, and release guidelines.

6. Guidelines for the use of "lint" or other code checking programs

7. Standardization of the development environment - compiler and linker options and directory structures.

3. FILE and MODULE GUIDELINES

3.1 Module Design Guidelines

All source code will be grouped into modules. Each module will deal with a single, unique domain. Deciding how to decompose a specific system into constituent modules can be complex and not within the scope of this paper. However, emphasize simplicity, clarity, cohesion, and decoupling. In general, one source code file will contain the implementation of one module.

3.2 Header (Include) Files

If the language permits, header files are support files referenced by other files prior to compilation. Some, such as **stdio.h** are defined at the system level and must be included by any program using the standard I/O library. Header files are also used to contain data declarations and defines that are needed by more than one program. Header files should be functionally organized, i.e. declarations for separate subsystems should be in separate header files. Also, if a set of declarations is likely change when ported from one machine to another, those declarations should be in a separate header file. Include file Guidelines Any source file that uses the facilities made available by another module needs to include the associated header file.

- Use relative (not absolute) path names for include files of source code being created by the project. This approach allows for the copying and compiling of project modules into different subdirectories without the need to change the contents of the source code.

C language example: #include “filename.h”, or “..\filename.h”

- For compiler provided header files (i.e. stdio.h) it’s recommended to specify path names that allow the compiler to be configured to search for header file.

C language example: **#include <stdio.h>**

3.2.1 Use consistent implementation strategy for header files

It’s most common to implement header files in one of two ways, as described below. Either approach is valid, but should be defined and applied consistently across a project.

- Each module has it’s own corresponding header file that describes the interface to that module, and provides external declarations for global functions, data types, and variables that are allocated in the source file for the specific module.
- Use one (or a few) header files to group global information for a project or a subsystem, making it easier to locate function prototypes, global data types, constants, macros, and data.

3.2.2 Header file Guidelines

- Header files will not allocate variables or contain code. All global data will be declared in a single source file and referenced via external declarations in corresponding header files. That way it is clear which source files owns the data.
- Header files that declare functions or external variables should be included in the file that defines the function or variable. That way, the compiler can do type checking and the external declarations will always agree with the definition.

- Header files can define global data types constants and macros, and contain external declarations that is shared between multiple modules.
- Header files will not be used to initialize data
- Do not put any private information in a header file

3.3 Source File Layout Guideline

The specific contents of sections in a file may vary from language to language, but following order should be used as a guideline for layout of a file:

- File or Module Header (defined later in this document)
- Include Definitions: Any header file includes should be next. If the include is for a non obvious reasons, the reason should be commented
- Constant and Macro Definitions
- Data Type definitions
- Variable Definitions
- Subroutines

3.4 File Naming Guideline

A consistent naming convention for files and for directories shall be developed and used on a per-project basis.

- A file naming convention makes project files easily distinguishable from other projects, and it helps associate different file types within the same project.
- Directories and subtrees can be used to link portions of a project together.

3.5 File Informational Headers

A File Header shall be placed at the beginning of each source file. Fields of the headers should never be deleted; non-applicable fields shall be marked "none". Note that a software "module" (a logical grouping of
related subroutines) may cross many files.

The File Header should include the following:

- Copyright, 199x, All Rights Reserved. For internal use only.
- File Name
- Project Name
- Module Name
- Brief Description of purpose of file. Enough detail should be provided to give an overview of the module and how subroutines interact with each other. It should not restate information that is included in the function's comment header.
- Revision History notes

- File-wide compiler dependencies

3.5.1 File Header

This is a C language example of a file header - a project specific coding standard that can be used as a template for any project. Projects may require additional specification or clarification beyond those detailed in the standards documents.

```
/*************************************************************************
     1.   *  Copyright 1999, All rights reserved, For internal use only
*
*  FILE:
*  PROJECT:
*  MODULE:
*
*  Description:
*
*  Notes:
*
*  Compiler dependencies or special instructions:
*
*  REVISION HISTORY
*  Date:                    By:              Description:
*
*************************************************************************/
```

3.6 Subroutine Header

For each program unit in the file, a header should be inserted at or near the beginning of the declaration of the unit, as specified for the given language. The exact form of the header may vary somewhat according to the language used and the type of program unit, but should always include:

- Name
- Brief description of the purpose of the unit
- Subroutines called in support of this subroutine.
- Global variables written and referenced.
- Return (If applicable)
- Designer(s) or Reference to Design Document(SDDD).
- Programmers(s)
- Tested By/Date: ????
- Assumptions and limitations, including compiler, assembler, and machine dependencies - note any assumptions about the current processing state. It might be noted that it is assumed that a certain variable has been initialized, or that a certain variable has on of a certain subset of values. The section shall also point out any obstructions to reusability which are not made clear by the name of the subroutine. For instance, for a subroutine named "Is_Digit", this section might note that it only works for the ASCII character set. This section should note any system limitations, e.g. execution of a unit may depend on low byte/ high byte ordering of a number in processor memory.

- Exception processing - describe any unusual actions taken by the unit, including its handling of invalid input data.
- Notes - give commentary on the algorithm, efficiency evaluations, alternative methods, etc. Note that the purpose of this section is **not** to provide a general description of the code's purpose and structure; this is provided by the Distributed Code Description, as described below.
- Revision history, included for origin and each set of changes:
  - Date of revision
  - Name of programmer making change(s)
  - Description of change(s), including reasons for change and any other statements of interest

3.6.1 This is a C language example of a subroutine header:

```
/**************************************************************************
* NAME:
* Description:
*
* Subroutines Called:
*
* Returns:
*
* Globals:
*
* Designer(s):/ Design Document:
* Programmer(s):
* Tested By:                    Date:
* Assumptions and Limitation:
*
* Exception Processing:
*
* NOTES:
*
* REVISION HISTORY
* Date:                By:                    Description:
*
**************************************************************************/
```

4. Constants and Macros

4.1 Use constants and macros instead of hard coded literal values

If supported by the language,

- Literal values shall be avoided in code statements; rather, a symbolic constant for each shall be defined reflecting its intent. 0 should not be assumed to mean “OFF”, a symbolic constant OFF should be defined to be equal to 0.
- The numeric literals 0 and 1 shall not be used as Boolean constants. Booleans are not to be treated as integers.
- Whenever different constants must have fixed relationships and whenever allowed by the language, the fixed relationships shall be forced to hold true. For instance, if ConstantB must be twice the value of ConstantA, define ConstantB as being equal to 2 * ConstantA.

The exceptions to this rule include

- The numeric literals 0 and 1 (where their use is to initialize, increment or to test).
- Certain fixed-purpose character literals (e.g., ‘ ‘ will always be a blank), and strings giving messages or labels.

4.2 Only Define constants and macros once

Constants and macros shall not be defined in more than one textual location in the program, even if the multiple definitions are exactly the same.

4.3 Place parenthesis around each macro parameters

This will avoid unexpected precedence problems when the macro is expanded into code.
C language example: #define PRODUCT(a , b) = ((a) * (b)), not (a * b)

4.4 Notes about style

“Magic numbers” are discouraged. They often make the author’s intent unclear and make global changing of a value undependable.

Wrong:

```
if (fKilnTemperature < 92.46) .....
```

Correct:

```
#define  MAX_KILN_TEMP   (92.46)

if (fKilnTemperature < MAX_KILN_TEMP) ....
```

There are times when expected modifiability would dictate that constants be placed in a dependable area (e.g. in the global header file, etc.) for easy access and changing.

5. Global Data Guidelines

- Global data is generally to be avoided. Parameters are the preferred method of communication among subroutines.
- Limit scope of module-level data to the module (i.e. use the **static** storage class specifier to declare functions and data local to the file).
- Any variable whose initial value is important to be initialized should be with executable code, don't initialize static data at time of allocation.

6. Subroutines

6.1 Subroutine Scope Guideline

Repetitive sections of code should be made subroutines so that parallel maintenance of several copies of the same code can be avoided, and so that maintainers can be sure of the similarity of the passages. Whenever appropriate, sections of code which are almost the same, except for the identity of some variables, should be made subroutines with parameters to allow for the differing variables.

6.2 Subroutine Declaration Guidelines:

- Minimize scope by declaring subroutines used only within the module as "static"
- The subroutine's return type should be declared (do not allow the compiler to select a default value).
- Each parameter type should be declared (do not allow the compiler to select a default value).

6.3 Subroutine Layout Guidelines:

The specific contents of sections in a subroutine may vary from language to language, but following order should be used as a general guideline for layout of a subroutine:

- Subroutine Header
- Variable declarations
- Comments and Code (intermixed, using "Distributed Code Description")

6.4 Subroutine Size Guideline

Subroutines shall be as long as is necessary to accomplish one independent, cohesive, decoupled function. Size guidelines which limit the number of lines of code, or limit the number of pages of text, are addressing an inconsequential aspect of subroutine design. However, the association between excessive length and the number of independent jobs being performed within a routine is probably a strong one. Review very closely long routines, and judge whether they are performing many tasks that are better parsed out to subordinate subroutines.

6.5 Parameter List Guideline

- If a subroutine parameter list is longer than one line, lines after the first one will be indented from the left margin so that the second parameter will be listed directly below the first parameter
- The list of the function parameters should have a definite order. The most conventional ordering of parameters is: Input, modified (input and output), and finally output parameters.
- A function that returns information via one or more of it's parameters may return only status information in its name
- For complicated function calls (with multiple parameters) listing each parameter passed on a separate line with a short comment describing it's function makes the function easier to comprehend.
- For complicated function calls, you can use a prefix for each of the parameters to clearly distinguish input, modify, and output parameters. For example: In_, Mod_, Out_ for input, modify, and output parameters respectively).

6.6 Variable Declaration Guidelines:

- The variable's type should be declared (do not allow the compiler to select a default value).
- Only declare one variable per line.

7. Comments

7.1 "Distributed Code Description "

Note that there is no Code Description section in the Program Unit Header, even though such a section is usually a part headers of this sort. Instead, a **Distributed Code Description** shall be embedded throughout the code. Each logical passage of code -- typically, on the order of 5 to 15 lines, though some may be 1 line and some may be 100 lines -- shall be preceded by an internal comment describing the action of the code that follows. This will make the code easier to read, and will form a Distributed Code Description that will be easier to verify and maintain as the code is modified. This will also make a first-pass reading of the code much easier than reading the code alone, no matter how self-descriptive the code is. Internal comments should not simply restate the code, but should clarify the encoded data structures or algorithms at a more descriptive level than the code. Comments should also be used to describe alternative methods that were considered and abandoned, to prevent duplicate effort by maintainers.

7.2 Comment Block Standard:

Code is more readable when comments are presented in paragraph form prior to a block of code, rather than a line of comment for a line or two of code.

- Comments should be preceded by and followed by a single blank line
- Indent block comments to the same level as the block being described
- Highlight the comment block in a manor that clearly distinguishes it from the code

For example:

```
/*
==============================
This is an example of a blocked comment that
is easily distinguished from the rest of this document,
and is easily edited.
==============================
*/
```

7.3 In line comments:

Block comment should be the primary approach for commenting code, but where appropriate, In-line comments adding important information are also encouraged. For example:

```
int gStateFlag;     /* This state variable is defined here, initialized in Main */
```

A comment shall accompany each definition of a constant, type, or variable, giving its purpose. The comment shall either be on the same line as the definition (to the right), or on the previous line(s).

7.4 Commenting control constructs

For constructs (such as "end if") which close specific other structures (such as "if ConditionA"), a comment after the closing construct shall in certain circumstances note the identity of the opening construct. This shall be done if the opening and closing components are textually far apart from one another, or if they are part of a somewhat complex nesting of control structures. This may seem superfluous for short blocks, but is a lifesaver for long and many-nested passages.

```
while (indTotal < 1000)
{

   Pristine, well-designed code, indented  here

} /* end while indTotal < 1000 */
```

8. Code Layout

8.1 One statement per line

- There shall normally be no more than one statement per line; this includes control statements such as "if" and "end" statements.
- When a single operation or expression is broken over several lines, break it between high-level components of the structure, not in the middle of a sub-component. Also, place operators at the

ends of lines, rather than at the beginning of the next line, to make it clear at a glance that more is coming.

8.2 Indentation Guidelines

A consistent use of indentation makes code more readable and errors easier to detect.

- 3 spaces is recommended per indent, but the exact number of blanks per indentation quantum may vary with the language.
- Statements that affect a block of code (i.e. more than one line of code) must be separated from the block in a way that clearly indicates the code it affects.

Use vertical alignment of operators and/or variables whenever this makes the meaning of the code clearer

C language example:

```
if ((Weather  ==  CLOUDY) andand
   (Temperature  == COLD)  andand
   ( LengthOfDay ==  SHORT))
{
   Season = WINTER;
   State    =  ALASKA;
}
else
{
   Season  = SUMMER;
   State =      FLORIDA;
}  /* end if Weather == CLOUDY......*/
```

8.3 Brackets, Begin...End, and Delimiting Control Blocks

Nothing brings out the territorial and protective instinct in programmers like the issue of bracket, begin..end, or any delimiter placement. People generally consider correct the format they learned in school, yet there must be reason, organization, and consistency. Most importantly, it must allow for the quick separation of a condition from its resulting functional code (i.e. the code within the brackets), and the logical grouping together of that functional code with further indents and white space.

Any of the following three are acceptable. Note that the brackets are either considered part of the conditional statement (first and second example) or part of the resultant functional block (third example, called a *pure-block* scheme). Mixing bracket attachment (one part of the conditional, one part of the functional block, as in the fourth and fifth examples) is discouraged. In example 6, the bracket is not attached to either the conditional or the functional block.

recommended:

```
if (gTestFlag = = TRUE)
{
   Run_Machine_1( );
} /* end if gTestFlag is True */
```

not recommended: multiple conditionals confuse the first bracket placement.

```
if (gTestFlag = = TRUE) {
   Run_Machine_1( );
} /* end if gTestFlag is True */
```

widely used, but not recommended:

```
if (gTestFlag = = TRUE)
   {
   Run_Machine_1( );
   } /* end if gTestFlag is True */
```

wrong: first bracket is part of the conditional, but second bracket is part of the functional block.

```
f (gTestFlag = = TRUE) {
   Run_Machine_1( );
   } /* end if gTestFlag is True */
```

wrong again:

```
f (gTestFlag = = TRUE)
   {
  Run_Machine_1( );
} /* end if gTestFlag is True */
```

wrong: unnecessary second indentation**.**

```
f (gTestFlag = = TRUE)
  {
     Run_Machine_1( );
  } /* end if gTestFlag is True */
```

Another STRONGLY SUGGESTED standard is that all functional code following a conditional be delimited, even a single line. The following example illustrates why... the two lines of functional code after the conditional appear from indentation to both run. Only the first one does.

wrong:

```
if  (gTestFlag = = TRUE)
     Run_Machine_1( );
     Write_Timestamp_To_Log( );
```

The following is clear as to intent:

```
if  (gTestFlag = = TRUE)
{
   Run_Machine_1( );
} /* end if gTestFlag is True */

Write_Tmestamp_To_Log( );
```

8.4 Error Handling

- Functions that can fail (i.e. file I/O) should always return a success or error as a return code parameter.
- Any time a subroutine calls a function that returns an error condition, the error condition should be tested for and acted on in accordance with the error handling conventions specified in the projects SDDD. Error recovery should e handled in the routine that is responsible for the domain in which the error occurs (e.g. A file error should not be passed up from file_IO( ) to Main for handling).

9. Naming Convention for Identifiers (Variables, Constants, and Subroutines)

9.1 Select Clear and Meaningful Names

The most important consideration in naming a variable, constant, or subroutine is that the name fully and accurately describe the entity or action that the structure represents. Clear, complete, and meaningful names makes the code more readable and minimizes the need for comments. For example; suppose a subroutine called “Process_Input_Line” calls “Push_Input_Character.” If“Push_Input_Character” happens also to echo the input character, then either a different subroutine should be extracted (named “Echo_Input_Character”) and called by “Process_Input_Line”, or the name of “Push_Input_Character” should be changed to “Push_And_Echo_Input_Character”.

9.2 Naming Subroutines (verb and object)

Names of procedures shall consist of a verb and (whenever appropriate) an object, such as "Push_Input_Character". This will make both the action and the object of the action clear. Subroutine names such as "Subroutine_1" are discouraged.

A Hungarian Prefix should be used to identify the return type on functions that return a value:

e.g. iGet_Time( ), blnEstablish_Communications( ), etc.

9.3 Naming Constants, variables (noun)

Names of constants, variables, and functions shall be nouns, with or without modifiers (e.g., "Line", "InputLine", "NumInputLines"), with the exception of Boolean identifiers as noted below. Constants (variables whose values can not be changed during runtime) should be capitalized: MAX_LINES.

9.4 Naming Boolean identifiers (verb and ((object or adjective))

Each name of a Boolean constant, Boolean variable, or Boolean function shall consist of a verb and (whenever appropriate) an object or an adjective. As an example, if this rule were not followed, would the subroutine call "Black_King (Checker)" mean make Checker a black king, or report whether Checker is a black king? This Boolean function would be better named "blnIs_Black_King". Note that the form "Is_Black" illustrates another appropriate construct. A subject for the verb may also be appropriate, as in a Boolean function named "blnStack_Is_Updated".

9.5 Naming Types

If allowed by the language, the names of types should have a distinguishable prefix or suffix. It is recommended that all names of types end with the letters "Type" . Further, if the type exists solely to define variable "Xyz", then the type shall be named "XyzType". Furthermore, the name descriptor (or abbreviation) part of the type (i.e. "Xyz" part of "XyzType") should be included in the name of any variable declared with that type. Likewise, Structures, Enumerators, Unions should have the suffix "Struct", "Enum", and "Union" respectively. Example:

```
struct  type
{
    integer  iAddress;
    string    strLastName;
}
EmployeeDataStruct   *EmployeeDataStructPtr;              /* struct and pointer to struct name */

EmployeeDataStruct      stEmployee;
```

```
EmployeeDataStructPtr    pstPermanentEmployee;  /* a variable that is a pointer to the struct */
```

9.6 Use of upper/lower case and underscores to differentiate Subroutines, Variables, and Constants.

Program-specific identifiers shall be differentiated from reserved words by use of upper/lower/mixed case; the exact scheme for doing so shall vary with the language, but not from project to project.

Assume that the language that you are programming in, is case insensitive, even if it isn't (as in C). This will prevent the creation of variables such as portnum, PortNum, and Portnum, which can in fact be three different variables performing three different functions... a very difficult thing to figure out and maintain.

9.7 Subroutines and Program Units:

Whenever appropriate for the language, names of program units shall contain underscores between words and use mixed upper and lower case letters (except for abbreviations).

9.8 Variables

Other sorts of identifiers (such as those for types, and variables) shall not contain underscores, and use upper case for the first letter in each unique word. For example; a Boolean function could be named "Stack_Is_Updated", and a Boolean variable with a similar purpose could be named "StackIsUpdated". This allows very differentiation between functions and variables. A variable "Color" can't be confused with the function "color".
<<Note C++ exception>>

9.9 Macros and Constants

Fixed identifiers such as Macros and Constants shall contain underscores between words and use all UPPER case letters. For example; a constant that defines the maximum of a motor could be named "MAX_MOTOR_RPM"

9.10 Use of prefix (Hungarian) notations to differentiate the scope and type of a data variable

The Hungarian naming convention is a set of guidelines for naming variables and routines. The convention is widely used the C and Visual Basic languages. (For the engineering department, a modified Hungarian notation will be suggested.)

The use of Hungarian Notation is strongly encouraged, but may vary in form on a per-project basis. If used, prefixes can vary from language to language and across applications. A list of the prefixes should to be defined as part of the project specific standards. A list of commonly used prefixes that should be used as a starting point or template for the project's list will be included below.

Strict Hungarian names are composed of three parts: one or more **prefixes** (i.e. bl for Boolean, a for array), the **base type** (i.e. wn for Window, ch for character), and a **qualifier** (descriptive part of the name that would probably make up the entire name if you weren't using Hungarian notation).

Example:
The apch prefix of apchFileSpec means that the variable is an array (a) of pointers (p) to characters (ch). Just from looking at the variable name you can guess that the variable is defined something like this:
char *apchFileSpec[10]. And that tells you a lot more about the variable than a simple name FileSpec.

The following three section are tables of common Hungarian type-defined prefixes:

9.10.1 Identifying data and variable types

Type	Prefix
Boolean	bl, bln
char	ch
unsigned char	uch
short	s
unsigned short	us
long	l
unsigned long	ul
float (or single)	f
double	d
bit	b
byte	by
function	fn
array	a
pointer	p
string	sz, str
void	v
file pointer	fp
file descriptor	fd
array of characters	ach
pointer to a structure	pst

9.10.2 Variable context

Usage	Prefix
State	sta
Register	reg
Counter	ctr
Index	idx
Flag	flg
Label	lbl
Pointer	ptr
Object	obj
Function	fn
Array	ary
String	str
Void	vd
Command	cmd
Dialog	dlg
Control	ctr
Data	dat
File	fil
Form	frm
Graph	gra
Image	img
Key	key
Menu	mnu
OLE	ole
Picture	pic
Report	rpt
Timer	tmr
Window	wnd
Window Handle	hwnd

9.10.3 Variable Scope

Scope	Description	Prefix
Global	Variable is valid only within any module in a project	g (or unique prefix identifying the source module)

Local	Variable is valid only within the subroutine that it is defined	none
Module	Variable is valid only within the module that it is defined	m
Public	In Object Oriented languages, useable by all	pb
Protected	In Object Oriented languages, useable by members of any class derived from the defining class	prt
Private	In Object Oriented languages, useable by members only of the defining class	prv

Additional Reading:

- Chapter 9 section 5 of the book "*Code Complete*" (Refer to References, section 1.5 of this document) describes the Hungarian naming convention, rules for use, and advantages and disadvantages.
- Speaking the Language of the PM API, Part 4 (PC magazine, March 14, 1989)

9.11 Abbreviations

An abbreviation shall only be considered if it saves a considerable number of characters (e.g., "FFT" for "Fast Fourier Transform" is acceptable, but "Snd" for "Send" is not), as long as the language does to restrict identifier lengths severely. An abbreviation list shall be created during the design phase of each project. However, smaller groups of programmers may create their own abbreviations for terms which are used within the domain of the code to which they are assigned; again, the use of these abbreviations shall be consistent. Any abbreviations which are on the list must be used by all programmers for any identifiers which include the corresponding phrase. For example, if series of procedures sends various types of messages using the identifiers Send_Hello_Msg, Send_Connect_Msg, and Send_Data_Msg, then the name of a new procedure in the series should not be Send_Disconnect_Message. To do otherwise simply encourages coding errors and frustrates text searches.

Common abbreviations - This is a list of commonly used abbreviations. It could be used as a starting point for a project's abbreviation list:

Alignment	align
average	avg
calibrate	calib
calibration	calib
channel	chn

coefficient	coef
column	col
control	cntl
controller	cntl
degrees	deg
detector	det
high	hi
length	len
low	lo
maximum	max
message	msg
minimum	minm
minutes	min
number	numb
quadrant	quad
seconds	secs
tolerance	tol
unit-under-test	UUT

9.12 Summary table for Naming Convention:

	Upper/lower case	Under -score	Prefixes or Suffix	Example
Subroutines	Mixed case separated by underscores	yes	Hungarian prefix for the return type	iGet_Color()
Constants, Macros, Enum Constants	Upper case	yes	none	COLOR_RED
Types and User-type declarations (Structures, Enumerators, Unions)	MixedCase	no	Suffix (Type, Struct, Enum, and Union)	ColorType, EmployeeStruct
Variables	MixedCase	no	Hungarian prefixes	blStackIsUpdated

10. Misc. Rules for Coding

10.1 Conditionals and comparisons

Always test floating-point numbers as <= or >= relational operator, never use exact comparisons (= = or !=).

No assumptions shall be made about the value of uninitialized variables, unless the language definition makes a clear statement about this.

Never use implied processing.... always state clearly a conditional's intent:

Wrong (note gTestFlag):

```
if ( gblnTestFlg )
{
   do_this( );
} /* end if gblnTestFlg */
```

Correct:

```
if ( gblnTestFlg = = TRUE )
{
   do_this( );
} /* end if gblnTestFlg */
```

10.2 Program Flow

Interrupt handlers shall perform minimal processing, and shall be meticulously commented.

In high-level languages, multiple exits from a unit are allowed if that avoids excessive control structure nesting.

Multiple entries into a unit are not allowed.

10.3 Binding time of variables and values

When data files are accessed in a tree-structured directory environment, the names of the file directories shall not be hardwired in the code; whenever possible, environment variables or some similar mechanism shall be used to provide exact directory names dynamically. The same applies to the names of nodes which are accessed in a network. (see section 5.4 on constants and coding style).

10.4 Go-tos, pointers, and issues of clarity

Go-tos are not to be "avoided at all costs". It is, instead, *serpentine code* that needs to be avoided. Simplicity and clarity should override most other design decisions. A go-to, in particular, is a powerful tool when used as a direct, no-nonsense jump under well-stated conditions, and can very closely follow problem-space behavior if used with some planning and forethought (Ada, a language designed from scratch by smart French people, contains a goto keyword). On the other hand, the indirection of a pointer tends to be a computer-space construct, that is often confusing and - if honesty should prevail - unnecessary (Java, the latest geek programming language, does not allow the use of pointers).

Other clarity-based suggestions**:**
Use case statements instead of nested ifs, use arrays instead of linked lists, optimize through solid design rather than bit-tuning, get a faster CPU instead of writing assembler, pay for the extra memory, buy code if it's available.

10.5 Strive to develop clear code

Engineers should strive to develop code that is both clear, and efficient in its use of CPU time, memory, and other resources. However, when efficiency and clarity conflict, then clarity should take strong precedence over resource stinginess, unless it is proven that using the clear but less efficient method impairs the program critically. Micro-optimizations to small areas of code are especially to be avoided if they impair clarity in any way, since it is generally only the program's overall algorithms that affect resource utilization significantly.

10.6 Use libraries when available

Whenever library routines, graphics packages, compiler/assembler features, or other sorts of utilities are helpful to the program, they should be utilized. The danger of losing the access to the utility if the hardware, the compiler/assembler, or the operating system should change is generally overridden by the savings in software creation time. In those cases in which there is no significant savings of software creation time, it is preferable to use the standard language features, for portability's sake.

10.7 Type casting integer and float variables makes code more portable

If the language allows (as with C's "typedef" or Ada's subtypes), all integer and floating types which are important in the program should be defined as new types within the program. This will allow for easy correction if the code is ported to a different compiler or machine with different default work sizes. This also often allows for much better type checking, depending on the language.

10.8 Compiler dependent code should include tests

Whenever the code makes assumptions about how the compiler represents data structures, the code should include a test (if possible) to determine whether the assumption holds true, and display a prominent message and abort the program if the test fails. This will notify future maintainers who may unwittingly spoil the assumption, or who may port the program to a different compiler and/or machine.

10.9 Use ASCII files for runtime or machine dependent constants and macros

Whenever possible, values which remain static throughout runtime but which can be used to tune or modify the program shall be read from an ASCII file at the start of runtime, to allow for dynamic modification without recompilation or relinking. Note that in some cases, the program design may specify features to support dynamic tuning or modification of some of these values (for example, machine setpoints).

10.10 Error Handling

10.11 Debugging

Rules for i.e. #define DEBUG_MODULE_NAME

10.12 Using structures and enumerators is recommended

11. Modularization and Information Hiding

11.1 Information Hiding , domain, and scope of variables

The use of information hiding is mandatory, to the extent allowed by the given language. The overall program shall be divided into various domains of interest, and different divisions of the code shall deal with different domains. The code that deals with a given domain shall protect, to the greatest extent possible in the given language, its data, its data structure design, and its internal operations on the data. It shall "export" to outside code modules only the operations required by the outside modules. The exported operations shall be presented to outside modules at a level of abstraction such that the internal implementation is not detectable.

11.2 Low Coupling , High Cohesion, and Clean interfaces

The quality of the modularization of a program depends on the linkage between modules, called coupling, and the binding within a module, called cohesion. Ideally, a module will have low coupling with other modules and a high level of cohesion with itself.[1]

It may not be easy to attain low coupling, high cohesion, and clean interfaces at the same time, but the final product will be cleaner and easier to maintain because of this extra effort.

11.3 Cohesion

Cohesion refers to the relation of the statements within a routine. There are different ways in which statements can be related. Functional cohesion, which means that the statements perform a single purpose or goal, is the strongest type of binding and the ideal to strive for.[1]

11.4 Coupling

Coupling refers to the degree to which two routines are dependent on each other, or the degree of difficulty one would have trying to change one routine without having to change the other. Data coupling would depend on the number and type of parameters passed into or out of a routine. Loose data coupling (low dependence between modules) is the ideal coupling method.[1]

11.5 Clean Interface

Clean interfaces refers to clear, standard and defined lines of communication between routines. In many languages, this is done by passing parameters. It is important to try to keep communication between procedures and functions confined to parameters, i.e. not referencing global data. If this is done and all non-output parameters are passed by value (so that their value isn't mistakenly changed

[1] Turner, R., Software Engineering Methodology, Reston Publishing Company, Inc., 1984

by a routine), then each routine will be isolated from what happens in other parts of the program. Each routine can then be tested and debugged and integration should then run smoothly.

11.6 Minimize scope of variables

Whenever allowed by the language, all constants, types, and variables shall be declared only within the scope in which they need to be known.

Data items must not be accessed or altered by some obscure process. Data should be local if at all possible. "Pass through" parameters (or "Tramp Data"), whose only function is to pass data down to called routines creates readability and maintainability problems. Each data linkage makes integration problems more probable.[2]

[2] Plum, T., C Programming Guidelines, Plum Hall, Inc., 1984