SOFTWARE DEVELOPMENT STANDARDS

Table of Contents

1. Overview

The goal of these guidelines is to create uniform coding habits among software developers so that reading, checking, and maintaining code written by different programmers becomes easier. The intent of these standards is to define a natural style and consistency yet leave source code authors with 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. This is especially true when developing GNU licensed commercial software products for sale.
- New developers can get up to speed quickly, which agencies and project teams need.
- Developers make fewer mistakes in consistent environments.
- Developers new to a language are spared making the same mistakes repeatedly, so reliability is increased.
- Developers new to a language are spared the need to develop a personal style and defend it to death. We've seen too many examples where code is written as:

```
$personal_style ===
in_array(array("spaghetti","cowboy","obfuscated",
"serpentine"));
```

- Idiosyncratic styles and college-learned behaviors are replaced with an emphasis on business concerns-high productivity, maintainability, shared authorship, etc.

Experience over many decades of professional application development points to the conclusion that coding standards help projects to run smoothly. They aren’t necessary for success, but they help. Most arguments against a particular standard come from the ego.

Few, if any, decisions in a common-sense standard are technically deficient; they are matters of taste. So, in the interests of establishing a showcase software development environment, be flexible, control the ego a bit, and remember any project is a team effort, even if you’re a team of one.

As software development practitioners, never forget that your "team" includes your clients. By following consistent standards, you're building your software legacy and delivering code that meets or exceeds their expectations.

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 maintenance costs by easing the learning or re-learning of a 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.

2. Scope

This document describes general software coding standards for Formidable Forms projects. Each language specific coding standard will be written to expand on these concepts with specific examples and define additional guidelines unique to that language.

3. Masterminds Software Development Standards License

The *Masterminds Software Development Standards* document is published under the GNU Free Documentation License. This means that the document is "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially.

Secondarily, the **GNU Free Documentation License** preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

4. Masterminds Software Development Standards Attribution

Masterminds Software Development Standards is an enterprise level Formidable/WordPress software standards document modeled after a paper written by Mike Buckley, Professor at the State University of New York at Buffalo. Buckley contributed his paper as a downloadable resource for the book *"The Ultimate Guide to the SDLC"* by Victor M. Font Jr, published 2010.

"SDLC" is an acronym for System Development Life Cycle. The paper is included on the content CD for Chapter 7–Design and Development. You may download Buckley's foundational document here: General Style and Coding Standards for Software Projects by Mike Buckley

5. Relation with Other Coding Standards

5.1. General Coding Standard

This will be used as the base document for Formidable Forms project programming languages, i.e. PHP, jQuery, CSS, HTML, and Forms

5.2. Language specific coding standards

Each language specific coding standard will be written to expand on the concepts captured in this document with specific examples and define additional guidelines unique to that language. These language standards shall supplement, rather than override, the General Coding standards as much as possible.

5.3. General 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 general coding standards and language specific coding standards. Where conflicts between standards 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.

5.4. 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, 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.

5.5. 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. Meet or exceed the expectations of your clients.

5.6. A 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 its rules, should dictate the place of standards and consistency within and across projects.

5.7. I Never Learned This in School...Is This a Joke?

- You must use some style, why not be consistent across the project?
- Individual styles are not best just because they're individual.
- Individual styles are learned in a non-business environment (school?).
- Any style becomes natural after 100,000 lines.
- Syntax-based editors can be configured to do the mundane tasks

6. Project Dependent Standards

The standards and guidelines described in this document are selected based on common coding practices and many language-specific programming standard documents collected throughout 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 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.

6.1. Naming conventions

What naming conventions should be followed? Systematic prefix conventions for functional grouping of global data and for names of structures, objects, and other data types may be useful.

1. Project specific contents of module and subroutine headers

2. File Organization:

    - What kind of Include file organization is appropriate for the projects data hierarchy
    - Directory structure
    - Location of composer.json file if using Composer
    - Note: "Actions taken before compilation or assembly is performed" should be the directory in which the source code resides, unless otherwise specified.
    - Specifications for Error Handling:
    - specifications for the detection and handling of errors
    - specifications for boundary condition checking for parameters passed to subroutines

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

4. Guidelines for the use of “lint” or other code checking programs

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

7. FILE and MODULE GUIDELINES

7.1. Module Design Guidelines

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

7.2. Include Files

If the language permits, include files are support files referenced by other files prior to compilation. Include files are also used to contain data declarations and defines that are needed by more than one program. Include files should be functionally organized, i.e. declarations for separate subsystems (classes) should be in separate include files.

Also, if a set of declarations is likely to change when ported from one machine to another, those declarations should be in a separate include file. Any source file that uses the facilities made available by another module needs to include the associated include file.

7.3. Path names for include files

Use relative (not absolute) path names for include files of source code being created for 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.

7.4. Include File Guidelines

- Include files can define global data types, constants, and macros. Multiple definitions of data types and structures should be avoided.

- While the use of global data should be generally avoided, a module that must “own” a global variable should allocate that variable. (It is generally useful to consider the one who writes to the global data as the owner.)
- The use of a single module to allocate all global data is generally discouraged.

7.5. Source File Layout Guideline

The specific contents of sections in a file may vary from language to language, but the 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 include files should be next. Any non-obvious reasons for the inclusion should be commented.
- Constant Definitions
- Data Type definitions, Function Prototypes, and Structure Templates
- Variable Definitions
- Subroutines

7.6. 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.

7.7. File Informational Headers

A File Informational Header shall be placed at the beginning of each source file. Header fields 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 Informational Header should include the following:

- Copyright Generic Company, 202x, 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

7.8. Program Unit Header

For each program unit (e.g. subroutine) 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 - the name of the program unit
- Brief description of the purpose of the unit
- Parameters: Description of each parameter passed to the function.
- Return (If applicable)
- 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.
- Programmers(s)
- 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

Other information that should be considered for each program unit header include:

- Outputs as language supports: Changes to global conditions (states, machine outputs, global data, ...) not included in the parameters passed as part of the call to the function
- Subroutines called in support of this program unit.
- Global variables written and referenced by this program unit.
- Designer(s) or Reference to Design Document(SDDD).

- Tested By/Date: (Indicates any unit testing that was done for this program unit)
- 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 one 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.

8. Subroutines

8.1. 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.

For example, Formidable Masterminds maintains a shared function code base that we use with every project. Ours is a full-blown code library. On the project level, our developers can pick and choose the functions they need to use.

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.

8.2. 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).

8.3. Subroutine Layout Guidelines:

The specific contents of sections in a subroutine may vary from language to language, but the 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”)

8.4. Subroutine Size Guideline

Subroutines shall be as long as 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.

8.5. 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 its 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).

8.6. 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.

9. Comments

9.1. "Distributed Code Description"

Note: there is no Code Description section in the Program Unit Header, even though such a section is usually a part of headers of this sort. Instead, **Distributed Code Descriptions** 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 makes the code easier to read and forms a Distributed Code Description that will be easier to verify and maintain as the code is modified.
- This also makes 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.

9.2. 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 manner that clearly distinguishes it from the code.

9.3. 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).

9.4. 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.

10. Code Layout

10.1. 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 immediately that more is coming.

10.2. Indentation Guidelines

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

- 4-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

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

Nothing brings out the territorial and protective instinct in programmers like the issue of brackets, begin..end, or any delimiter placement. People generally consider the format they learned in school correct, 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.

A STRONGLY SUGGESTED standard is that all functional code following a conditional be delimited, even a single line.

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

11.1. Summary table for Naming Convention:

This table summarizes the recommended use of underscores, upper/lower case, and Hungarian notation for naming identifiers.

Identifier	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

Identifier	Upper/lower case	Under -score	Prefixes or Suffix	Example
Types and User-type declarations (Structures, Enumerators, Unions)	MixedCase	no	Suffix (Type, Struct, Enum, and Union)	ColorType, EmployeeStruct
Variables	MixedCase	no	Hungarian prefixes	blStackIsUpdated

11.2. Select Clear and Meaningful Names

The most important consideration in naming a variable, constant, or subroutine is that the name fully and accurately describes the entity or action that the structure represents.

Clear, complete, and meaningful names make 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”.

11.2.1. Naming Subroutines (verb and object)

Names of methods and procedures shall consist of a verb and (whenever appropriate) an object, such as “Push_Input_Character”. This makes 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.

11.2.2. 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.

11.2.3. 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”.

11.2.4. 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.

11.3.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 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.

11.3.1. 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).

11.3.2. 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".

11.3.3. 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 RPMs of a motor could be named "MAX_MOTOR_RPM"

11.3.4. Acronyms:

When Acronyms are used as part of a name in a mixed case situation, the acronym should generally remain capitalized. For example, if "FFT" is used as

an abbreviation for “Fast Fourier Transform”, then a likely variable would be named “FFTName”, rather than “FftName.

The project specific standard should specify which approach is more usable on the project, as consistency in an application is more important than a hard and fast rule.

11.4.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 in the C and Visual Basic languages. For PHP, the **PHP Alternative Hungarian Notation** (or **PAHN**) is a naming convention for PHP based on Hungarian Notation, but in a more simplified format.

The use of **PAHN** 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 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 is included below.

11.4.1. PAHN Benefits

- By using a naming convention for variables that is different than functions, class methods, or class variable names, it helps make the code more readable so that you don't confuse a variable for a function call, for instance.
- It helps other programmers coming back to your project understand the intent of that variable.
- By sticking to a simple standard like PAHN that is easy to learn, it is one measure (of many) to make the code from several team members look identical. Otherwise, one team member may use one variable naming convention, while another team member may use another.
- There are not a lot of published standards for PHP-based variable naming conventions. By having one set down here, it is one opportunity to settle the issue, and to have a central document on the web to which many can refer.

- Class variables used for model objects, such as $Member, need to stand out more so than $nMember or $sMember, for instance. Using a naming convention helps improve readability for separating the two.
- Imagine we want to delineate that $sMemberID means an ID that may be alphanumeric, while $nMemberID is going to be an integer. If we do $MemberID, you don't pick up on that so easily. So, again, this naming convention improves readability.
- If we were to use $NamesArray, it's more typing than $asNames. Plus, we have no idea with $NamesArray whether it's an array of Names objects, an array of Names' strings, or what.

So, again, we can improve readability of the code by using this naming convention.

11.4.2. PAHN naming convention

The PAHN variable naming convention begins with a series of prefix characters, followed by a ProperCase variable name. Example:

```
global $gasNames;
$gasNames = $Members->getNames();
```

The $gasNames prefix (gas) means global + array + string, or global array of strings.

Prefixes

The prefixes are:

Prefix	Meaning
_	a private class variable
a+	array (often combined with the data type used inside the array)
c+	character
s+	string
o+	object
d+	date object–as in what's returned from a date() or gmdate()
v+	variant–used very infrequently to mean any kind of possible variable type
i+	integer–an integer
f+	float–a floating point number, e.g. an integer with a fractional part
n+	numeric (unknown if it's float, integer, etc. Use infrequently)

Prefix	Meaning
x+	to let other programmers know that this is a variable intended to be used by reference rather than value
rs+	db recordset (set of rows)
rw+	db row
h+	handle, as in db handle, file handle, connection handle, curl handle, socket handle, etc.
hf	handle to function, as in setRetrievalStrategy(callable $hfStrategy)
t+	a threaded object, use to indicate that an object may be safe to call\pass between threads
g+	global var (and used sparingly, and often combined with the datatype used for the variable)
b+	boolean

Some examples:

Variable	Meaning
$oMember	a Member object
$hFile	a handle to a file, for instance as passed from the fopen() statement
$cFirst	first character retrieved from a string
$rsMembers	records of Members, as returned from a database table
$rwMember	a single Member record from the database
$bUseNow	a boolean flag
$sxMemberName	a byref string variable for a name of a member.
$nCounter	a numeric counter
$dBegin	a beginning date
$sFirstName	a string to represent someone's first name
$_hDB	a private class variable to store a database connection handle (often addressed by $this->_hDB)

For class variable names, PAHN does not use this prefix. Thus, you might see something like:

```
$Members = new Members();
```

For constants, PAHN uses just an uppercase word like MEMBER, and this is often used with only inserting variables in PHP Alternative Syntax or CCAPS.

As for what comes after the prefix, it is preferred to stick with ProperCase, as in $sMemberName rather than $sMEMBERNAME, $sMember_Name, $s_Member_Name, or $smemberName.

There are several reasons for this. In the case of $sMEMBERNAME, it would imply that the variable is to be treated like a constant (where uppercasing is often seen), when it is not.

In $sMember_Name, it makes for more unnecessary typing, as does $s_Member_Name.

And $smemberName runs the s+ prefix against the word "member" and makes for a confusing variable name.

Now, saying this, there are some rare exceptions where adding an underscore does help with readability, and in those cases it can be used with PAHN. A good example of this rare exception is with an acronym like FIFO. So, $bFIFOIndicator might be more confusing than $bFIFO_Indicator and the latter would be more preferred.

There is also an exception to this prefix for variables used as loop iterators:

Many programmers may be familiar with the short variables: $a, $b, $c, $d, $i, $x, $y, $z used in many textbooks. These are great for when you want to have an iterator variable that you address in a loop and use within arrays. For example:

```
$asMemberNames = array();
for ($i = 1; $i <= 10; $i++) {
  $asMemberNames[$i-1] = $Member->getMemberByID($i);
}
```

Therefore, this exception for loop iterators is allowed because it saves time with less typing and does not reduce readability.

11.4.3. Use of prefixes for 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

Scope	Description	Prefix
Public	In Object Oriented languages, useable by all	pub
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

11.4.4. Table of Common Qualifiers

Qualifier	Description (follows Body)
First	First element of a set.
Last	Last element of a set.
Next	Next element in a set.
Prev	Previous element in a set.
Cur	Current element in a set.
Min	Minimum value in a set.
Max	Maximum value in a set.
Save	Used to preserve another variable which must be reset later.
Tmp	A "scratch" variable whose scope is highly localized within the code. The value of a Tmp variable is usually only valid across a set of contiguous statements.
Src	Source. Frequently used in comparison and transfer routines.
Dst	Destination. Often used in conjunction with Source.

11.5. 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.

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 a 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:

Word	Abbreviation
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

12. Misc. Rules for Coding

12.1. Use constants instead of hard coded literal values

"Magic numbers" (hard-coded literal values) are discouraged. They often make the author's intent unclear and make global changing of a value undependable

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.

12.1.1. Only Define constants once

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

12.2. Global Data Guidelines

- Global data is generally to be avoided. Parameters are the preferred method of communication among subroutines.
- The scope of module-level data should be limited to the module (i.e. use the **static** storage class specifier to declare functions and data local to the file, rather than declaring them as global).
- Any variable whose initial value is important should be initialized with executable code, don’t initialize static data at time of allocation.
- Performance considerations sometimes suggest the use of “static” variables in interrupt routines to avoid overuse of system resources. In general, these variables should not be global.

12.3. 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 project's design documents.
- Error recovery should be handled in the routine that is responsible for the domain in which the error occurs.

12.4. 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.

12.5. 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.

12.6. 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.

12.7. Go-to's, 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 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.

12.7.1. 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.

12.8. 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.

12.9. 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.

12.10. 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.

12.11. 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.

12.12. 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).

13. Modularization and Information Hiding

13.1. 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.

13.2. 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.

13.3. 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]

13.4. Coupling

Coupling refers to the degree to which two routines are dependent on each other, or the degree of difficulty one would have attempting 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]

13.5. Clean Interface

“Clean interface” 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.

---

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

If this is done and all non-output parameters are passed by value (so that their value isn't mistakenly changed 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.

13.6. 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 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

14. Document Revision History

Author	Revision Date	Revision Description
Victor M. Font Jr.	January 18, 2022	Initial Public Release
Victor M. Font Jr.	January 19, 2022	Corrected typo in section 11.4
Victor M. Font Jr.	January 20, 2022	1. Fixed table header rows for consistency 2. Fixed paragraph spacing for consistency 3. Added Document Revision History section