SCADA Software Coding Standard - Scripting

E354086-00000-271-050-0008

REV. 01

DOCUMENT APPROVAL PROCESS

NAME		POSITION/MEETING NO.	SIGNATURE	DATE
Originator:	Hugo Rust	SCADA Leading Engineer		15-08-2019
Approver:	Paulo De Sousa Gomes	Engineering Manager		15-08-2019
Original date: 05-12-2018				
Effective date: 15-08-2019				

DOCUMENT CHANGE HISTORY:

The owner of this document is responsible for the revision and control of the document, including updating of the table below, which contains the history of the document with details of each revision.

Date	Previous Rev No.	New Rev No.	Details of Revision
20-08-2021	00	01	As Built

This table summarises what has been changed in the document so that it is easy to keep track of the effected changes.

TABLE OF CONTENTS

INTRODUCTION

1.1 Purpose

The purpose of this document is to specify basic guidelines for creating user (customer) specific program code within SCADA display components,and as such covers recommendations for Visual Basic .NET (VB .NET) programming language.

The aim of this document is thus, to ensure all code developed for the SCADA system is consistent regardless of author, maintainable and of a high quality.

The document is mostly applicable for designing OASyS DNA displays and ACE routines.

All recommendations are based on Microsoft Standard for .NET platform and were taken from Microsoft official repository (https://docs.microsoft.com/en-us/dotnet/visual-basic/).

1.2 Scope

1.2.1 Requirements Included

The following will be included in this document:

- Naming convention for VB .NET programs
- Coding convention for VB .NET programs

This document details internal variables which are used in the scripts. Where external variables and parameters are used they will comply to the PCS Software Naming Standard [9].

1.2.2 Requirements Excluded

The following will be excluded from this document:

- Naming conventions for sites, field devices, database tags, display names.

  [Site, Plant, Equipment and Instrumentation (field devices) naming conventions are detailed in the TPL Specification PL101 [12]. Database Tag naming conventions and SCADA Display naming conventions are detailed in [9] Naming Standard. PLC software coding standard. Refer to PLC Software Coding Standard [10] for details.

- Baseline program code (goes with the product).
- Program code which is auto generated in SCADA components (i.e. display editor – XE, ACE editor).

1.3 Document Usage

In this specification,

- the word shall is to be understood as a mandatory requirement,
- the word should as a preference,
- the word may as a permissive (i.e. neither mandatory nor necessarily recommended),
- and the word will as a declaration on behalf of something/ someone else.
- The word DO and DO NOT is to be understood as recommendations, not mandatory.

1.4 Terms and Definitions

Adapter	Ethernet/IP device the Publishes data at a set RPI(Request Packet Interval) and sent as multicast messages to Scanner Devices.
Advanced Database Editor (ADE)	An OASyS DNA support and configuration program for editing the real-time database.
Application Software	The software written specifically to perform user requirements for an individual plant when standard software packages cannot be configured to meet the requirements. Application software works with the standard operating software and it does not modify any standard software
Archive	Saving measured values and messages in the operator station to history so the data can be called up over a long period of time
Availability	The probability that a system will perform its designed function when required to do so is expressed as the fraction (or percentage) of time a system or individual module remains on-line and performs as specified during an observation period. It is calculated as follows: A = MTTF/MTBF or A = MTTF/(MTTF + MTTR)
Blocks	Blocks are separate parts of a user control software configuration distinguished by their function, structure, and purpose
CFC	Continuous Function Chart is a high-level graphical language using function blocks for configuring continuous control systems
Camel Case	Naming convention that capitalised the first character of each word (without underscores). 2 types of Camel Case are defined: Upper Camel Case and Lower Camel Case / Pascal Case. A.k.a. Medial capitals. For the purpose of this document Camel Case refers to Lower Camel Case (first character small caps).
Control Panel	A standard graphic element that represents, for example, an analogue controller instrument, a hardwired push-button, or a switch, allowing operator monitoring and control of the device, and comprises on one RTDB object.
Display	Graphics which will show the information coming from the RealTime database statically or dynamically.
Engineering Server (ES)	Used for preparation and distribution of software binaries, displays and database changes to the SCADA servers.
Faceplate	A standard graphic element that represents, for example, an analogue controller instrument, a hardwired push-button, or a switch, allowing

	operator monitoring and control of the device, and comprises of multiple RTDB objects.
Fault Tolerance	The property of a system which permits it to carry out its assigned function even in the presence of one or more faults in its hardware or software components. Fault tolerance is to be achieved automatically without any user intervention
Function Block	A control bock as defined in IEC 61131-3. See also Block
HMI	The graphical interface program for allowing an operator to interact with and control a process
Instance	A copy of a function block, which is used again in the control configuration for a similar application
Ladder Logic (LAD)	Graphical representation of the automation task using relay symbols complying with DIN 19239
Lower Camel Case	Camel Case with first character small caps.
MTBF	MTBF is the expected time between failures of a system including time to repair. It is derived in its simplest form as: MTBF = MTTF + MTTR
MTTF	MTTF is the expected time to failure of a system in a population of identical systems
MTTR	MTTR is the statistical average of time taken to identify and repair a fault (including diagnostics)
Mode	Control block operational condition, such as manual, automatic, or cascade
Monitor	Physical device used to show displays.
OPC	Software applications which allow bi-directional data flow between two separate applications. These applications may be running on the same or separate servers. OPC refers to the complete OPC specification
Operator Workstation	Electronic equipment on which the HMI resides, including, at a minimum, PC workstation, a monitor, keyboard, and pointing device used by an operator to monitor and control his assigned process or manufacturing units
Operator / Controller	One who exercises central surveillance and control of the field using SCADA.
Pascal Case	Camel Case convention with the first letter capitalised. A.k.a. Upper Camel Case.
PCS	The Process Control System (PCS) refers to the complete control system required for the operation of the TPL sites from the field interface to the operator interface.
Personal Computer (PC)	A workstation or server, typically running MS-Windows when referred to in this way.
PLC	Programmable Logic Controller, used for discrete and continuous control in processing and manufacturing plants

Point	A process variable derived from an input signal or calculated in a process calculation
Process Object	A collection of variables and parameters that performs a control function (e.g. motor, block valve, PID Controller) which may consist of more than one I/O point
Plug and Play	The ability of hardware equipment to automatically identify itself to the system. When the equipment is powered up it is automatically assigned a unique identity without the need to set any dipswitches
Real-time	The inherent property of a system to distribute data such that the users of the data always have the most current data at all times.
Reliability	The probability that when operating under stated environmental conditions, the system will perform continuously, as specified, over a specific time interval
Redundant	A system/subsystem with two modules that provides automatic switchover to a backup in the event of a failure, without loss of a system function
Scanner	Ethernet/IP – A scanner device opens connections and initiates data transfers. This device is typically the subscriber of data. (See Adapter for Publisher)
Screen	Part of the monitor which is shown to arrange displays.
Sequential Function Chart (SFC)	Sequential Function Charts are a high-level graphical configuration language for sequential control applications
Statement List (STL)	Statement List is a textual programming language resembling machine code and complying with IEC 1131-3
Structured Control Language (SCL)	A high-level language complying with IEC 1131-3 and resembling Pascal for programming complex or custom logic tasks within the controller
System Bus	The network used for communication between controllers and HMI servers
System Software	The software components that are required to make the system functional and fit for purpose. System software shall include any firmware, operating software and tools that are supplied as standard items (for example configuration software, operating system and human interface configuration software). Typically, system software is configured to meet user requirements
TVDA	Tested, Validated and Documented Architecture - A guideline document published by Schneider Electric on specific applications, which have been setup, tested and validated in Schneider Electric Laboratories.
Upper Camel Case	See Pascal Case.
User Requirements	Those requirements that describe what functions the system must perform to achieve the objectives of operating the physical plant. Typically, the system is configured to meet user requirements

Works	PCS Upgrade Project which includes design, engineering, supply, installation, commissioning and handover of the Process Control, Integrated Custody Metering and Pipeline Monitoring Systems and Deployment to all Crude Oil Pipeline Stations – Main Automation Contract.

1.5 Abbreviations

ACDB	Alarm Configuration Database
API	American Petroleum Institute
AS	PL723 Automation Standard
ASCII	American Standard Code for Information Interchange
CCOTF	Change of Configuration on The Fly
CO	Co-ordinating Officer
DCS	Distributed Control System
DDDT	Device Derived Data Type
DDF	Detected Dangerous Failure
DDS	Detailed Design Specification
DDT	Derived Data Type
DIE	Diesel
DOL	Direct Online
DTM	Device Type Manager
ECP	Effluent Control Panel
EDS	Engineering Design Specification
EIO	Ethernet (Remote) IO
ePAC	Ethernet Programmable Automation Controller
ES	Engineering System
FAST	PLC Fast Task which runs periodically at a pre-determined rate measured in ms
F&G	Fire and Gas
FBD	Function Block Diagram
FC	Flow Computer
FDS	Functional Design Specification
FDT	Field Device Type
FFB	Collective term for EF, EFB and DFB
FRS	Functional Requirements Specification
HART	Highway Addressable Remote Transducer
HMI	Human Machine Interface

HSBY	Hot Standby
I/O	Input/output
IP	Industrial Protocol
IS	Intrinsically Safe
LAN	Local Area Network
MAST	PLC Master Task which runs periodically at a pre-determined rate measured in ms
MCC	Master Control Centre
MDS	Metering System
MIS	Manufacturing Information System
MMS	Machine Monitoring System
MoC	Mode of Control
MoO	Mode of Operation
MTBF	Mean Time Between Failure
MTTF	Mean Time To Failure
MTTR	Mean Time To Replacement
MV	Medium Voltage
NOC	National Operations Centre
OPC	OLE for Process Control
OS	Operating System
P&ID	Piping and Instrumentation Drawing
PCS	Process Control System
PFD	Process Flow Diagram
PID	Proportional, Integral & Derivative Controller
PLC	Programmable Logic Controller
PLC	Programmable Logic Controller
RIO	Remote Input/Outputs
RPI	Request Packet Interval
RSTP	Rapid Spanning Tree Protocol
RTU	Remote Terminal Unit
SCADA	Supervisory, Control and Data Acquisition
SCC	Secondary Control Centre
SFC	Sequential Flow Chart
SIF	Safety Instrumented Function
SIL	Safety Integrity Level
SIS	Safety Instrumented System

SNMP	Simple Network Management Protocol
SO	Station Operator
ST	Structured Text
TBA	To be Advised
TBC	To be Confirmed
TBD	To be Defined
TCP	Transmission Control Protocol
TGS	Tank Gauging System
URS	User Requirements Specification
VSD	Variable Speed Drive
WAN	Wide Area Network

APPLICABLE DOCUMENTS

All documents of the exact revision cited in the Applicable Documents form part of this specification to the extent specified. In the event of conflict between the text of this specification and the documents invoked herein, the text of this specification shall take precedence.

However, nothing in this specification supersedes applicable laws and regulations.

2.1 TPL Applicable Specifications and Standards

No. and Title		Doc. No.	Rev.
[1]	PCS Control Module Specification	E354086-00000-271-078-0005	Latest
[2]	SCADA Functional Design Specification	E3544086-00000-271-078-0018	Latest
[3]	Metering Functional Design Specification	E3544086-00000-271-078-0020	Latest
[4]	LDS Functional Design Specification	E3544086-00000-271-078-0007	Latest
[5]	SCADA/PLC Communication Plan	E354086-00000-271-078-0012	Latest
[6]	HMI Style Guide	E354086-00000-271-078-0006	Latest
[7]	Software Configuration Management Plan	E354086-00000-271-050-0002	Latest
[8]	Software Lifecycle Plan	E354086-00000-130-050-0001	Latest
[9]	PCS Software Naming Standard	E354086-00000-271-050-0006	Latest
[10]	PLC Software Coding Standard	E354086-00000-271-050-0004	Latest

2.2 Other Applicable Specifications and Standards

The following national and international standards are required to be complied with and shall be read in conjunction with this Specification.

No. and Title		Doc. No.	Rev.
[11]	Nil		

2.3 Reference Documentation

The documents included in this section do not form part of the specification, but are included for background and context.

No.	Doc. No.	Rev.
[12] TPL Plant & Equipment Numbering Standard	TPL-TECH-DO-STD-002 (PL101)	06

VISUAL BASIC PROGRAMMING CONVENTION DETAILS

Note: Where there is a conflict between the PCS Software Naming Standard [9] and this document, the PCS Software Naming Standard shall take preference over this document.

All code developed shall be based on US English only for names.

3.1 Capitalization

- **DO NOT** use underscores to differentiate words, or for that matter, anywhere in identifiers.

There are two appropriate ways to capitalize identifiers, depending on the use of the identifier:

- PascalCasing
- CamelCasing

The PascalCasing convention, used for all identifiers except parameter names, capitalizes the first character of each word (including acronyms over two letters in length), as shown in the following examples:

PropertyDescriptor

HtmlTag

A special case is made for two-letter acronyms in which both letters are capitalized, as shown in the following identifier:

IOStream

The camelCasing convention, used only for parameter names, capitalizes the first character of each word except the first word, as shown in the following examples. As the example also shows, two-letter acronyms that begin a camel-cased identifier are both lowercase:

propertyDescriptor

ioStream

htmlTag

- **DO** use PascalCasing for all public member, type, and namespace names consisting of multiple words.
- **DO** use camelCasing for parameter names.

The following table describes the capitalization rules for different types of identifiers to be used in the scripts on this project.

Identifier	Casing	Example
Namespace	Pascal	namespace System.Security

Identifier	Casing	Example
Type	Pascal	public class StreamReader
Interface	Pascal	interface IEnumerable
Method	Pascal	Public ToString() as String
Property	Pascal	ReadOnly Property QuoteForTheDay() As String
Event	Pascal	Event AnEvent(ByVal EventNumber As Integer)
Field	Pascal	Private AccountNumber As String
Enum value	Pascal	enum FileMode Append ... end enum
Parameter	Camel	public class Convert public static int ToInt32(string valueToConvert)

3.1.1 Capitalizing Compound Words and Common Terms

Most compound terms are treated as single words for purposes of capitalization.

- **DO NOT** capitalize each word in so-called closed-form compound words.

These are compound words written as a single word, such as endpoint. For the purpose of casing guidelines, treat a closed-form compound word as a single word. Use a current dictionary to determine if a compound word is written in closed form.

Pascal	Camel	Not
BitFlag	bitFlag	Bitflag
Callback	callback	CallBack
Cancelled	cancelled	Cancelled
FileName	fileName	Filename

Pascal	Camel	Not
Id	id	ID
LogOff	logOff	LogOut
LogOn	logOn	LogIn
Namespace	namespace	NameSpace
Ok	ok	OK
UserName	userName	Username

3.2 General

3.2.1 Naming Conventions

3.2.1.1 Word Choice

- **DO** choose easily readable identifier names.

For example, a property named HorizontalAlignment is more English-readable than AlignmentHorizontal.

- **DO** favor readability over brevity.

The property name CanScrollHorizontally is better than ScrollableX (an obscure reference to the X-axis).

- **DO NOT** use underscores, hyphens or any other nonalphanumeric characters.
- **DO NOT** use Hungarian notation.
- **AVOID** using identifiers that conflict with keywords of widely used programming languages.
- **DO** begin function and method names with a verb, as in InitNameArray or CloseDialog.
- **DO** begin class, structure, module, and property names with a noun, as in EmployeeName or CarAccessory.
- **DO** begin interface names with the prefix "I", followed by a noun or a noun phrase, like IComponent, or with an adjective describing the interface's behaviour, like IPersistable. Do not use the underscore, and use abbreviations sparingly, because abbreviations can cause confusion.

- **DO** begin event handler names with a noun describing the type of event followed by the "EventHandler" suffix, as in "MouseEventHandler".
- **DO** include the "EventArgs" suffix in the names of event argument classes.
- **DO** use a suffix in present or past tense if an event has a concept of "before" or "after" as in "ControlAdd" or "ControlAdded".
- **DO** use abbreviations to keep name lengths reasonable for long or frequently used terms, for example, "HTML", instead of "Hypertext Markup Language".

  In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" can lead to confusion.
- **AVOID** using names in an inner scope that are the same as names in an outer scope. Errors can result if the wrong variable is accessed.

3.2.1.2 Using Abbreviations and Acronyms

- **DO NOT** use abbreviations or contractions as part of identifier names.

For example, use GetWindow rather than GetWin.

- **DO NOT** use any acronyms that are not widely accepted, and even if they are, only when necessary.

3.2.1.3 Avoiding Language-Specific Names

- **DO** use semantically interesting names rather than language-specific keywords for type names.

For example, GetLength is a better name than GetInt.

- **DO** use a generic CLR type name, rather than a language-specific name, in the rare cases when an identifier has no semantic meaning beyond its type.

For example, a method converting to Int64 should be named ToInt64, not ToLong (because Int64 is a CLR name for the C#-specific alias long). The following table presents several base data types using the CLR type names (as well as the corresponding type names for C#, Visual Basic, and C++).

C#	Visual Basic	C++	CLR
sbyte	SByte	Char	SByte
byte	Byte	unsigned char	Byte

C#	Visual Basic	C++	CLR
short	Short	Short	Int16
ushort	UInt16	unsigned short	UInt16
int	Integer	Int	Int32
uint	UInt32	unsigned int	UInt32
long	Long	__int64	Int64
ulong	UInt64	unsigned __int64	UInt64
float	Single	Float	Single
double	Double	Double	Double
Bool	Boolean	Bool	Boolean
Char	Char	wchar_t	Char
String	String	String	String
Object	Object	Object	Object

- **DO** use a common name, such as value or item, rather than repeating the type name, in the rare cases when an identifier has no semantic meaning and the type of the parameter is not important.

3.2.2 Coding Conventions

- No compile warnings are to be introduced
- Be cautious when modifying autogenerated code

VS.NET and other tools often generate code that may be deleted and regenerated later without your direct knowledge. If you modify these sections, your additions and modifications may be deleted later. In some cases, programmer-driven modification of these sections is logical, since the modification is coupled to the autogenerated code itself.

3.2.2.1 Layout

- Insert tabs as spaces, and use smart indenting with four-space indents.
- Use only one statement per line.
- Do not use the Visual Basic line separator character (:).
- Avoid using the explicit line continuation character "_" in favour of implicit line continuation wherever the language allows it.
- Use only one declaration per line.
- However, always left-align items in a list.

Dim a As Integer

Dim b As Integer

- Add at least one blank line between method and property definitions.

3.2.2.2 Language Guidelines

3.2.2.2.1 Unsigned Data Type

- Use Integer rather than unsigned types, except where they are necessary.

3.2.2.2.2 Arrays

- Use the short syntax when you initialize arrays on the declaration line.

Use the following syntax.

Dim letters1 As String() = {"a", "b", "c"}

Do not use the following syntax.

Dim letters2() As String = New String() {"a", "b", "c"}

- Put the array designator on the type, not on the variable.

Use the following syntax:

Dim letters4 As String() = {"a", "b", "c"}

Do not use the following syntax:

Dim letters3() As String = {"a", "b", "c"}

- Use the { } syntax when you declare and initialize arrays of basic data types.

Use the following syntax:

Dim letters5 As String() = {"a", "b", "c"}

Do not use the following syntax:

```
Dim letters6(2) As String
letters6(0) = "a"
letters6(1) = "b"
letters6(2) = "c"
```

3.2.2.2.3 Use the With Keyword

When you make a series of calls to one object, consider using the With keyword:

```
With orderLog
  .Log = "Application"
  .Source = "Application Name"
  .MachineName = "Computer Name"
End With
```

3.2.2.2.4 Use the IsNot Keyword

Use the IsNot keyword instead of Not...Is Nothing.

3.2.2.2.5 Event Handling

- Use Handles rather than AddHandler:

```
Private Sub ToolStripMenuItem1Click() Handles ToolStripMenuItem1.Click
End Sub
```

- Use AddressOf, and do not instantiate the delegate explicitly:

```
Dim closeItem As New ToolStripMenuItem(
    "Close", Nothing, AddressOf ToolStripMenuItem1Click)
Me.MainMenuStrip.Items.Add(closeItem)
```

- When you define an event, use the short syntax, and let the compiler define the delegate:

```
Public Event SampleEvent As EventHandler(Of SampleEventArgs)
' or
Public Event SampleEvent(ByVal source As Object, ByVal e As SampleEventArgs)
```

- Do not verify whether an event is Nothing (null) before you call the RaiseEvent method. RaiseEvent checks for Nothing before it raises the event.

3.2.2.2.6 Using Shared Members

Call Shared members by using the class name, not from an instance variable.

3.3 Assemblies

An assembly is the unit of deployment and identity for managed code programs. Although assemblies can span one or more files, typically an assembly maps one-to-one with a DLL.

3.3.1 Naming Conventions

- **DO** choose names for your assembly DLLs that suggest large chunks of functionality, such as System.Data.

Assembly and DLL names don't have to correspond to namespace names, but it is reasonable to follow the namespace name when naming assemblies. A good rule of thumb is to name the DLL based on the common prefix of the namespaces contained in the assembly. For example, an assembly with two namespaces, MyCompany.MyTechnology.FirstFeature and MyCompany.MyTechnology.SecondFeature, could be called MyCompany.MyTechnology.dll.

- **CONSIDER** naming DLLs according to the following pattern:

<Company>.<Component>.dll

where <Component> contains one or more dot-separated clauses. For example:

Litware.Controls.dll.

3.3.2 Coding Conventions

- An assembly shall contain classes defined within one namespace only
- Each assembly shall be strongly signed

3.4 Classes, Structs, and Interfaces

3.4.1 Classes

3.4.1.1 Naming Conventions

- **DO** name classes with nouns or noun phrases, using PascalCasing.

This distinguishes type names from methods, which are named with verb phrases.

- **DO NOT** give class names a prefix (e.g., "C").
- **CONSIDER** ending the name of derived classes with the name of the base class.

This is very readable and explains the relationship clearly. Some examples of this in code are: ArgumentOutOfRangeException, which is a kind of Exception, and SerializableAttribute, which is a kind of Attribute. However, it is important to use reasonable judgment in applying this guideline; for example, the Button class is a kind of Control event, although Control doesn't appear in its name.

3.4.1.2 Constructors

There are two kinds of constructors: type constructors and instance constructors.

Type constructors are static and are run by the CLR before the type is used. Instance constructors run when an instance of a type is created.

Type constructors cannot take any parameters. Instance constructors can. Instance constructors that don't take any parameters are often called default constructors.

Constructors are the most natural way to create instances of a type. Most developers will search and try to use a constructor before they consider alternative ways of creating instances (such as factory methods).

- **CONSIDER** providing simple, ideally default, constructors.

A simple constructor has a very small number of parameters, and all parameters are primitives or enums. Such simple constructors increase usability of the framework.

- **CONSIDER** using a static factory method instead of a constructor if the semantics of the desired operation do not map directly to the construction of a new instance, or if following the constructor design guidelines feels unnatural.
- **DO** use constructor parameters as shortcuts for setting main properties.

There should be no difference in semantics between using the empty constructor followed by some property sets and using a constructor with multiple arguments.

- **DO** use the same name for constructor parameters and a property if the constructor parameters are used to simply set the property.

The only difference between such parameters and the properties should be casing.

- **DO** minimal work in the constructor.

Constructors should not do much work other than capture the constructor parameters. The cost of any other processing should be delayed until required.

- **DO** throw exceptions from instance constructors, if appropriate.
- **DO** explicitly declare the public default constructor in classes, if such a constructor is required.

If you don't explicitly declare any constructors on a type, many languages will automatically add a public default constructor (Abstract classes get a protected constructor.)

Adding a parameterized constructor to a class prevents the compiler from adding the default constructor. This often causes accidental breaking changes.

- **AVOID** explicitly defining default constructors on structs.

This makes array creation faster, because if the default constructor is not defined, it does not have to be run on every slot in the array. Note that many compilers, including C#, don't allow structs to have parameterless constructors for this reason.

- **AVOID** calling virtual members on an object inside its constructor.

Calling a virtual member will cause the most derived override to be called, even if the constructor of the most derived type has not been fully run yet.

3.4.1.2.1 Type Constructor Guidelines

- **DO** make static constructors private.

A static constructor, also called a class constructor, is used to initialize a type. The CLR calls the static constructor before the first instance of the type is created or any static members on that type are called. The user has no control over when the static constructor is called. If a static constructor is not private, it can be called by code other than the CLR. Depending on the operations performed in the constructor, this can cause unexpected behavior. The C# compiler forces static constructors to be private.

- **DO NOT** throw exceptions from static constructors.

If an exception is thrown from a type constructor, the type is not usable in the current application domain.

- **CONSIDER** initializing static fields inline rather than explicitly using static constructors, because the runtime is able to optimize the performance of types that don't have an explicitly defined static constructor.

3.4.2 Structs

3.4.2.1 Naming Conventions

- **DO** name structs with nouns or noun phrases, using PascalCasing.

This distinguishes type names from methods, which are named with verb phrases.

3.4.3 Enumerations

3.4.3.1 Naming Conventions

- **DO NOT** use a suffixes "Enum", "Flag" or "Flags" in enum type names.
- **DO NOT** use a prefix on enumeration value names (e.g., "ad_rtf" : ""ad" for ADO enums, "rtf" for rich text enums, etc.).
- **DO** use singular names for basic enumerations. For example:

```
Enum Protocol
        Tcp
        Udp
        Http
        Ftp
End Enum
```

- Use plural names for enumerations representing bitfields. For example:

```
Enum EggSizes
        Jumbo
        ExtraLarge
        Large
        Medium
        Small
End Enum
```

- Enumerations shall be Pascal case. For example:

```
Enum SearchOptions
        CaseInsensitive
        WholeWordOnlyEnd
Enum
```

3.4.3.2 Coding Conventions

- Enumeration numeric values shall not be explicitly assigned. For example:

```
enum LicensePlateType
        Vanity = 0              'BAD
        AssignedByDMV           'OK
End Enum
```

Exception: An enumeration may need explicit values assigned to interact with other external subsystems.

Here is an example where enumeration values *must* be explicitly defined in order to match externally defined color-masking conventions:

```
enum RGBColorMask
        Red = 1
        Green = 2
        Blue = 4
        ...
End Enum
```

- Enumerations that are bitmasks shall use the <*Flags*> attribute

```
<Flags()> Public Enum FilePermissions
    None
    Create
    Read
    Update
    Delete
End Enum
```

- Enumerations that are bitmasks shall assign enumeration values increasing powers of 2

```
<Flags()> Public Enum FilePermissions
    None = 0
```

```
    Create = 1
    Read = 2
    Update = 4
    Delete = 8
End Enum
```

- Avoid specifying a type for an enumeration

Type may be specified if the bitmask requires more than 32 bits. If type is specified beyond the default (of a 32 bit integer), then favour Int64, since this will align with word boundaries and gain efficiency over other byte size types (E.g. byte).

```
Enum MyEnum As Byte
...
End Enum
```

- Do not use an enum for open sets

For example: Operating system version.

3.5 Type Members

3.5.1 Methods

3.5.1.1 Naming Conventions

Because methods are the means of taking action, the design guidelines require that method names be verbs or verb phrases. Following this guideline also serves to distinguish method names from property and type names, which are noun or adjective phrases.

- **DO** give methods names that are verbs or verb phrases.
- Method names shall be Pascal case. For example,

public TickleMeTimbers(numberOfTimbers as Integer)

- Method arguments shall be Camel case. For example:

protected MyFunc(redIsMyFavouriteColour as Boolean)

- Name methods using a verb-object pair. For example*:*

ShowDialog()

3.5.1.2 Coding Conventions

- All variants of an overload method shall be used for the same purpose and have similar behaviour
- Use consistent names for parameters in method overloads

public class Sunglasses
… WearGlasses(size as Integer)
… WearGlasses(size as Integer, atNight as Boolean, duringDay as Boolean)
*… WearGlasses(**sizeOfHead** as Integer**, night** as Boolean, **day** as Boolean, **pool** as Boolean) 'BAD*

- Use consistent ordering for parameters in method overloads

public class Sunglasses
… WearGlasses(size as Integer)
… WearGlasses(size as Integer, atNight as Boolean, duringDay as Boolean)
*… WearGlasses(**atNight as Boolean, size as Integer,** duringDay as Boolean, atThePool as Boolean) 'BAD*

- Do not return error codes (boolean or enumeration) to indicate exceptional conditions. Use exceptions for this purpose.

```
public class Sunglasses

    public sub WearGlasses()
        success as Boolean = true
        …
        return success;          'BAD
    end sub
end Class
```

```
public class Sunglasses
    public sub WearGlasses()
        …
        if failed = true then
            throw new DNAException("Glasses don't fit") 'GOOD
        end if
    end sub
end Class
```

- Methods shall be kept as short as possible

A good rule of thumb is that each method shall be visible in its entirety in the edit window. Each method shall have only one functional goal, and produce as few side effects (i.e. changing the state of the class) as possible.

- Use *ByRef* on value-type arguments only when you intend on modifying the callers copy

sub CalculateOffset(ByRef size as Integer)

- Use a property rather than a method when the member is a logical data member

- Methods with multiple parameters shall either be visible on a single line, or if multiple lines are required, then each parameter shall be on a separate line. For example:

```
virtual sub ComplicatedAction(modeValue as Integer, filename as String)          'OK
End sub
virtual sub ComplicatedAction(modeValue as Integer, string fileName, _
        numberOfFiles as Integer, fondMemoryOfChildhood as Boolean)              'BAD
End sub
virtual sub ComplicatedAction(                                                   'BETTER
        modeValue as Integer
        filename as String
        numberOfFiles as Integer
)
...
End sub
```

3.5.2 Properties

3.5.2.1 Naming Conventions

Unlike other members, properties should be given noun phrase or adjective names. That is because a property refers to data, and the name of the property reflects that. PascalCasing is always used for property names.

- **DO** name properties using a noun, noun phrase, or adjective.
- **DO NOT** have properties that match the name of "Get" methods as in the following example:

```
public property TextWriter { get {...} set {...} }
public property GetTextWriter(value as Integer) { ... }
```

This pattern typically indicates that the property should really be a method.

- **DO** name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by "List" or "Collection."
- **DO** name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with "Is," "Can," or "Has," but only where it adds value.
- **CONSIDER** giving a property the same name as its type.

For example, the following property correctly gets and sets an enum value named Color, so the property is named Color:

```
public enum Color {...}
public class Control {
    public Color Color { get {...} set {...} }
}
```

- Property names shall be Pascal case. For example:

IsMySandwichGood as Boolean

- Boolean property names shall start with the word ***Is.*** For example:

```
Property IsMySandwichGood() as Boolean
        Get
                return isMySandwichGood;
        End Get
End Property
```

3.5.2.2 Coding Conventions

- Allow properties to be set in any order
- Avoid writing properties that result in extensive side effects

If a property begins to cause side effects, consider making it a method.

- Consider providing property-changed events

Consider if users of your class will want to know if a given property has changed. If so, then property change events should be created.

If provided, ensure that the event is named *Property*Changed where *Property* is replaced with the name of the property that changed. Refer to CS_DEN_006 for further details on naming.

- The following conditions identify when a method should be used instead of a property
  - The operation is a conversion (e.g. Object.ToString())
  - The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
  - Obtaining the property using a get accessor would have an observable side effect.
  - Calling the member twice in succession produces different results.
  - The order of execution is important.
  - The member is static, but returns a value that can be changed.
  - The member returns a copy of an internal array or other reference type.
  - Only a set accessor will be supplied. Write-only properties tend to be confusing.

    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ cpgenref/ html/ cpconpropertyusageguidelines.asp
- Consider implementing *ISupportInitialize* if multiple related properties are supported

If you have related properties, such as **DataSource** and **DataMember**, you should consider implementing the *ISupportInitialize Interface*. This will allow the designer (or user) to call the *ISupportInitialize.BeginInit* and *ISupportInitialize.EndInit* methods when setting multiple properties to allow the component to provide optimizations.

3.5.3 Delegates and Events

3.5.3.1 Naming Conventions

Events always refer to some action, either one that is happening or one that has occurred. Therefore, as with methods, events are named with verbs, and verb tense is used to indicate the time when the event is raised.

- **DO** name events with a verb or a verb phrase.

Examples include Clicked, Painting, DroppedDown, and so on.

- **DO** give events names with a concept of before and after, using the present and past tenses.

For example, a close event that is raised before a window is closed would be called Closing, and one that is raised after the window is closed would be called Closed.

- **DO NOT** use "Before" or "After" prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.
- **DO** name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

Delegate Sub ClickedEventHandler(e as ClickedEventArgs)

- **DO** use two parameters named sender and e in event handlers.

The sender parameter represents the object that raised the event. The sender parameter is typically of type object, even if it is possible to employ a more specific type.

- **DO** name event argument classes with the "EventArgs" suffix.
- Delegate definitions used as events shall be suffixed with the string ***EventHandler***

For example, the ***CloseEventHandler*** below is used to define the signature that will be used for the Close event:

*public delegate **CloseEventHandler**(sender as object, args as EventArgs)*
public event Close as CloseEventHandler 'an instance of CloseEventHandler

- Event ***instances*** shall not be suffixed. For example, the ***Close*** event below is not suffixed:

public delegate CloseEventHandler(sender as object, args as EventArgs)
*public event **Close** as CloseEventHandler 'an instance of CloseEventHandler*

- Delegates that are not used for events shall be suffixed with the string ***Callback***

For example, the ***AsyncIOFinishedCallback*** delegate below is not used in any event definition:

*public delegate AsyncIOFinished**Callback**(client as IpcClient)*

- Methods that are called via a delegate that is ***not*** related to an event should ***not*** have any suffix

Users of such a method may call it through any means (via a delegate, or directly), and any suffix chosen for this method is likely to not express all future use cases of the method. For example:

```
public sub WorkCompleted(client as IpcClient)
end sub

public sub GetWorkCompletedDelegate() as WorkCompletedCallback
        return new AsyncIOFinishedCallback(WorkCompleted)        'hand back delegate
end sub

public sub DoStuffDirectly()
        WorkCompleted(client);          'direct call
End sub
```

- Use a verb for naming an event or delegate
- Property Change events shall be suffixed with *Changed*

Events that notify consumers of changed properties should be named <PropertyName>Changed. Protected helper methods that raise these events should be named "Raise<PropertyName>Changed".

- Methods that are called via an event handler shall be prefixed with the variable instance name, followed by an underscore, followed by the event name

This is the standard that the VS.NET environment uses when it autogenerates code.

For example:

```
public class ScoobyDoo
        public delegate sub JumpEventHandler(heightInMeters as Integer, yelp as Boolean)
        public event Jump  as JumpEventHandler

        public sub EnterHauntedHouse()
                Jump(10, true )                 'Raise the event
        End sub
End Class

public class Scrappy
        public sub Scrappy (sdoo as ScoobyDoo)
                'Register for the Event
                sdoo.Jump += new ScoobyDoo.JumpEventHandler(sdoo_Jump);
        end sub

        'We are called when the event is raised
        private sub sdoo_Jump(heightInMeters as integer, yelp as boolean)
                MessageBox.Show( String.Format("Scooby jumped {0} meters",
                        heightInMeters))
        end sub
End class
```

```
static sub Main()
        scooby  as ScoobyDoo = new ScoobyDoo()
        ScrappyscrappyToo = new Scrappy(scooby)
        scooby.EnterHauntedHouse()
end sub
```

3.5.3.2 Coding Conventions

- **DO NOT** explicitly qualify the arguments (Object and EventArgs) to event handlers. If you are not using the event arguments that are passed to an event (for example, sender as Object, e as EventArgs), use relaxed delegates, and leave out the event arguments in your code:

```
Public Sub Form1_Load() Handles Form1.Load
End Sub
```

- When raising events within a base class designed for inheritance, design the parent class to allow children to intercept the event

That is, if you are designing a class that:

- Is designed for inheritance (you are developing a class that is ***not*** sealed)
- Raises an event

Then you must design the base class to allow any future child class to intercept the event. While a child class could register to receive the event on a private method, this overhead can be avoided with the following pattern:

- Use a **protected** virtual method to raise each event.
- The name of the method takes the form *On<EventName>*, where *<EventName>* is the name of the event being raised.

The purpose of the protected method is to provide a way for a derived class to intercept the event using a simple method override.

For example:

```
public class Button
        public MouseUp as MouseUpEventHandle
        protected sub DoMouseUp(x as integer, y as integer)
                try
                        'Raise the event to any inheriting classes
                        OnMouseUp(new MouseEventArgs(x,y))
                catch
                finally
                End try
        protected virtual sub OnMouseUp(e  as MouseEventArgs)
                if (MouseUp != null) then
                        'Raise the event to any subscribers
                        MouseUp(this, e)
                End if
```

```
        End sub
End Class
public sealed class SpecialButton : Button
        protected override sub OnMouseUp(e as MouseEventArgs)
                'Our parent class is telling us that the mouse
                'is now up.  Do special processing!
                ...
                'Finally, call the parent OnMouseUp who will raise the
                'the event to externally registered classes.
                base.OnMouseUp(e);
        end sub
End Class
```

Note that any derived class can choose not to call the base class during the processing of *OnMouseUp* (see the override for *OnMouseUp* in this example).

Be prepared for this by not including any processing in the *On<EventName>* method that is required for the base class to work correctly.

- Copy delegates to local variables before publishing to avoid concurrency race conditions

This is required because there is a race condition whereby a caller may be deregistering for the event at the same time that the event is about to be raised. See example below in CS_DAE_003.

- Always check an event for null before invoking it

For example:

```
public class MySource
        private event MyEvent as EventHandler
        public sub FireEvent()
                temp as EventHandler = MyEvent          ' CRITICAL
                if (temp != null)                       ' CRITICAL
                        temp( this, EventArgs.Empty)
                End if
        End sub
End class
```

- When you need to limit the number of registered subscribers to an event, use event accessors
- **DO NOT** force all events to have the same parameters (i.e. object sender, EventArgs args)

While Microsoft may advocate this standard signature to reduce namespace pollution in a WinForms environment, it reduces the type safety of the arguments at runtime. It also complicates the code of the receiver of the event.

By contrast, using a signature that properly matches the true purpose of your event ensures compile time verification for both the event caller and the event receiver. For example:

```
// Good
public class ScoobyDoo
```

```
        public delegate sub JumpEventHandler(heightInMeters as integer, yelp as
boolean)
        public event Jump as JumpEventHandler

// Bad
public class ScoobyDoo
        public delegate sub JumpEventHandler(sender as object, args as EventArgs)
        public event Jump as JumpEventHandler
```

Exception: EventArgs is a universal standard in handling events within the WinForms universe, and therefore use of this event signature will be beneficial in these environments.

- If you choose to define a new event using *EventArgs*, then pass an object instance that inherits from *EventArgs*

For example:

```
public delegate sub MouseUpEventHandler(sender as object, e as MouseEventArgs)

public class MouseEventArgs of EventArgs
        int x;
        int y;

        public sub MouseEventArgs(ix as integer, y as integer)
                this.x = x;
                this.y = y;
        end sub

        public sub X as Integer
                get
                        return x
                end get
        end sub

        public sub Y as Integer
                get
                        return y
                end get
        end sub

public class Button
        public event MouseUp as MouseUpEventHandler
        …
```

- Objects that raise events should wrap the event in a *try/catch/finally*

Consumers of events can contain any code, including code that calls into the object raising the event. Because of this, objects that raise events should include a try-finally block that ensures that the object is returned to a known state after the event is raised.

- Use or extend the *System.ComponentModel.CancelEventArgs* class to allow the developer to control the events of an object.

For example, the *TreeView* control raises a *BeforeLabelEdit* when the user is about to edit a node label. The following code example illustrates how a developer can use this event to prevent a node from being edited.

```
public class Form1 as Form
        treeView1 as TreeView = new TreeView()
        sub treeView1_BeforeLabelEdit(source as object, e as NodeLabelEditEventArgs)
                e.CancelEdit = true
        end sub
End class
```

Note that in this case, no error is generated to the user. The label is read-only.

Cancel events are not appropriate in cases where the developer would cancel the operation and return an exception. In these cases, you should raise an exception inside of the event handler in order to cancel. For example, the user might want to write validation logic in an edit control as shown.

```
public class Form1 as Form
        edit1 as EditBox = new EditBox()
        sub TextChanging(source as object, e as EventArgs)
                throw new RuntimeException("Invalid edit")
        end sub
End class
```

See:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconconstructorusageguidelines.asp

3.5.4 Fields

3.5.4.1 Naming Conventions

The field-naming guidelines apply to static public and protected fields. Internal and private fields are not covered by guidelines, and public or protected instance fields are not allowed by the member design guidelines.

- **DO** use PascalCasing in field names.
- **DO** name fields using a noun, noun phrase, or adjective.
- **DO NOT** use a prefix for field names.

For example, do not use "g_" or "s_" to indicate static fields.

3.6 Exceptions

3.6.1 Naming Conventions

- Add the ***Exception*** suffix to all custom exception classes

*public class **PubSubException** : DNAException*

{

3.6.2 Coding Conventions

3.6.2.1.1 Use the Try...Catch and Using Statements when you use Exception Handling

- **DO NOT** use On Error Goto.
- Only throw exceptions in exceptional situations

DO NOT throw exceptions in normal or expected conditions (e.g. end-of-file); return booleans or enumerations instead.

- Only catch and re-throw an exception when you want to specialize or log the exception

That is, if you intend to add additional information, change the type of the exception to a more specific exception, or write a message describing the exception to the log.

```
' Set up structured error handling.
Try
    ' Do something.
Catch e As System.IO.IOException
    ' Code that reacts to IOException.
DNALog.DumpNonFatalExceptionToLog(e)
Catch e As NullReferenceException
    ' Code that reacts to NullReferenceException.
    DNALog.DumpNonFatalExceptionToLog(e)
Catch e As Exception
    ' Code that reacts to any other exception.
    DNALog.DumpNonFatalExceptionToLog(e);
Finally
    ' This line executes whether or not the exception occurs.
    DNALog.DumpNonFatalExceptionToLog("in Finally block");
End Try
```

- Describe the recoverable exceptions using the <exception> XML documentation tag

Explicit exceptions are the ones that the method or property explicitly throws from its implementation and which users are allowed to catch. Exceptions thrown by the .NET framework classes and methods used by this implementation do not need to be documented.

```
''' <exception cref="System.OverflowException">
''' Thrown when <paramref name="denominator"/><c> = 0</c>.
''' </exception>
Public Function IntDivide(
    ByVal numerator As Integer,
    ByVal denominator As Integer
```

```
) As Integer
    Return numerator | denominator
End Function
```

- Use standard exceptions where appropriate over custom exceptions.

For a complete list of standard exceptions, refer to the documentation on *System.Event*. Here is a partial list of exceptions that are typically of interest:

IndexOutOfRangeException	Indexing an array or indexable collection outside its valid range.
InvalidOperationException	An action is performed which is not valid considering the object's current state.
NotSupportedException	An action is performed which is not supported today.
ArgumentException	An incorrect argument is specified.
ArgumentNullException	A null reference is supplied to an argument that does not allow null.
ArgumentOutsideOfRangeException	An argument is not within the desired range.

- Throw informational exceptions

That is, set the exception *Message* property to help the caller diagnose the problem.

- Localize the event description string
- Throw the most specific exception possible

3.7 Flow Control

- **DO NOT** change a loop variable inside a *for* statement

```
For index As Integer = 1 To 5

Index = index + 1 ' BAD
Next
```

3.8 Variables

3.8.1 Naming Conventions

- Local instance variables shall be camel case

That is, starting with a lower case letter, and words within the name shall be separated by a capital letter. For example:

Dim velocityMetersPerSec As Integer

- Private class variables shall be Camel case

For example:

```
Class Class1
    ' Define a local variable to store the property value.
    Private propertyValue As String
    ' Define the property.
End Class
```

Exception: Autogenerated code is not required to follow this pattern. For example, Windows Form Controls placed on a form will autogenerate class variables based on the instance name of the visual control.

- Constant variable names shall be all upper case

Words in the constant can be separated by an underscore

For example:

```
Public Const  MAX_BUFFER_SIZE_KB as Integer = 1024
```

- Underscores shall not be used in the name of any type, other than constants and event handlers

3.8.2 Coding Conventions

- **DO** use private *const*, or public *readonly*, instead of embedding numeric values into the code

Assign any numbers to a constant variable that describes its meaning, and then use the constant within the code. For example:

```
Public Const SERVER_NOT_AVAILABLE_ERRNO as Integer = 1
        ...
        if errNo = SERVER_NOT_AVAILABLE_ERRNO then
        .
```

Exceptions:

The values 0, 1, and null can nearly always be used safely.

Often, 2 and –1 can fall into this category.

Strings intended for the oasErrLog are exempt from this rule.

Literals are allowed when their meaning is clear from the context, and not subject to future changes. For example:

```
mean = (a + b) / 2  'this will always be divided by 2
```

- Each variable is to be declared in a separate declaration statement

For example:

```
Dim velocityMetersPerSec As Integer      'GOOD
```

```
Dim numberOfPeople As Integer          'GOOD
Dim velocityMetersPerSec, totalTimeSecs, meterLimit As Integer 'BAD
```

- Local variables shall be defined within the lowest scope that they are used

Declare the variable just prior to use. For example:

```
DIM success as BOOLEAN = IsRequestComplete()
if success = true then

        Dim myElmo as Elmo      = new Elmo()
        Dim counter as Integer  = 0
        while counter < maxItems

                ...
                Dim myScooby as ScoobyDoo     = new ScoobyDoo()
                myScooby.Jump()
                elmo.Giggle()
        End While
End If
```

NOTE: For those reference objects that may be allocated inside the loop, consider whether they can be constructed outside of the loop and have the single instance reused.

For example, here an instance of myScooby is allocated on each iteration, but elmo is reused.

- If possible, initialize variables at the point of declaration

NOTE: If you use field initialization for a class (i.e. giving a value at the point of the field declaration), then instance fields will be initialized before the instance constructor is called.

- Use properties to expose class variables instead of marking class variables as public or protected

Exception: Constants may be exposed publicly.

3.9 OASyS infrastructure reuse

- **DO NOT** duplicate the functionality within the OASySDNA assemblies

Be aware of the capabilities within the OASySDNA infrastructure assemblies. This can be learned through the documentation found on each machine that has OASySDNA Infrastructure installed within the following helpfile:

Telvent | DNA | Documentation | Helpfiles | OASySDNADotNet.chm

- Make use of the DNALog for outputting log messages, and use the appropriate error level for the message. Do not output messages directly with System.Console or direct file I/O

3.10 Debugging instrumentation

- **DO** favour the DNALog class over System.Diagnostics.Debug and System.Diagnostics.Trace

DNALog enables logging that can be toggled during runtime without restarting your process. It can produce a stack trace, or a simple source file and line number display. It can write to a common log file, or a process specific file.

Unlike the Debug and Trace classes, DNALog not depend on any switches or compile target to be effective (e.g. DEBUG vs. RELEASE, or the Trace macro).

Therefore, you can instrument your code with *DNALog* calls, and have the capability to debug the product even in a running production environment, without restarting your process.

Finally, *DNALog* has been tested against the performance constraints of a control system, whereas Debug and Trace have not.

3.11 User friendliness

- Operator error and information messages that uses standard OASyS message boxes must be user-friendly
- **DO NOT** suggest action with an error message, unless the way to correct the error is obvious
- **DO** log appropriate messages for error and warning conditions, and for other useful information

Also make liberal use of DNALog using the LogLevel enumeration values *Warning, Info,* and *Verbose*

3.12 Comments and emdedded documentation

As you read the code examples, you often encounter the comment symbol ('). This symbol tells the Visual Basic compiler to ignore the text following it, or the *comment*. Comments are brief explanatory notes added to code for the benefit of those reading it.

It is good programming practice to begin all procedures with a brief comment describing the functional characteristics of the procedure (what it does). This is for your own benefit and the benefit of anyone else who examines the code. You should separate the implementation details (how the procedure does it) from comments that describe the functional characteristics. When you include implementation details in the description, remember to update them when you update the function.

Comments can follow a statement on the same line, or occupy an entire line. Both are illustrated in the following code.

```
' This is a comment beginning at the left edge of the screen.

text1.Text = "Hi!"   ' This is an inline comment.
```

If your comment requires more than one line, use the comment symbol on each line, as the following example illustrates.

```
' This comment is too long to fit on a single line, so we break
' it into two lines. Some comments might need three or more lines.
```

3.12.1 Commenting Guidelines

The following table provides general guidelines for what types of comments can precede a section of developed custom code.

Purpose	Describes what the procedure does (not how it does it)
Inputs	Specifies the purpose of the argument
Returns	Explains the values returned by the procedure

TPL Requirement:

Each file shall contain a header block. Each header block should contain a short summary of the file (or display), file name (per Naming convention document), and audit of the file per example below.

```
#region "Header"
'Filename: displayName.XEM
'Description: This display indicates LP Manifold subsystem for Fynnlands station
'Version Number: XXX
'Change history
'-------------------------------------------------------------------------------------------------------
' VersionNum    Date          Name                Proj    Description
        ChangeSet(marker)
'-------------------------------------------------------------------------------------------------------
XXX'          06-Dec-18     FirstName LastName  TPL     describe changes      TPL-
123
'
'ChangeSet(marker)- unique identificatory of a change which marking the specific changes
inside the file and, if required, is used for tracing changes in Management of Change System
#endregion
```

Remember the following points:

- Every important variable declaration should be preceded by a comment describing the use of the variable being declared.
- Variables, controls, and procedures should be named clearly enough that commenting is needed only for complex implementation details.
- Comments cannot follow a line-continuation sequence on the same line.

- Put comments on a separate line instead of at the end of a line of code.
- Start comment text with an uppercase letter, and end comment text with a period.
- Insert one space between the comment delimiter (') and the comment text e.g. ' Here is a comment.
- Do not surround comments with formatted blocks of asterisks.