**C# Coding Guidelines**

# Revision History:

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Owner | Change Description |
| 1.0 | May 21, 2009 | v-mengzh | Document creation |
| 1.1 | Sep. 17 2009 | sisun | Update content |
| 1.2 | Nov. 25 2009 | sisun | Update content according to sprint2 postmortem |
|  |  |  |  |

**Table of Contents:**

1. [Introduction 4](#_bookmark0)
2. [Code Organization and Style 5](#_bookmark1)
   1. [File Layout 5](#_bookmark2)
   2. [Class Definition Order 5](#_bookmark3)
   3. [Indentation 6](#_bookmark4)
3. [Comments 7](#_bookmark5)
   1. [Comments for interface 7](#_bookmark6)
   2. [Comments for class 7](#_bookmark7)
   3. [Comments for method 7](#_bookmark8)
4. [Naming 10](#_bookmark9)
   1. [Case Sensitivity 10](#_bookmark10)
   2. [Capitalization Styles 11](#_bookmark11)
      1. [Pascal Casing 11](#_bookmark12)
      2. [Camel Casing 11](#_bookmark13)
      3. Upper case Error! Bookmark not defined.
      4. [Capitalization summary 11](#_bookmark14)
   3. [Word Choice 12](#_bookmark15)
   4. [Namespaces 12](#_bookmark16)
   5. [Class and Class members 12](#_bookmark17)
      1. [Class Naming Guidelines 12](#_bookmark18)
      2. [Interface Naming Guidelines 13](#_bookmark19)
      3. [Enum Naming Guidelines 13](#_bookmark20)
      4. [ReadOnly and Const Field Names 14](#_bookmark21)
      5. [Parameter Names 14](#_bookmark22)
      6. [Method Naming Guidelines 15](#_bookmark23)
      7. [Property Naming Guidelines 15](#_bookmark24)
      8. [Event Naming Guidelines 15](#_bookmark25)
5. [Class Members 18](#_bookmark26)
   1. [Variable Usage Guidelines 18](#_bookmark27)
   2. [Property Usage Guidelines 18](#_bookmark28)
      1. [Read-Only and Write-Only Properties 19](#_bookmark29)
      2. [Indexed Property Usage 19](#_bookmark30)
   3. [Event Usage Guidelines 19](#_bookmark31)
   4. [Constructor and Destructor Usage Guidelines 20](#_bookmark32)

# Introduction

Coding standards are important because they lead to greater consistency within your code and the code of your teammates. Greater consistency leads to code that is easier to understand, which in turn means it is easier to develop and to maintain. This reduces the overall cost of the applications that you create.

This document is an abridged version of C sharp coding guidelines. The aim of this document is to define a set of general rules that will enable to standardize the developments carried out by different people, and in different type of applications, so that we can guarantee to the maximum the legibility, reliability, re-usability and homogeneity of our developments.

# Code Organization and Style

## *Clarity and Constency*

* ***MUST*** ensure that clarity, readability and transparency are paramount. These coding standards strive to ensure that the resultant code is easy to understand and maintain, but nothing beats fundamentally clear, concise, self-documenting code.
* ***MUST*** ensure that when applying these coding standards that they are applied consistently.

## *Formatting and Style*

** *MUST NOT*** use tabs. It's generally accepted across Microsoft that tabs shouldn't be used in source files - different text editors use different spacing to render tabs, and this causes formatting confusion. All code should be written using four spaces for indentation.

Most source code editors have an option to uses spaces instead of tabs. You can also turn on 'visible whitespace' to show whitespace as characters - in Visual Studio you can enable this using the "EditAdvancedView Whitespace" option. In WinDiff you can use "ViewShow Whitespace".

* ***MUST*** limit the length of lines of code. Having overly long lines inhibits the readability of code. Some standards recommend limiting lines to 80 characters, but since displays have improved over the years and more text can fit on screen and remain readable, limiting lines to 120 or 140 characters is fine.

## *File Layout*

Only one public class is allowed per file.

The file name is derived from the class name. Example: Class: Observer

Filename: Observer.cs

C# files must have content structured in the following order:

* + - File comment header
    - Using statements
    - Namespace definition
    - User defined type definitions (enums and structs)
    - Class definition

## *Class Definition Order*

The class members must be ordered by category, and for the same category they are ordered by scope. From less restricted scope (public) to more restrictive (private).

The class definition contains class members in the following order:

* + - Nested types
    - Field members
    - Member functions
      * Constructors
      * Finalizer
      * Methods (Properties, Events, Operations, Overridables, Static)
    - Private nested types

## *Indentation*

**Spaces in lists.** To improve clarity a space is always required after a colon or semicolon in a list.

for (Count = 0, Flags = 0; Count < 5; Count++)

**Block encapsulation.** A compound statement or code block is encapsulated in set brackets as follows:

For(…) // bracket must be set on next line

{

//…

}

**Curly brackets.** Curly brackets are always used, even if only a single statement compounds the block.

If (bFirstTime)

{

bFirsTime = false;

}

**Code Indentation.** Each new code block must be indented by a tab character. Tabs are not replaced by spaces.

**Line length.** Whenever practical, lines are kept to less than one screen-wide. This makes it easier to edit the source file.

**Functions size.** Methods are supposed to be designed short enough to never have more than 100 lines. Is recommended redesign larger methods into smaller functions.

# Whitespace and Comments

Write meaningful comments for every interface, class and method. The comment format is listed below.

## *Comments for interface*

/// <summary>

/// IWRMResourceGroupAdapter defines all actions of interface IWRMResourceGroup listed in

/// document [MS-WSRM]

/// </summary>

public interface IWRMResourceGroupAdapter : IAdapter

## *Comments for class*

/// <summary>

/// Implement methods of interface IWRMResourceGroup to perform all Resource Group operations

/// of WSRM.

/// </summary>

public partial class WRMResourceGroupAdapter : ManagedAdapterBase, IWRMResourceGroupAdapter

## *Comments for method*

/// <summary>

/// An action of IWRMConfig, that activates or deactivates WSRM service of the server.

/// </summary>

/// <param name="willActivate">Specify TRUE to activate WSRM, while FALSE to deactivate WSRM.

/// </param>

/// <param name="intendedErr">Specify what error code want to get.</param>

/// <returns>Then real error code returns from the server.</returns> public HRESULT Activate(bool willActivate, HRESULT intendedErr)

## *File Banner Comments*

* You ***SHOULD*** have a banner comment at the start of every code file. An example of a minimal C# file banner is:

//------------------------------------------------------------------------------

// Copyright (c) Microsoft Corporation. All rights reserved.

//------------------------------------------------------------------------------

* You ***MAY*** provide more detail in the file banner, but you must keep it up-to-date if you choose to do so. An example of a more detailed C# file banner is:

//------------------------------------------------------------------------------

Any notes or multiple paragraphs are formatted like this.

justified/indented like this.

//

//

//

//------------------------------------------------------------------------------

// Copyright (c) Microsoft Corporation. All rights reserved.

//

// Description: This is the description of the file. Long descriptions should be

## *Blank Lines*

* You ***SHOULD*** use a single blank line to separate areas of functionality within a method. No more than a single blank line should be used within the body of a method. For example, you can delineate local variable declarations from the code within the method:

void MyMethod(ref int i)

{

int counter = 0;

if (...)

{

}

}

Notice the single blank line between the local variable declaration and the ‘if’ statement. Bad vertical spacing would be;

void ProcessItem(ref int i)

{

int counter = 0; // local variable

// Implementation starts here

//

if (...)

{

}

}

In this example of bad usage of blank lines, there are multiple blank lines between the local variable declarations, and multiple blank lines after the ‘if’ block.

**** You ***SHOULD NOT*** have two or more consecutive blank lines *within* a method implementation.

* You ***SHOULD*** separate method implementations with two blank lines.

## *Spaces*

Standard spacing is used in mathematical texts. Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code:

* Spaces ***MUST*** surround binary/ternary operators.

Right: x = y + 1; x = y; if (a > b);

Wrong: x=y+1; x =y+1; x = y +1; if (a==b);

* There **MUST** be no spaces between a unary operator and associated variable.

Right: x = -y; x = --y; x++;

Wrong: x= - y; x ++; -- y;

**** ***MUST NOT*** use spaces between a function name and parenthesis.

Right: CreateFoo()

Wrong: CreateFoo ()

**** You ***SHOULD*** use a single space after a comma between function arguments.

Right: Method(myChar, 0, 1); Wrong: Method(myChar,0,1);

**** You ***SHOULD NOT*** use a space after the parenthesis and function arguments

Right: CreateFoo(myChar, 0, 1)

Wrong: CreateFoo( myChar, 0, 1 )

**** You ***SHOULD NOT*** use spaces inside brackets.

Right: x = dataArray[index]; Wrong: x = dataArray[ index ];

* You ***SHOULD*** use a single space before flow control statements

Right: while (x == y)

Wrong: while(x==y)

* You ***SHOULD*** use a single space before and after comparison operators

Right: if (x == y)

Wrong: if (x==y)

* You ***SHOULD*** use multiple lines for complex conditional expressions. 4 spaces will be used from the starting parentesis.

Right:

if (expected.Length == 0

|| (!usePassword && !response.IsMatch(expected))

|| (usePassword && !response.IsTemplateMatch(expected)))

{

Wrong:

// || should in next line

// wrong spaces before (

// wrong spaces before ||

if (expected.Length == 0 ||

(!usePassword && !response.IsMatch(expected))

|| (usePassword && !response.Contains(special)))

{

# Naming

One of the most important elements of predictability and discoverability is the use of a consistent naming pattern. Many of the common user questions don't even need to come up once these conventions are understood and widely used. There are three elements of naming guidelines:

casing – use of the correct capitalization style

mechanical — use nouns for classes, verbs for methods, etc. word choice — use consistent terms across libraries.

The following section lays out rules for the first two elements, and some philosophy for the third.

We will be discussing naming conventions throughout the standards, so let’s set the stage with a few basics:

**Use full English descriptors that accurately describe the variable/field/class** For example, use names like **firstName**, **grandTotal**, or **CorporateCustomer**. Although names like **x1**, **y1**, or **fn** are easy to type because they are short, they do not provide any indication of what they represent and result in code that is difficult to understand, maintain, and enhance.

**Use abbreviations sparingly, but if you do so then use them intelligently.** This means you should maintain a list of standard short forms (abbreviations), you should choose them wisely, and you should use them consistently. For example, if you want to use a short form for the word "number," then choose one of **nbr**, **no**, or **num**, document which one you chose (it does not really matter which one), and use only that one.

**Avoid names that are similar or differ only in case.** For example, the variable names **persistentObject** and **persistentObjects** should not be used together, nor should **anSqlDatabase** and **anSQLDatabase**.

**Avoid leading or trailing underscores.** Names with leading or trailing underscores are usually reserved for system purposes, and may not be used for any user-created names except for pre-processor defines. More importantly, underscores are annoying and difficult to type so try to avoid their use whenever possible.

## *Case Sensitivity*

MUST NOT use names that require case sensitivity. Components must be fully usable from both case-sensitive and case-insensitive languages. Since case-insensitive languages cannot distinguish between two names within the same context that differ only by case, components must avoid this situation.

Examples of what not to do:

MUST NOT have two namespaces whose names differ only by case.

namespace ee.cummings; namespace Ee.Cummings;

MUST NOT have a function with two parameters whose names differ only by case.

void foo(string a, string A)

MUST NOT have a namespace with two types whose names differ only by case.

System.WinForms.Point p; System.WinForms.POINT pp;

MUST NOT have a type with two properties whose names differ only by case.

int Foo

{

get

{

}

}

return foo;

int FOO {get, set}

MUST NOT have a type with two methods whose names differ only by case.

void foo(); void Foo();

## *Capitalization Styles*

The following section describes different ways of capitalizing identifiers. We will refer to these terms throughout the rest of this document.

* + 1. Pascal Casing

This convention capitalizes the first character of each word. For example:

**B**ack**C**olor

* + 1. Camel Casing

This convention capitalizes the first character of each word except the first word. For example:

**b**ack**C**olor

* + 1. Capitalization summary

The following describes the capitalization of different types of identifiers.

|  |  |  |
| --- | --- | --- |
| **Type** | **Case** | **Notes** |
| Class | PascalCase |  |
| Enum values | PascalCase |  |
| Enum type | PascalCase |  |
| Events | PascalCase |  |
| exception class | PascalCase | ends with **Exception** |

|  |  |  |
| --- | --- | --- |
| Final Static field | PascalCase |  |
| interface | PascalCase | Starts with **I** |
| Method | PascalCase |  |
| namespace | PascalCase |  |
| property | PascalCase |  |
| Public Instance Field | PascalCase | Rarely used, prefer properites |
| Protected Instances Fields | camelCase | Rarely used, prefer properites |
| parameter | camelCase |  |

## *Word Choice*

MUST avoid using class names duplicated in heavily used namespaces.

MUST avoid using identifier that conflict with common words(e.g. data, type, value, length) or keywords.

MUST NOT use abbreviations in identifiers (including parameter names).

If you must use abbreviations, MUST use camelCasing for any abbrevation over 2 characters even if this is not the "standard" abbreviation.

SHOULD NOT use “AdjustedXxxx”, “ChangedXxxx”, “ModifiedXxxx” for a variable name to store the processed result. It is meaningless that the variable is just changed but miss the context that how/why. Instead, find a more suitable word according to the context.

## *Namespaces*

The general rule for namespace naming is: AW2K.GroupName.DomainName. GroupName could be DA for Development Architecture or EA for Execution Architecture.

MUST use PascalCasing, and separating logical components with periods (e.g., Microsoft.Office.PowerPoint).

MUST use plural namespace names where appropriate. For example, use System.Collection***s*** not System.Collection. Exceptions to this rule are brand names and abbreviations. For example, use System.IO not System.IOs.

MUST NOT have name namespaces and classes with the same name. For example, don't use "Debug" for a namespace name and have a class named "Debug".

## *Class and Class members*

* + 1. Class Naming Guidelines

MUST name classes with nouns or noun phrases. MUST use PascalCasing.

MUST use abbreviations in class names sparingly. MUST not use any class prefix (such as "C").

MUST not use an underscore.

public class **FileStream** public class **Button** public class **String**

* + 1. Interface Naming Guidelines

MUST name interfaces with nouns or noun phrases, or adjectives describing behavior. E.g., IComponent (descriptive noun), ICustomAttributeProvider (noun phrase), and IPersistable (adjective).

MUST use PascalCasing.

MUST use abbreviations in interface names sparingly. MUST not use the underscore.

MUST prefix interface names with the letter "I", to indicate that the type is an interface. Don't prefix class names with the letter "C". Occasionally, it will be necessary to have a class name that begins with I that is not an interface. This is acceptable as long as the next character is lower case. Such as IdentityStore.

MUST use similar names when defining a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the "I" prefix on the interface name. E.g., these guidelines are used for the interface IComponent and its standard implementation, the class Component.

public interface **IComponent**

public class Component : **IComponent** public interface **IServiceProvider** public interface **IFormatable**

* + 1. Enum Naming Guidelines

MUST use PascalCasing for enums.

MUST use PascalCasing for enum value names. MUST use abbreviations in enum names sparingly

MUST not use a prefix on enum names (e.g. adXXX for ADO enums, rtfXXX for rich text enums, etc.).

MUST not use any "Enum" suffix on enum types. MUST use a singular name for enums

MUST use a plural name for bit fields

MUST define enumerated values using an enum if they are used in a parameter or property. This allows the tool to know the possible values for a property or parameter.

public enum FileMode{ Create, CreateNew,

Open, OpenOrCreate, Truncate

}

MUST use the Flags custom attribute if the numeric values are meant to be bitwise or are used together

[Flags]

public enum Bindings { CreateInstance, DefaultBinding, ExcatBinding, GetField, GetProperty, IgnoreCase, InvokeMethod, NonPublic, OABinding, SetField SetProperty, Static

}

An exception to this rule is when encapsulating a Win32 API, it's common to have internal definitions that come from a Win32 header. It's ok to leave these with the Win32 casing, which is usually all-caps.

MUST use Int32 as the underlying type of an enum.

An exception to this rule is if the enum represents flags and there are many flags (>32) or the enum may grow to many flags in the future or the type needs to be different than int for backward compatibility.

MUST use enums only if the value can be completely expressed as a set of bitflags. MUST not use enums for open sets (such as Operating system version etc).

* + 1. ReadOnly and Const Field Names

MUST name static fields with nouns, noun phrases, or abbreviations for nouns. MUST name static fields with PascalCasing.

MUST not prefix static field names Hungarian type notation.

* + 1. Parameter Names

MUST use descriptive parameter names. Parameter names should be descriptive enough that in most scenarios the name of the parameter and its type can be used to determine its meaning.

MUST name parameters with camelCasing.

MUST prefer names based on a parameter's meaning to names based on the parameter's type. We expect development tools to provide the information about type in a handy manner, so the parameter name can be put to better use describing semantics rather than type. Occasional use of type-based parameter names is entirely appropriate.

MUST not use **reserved** parameters. If more data is need in the next version a new overload can be added.

MUST not prefix field names Hungarian type notation.

Type GetType (string **typeName**)

string Format (string **format**, object [] **args**)

* + 1. Method Naming Guidelines

MUST name methods with verbs or verb phrases. MUST name methods with PascalCasing

RemoveAll(), GetCharArray(), Invoke()

* + 1. Property Naming Guidelines

MUST name properties using noun or noun phrase MUST name properties with PascalCasing

Recommendation not using properties and types with the same name.

Defining a property with the same name as a type can cause some ambiguity languages. It is best to avoid this ambiguity unless there is a clear justification for not doing so.

For example: System.WinForms has an Icon property even though there is an Icon class because Form.Icon is so much easier to understand than Form.FormIcon or Form.DisplayIcon etc.

However, System.WinForms.UI.Control has a color property. Because there is a Color class, the Color property was named BackgroundColor as it's a more meaningful name that does not conflict.

Text, LastIndex, Value[5]

* + 1. Event Naming Guidelines

MUST name Event handlers with the "EventHandler" suffix.

public delegate void **MouseEventHandler**(object sender, MouseEvent e);

MUST use two parameters named *sender* and *e*.

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

The state associated with the event is encapsulated in an instance of an event class named e. Use an appropriate and specific event class for its type.

public delegate void MouseEventHandler(object **sender**, MouseEvent **e**);

MUST name event argument classes with the "EventArgs" suffix.

public class MouseEventArgs : EventArgs { int x;

int y;

public MouseEventArgs(int x, int y)

{ this.x = x; this.y = y; } public int X { get { return x; } } public int Y { get { return y; } }

}

* + - * MUST name event names that have a concept of pre and post will be prefixed using the present and past tense (do not use BeforeXxx\AfterXxx pattern). For example, a close event that could be canceled would have a Closing and Closed event.

public event ControlEventHandler ControlAdded {

//..

}

* + - * Consider naming events with a verb.

## *Directory and File Names*

* ***MUST*** use ‘PascalCasing’ for directory names.

D:\Code\Infrastructure\Src\ProtoSdk\SspiLib D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\

Note: SSPI, SDK are acronyms, src is abbreviation.

* You ***MAY*** use all Capital Letters for protocols like MS-XXXX under ProtoSdk.

** *MUST NOT*** use dash(-), underscore(\_), or other special characters with only one exception that dash could be used for protocol names under $(INETROOT)\Infrastructure\Src\ProtoSdk.

** *SHOULD*** avoid reusing or embeding upper level names.

Right:

D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\CsraCodec.csproj D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\Codec.cs D:\Code\Infrastructure\Src\ProtoSdk\MS- CSRA\Codec\Microsoft.Protocols.TestTools.StackSDK.CA.Csra.Codec.asmmeta D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\sources D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\placefile

Wrong:

D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\**MS-CSRA**\_Codec\**MS-CSRA\_Codec**.csproj D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\**CSRA**Codec.cs D:\Code\Infrastructure\Src\ProtoSdk\MS- CSRA\Codec\Microsoft.Protocols.TestTools.StackSDK.CertificateAuthorization.CSRA.Codec

.asmmeta

D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\**Sources** D:\Code\Infrastructure\Src\ProtoSdk\MS-CSRA\Codec\**PLACEFILE**

* ***MUST*** use lowver case for the binary folder “bin”. ***MUST NOT*** use “Bin”.
* ***MUST*** use the following folder structure(the folder names are bold & underline):

**ProtocolFamily**

|- ... (common files in this protocol family)

**MS-XXXX**

|-**Role** (folder, e.g. Client, Server, Initiator...)

|-XxxxRole.cs

|-**Messages** (folder, could have subfolders in it)

|-...

|-XxxxStack.sln

|-Microsoft.Protocols.TestTools.StackSdk.ProtocolFamily.Protocol.asmmeta

|-placefile

|-sources

# Class Members

MUST define a class as public only if it can be used out of the assembly. MUST group references according to their namespace.

For example:

using System.Net;

using System.Net.Sockets; using System.IO;

using Microsoft.Protocols.TestTools.Messages; using Microsoft.Protocols.TestTools;

## *Variable Usage Guidelines*

MUST not define public variables.

MUST declare variables as locally as possible MUST initialize variables when they are declared.

MUST declare and initialize/assign local variables on a single line where the language allows it

Define a public property to access a private or internal variable instead. MUST use this directive before a variable when used inside the class. For example:

public class StackTcpClient

{

string serverIp;

int serverPort;

public StackTcpClient(string ip, int port)

{

this.serverIp = ip; this.serverPort = port;

}

}

MUST set a variable to null after releasing it. For example:

public void Disconnect()

{

tcpClient.Close();

tcpClient = null;

}

## *Property Usage Guidelines*

MUST use PascalCasing for Property names.

public class Foo {

public Color **BackColor** { }

}

MUST not use Hungarian notation.

MUST use the expanding “get” and “set” instead of abbreviation mode.

int Foo

{

Get

{

}

Set

{

}

}

Return foo;

Foo = value;

* + 1. Read-Only and Write-Only Properties

MUST use Read-only properties when the user cannot change the logical backing data field.

MUST not use Write-only properties.

* + 1. Indexed Property Usage

MUST use only one indexed property per class and make it the default indexed property for that class.

MUST not use non default Indexed properties.

MUST use the name "Item" for indexed properties unless there is an obviously better name (for example a Chars property on String)

MUST use indexed properties when the logical backing store is an array MUST not mix indexed properties and overloaded methods.

MethodInfo Type.Method[string name]

MethodInfo Type.GetMethod (string name, boolean ignoreCase)

## *Event Usage Guidelines*

MUST name events using PascalCasing MUST not use Hungarian notation

MUST use the terminology "raise" for events rather than fired.

When referring to events in documentation, use the 'an event was raised' terminology instead of 'an event was fired'.

Consider naming events with a verb.

MUST nameEvent handler names ending in "EventHandler". MUST use a return type of void for Event handlers.

MUST use two parameters named sender and e.

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

The state associated with the event is encapsulated in an instance of an event class named e. Use an appropriate and specific event class for its type.

public delegate void MouseEventHandler(object sender, MouseEventArgs e);

## *Constructor and Destructor Usage Guidelines*

MUST not create reference objects in the field of a class, new and initialize reference objects in the constructor.

For example:

Bad:

public class StackTcpClient

{

string serverIp;

int serverPort;

TcpClient tcpClient = new TcpClient(); public StackTcpClient(string ip, int port)

{

this.serverIp = ip; this.serverPort = port;

}

}

Good:

For example:

public class StackTcpClient

{

string serverIp;

int serverPort;

TcpClient tcpClient;

public StackTcpClient(string ip, int port)

{

this.serverIp = ip; this.serverPort = port;

this.tcpClient = new TcpClient();

}

}

MUST not use destructor, inherit interface IDisposable instead. For example:

Bad:

public class StackTcpClient

{

~StackTcpClient()

{

}

}

Good:

For example:

// A base class that implements IDisposable. public class MyClass: IDisposable

{

// Implement IDisposable.Dipose method public void Dispose()

{

Dispose(true);

//Take this object out of the finalization queue of the GC GC.SuppressFinalize(this);

}

private void Dispose(bool disposing)

{

// If disposing equals true, dispose all managed

// and unmanaged resources. if(disposing)

{

// Free managed resources & other reference types

}

// Call the appropriate methods to clean up unmanaged resources.

// If disposing is false, only the following code is executed.

}

// This destructor will get called only from the finalization queue

~MyClass()

{

Dispose(false);

}

}

# CoreXT

* ***MUST*** ensure filenames for dirs, sources, and placefile are all lower cases.
* ***MUST*** ensure macro defined in dirs, sources, and placefile are in alphabetic order

** *MUST NOT*** use hard coded version macros. For example:

= \

$(PTFREFERENCES\_V2\_1\_1777) \

REFERENCES

** *MUST NOT*** use local key files. For example:

ASSEMBLY\_IDENTITY\_PRIVATE\_KEY\_NAME = CifsStrongName.snk

** *MUST NOT*** disable warnings and/or rules in local project. For example:

= $(FXCOP\_USER\_FLAGS) \

/directory:$(PTF\_REF\_PATH) \

/directory:..\SspiLib\$(O) \

/ruleid:-Microsoft.Design^#CA1008 \

/ruleid:-Microsoft.Design^#CA1028 \

/ruleid:-Microsoft.Maintainability^#CA1505 \

= $(USER\_CS\_FLAGS) \

/warnaserror-

/warn:4

/nowarn:649

FXCOP\_USER\_FLAGS

USER\_CS\_FLAGS

The lab admin and release team will use global macro to control the version, rules, scope, etc. Hard these into your project will cause a lot efforts for upgrade and maintainence. A normal sources file look like:

= $(FXCOP\_USER\_FLAGS) \

/directory:$(PTF\_REF\_PATH) \

/directory:..\SspiLib\$(O) \

/directory: $(INFRA\_SDK\_REF) \

$(INFRA\_SDK\_PATH)\FileAccessService\$(O)

FXCOP\_USER\_FLAGS

= \

Client\Smb2Client.cs \ Client\Smb2ClientCollection.cs \ Packets\Smb2EchoResponsePacket.cs \ Packets\Smb2ErrorResponsePacket.cs \ Packets\Smb2FlushRequestPacket.cs \ Packets\Smb2WriteRequestPacket.cs \ Packets\Smb2WriteResponsePacket.cs \ Packets\SmbNegotiateRequestPacket.cs \ Server\Smb2Server.cs \ Server\Smb2ServerContext.cs \ Server\Smb2ServerDecodePacket.cs \ Server\Smb2ServerPerConnection.cs \ Server\Smb2ServerPerSession.cs \ SMB2Message.cs \

SMB2MessageUtil.cs \ Smb2Types.cs \ SmbMessage.cs \ StackConfig.cs\ Utilities.cs \

SOURCES

$(INFRA\_SDK\_PATH)\FileAccessService\$(O)\Microsoft.Protocols.TestTools.StackSDK.FileA

ccessService.metadata\_dll; \

= \

$(PTFREFERENCES) \

$(INFRA\_SDK\_REF) \

REFERENCES

= 1

MANAGED\_CODE

!INCLUDE $(INETROOT)\build\sources.all

TARGETNAME =

Microsoft.Protocols.TestTools.StackSDK.FileAccessService.SMB2 TARGETTYPE = DYNLINK

* ***MUST*** sort the file and folder names alphabetically. This is good to maintain.
* ***MUST*** add all the reference paths into FXCOP\_USER\_FLAGS. This is to avoid the fxcop build error.
* ***MUST*** use the absolute path macro, e.g. $(INFRA\_SDK\_REF), in sources. Those macros are generally defined in sources.all.

# Generics

* ***MUST*** use generics when creating Collection classes
* You ***SHOULD*** use generic classes and interfaces from System.Collections.Generic namespace as opposed to from System.Collections namespace

For example, prefer **List<T>** class in places where you want to use **ArrayList** class.

# Exceptions

## *Throwing Exceptions*

* ***MUST*** throw the most specific (the most derived) exception that makes sense. Use existing exceptions for error conditions. Don’t reinvent the wheel for existing error conditions. For example, throw ArgumentNullException and not its base type ArgumentException if a null argument is passed.
* ***MUST*** use resource ID rather than hard coded string in the code.

** *MUST NOT add other comments to ArgumentNullException than the parameter itself,*** .

** *MUST NOT*** use exceptions to alter the normal flow of control. Exceptions should be used to report/handle error conditions.

** *MUST NOT*** have public members that can either throw or not based on some option.

// bad design

public Type GetType(string name, bool throwOnError)

** *MUST NOT*** throw System.Exception, the base class for all CLS-compliant exceptions.

** *MUST NOT*** throw exception in Dispose function. It is probably system thread, e.g. GC, to invoke the Dispose(), in this case the user thread is unable to catch the exception, and the behavior is unpredictable.

## *Exception Handling*

** *MUST NOT*** catch non-specific exceptions, such as System.Exception, System.SystemException, etc. in framework code.

Try

{

File.Open(...);

}

catch(Exception e)

{

} // swallow “all” exceptions - don’t do this!

** *MUST*** use try-finally for clean up (say closing connections to a database) instead of using try-catch.

FileStream stream = null;

**Try**

{

stream = new FileStream(…);

…

}

**finally**

{

if(stream != null) stream.Close();

}

* ***MUST*** use an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.

public void DoSomething(FileStream file)

{

long position = file.Position; try

{

... // do some reading with fs

}

catch(MyException e)

{

file.Position = position; // unwind on failure

….

**throw**; // rethrow

}

}

## *Custom Exceptions*

* **MUST** use existing exceptions for error conditions. Don’t reinvent the wheel for existing error conditions

(e.g. use ArgumentNullException if you get a null parameter, rather than rolling your own exception).

* ***MUST*** derive from System.Exception or one of the other common base Exceptions when designing custom exceptions.
* ***MUST*** end exception class names with the ‘Exception’ suffix.
* ***MUST*** make exceptions serializable. An exception must be serializable to work correctly across application domains and remoting boundaries.

[Serializable]

public class MyException : Exception, ISerializable

{

….

//Serialization constructor

protected MyException(SerializationInfo info, StreamingContext context)

{

…..

}

}

# Design

* ***MUST refer to <DotNet Framework Design Guidelines.doc>.***

** *MUST NOT*** use magic number in code, instead, add comments to explain why it is defined in this way(could use TD as reference), and define it as a const.variable if it appears in several places. Sometimes you need to define the const variable in ptfconfig if this is configurable.

## *From Licensee’s Viewpoint*

** *MUST not assume that Licensee implement all the required features (normative requirement identified by***

***“MUST”) in TD, instead, sometimes Licensee just implement a subset of functions.***

* ***This actually requires the StackSdk decouple the functions and implement in such a way that when one sub-function is disabled, the other sub-functions do not have side-effect.***

***Generally, the following modules should be examined in detail whether they need be able to be plug-in or not(switched on/off according to the configuration).***

1. ***The under layer transport module;***
2. ***The algorithms, e.g. used for encryption. Licensee may implement not all of the algorithms defined in this TD.***
3. ***The logical independent functions. One typical example for this is that some Licensee of SMB/SMB2 do not implement Named-Pipe function at all, though it is a MUST in the TD.***

## *From FuzzyTest’s Viewpoint(Optional)*

* ***For those PacketAPIs(PacketAPI is to generate the formatted message which is used as the under layer transport payload) that do some encryption inside, SHOULD decouple*** the encryption step from the PacketAPI or provide a configuration parameter to switch on/off the internal encryption steps.

# Unit Test

* ***MUST provide the corresponding unit test when checking in the source code. if the source code is changed, the unit test cases should also be checked if there is any change.***

## *Testability*

* ***The code MUST have good testability, i.e.. all the paths or conditions in a function SHOULD be satisfied easily, AND the result of the function MUST be able to expect.***
* ***MUST*** decouple the complex function if:
  + it could not be tested;
  + or hard to construct the parameters for a certain code path;
  + Or hard to predict the result.

## *Private Method*

* ***MUST use private accessor which is supported by Visual Studio Test Framework When access to private methods.***
* ***MUST check in the source file of the private accessor supporting files(usually with “\_Accessor.cs” prefix) to***

***utest folder, to get coreXT build pass.***.

## *Dependent Class*

** *MUST NOT use macro(e.g. “#if UT” to differenciate build condition, which will produce two binaries. It is not so convincing that the binary we released is just the same with the one we tested.***

* ***MUST*** use class factory mode to dynamically binding to a class.

Define an abstract class, could be instantiated as either Mock class or an encapsulation class;

Define an encapsulation class, which inherits the above abstract class and invokes the real dependent class/methods;

Define a Mock class, which also inherits the above abstract class. Used in UT;

e.g. the abstract class is as follow: public abstract class AbstractTcpClient

{

public static Type InstanceType = typeof(MyTcpClient);

public static AbstractTcpClient CreateInstance()

{

return (AbstractTcpClient)Activator.CreateInstance(InstanceType);

}

… other methods and constructor

}

When used in stack code:

AbstractTcpClient.InstanceType = typeof(MyTcpClient);//could be omitted since it is the default value. AbstractTcpClient tc = AbstractTcpClient.CreateInstance();

When used in UT code:

AbstractTcpClient.InstanceType = typeof(MockTcpClient); AbstractTcpClient tc = AbstractTcpClient.CreateInstance();

# Process

* ***MUST*** use jjpack (instead of bbpack) when doing integration from different branches. Because jjpack could

save the branch “integrate” information.

* MUST file a bug if public interface changed. Bug title: [MS-XXXX] [Breaking Change] …
* Inform others that your public interface is changed.
* It’s easy to track that all related codes are changed.
* It’s easy to write release notes.
* When resolve a bug, MUST identify the version with both branch name and version number.

Ex: ts\_dev 1.0.500.0

The most current version number could be found under

\ts\_dev\build\automation\versions\version.html

**** MUST not check in the following files to coreXT, for corext could take care of version and signature issue globally,

1. versioninfo.rc
2. Assembly.cs
3. Key.snk

## *Hotfix*

* MUST reverse integrate (RI) the hotfix change from ts\_release\_vbl# back to ts\_main, and then forward integrate(FI) the change from ts\_main to ts\_dev after the regression test.
* MUST use the 4th digit for the hotfix version number. E.g. 1.0.500.0 is the official release version, 1.0.500.1 is the hotfix version. And the 4th digit MUST increate 1 for each build.