Coding Guidelines for Rightware

March 21, 2012

Contents

[1 Introduction 4](#_Toc320085867)

[2 General 4](#_Toc320085868)

[2.1 Compiler and build process 4](#_Toc320085869)

[2.2 Modularity 4](#_Toc320085870)

[2.2.1 Includes (C / C++) 4](#_Toc320085871)

[2.2.2 Code size 5](#_Toc320085872)

[2.2.3 Visibility 5](#_Toc320085873)

[2.2.3.1 C / C++ 5](#_Toc320085874)

[2.2.3.2 C 5](#_Toc320085875)

[2.2.3.3 C# 5](#_Toc320085876)

[3 Portability (C-Specific) 5](#_Toc320085877)

[3.1 Empty lines 6](#_Toc320085878)

[3.2 Localization 6](#_Toc320085879)

[3.3 Evaluation order 6](#_Toc320085880)

[3.4 Endianness 6](#_Toc320085881)

[3.5 Paths 6](#_Toc320085882)

[4 Optimization 6](#_Toc320085883)

[5 Code style 7](#_Toc320085884)

[5.1 Magic numbers 7](#_Toc320085885)

[5.2 Indenting 7](#_Toc320085886)

[5.2.1 C / C++ 7](#_Toc320085887)

[5.2.2 C# 7](#_Toc320085888)

[5.3 Readability 7](#_Toc320085889)

[5.4 Formatting 7](#_Toc320085890)

[5.4.1 Spaces 7](#_Toc320085891)

[5.4.1.1 C# 8](#_Toc320085892)

[5.4.2 Empty lines 8](#_Toc320085893)

[5.4.2.1 C / C++ 8](#_Toc320085894)

[5.4.2.2 C# / C++ 8](#_Toc320085895)

[5.4.3 Pointers (C-Specific) 9](#_Toc320085896)

[5.4.4 Braces 9](#_Toc320085897)

[5.4.5 Comments 9](#_Toc320085898)

[5.4.5.1 C 9](#_Toc320085899)

[5.4.5.2 C++ 9](#_Toc320085900)

[5.4.5.3 C# 9](#_Toc320085901)

[5.4.6 If / else 9](#_Toc320085902)

[5.4.7 Switch / cases 10](#_Toc320085903)

[5.4.7.1 C / C++ 10](#_Toc320085904)

[5.4.7.2 C# 11](#_Toc320085905)

[5.4.8 Return 11](#_Toc320085906)

[5.4.9 Splitting 11](#_Toc320085907)

[5.4.9.1 C / C++ 11](#_Toc320085908)

[5.4.9.2 C# 12](#_Toc320085909)

[5.4.10 Functions 12](#_Toc320085910)

[5.5 Example 12](#_Toc320085911)

[5.5.1 C 12](#_Toc320085912)

[5.5.2 C++ 13](#_Toc320085913)

[5.5.3 C# 13](#_Toc320085914)

[5.6 Naming 13](#_Toc320085915)

[5.6.1 Files and folders 14](#_Toc320085916)

[5.6.1.1 C / C++ 14](#_Toc320085917)

[5.6.1.2 C# 14](#_Toc320085918)

[5.6.2 Constants 14](#_Toc320085919)

[5.6.3 Variables 14](#_Toc320085920)

[5.6.3.1 C 14](#_Toc320085921)

[5.6.4 Functions (C-Specific) 14](#_Toc320085922)

[5.6.5 Classes and structs 14](#_Toc320085923)

[5.6.5.1 C / C++ 14](#_Toc320085924)

[5.6.5.2 C# 14](#_Toc320085925)

[5.6.6 Enums 15](#_Toc320085926)

[5.7 Safety 15](#_Toc320085927)

[5.7.1 Functions 15](#_Toc320085928)

[5.7.2 Initialization 15](#_Toc320085929)

[5.7.2.1 C / C++ 15](#_Toc320085930)

[5.7.2.2 C# 15](#_Toc320085931)

[5.7.3 Type casting 15](#_Toc320085932)

[5.7.3.1 C 15](#_Toc320085933)

[5.7.4 Scopes 15](#_Toc320085934)

[5.7.4.1 C 16](#_Toc320085935)

[5.7.4.2 C# / C++ 16](#_Toc320085936)

[5.7.5 Types (C-Specific) 16](#_Toc320085937)

[5.7.6 Typedefs (C-Specific) 16](#_Toc320085938)

[5.7.7 Consts 16](#_Toc320085939)

[5.7.7.1 C / C++ 16](#_Toc320085940)

[5.7.7.2 C# 17](#_Toc320085941)

[5.7.8 Macros (C-Specific) 17](#_Toc320085942)

[6 Documentation 17](#_Toc320085943)

[**6.1** **Doxygen formatting** 18](#_Toc320085944)

[6.2 Dates 18](#_Toc320085945)

[6.3 Unfinished work 18](#_Toc320085946)

[6.4 File headers 18](#_Toc320085947)

[6.4.1 C 18](#_Toc320085948)

[6.4.2 C++ 19](#_Toc320085949)

[C# 19](#_Toc320085950)

[7 Error handling 19](#_Toc320085951)

[7.1 Assertions 19](#_Toc320085952)

[7.2 Error checking 19](#_Toc320085953)

[7.2.1 C 19](#_Toc320085954)

[7.2.2 C# / C++ 19](#_Toc320085955)

[8 Kanzi framework 20](#_Toc320085956)

[8.1 Introduction 20](#_Toc320085957)

[8.2 Types 20](#_Toc320085958)

[8.3 Memory management 21](#_Toc320085959)

[8.4 Logging 21](#_Toc320085960)

[8.5 Error handling 21](#_Toc320085961)

[9 Kanzi Tool 21](#_Toc320085962)

[9.1 Introduction 21](#_Toc320085963)

[9.2 Localization 21](#_Toc320085964)

[9.2.1 Code / Code-Behind Strings 21](#_Toc320085965)

[9.2.2 XAML Strings 21](#_Toc320085966)

[10 Example code 22](#_Toc320085967)

[10.1 kzu\_example.h 22](#_Toc320085968)

[10.2 kzu\_example.h 23](#_Toc320085969)

# Introduction

This document describes the coding guidelines for Kanzi and other Rightware products. The languages covered are C and C#. For each coding aspect the points concerning both the languages are presented first. The language specific conventions are listed under their own headings. If the whole section is language-specific this is noted in its header.

# General

## Compiler and build process

Every single warning should be treated seriously. Code must compile with zero warnings.

## Modularity

Don’t be afraid of large amounts of files, it’s always better than big files.

Try to keep functions as simple as possible; they should only do one thing with minimum amount of dependencies to other components.

### Includes (C / C++)

Include own header file as the very first code line in a .c file. Include other project headers next, and library headers last. Place high level APIs first and low level APIs last. Group includes by modules and components by placing empty line between.

Use unique #include guards in header files, e.g.  
#ifndef KZU\_FILE\_NAME\_H  
#define KZU\_FILE\_NAME\_H

All #include directives appear at the top of source files, and just after the #include guards in header files.

Avoid header file dependencies by using forward declaration where possible.

Use relative include paths for sub-folders in .c files, but don’t use ‘..’ in includes. Use full include paths in .h files due to LUA wrapper requirements.

### Code size

Avoid long functions. If a function does more than one logical thing, consider splitting to several functions instead.

In switch-case structures if a case is many lines long, extract it to a function.

Avoid overly deep nesting. Use functions instead.

Avoid long files. If a file can be split to several logical entities, do so.

### Visibility

#### C / C++

If a function is not shared between files, set it as static. This will completely hide the function from other files and prevents accidental naming conflicts.

Declare structs and classes in .c files if possible and use forward declaration in header files.

Avoid global variables. Variables are usually implementation detail and should be hidden from the API. Also global and static variables are problematic for initialization / uninitialization and for multithreading.

#### C

In Kanzi Engine code add \_internal postfix for names of static functions.

#### C#

Use *public* visibility for classes/structs/interfaces only when necessary, use *internal* visibility instead. Use explicit visibility declaration always instead of using the default one.

Use *public* visibility modifier instead of *internal* for methods in internal classes. This enables making the class public afterwards more easily should the need arise.

Use visibility modifiers before other modifiers in member declarations:

* like this:
  + public static foo()
* not like this:
  + static public bar()

# Portability (C-Specific)

Make sure to write code which compiles in as many language dialects as possible. Don’t use C89 specific syntax; instead make sure the code is valid C89, C99 and C++. For example in C89 it is possible omit the type cast from void pointers to concrete types, but this is disallowed. Also don’t use C99/C++ reserved keywords as identifier names.

## Empty lines

Use empty line at the end of each file. Some compilers require this.

## Localization

Assume that all strings are UTF-8 encoded unless there is better knowledge of the case. This means that most string operations work as with ASCII text, but individual chars cannot be accessed from the string.

## Evaluation order

Never pass multiple parameters with side-effects to function calls. Evaluation order of parameters isn’t specified. For example setColor(random(), random(), random()); is very unsafe.

## Endianness

Never assume any specific endianness for the target platform. Always support both big-endian and little-endian systems.

File formats use specific endianness, which might be different than the platform endianness. For example TGA is little-endian. It’s safest to read only bytes and form higher-bit numbers manually in code.

KzcInputStream supports reading ints and other non-8bit primitives with both little- and big-endian.

## Paths

Always use forward slash (‘/’) as path separator in generic code. Backslash is only used in System layer and conversion between forward slash and backslash is done automatically. If input data supports backslash, convert them to forward slashes with File utility after parsing.

## KZ\_HEADER (C-Specific)

In each header file, include <system/kzs\_header.h> as the last system include file. After all includes, specify KZ\_HEADER\_BEGIN and before the end of the include guard, specify KZ\_HEADER\_END. These macros ensure that the header can be used from C++ code directly. Put two empty lines before and after those macros.

# Optimization

Use pre-increment (++i) and pre-decrement (--i) if possible. It produces faster code at least on ARM.

Avoid code duplication as much as possible. Introduce local variables to hold values which would otherwise be fetched more than once. For example instead of:

if (array[i].size > 0)

{

int x = array[i].x;

int y = array[i].y;

...

}

Use the following:

ItemType item = array[i];

if (item.size > 0)

{

int x = item.x;

int y = item.y;

...

}

Keep your loops simple by avoiding complex continue tests. For example  
for (i = 0; i < s->getSize(); ++i). Here s->getSize() has to be fetched on every iteration.

# Code style

## Magic numbers

Avoid using magic numbers. Use constants (#define) and enums instead. Have some common sense here; if you are parsing decimal numbers and multiply by ten, it is okay. If you initialize an index to -1, use INVALID\_INDEX instead.

## Indenting

Indent blocks with 4 spaces. Tabs are not allowed. Remember to configure Visual Studio and other editors respectively and for all file types.

### C / C++

When splitting long lines, use double indent (8 spaces) or align on parenthesis.

Labels use zero indentation, except for switch cases.

### C#

When splitting long lines, use single indent (4 spaces). The difference to C is due to Visual Studio standard formatting for C#.

## Readability

Do not rely on operator precedence rules. Use parentheses to explicitly state evaluation order in expressions.

Use hex code (0xFFFF) for bit masks. Use capital letters for A-F in hex codes. Use leading zeros to indicate the number of meaningful bits (e.g. 0x00FF00 instead of 0xFF00 for RGB values). Only use hexadecimal numbers when working with binary data.

Don't use function calls or any other expression with side effects inside if statements. For example if (setValue(10) != 0) is not allowed. Use int value = setValue(10); if (value != 0) instead. This also makes you come up with a descriptive variable name for the value.

Use read-only tests inside ifs, whiles, etc. For example if ((j = 10 \* i) < 20) is not allowed, because the test has side-effects. Remember that the evaluation order is often unspecified due to C sequence points, for example f(i++) + g(j++) does not specify the order in which i and j are incremented or the order in which f and g are called.

## Formatting

### Spaces

Use space around the following operators:

* Arithmetic: +, -, \*, /, %
* &&, ||
* <<, >>, &, |
* ==, <, <=, >, >=, !=
* +=, -=, \*=, <<=, etc.

Don’t use space around the following operators:

* !, ~, ++, --, -number
* \*pointer
* ., ->

To sum it up, use space around the operator if it takes two operands, otherwise don’t.

Using space between control keyword such as if or while etc. and parentheses is preferred to distinguish them from function calls.

Don't use space after typecast.

Don’t use space inside parentheses or brackets. e.g. ((10 \* i) + x), not ( ( 10 \* i ) + x )

Don’t use space between function name and parameters.

Don’t use space before semicolon.

Use space after semicolon when it doesn’t end the line.

#### C#

Use Visual Studio->Edit->Advanced->Format Document to format the spacing and braces your code.

### Empty lines

Use single empty lines to separate different functions and functionality. Use them extensively in the code as well. End all source files with an empty line.

#### C / C++

Use double empty lines to separate different parts of source files:

* After include guards and before the final #endif in header files
* Includes
* Constants
* Variables
* Structs
* Enums
* Functions

#### C# / C++

Use double empty lines to separate different parts of source files:

* Between classes and structs
* Between member groups
  + Fields (private/protected)
  + Properties
  + Constructors
  + Methods

### Pointers (C-Specific)

Declare pointers as int\* x, not int \*x. Even though technically in variable declarations the asterisk binds the variable instead of the type, everywhere else it affects the type. Therefore it makes more sense to tie it together with the type. Multiple variable declarations in a single line should not be used anyway, so this is not a problem.

### Braces

Braces (‘{‘ and ‘}’) should be on their own line with the same indentation as the enclosing scope.

Always use braces with if/else, switch, while and for statements. For example for (int i = 0; i < 10; i++) print(i); is not allowed.

### Comments

Do not decorate comments with ASCII graphics.

Avoid commented out code when checking into source control. Use version control to retrieve old deleted code instead in the unlikely case that it is needed.

#### C

In C it is mandatory to use block comments (/\* Comment here \*/).

#### C++

In C++ line comments (// Comment here) are also allowed and recommended when appropriate.

#### C#

Use C# XML comments (start comment with /// to automatically insert the XML) for documenting classes and their members. IntelliSense is able to take advantage of these comments.

Use line comments (//) for documenting code. Use these comments even if the comment spans over a couple of lines. Reserve a line for each comment.

### If / else

Always have else if together in the same line. The final else should be alone on a single line. For example:

if (test1)

{

doSomething();

}

else if (test2)

{

doSomethingElse();

}

else

{

doTheElse();

}

### Switch / cases

Always declare default case. The default case should include an error check or assert if error check is not possible. If it doesn’t, document the reason clearly in a comment.

Avoid case fall-through. Always add a comment when falling-though from non-empty case to confirm that the fall-through is intentional.

#### C / C++

Always use brackets for switch cases and separate cases with an empty line.

Indent switch cases as follows:

switch (variable)

{

case VALUE\_1:

{

doValue1();

break;

}

case VALUE\_2:

case VALUE\_3:

{

doValue23();

break;

}

/\* Avoid, but comment when used. \*/

case VALUE\_4:

{

doValue4();

}

/\* Fallthrough \*/

case VALUE\_5:

{

doValue5();

break;

}

default:

{

error(INVALID\_VALUE);

}

}

For simple mapping from enumerations to other values a more compact formatting can be used:

kzString description;

switch (variable)

{

case VALUE\_1: description = “Value 1”; break;

case VALUE\_2: description = “Value 2”; break;

case VALUE\_3: description = “Value 3”; break;

default:

{

kzsAssert(KZ\_FALSE);

description = KZ\_NULL;

break;

}

}

#### C#

Indent switch cases as follows:

switch (variable)

{

case VALUE\_1:

doValue1();

break;

case VALUE\_2:

case VALUE\_3:

doValue23();

break;

// Avoid, but comment when used.

case VALUE\_4:

doValue4();

// Fallthrough

case VALUE\_5:

doValue5();

break;

default:

thrown new ArgumentException(“Invalid case.”);

}

### Return

Don’t use parentheses with return, it’s not a function.

### Splitting

Split overly long lines. Use common sense and consider what the cleanest way to format multiple lines is.

#### C / C++

Align parameters to the same level as the first parameter. For example:

longFunctionName((expressionOperand1 + expressionOperand2),  
 parameter2, parameter3,

anotherLongerParameter);

If the function name and assignment is so long that there is no space for parameters, separate the call parameters on their own line, indent with 2 “tabs” (8 spaces). For example:

assignmentToLongVariableName = longFunctionName(  
 (expressionOperand1 + expressionOperand2),  
 parameter2,  
 parameter3);

#### C#

Separate long function call parameters on their own line, indent with 1 “tab” (4 spaces).

The difference to C convention is due to Visual Studio default formatting. The motivation for the 2 tab convention would be to enable the reader to easily distinguish scopes from line splits. However, now that the convention to reserve a line for every scope brace in use, the separation should be trivial.

An example:

LongFunctionName(  
 (expressionOperand1 + expressionOperand2),  
 parameter2,  
 parameter3

);

### Functions

Use void instead of empty parameter list in function declaration. (E.g void doStuff(void); instead of void doStuff();, since void means no parameters, whereas empty means unspecified number of unspecified parameters.

## Example

### C

An example function:

kzInt functionName(kzInt parameter1, kzByte parameter2,

kzInt\* out\_returnValue1)

{

kzInt localVariable;

if (parameter1 < (kzInt)parameter2)

{

firstFunction(parameter1);

}

else if ((parameter1 % 2) == 0)

{

secondFunction();

}

\*out\_returnValue1 = 0;

}

TODO: Improve the example

### C#

An example function:

public int MethodName(int parameter1, byte parameter2)

{

int returnValue = 0;

if (parameter1 < (int)parameter2)

{

returnValue = FirstFunction(parameter1);

}

else if (parameter1 % 2 == 0)

{

returnValue = SecondFunction();

}

return returnValue;

}

## Naming

Choose names to indicate purpose (e.g. knownDevices) rather than implementation (e.g. deviceList); the implementation is already visible in the declaration.

Avoid abbreviations and compound words: “pointer” is better than “ptr”. A short word is better than a shortened word. Simple names can be used in some restricted cases such as in for loops.

All function names should have a verb. There are some exceptions currently in the Engine (e.g. kzcStringLength), but don't use those as an example.

### Files and folders

#### C / C++

Use lower\_case file and folder names.

C source files have .c extension, C include files have .h extension.

C++ source files have .cpp extension, C++ include files have .hpp extension.

#### C#

Use UpperCamelCase file and folder names. Directory and file names must conform to namespace and class/struct names respectively.

### Constants

Use ALL\_CAPS for constants.

### Variables

Use count postfix for counts.

Use plural names for arrays and for other data structures holding multiple items of the same type. In other words name the variable itself, not single elements inside it.

Use lowerCamelCase for variable names.

#### C

Use prefix s\_ for static variables, but avoid these.

Use prefix g\_ for global variables, but avoid these.

### Functions (C-Specific)

Use lowerCamelCase for function names.

Return value is usually reserved for error codes thus output parameters are used instead. Use prefix out\_ for output parameters and place them as the last parameters. Do not assign to output parameter until the very end of the function. Never read values from output parameters.

### Classes and structs

Use UpperCamelCase for class and struct names.

#### C / C++

Use the same naming conventions inside structs and classes as outside them.

#### C#

Use lowerCamelCase for private and protected field names (both instance and static).

Use UpperCamelCase for public fields and all properties.

Use UpperCamelCase for method names (both instance and static).

Use lowerCamelCase for method arguments.

In class constructors use the same name as the constructor argument as the corresponding field name and use this object initialize the field. This removes the need to pre/postfix the constructor arguments.

internal class FooClass

{

private int barField;

public FooClass(int barField)

{

this.barField = barField;

}

}

### Enums

Use UpperCamelCase for enum names. Use ALL\_CAPS for enum constants.

## Safety

### Functions

Do not overwrite function input parameters with new values. Only write to local variables and to the output parameters.

Function parameter order is input parameters first, output parameters last. Don’t use parameters which are both read and written.

### Initialization

#### C / C++

Initialize all struct member variables in every “constructor. Never initialize local variables with a safety value and overwrite it later with the real value as this will just prevent static code analysis tools from finding incorrect control flow.

#### C#

Class/Struct members are initialized to default values by the CLR, therefore constructors need to initialize only the necessary ones.

### Type casting

Use explicit type casting even if it is not required. Examples:

kzFloat pii = 3.14f; kzInt s = (kzInt)pii;

kzInt pii = 3; kzFloat s = (kzFloat)pii;

void\* userData = ...; struct Type\* value = (struct Type\*)userData;

An exception is when casting to void\*, don’t do it explicitly.

#### C

kzInt\* p = (kzInt\*)malloc(sizeof(kzInt));

### Scopes

Declare and initialize local variables as near as possible of the first usage.

Do not re-use any variables. Re-using variable names such as i and j is ok.

#### C

Using extra scope { } can be used in C to declare variables later than at the top of the function.

#### C# / C++

Extra scopes { } are a good way to restrict local variable scopes and to divide code into logical pieces.

### Types (C-Specific)

Always work with predefined unsigned or signed type. See Chapter 8.1 for details.

Never use numeric types for Booleans. Also don’t use implicit conversions between these types. For example kzInt x = 1; if (x) is not allowed, use if (x != 0) instead. Same with pointers, use if (pointer != KZ\_NULL) instead of if (pointer).

### Typedefs (C-Specific)

Use typedef for:

* Complex function pointers
* Kanzi uses typedef to wrap standard types

Don’t use typedef:

* Structs
* Pointers

### Consts

#### C / C++

Declare variables as consts whenever possible. This gives a compile-time checking for accidental variable modifications. It might also help compiler for optimizing.

Do not use const for function parameters, which are not pointers.

Usage of consts is easiest to explain with the following examples:

/\* Constant variable \*/

const int CONSTANT = value;

int const CONSTANT = value;

/\* Constant pointer, can’t assign a new value to the pointer \*/

int\* const pointer = &target;

/\* Pointer to constant memory, can’t modify the data \*/

const int\* data;

int const\* data;

/\* Constant pointer to constant memory, can’t modify either one \*/

const int\* const data = &source;

int const\* const data = &source;

/\* Pointer to constant array of pointers to constant memory \*/

const char\* const\* buffers;

char const\* const\* buffers;

As seen in the examples above, it is possible to place the const modifier of the actual variable in two different places. Use the following rules to determine appropriate place: For simple types place const in the beginning. For complex pointer types place const after the base type it modifies. For example:

const int CONSTANT = 10;

const void\* pointer;

void const\* const\* doublePointer;

#### C#

Declare variables as *const* or *readonly* whenever possible. This gives a compile-time checking for accidental variable modifications. It might also help compiler for optimizing.

* Const
  + implicitly static
  + evaluated at compile time
  + initialized in declaration
* Readonly
  + evaluated at run time
  + initialized in declaration or in constructor

### Macros (C-Specific)

Avoid macros. They can be useful, but extremely hard to follow when trying to understand the code. Macros aren’t type safe either.

Never introduce macros without discussing with other developers first.

There are several widely used macros in Kanzi. These are special cases with common agreement around the developers.

Macros are named in the same way as functions (camelCase) or constants (UPPER\_CASE) depending on the type of the macro.

# Documentation

Comment the functionality in API. Comment implementation inside the function body. All functions, non-local variables, structs etc. should be documented with Doxygen formatting. Also all members of structs, classes etc. should be documented. The same does not apply to parameters, instead only non-trivial parameters needs to be documented.

Use correct English sentences with proper spelling, capitalization and punctuation.

Write names as they appear in code. Function, variable and type names are lower-case even at the beginning of a sentence. Function names have ’()’ appended.

Use Doxygen formatting for commenting the API.

* 1. **Doxygen formatting**

/\*\* One-line comment before function or other symbol. \*/

struct MyStruct

{

kzUint member; /\*\*< One-line comment after the symbol. \*/

/\*\*

\* You can also use longer comments for struct and enum members,

\* if necessary.

\*/

kzBool anotherMember;

};

/\*\*

\* Multi-line comment before function or other symbol. The first sentence

\* of comments becomes the “brief” description in generated API documentation.

\*/

void function(void);

When a struct is defined in .c file, but the struct is visible in the .h file, comment it as follows.

/\*\*

\* \struct MyStruct

\* This is the description of the struct.

\*/

struct MyStruct;

## Dates

Always use YYYY-MM-DD formatting for dates. It is unambiguous unlike DD/MM and it also makes sorting by date easy. This applies to comments, file names etc.

## Unfinished work

Use “TODO: Description” in comments to indicate unfinished work. Use “FIXME: Description” to indicate bugs in the code.

## File headers

Every file needs to have a copyright header. The copyright year should span from the beginning of the project to the current date, for Kanzi the beginning is 2008.

### C

Use the following header for both header and source files. This acts as a Doxygen comment for the file also.

/\*\*

\* \file

\* File description here.

\*

\* Copyright 2008-2012 by Rightware. All rights reserved.

\*/

### C++

In C++ if a file has one main class, there is no need for Doxygen file description. Instead document the class properly. A file header is still needed for the copyright.

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

// Copyright 2009-2012 by Rightware. All rights reserved.

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

TODO: Check that the formatting is proper Doxygen formatting.

### C#

Use the following header for source files.

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

//// Copyright 2009-2012 by Rightware. All rights reserved.

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

# Error handling

## Assertions

Use assertions to test against code bugs. Never replace error checking with assertions though. For example it is ok to assert parameter values in a private function, but in a public API function an actual error checking is required instead.

## Error checking

### C

Errors are raised by returning a non-zero return value from functions. Return value zero means success. Actual function return values are returned with output parameters. If a function returns a value normally instead of error, the function name should make this clear (e.g. getData()). Error values must be enumerated constants, no magic numbers are allowed. Do not use C++ exceptions.

Errors are checked by comparing function return value against zero. Always check for error, never for success. Don’t use nesting if statements to keep track of success.

Use Kanzi error macro when possible. See Chapter 8.5 for details.

### C# / C++

Use the exception mechanism to deal with error conditions. Throw an exception when a method fails due to an unexpected condition (disk full, out of memory, network connection broken) or when a method is misused (invalid arguments supplied, method called when the object is in wrong state).

Do not use exception catching as a part of the normal control flow of the application.

Do not catch exceptions that are the result of method misuse, correct the misuse instead.

Catch only exceptions that can be handled at the current scope. It is better to crash straight away than continue when the program state is known to be broken. Of course at the top level of the application failing gracefully (notifying user using custom message & saving the data) is better than crashing without any exception handling.

Order exception catching from the most specific to the most general:

try

{

DoSomethingErrorProne();

}

catch(InvalidOperationException invalidOperationException)

{

RemedyTheSituation();

}

catch(SystemException systemException)

{

RemedyTheSituationSomeOtherWay();

}

catch(Exception exception)

{

throw new Exception(

“Extra information about the situation.”, exception);

}

When re-throwing exceptions try to add some extra information to the exception and include the original exception as the inner exception.

# Kanzi framework

## Introduction

Kanzi framework is written in C89, also called ANSI-C.

## Types

Always use the following types instead of the standard C types.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Size (bits) | Sign | Description |
| kzBool | unspecified | unspecified | Enum {KZ\_FALSE, KZ\_TRUE} |
| kzChar | 8 | unspecified | Character |
| kzString |  |  | Const array of characters |
| kzMutableString |  |  | Non-const array of characters |
| kzWchar | 16 | unspecified | Unicode character |
| kzInt | unspecified | signed | Native integer |
| kzUint | unspecified | unsigned | Native unsigned integer |
| kzFloat | 32 | signed |  |
| kzDouble | 64 | signed |  |
| kzU8 | 8 | unsigned |  |
| kzS8 | 8 | signed |  |
| kzU16 | 16 | unsigned |  |
| kzS16 | 16 | signed |  |
| kzU32 | 32 | unsigned |  |
| kzS32 | 32 | signed |  |

Use the following additional types in Kanzi.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Size (bits) | Sign | Description |
| kzsError | unspecified | unspecified | Error code |

## Memory management

Use Kanzi memory manager for allocating memory. Do not assume any default values such as zeros for the allocated memory block. TODO: Example on how to use Kanzi memory manager.

## Logging

Use Kanzi logger for error and verbose logging. Never use direct console functions. TODO: Example on how to use Kanzi logger.

## Error handling

TODO: Explain Kanzi error macro here.

# Kanzi Tool

## Introduction

Kanzi Tool is written mainly in C#. Interoperability with Kanzi Framework is handled using manager C++ wrapper classes.

## Localization

Every end-user-visible string should be localized.

### Code / Code-Behind Strings

Use the localization manager (TODO: implement) to retrieve localized string. The localization manager should fetch the strings from resource files residing in locale-specific satellite assemblies.

### XAML Strings

TODO: Research how to localize string inside XAML files.

# Example code

## kzu\_example.h

/\*\*

\* \file

\* File description here.

\*

\* Copyright 2008-2011 by Rightware. All rights reserved.

\*/

#ifndef KZU\_EXAMPLE\_H

#define KZU\_EXAMPLE\_H

#include <core/kzc\_xyz.h>

#include <system/kzs\_types.h>

#include <system/debug/kzs\_error.h>

#include <system/kzs\_header.h>

KZ\_HEADER\_BEGIN

/\* Forward declarations \*/

struct KzcMemoryManager;

/\*\*

\* \struct KzuExample

\* Description of what Example is for.

\*/

struct KzuExample;

/\*\* Xyz identifier for example. \*/

extern const KzcXyz KZU\_XYZ\_EXAMPLE;

/\*\* Allocates memory and returns a new example. \*/

kzError kzuExampleCreate(const struct KzcMemoryManager\* memoryManager,

int type, struct KzuExample\*\* out\_example);

/\*\* Frees the memory allocated for the example. \*/

kzError kzuExampleDelete(struct KzuExample\* example);

/\*\* Returns the type of the example. \*/

kzInt kzuExampleGetType(struct KzuExample\* example);

KZ\_HEADER\_END

#endif

## kzu\_example.c

/\*\*

\* \file

\* File description here.

\*

\* Copyright 2008-2011 by Rightware. All rights reserved.

\*/

#include "kzu\_example.h"

#include "kzc\_memory\_manager.h"

#define KZU\_EXAMPLE\_TYPE\_BASIC 0

#define KZU\_EXAMPLE\_TYPE\_ADVANCED 1

struct KzuExample

{

kzInt type;

void\* data;

};

kzError kzuExampleCreate(const struct KzcMemoryManager\* memoryManager,

kzInt type, struct KzuExample\*\* out\_example)

{

kzError result;

struct KzuExample\* example;

result = kzcMemoryAllocVariable(memoryManager, example, "Example");

kzErrorForward(result);

example->type = type;

example->data = KZ\_NULL;

\*out\_example = example;

kzSuccess();

}

kzError kzuExampleDelete(struct KzuExample\* example)

{

kzError result;

result = kzcMemoryFreeVariable(example);

kzErrorForward(result);

kzSuccess();

}

kzInt kzuExampleGetType(struct KzuExample\* example)

{

return example->type;

}