Portable Systems Group

NT Utilities Coding Conventions

Author: *David J. Gilman*

Revision 1.1, October 29, 1990
Revision 1.0, October 18, 1990

1. Introduction

This document describes the coding conventions that are used by the **NT OS/2 Utilities** group. Both the document and the conventions are heavily based on the document, "NT OS/2 Coding Conventions" written by Helen Custer and Mark Lucovsky.

There are primarily two reasons why the **NT OS/2 Utilities** group warrants a separate convention. First, work is done on existing code from many different sources. Second, all new code will be written in C++. This requires a number of changes and additions from the convention documented in the above mentioned document.

All code written for **NT OS/2 Utilities** adheres to a common coding style. This style gives the utilities a uniform appearance which allows group members to read, modify, and maintain each other's modules without learning several different coding conventions.

The following items are standardized:

- Module headers
- Function (member and non-member) headers and declarations
- Header file format
- Names of variables, data types (including classes), structure fields, macros, and constants
- Control structure indentation and placement of braces

2. The Existing Code Rule

When existing code is being ported to **NT OS/2**, every effort should be made to maintain the conventions and style that already exist in that code.

3. Module Headers

The following prototype should appear at the beginning of each module. The source to the prototype can be found in the file *\nt\public\oak\inc\modhdr.c.*

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   name-of-module-filename

Abstract:

   abstract-for-module.

Author:

   name-of-author (email-name) creation-date-dd-Mmm-yyyy

[Environment:]

   optional-environment-info (e.g. kernel mode only...)

[Notes:]

   optional-notes

Revision History:

   most-recent-revision-date email-name
      description
         .
         .
         .
   least-recent-revision-date email-name
   description

--*/
```

/ Note that no Revision History will be maintained until after the product has been released. /

The following is a sample of a completed module header:

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   object.hxx

Abstract:

   Definition of the root class for the ULIB class hierarchy.

Author:

   David J. Gilman (davegi) 12-Oct-1990

Environment:

   ULIB, User Mode

Notes:

   Note the PURE VIRTUAL functions.

--*/
```

| The /++ <text> --*/ construct is used by a comment extractor program that will be developed to assist in our documentation efforts.|*

4. Function Headers

In C++ member functions are declared within a class definition. These declarations contain a lot of information and as such will be enhanced by the use of *modifiers.* Some of these modifiers are also used by the function definition.

4.1 Modifiers

There are essentially three different types of modifiers; function specifiers, type specifiers and argument direction. All can be used by function declarations. Those that can also be used in function definitions are noted.

- All member function declarations are preceded by one of the following modifiers:

VIRTUAL
Indicates that the implementation of a member function can be overridden by a derived class

NONVIRTUAL
Indicates that the implementation of a member function can not be overridden by a derived class

STATIC
Indicates that the member function is static and therefore callable without an object instance of the class.

- Member function declarations may also be preceded by the following modifier:

  CONST (definition)
  Indicates that the function returns a constant value (usually a pointer).

- All formal arguments are preceded by one of the following modifiers:

  IN (definition)
  Indicates that the argument is a non-modifiable input value (i.e., call-by-value semantics)

  OUT (definition)
  Indicates that the argument is an address which refers to a variable or structure that will be modified by the function (i.e., call-by-reference semantics)

  IN OUT (definition)
  Indicates that the argument is the address of an input variable or structure that is both read and written by the function (i.e., call-by-reference semantics).

- Formal arguments may also be followed by one of the following modifiers:

  OPTIONAL
  Indicates that an argument can be or NULL (or zero). To determine whether the actual value supplied is NULL, the programmer must use the macro ARGUMENT_PRESENT, which takes the argument and returns a value of

type BOOLEAN. OPTIONAL arguments must be specified by the caller and can occur at any position in the argument list

DEFAULT
Indicates that the argument is optional and need not be specified by the caller. DEFAULT arguments may only occur at the end (*i.e.* right end) of an argument list and must be initialized in the class definition

- Member function declarations may also be followed by the following modifiers:

  CONST (definition)
  Indicates that the member function is *safe*. That is, it does not directly, or indirectly via a call, modify the object's state

  PURE
  Indicates that the member function is a pure virtual function. That is, all derived classes must supply their own implementation

- The order of the arguments in the comment block is the same as the order in which they appear in the function declaration.

4.2 Function Declarations

When a member function is declared in a class definition, it's declaration contains the function prototype and appropriate modifiers. For example:

```
NONVIRTUAL
CONST
POBJECT
GetNext (
  IN  POBJECT    LastObject,
      OUT BOOLEAN    WrapAround
  ) CONST
```

Note that modifiers, types and argument names should be aligned.

4.3 Function Definitions

Below is a prototype function definition and declaration. The definition form is to appear with the implementation of the function. The source to the prototype can be found in the file *\nt\public\oak\inc\prochdr.cxx*.

Note that a form-feed character should appear one line before the "return-type" line. This convention is noted in this document with the string "<form-feed>".

The function declaration follows:

```
<form-feed>
modifier
 .
 .
 .
return-type
procedure-name (
   direction type-name argument-name [modifier],
   direction type-name argument-name [modifier]
     .
     .
     .
   ) [modifier]

/*++

Routine Description:

   description-of-function.

Arguments:

   argument-name - Supplies (IN) | Returns (OUT) description of argument.
     .
     .

Return Value:

   return-value - Description of conditions needed to return value.
 - or -
   None.

--*/

{
 .
 .
 .
}
```

/ Note the space between the procedure name and the opening parenthesis for it's argument list. This is needed so that overloaded operators will be more readable. /

The following is a sample of a completed member function declaration:

```
<form-feed>
NONVIRTUAL
CONST
POBJECT
COLLECTION::GetNext (
   IN POBJECT    LastObject OPTIONAL
   ) CONST

/*++

Routine Description:

   Get the next object from the collection.

Arguments:

   LastObject - Supplies the current object.

Return Value:

   POBJECT - A constant pointer to the next OBJECT in the
         COLLECTION.

--*/

{
  .
  .
  .
}
```

5. Header Files

The following sections define the requirements for inclusion and format of header files.

5.1 Header File Inclusion

5.1.1 Description

There are two types of header files used by the **NT OS/2 Utilities**:

- Header files that are private to a single class:
    - Types, constants etc.

- Inline functions

- A public header file that contains the class declaration and associated types, constants etc.

The naming convention for private header files is <class-name>*p.h*xx. For example, the private header file for the object class, would be called *objectp.h*xx.

The public style of header files are the most important as they define class interfaces. An example can be found in section 1.

Header files should not be nested. That is, one header file should not include another.

5.1.2 Special Header Files

There are two special header files used by the ULIB class library: ulibdef.hxx and ulib.hxx. These files are exceptions to the nested include file rule.

5.1.2.1 Ulibdef.hxx

The file *\nt\private\os2\programs\ulib\inc\ulibdef.hxx* contains global information which is required by all classes and client's of ULIB. It should *not* be directly included. Rather, it will be included by *ulib.hxx*.

5.1.2.2 Ulib.hxx

The file *\nt\private\os2\programs\ulib\inc\ulib.hxx* is the master header file for the ULIB library. It should be included by all classes and clients of ULIB by the statement

```
#include "ulib.hxx"
```

In turn *ulib.hxx* will include, in the correct order, the header files that are needed by a particular class. This will be controlled by symbols of the form

```
_CLASSNAME_
```

which will be defined by the class client.

Class definitions will support this architecture by conditionally expanding to themselves, or to nothing if they have already been expanded.

As mentioned, class writers will use *ulib.hxx*. This will ensure that it is accurate and usable by any class clients. This means that special care should be taken to ensure that private header files are not listed within *ulib.hxx.*

In the example below, if the class definition in *collection.hxx* was not previously referenced, then the macro _COLLECTION_ is defined and the header file is expanded. Otherwise, _COLLECTION_ is already defined and the remainder of the header file is ignored. This results in the header file being included only once.

The following header file style should be used:

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   object.hxx

Abstract:

   Definition of the abstract container class.

Author:

   David J. Gilman (davegi) 12-Oct-1990

Environment:

   ULIB, User Mode

Notes:

Revision History:

--*/

#if ! defined( _COLLECTION_ )
#define _COLLECTION_
  .
  .
  .

//
// body
//

#endif // _COLLECTION_
```

6. Naming

The following sections describe the naming conventions for variables, structure fields, types, constants, and macros.

6.1 Variable Names

Variable names are either in "initial caps" format, or they are unstructured. The following two sections describe when each is appropriate.

Note that the **NT OS/2** system, utilities included, do not use the Hungarian naming convention used in some of the other Microsoft products.

6.1.1 Initial Caps Format

All global variables and formal argument names must use the initial caps format. The following rules define this format:

- Words within a name are spelled out; abbreviations are discouraged.
- The first character of each word in a name is capitalized.
- Acronyms are treated as words, that is, only the first character of the acronym is capitalized.

The following list shows some sample names that conform to these rules:

NumberOfBytes

TcbAddress

BilledProcess

6.1.2 Unstructured Format

Local variables may appear in either the initial caps format, or in a format of the programmer's preference. The following list shows some possibilities for local variable names:

loopindex

LoopIndex

loop_index

6.2 Data Type Names

A set of primitive data types for use in the **NT OS/2 Utilities** is defined in *ulibdef.hxx.* All **NT OS/2 Utilities** software must declare variables using these defined types rather than standard C++ types, where appropriate. The following are some examples of **NT OS/2 Utilities** types:

ULONG

PULONG

VOID

PVOID

BOOLEAN

PBOOLEAN

All new type names should be created in uppercase using **typedef**. Words within the name may either be packed together or separated by underscores. All types should have a corresponding **typedef** which defines a pointer and a reference to the type. The name for the pointer is the type name with a "P" prefix. Similarly the reference is the type name with a "R" prefix.

The following example illustrates how to use **typedef** to create a class type:

```
typedef class COLLECTION : public OBJECT {

   public:

      NONVIRTUAL
      COLLECTION (
         IN ULONG    InitialNumberOfElements,
         IN ULONG    IncrementNumberOfElements DEFAULT = 10
         );

      VIRTUAL
      ~COLLECTION (
         );

      NONVIRTUAL
      POBJECT
      QueryNextElement (
         IN POBJECT    CurrentElement
         ) CONST;

      VIRTUAL
      CONST
      POBJECT
      GetNextElement (
         IN POBJECT    CurrentElement
         ) PURE;

   protected:

      POBJECT       mCollection;

   private:

      ULONG    _InitialNumberOfElements;
      ULONG    _IncrementNumberOfElements;
} POINTER_AND_REFERENCE_TYPES( COLLECTION );
```

Note that there should only be one *public:*, one *protected:* and one *private*: section in each class definition. In addition constructors and destructors should appear at the top of the list followed by logical groupings of other member functions.

C++ does not require a typedef for structures, and enumerated types as it considers them to be types when they are defined. However typedefs should be used so that a pointer and reference to the type are defined at the same time as the underlying type. For example,

```
typedef struct RANGE {
   ULONG    Start
   ULONG    Count;
} POINTER_AND_REFERENCE_TYPES( RANGE );


typedef enum COLLECTION_TYPE {
   Array,
   List,
   Table
} POINTER_AND_REFERENCE_TYPES( COLLECTION_TYPE );
```

6.3 Returning or Accepting Pointers

In order to minimize performance impacts of using objects, the following conventions are used when pointers to objects, or other dynamic structures, are passed to and from an object:

- Member function names that have the prefix:
  - **Query**
    Return a pointer to an object which will be de-allocated by the client.
  - **Get**
    Return a constant pointer which will be de-allocated by the object.
  - **Set**
    Take a pointer to an object which will be de-allocated by the object.
  - **Put**
    Take a pointer to an object which will be de-allocated by the client.

6.4 Structure Fields, Class Member Data and Enumeration Constants

Notice from the above examples that structure field names, enumeration constants and class member data should follow initial caps format. They should not have field name prefixes tied to a type.

The subtle exception to this rule is for member data. The names used for a class' member data should be preceded by an '_' so that they can be more easily recognized in member function implementations.

6.5 Macro and Constant Names

All macros and manifest constants should have uppercase names. Words within a name may either be packed together, or separated by underscores.

The following statements illustrate some manifest constant and macro names:

```
#define PAGE_SIZE   4096
#define CONTAINING_RECORD(address, type, field) \
      ((type *)((LONG)(address) - \
            (LONG)(&((type *)0)->field)))
```

Any macro that is likely to be replaced by a function at a later time should use the naming conventions for functions.

In C++ it is preferable to use constant variables and inline functions instead of manifest constants and macros.

7. Indentation and Placement of Braces

Source files should contain real tab characters. Tab stops should be set to four characters. This can be accomplished for the following tools by adding the following entry to the *tools.ini* file:

```
[pwb]
   entab:1
   filetab:4
   tabstops:4
   realtabs:yes

[list]
   tabamt:4

[ppr]
   flags = -e 4
```

/ The entries for list and ppr do not work. /

The following skeletal statements illustrate the proper indentation and placement of braces for C++ control structures.

```
<form-feed>
INT
FooBar(
   INT ArgumentOne,
   PULONG ArgumentTwo
   )

/*++

Routine Description:

   This is the routine description.

Arguments:

   ArgumentOne - Supplies the value for argument 1.

   ArgumentTwo - Supplies the address of argument 2.

Return Value:

   0 - Success

   1 - Failure

--*/

{
   //
   // Local variables are indented one tab (tabs are 4 spaces)
   //

   ULONG LocalVariable1;
   LONG Counter;

   //
   // for loops
   //    - all for loops must have braces
   //    - closing brace is at same indentation level as
   //      for statement
   //

   for ( Counter = 0; Counter < 10; Counter++ ) {

      //
      // Body of loop
      //

   }

   //
   // if statement
   //
```

```
if ( Counter == 0 ) {

    //
    // Then statements
    //

}

//
// if then else
//

if ( Counter == 1 ) {

    //
    // Then statements
    //

} else {

    //
    // Else statements
    //

}

//
// switch statement
//

switch ( Counter ) {

case 1 :

    //
    // case 1 statements
    //
    break;

case 2 :

    //
    // case 2 statements
    //
    break;

default :

    //
    // default case
    //
    break;
```

```
    }
}
```

8. Language Usage Guidelines

The **NT OS/2 Utilities** are written in portable C++ as defined by "The Annotated C++ Reference Manual" written by Margaret A. Ellis and Bjarne Stroustrup[1]. Care should be taken not to write any code that breaks with this language definition or with the ANSI C standard. When the two language definitions are at odds, side with the C++ definition.

8.1 Known Problems

There are two known problems that have been encountered by the **NT OS/2** group:

- Left Hand Side Typecasts

```
ULONG          i;
( FLOAT ) i = 2.0;    // PROBLEM!
```

- Zero Length Arrays in Structures

```
struct X {
        ULONG          i;
        ULONG          arr[ ];    // PROBLEM!
};
```

Fortunately, C++ will not allow either of these constructs.

8.2 C++ Specific Guidelines

Following are a number of C++ specific guidelines which will aid in readability, consistency and debugging:

- File names should have the following extensions:
  - **hxx**
    for class definitions and related types and constants

[1] This book is also referred to as the "ANSI Base Document".

- **inl**
  for inline function implementations
- **cxx**
  for non-inline function implementations.

- In order to benefit from C++'s strong type checking:
  - Dynamic allocation and de-allocation should be performed with the C++ *new* and *delete* operators.
  - Constant, global variables should be used in lieu of pre-processor definitions.
- Do not declare inline functions within a class definition.
- Declare inline functions in the appropriate *.inl* file as described above and as shown in 2.
- Do not use multiple inheritance.
- Avoid using global, static objects.
- Do not use the C++ specific form of casting (*i.e.* ULONG(x)).
- Do not declare protected member data (use private data and access member functions).

8.3 Debugging Support

Debug code is enabled by the compiler symbol *DBG*. Debug code should not be defined within the body of non-debug code. Instead a macro should be defined which conditionally compiles to a, possibly inlined, function call. For example (from *ulibdef.hxx*),

```
#if defined( DBG )
        #define DebugAssert( b )    DbgAssert( b )
#else
        #define DbgAssert
#endif // DBG
```

Programmers should use the symbol *REGISTER* instead of the C++ storage specifier, *register*. This will disable register storage when *DBG* is enabled.

The macro, *INLINE_INCLUDE*, should be used to conditionally (depending on *DBG*) include (or compile) inline functions. See 3 for an example. Note that usage of this macro will cause the *DBG* symbol to effect the list of source files to be compiled. This macro will make tracing and stepping of inline functions easier.

9. Appendix A - Example : Class EXAMPLE

9.1 Ulib.hxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   ulib.hxx

Abstract:

   Master include file for the ULIB class hierarchy.

Author:

   David J. Gilman (davegi) 19-Oct-1990

Environment:

   ULIB

Notes:

Revision History:

--*/

#include "ulibdef.hxx"
#include "object.hxx"
  .
  .
  .
#if defined( _EXAMPLE_ )

//
//  include files that the EXAMPLE class definition (not
//  implementation) is dependent upon
//

   #include "example.hxx"

#endif // _EXAMPLE_
```

9.2 Example.hxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   example.hxx

Abstract:

   Definition for class EXAMPLE.

Author:

   David J. Gilman (davegi) 19-Oct-1990

Environment:

   ULIB

Notes:

Revision History:

--*/

#if ! defined( _EXAMPLE_ )
#define _EXAMPLE_

typedef class EXAMPLE : public OBJECT {

   public:

      NONVIRTUAL
      EXAMPLE (
         IN ULONG    Value
         );

      VIRTUAL
      ~EXAMPLE (
         );

      VIRTUAL
      ULONG
      SetValue (
         IN ULONG    Value
         );

      NONVIRTUAL
      ULONG
      QueryValueDoubled (
         ) CONST;
```

```
    private:

        ULONG    mValue;

} POINTER_AND_REFERENCE_TYPES( EXAMPLE );

INLINE_INCLUDE( example.inl );

#endif // _EXAMPLE_
```

9.3 Examplep.hxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

    examplep.hxx

Abstract:

    Private header file for class EXAMPLE.

Author:

    David J. Gilman (davegi) 19-Oct-1990

Environment:

    ULIB

Notes:

Revision History:

--*/

#if ! defined( _EXAMPLE_P )
#define _EXAMPLE_P

CONST DOUBLEVALUE = 2;

#endif // _EXAMPLE_P
```

9.4 Example.inl

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   example.inl

Abstract:

   Inline functions for class EXAMPLE.

Author:

   David J. Gilman (davegi) 19-Oct-1990

Environment:

   ULIB

Notes:

Revision History:

--*/

#define _EXAMPLE_
#include "ulib.hxx"

#include "examplep.hxx"

<form-feed>

ULONG
EXAMPLE::QueryValueDoubled (
   ) CONST

/*++

Routine Description:

   Compute double the value.

Arguments:

   None.

Return Value:

   ULONG - double the value.

--*/
```

```
{
   return( mValue * DOUBLEVALUE );
}
```

9.5 Example.cxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   example.cxx

Abstract:

   Implementation for class EXAMPLE.

Author:

   David J. Gilman (davegi) 19-Oct-1990

Environment:

   ULIB

Notes:

Revision History:

--*/

#define _EXAMPLE_
#include "ulib.hxx"

#include "examplep.hxx"


<form-feed>

EXAMPLE::EXAMPLE (
   ULONG    Value
   )

/*++

Routine Description:

   Construct an EXAMPLE object.

Arguments:

   Value - Initial value for the EXAMPLE object.

Return Value:

   None.

--*/
```

```
{
   mValue = Value;
}


<form-feed>

EXAMPLE::~EXAMPLE (
   )

/*++

Routine Description:

   Destroy an EXAMPLE object.

Arguments:

   None.

Return Value:

   None.

--*/

{
   DbgPrint( "Example destroying...\n" );;
}
```

```
<form-feed>

ULONG
EXAMPLE::SetValue (
    ULONG    Value
    )

/*++

Routine Description:

    Set an EXAMPLE's value.

Arguments:

    Value - The value to set in EXAMPLE.

Return Value:

    ULONG - The set value.

--*/

{
    return( mValue = Value );
}
```

9.6 Client.cxx

```
/*++

Copyright (c) 1990  Microsoft Corporation

Module Name:

   client.cxx

Abstract:

   Sample usage of class EXAMPLE.

Author:

   David J. Gilman (davegi) 19-Oct-1990

Environment:

   ULIB

Notes:

Revision History:

--*/

extern "C" {
   #include <stdio.h>
};

#define _EXAMPLE_
#include "ulib.hxx"
```

```
<form-feed>

VOID
main (
    )

/*++

Routine Description:

    Constructs and demonstrates usage of an EXAMPLE object.

Arguments:

    None.

Return Value:

    None.

--*/

    EXAMPLE    example = 4;
    PEXAMPLE   pexample;

    pexample = &example;

    printf( "Value = %d\n", example.QueryValueDoubled( ) / 2 );
    printf( "Value = %d\n", pexample->QueryValueDoubled( ) / 2 );
}
```

Revision History

Revision 1.1, October 29, 1990 - djg

1. Changed definition of IN argument modifier.
2. Added IN OUT argument modifier.
3. Added POINTER_AND_REFERENCE_TYPES macro.
4. Added STATIC member modifier.
5. Clarified OPTIONAL versus DEFAULT argument modifiers.
6. Changed member data prefix from 'm' to '_'.
7. Added style guidelines for public, private and protected sections.
8. Miscellaneous edits for clarity.

Revision 1.0, October 18, 1990 - djg

1. Incorporated comments from stever and loup.
2. Added reference types.
3. Fixed formatting errors.

Original Draft, October 16, 1990 - djg