Portable Systems Group
NT OS/2 Coding Conventions
Author: Mark Lucovsky, Helen Custer
Revision 1.5, January 21, 1991

Introduction

All code written for NT OS/2 by members of the Portable Systems Group adheres to a common coding style. This style gives the system a uniform appearance that 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 headers and declarations
- Header file format
- Names of variables, data types, structure fields, macros, and constants
- Control structure indentation and placement of braces

Module Headers

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

```
/*++

Copyright (c) 1989  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

--*/
```

```
The following is a sample of a completed module header:
/*++

Copyright (c) 1989  Microsoft Corporation

Module Name:

    pool.c

Abstract:

    This module contains the pool allocator for the NT OS/2
    executive.

Author:

    Mark Lucovsky (markl) 16-Feb-1989

Environment:

    Kernel mode only.

Revision History:

    22-Feb-1989     markl

        Modified module to conform to the new naming and coding
        standards agreed to 21-Feb-1989.

    20-Feb-1989     markl

        Added module and function headers.

--*/
```

Note that the revision history portion is not completed. Until we get further along in the project, we will not keep a revision history.

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

Function Headers

The following is a prototype function declaration. This declaration is to appear with the implementation of the function. The source to the prototype can be found in file \nt\bak\inc\prochdr.c.
Notice the following details in the function declaration:

- 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>".
- All formal arguments are preceded by one of the following macro definitions:

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

OUT 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 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)

- The OPTIONAL macro appears after a formal argument of type pointer, HANDLE, or ULONG when the function accepts either a NULL or non-NULL value. To determine whether the actual value supplied is NULL or non-NULL, the programmer must use the macro ARGUMENT_PRESENT, which takes the pointer, HANDLE, or ULONG variable as an argument and returns a value of type BOOLEAN.

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

- The function declaration follows:

```
<form-feed>
return-type
function-name(
    direction type-name argument-name,
    direction type-name argument-name...
    )

/*++

Routine Description:

    description-of-function

Arguments:

    argument-name - [Supplies | Returns]  description-of-argument
    .
    .
    .

Return Value:

    return-value - description-of-return-value
                                    -or-
                           None

 --*/

{
    .
    .
    .
```

```
}

The following is a sample of a completed function declaration:
<form-feed>
VOID
IoBuildPartialMdl(
    IN PMDL SourceMdl,
    IN PMDL TargetMdl,
    IN PVOID VirtualAddress,
    IN ULONG Length OPTIONAL
    )

/*++

Routine Description:

This routine maps a portion of a buffer as described by an
MDL.  The portion of the buffer to be mapped is specified via a virtual
address and an optional length.  If the length is not supplied, then
the remainder of the buffer is mapped.

Arguments:

    SourceMdl - MDL for the current buffer.

    TargetMdl - MDL to map the specified portion of the buffer.

    VirtualAddress - Base of the buffer to begin mapping.

    Length - Optional length of buffer to be mapped;  if zero,
                  remainder.

Return value:

    None.

When a function is declared externally in a header file, its
declaration contains only the function prototype and not the comment
section. For example:
VOID
IoBuildPartialMdl(
    IN PMDL SourceMdl,
    IN PMDL TargetMdl,
    IN PVOID VirtualAddress,
    IN ULONG Length OPTIONAL
    );
```

Header Files

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

Header File Inclusion

There are three types of header files in the NT OS/2 system:

- Header files that are private to a single operating system component (the kernel or the I/O system, for example)
- A header file that is shared by the internal components of the operating system (the kernel and the executive)
- A public header file that defines external application programming interfaces (APIs) for system components outside the kernel and executive

Each component of the operating system has a private header file. The naming convention for these header files is <component-name>p.h. For example, the private header file for kernel component, ke, is called kep.h.

The NT OS/2 shared header file, \nt\private\src\ntos\inc\ntos.h, is included by each component of the executive and by the kernel, using the following statement:
#include "ntos.h"

(This file is included by a component's private include file.)

File ntos.h contains a list of #include statements, one for each operating system component. Each operating system component has a corresponding header file that defines prototypes for the functions that are shared with other components within the executive. The naming convention for these header files is <component-name>.h. For example, the header file containing shared prototypes for kernel component, ke, is called ke.h.

The public header file, \nt\sdk\inc\ntos2.h, is included by all components outside the NT OS/2 kernel and executive, using the following statement:
#include <ntos2.h>

Header File Format

Modules should be able to nest header files without causing multiple definition problems. To accomplish this, each header file should be conditionally expanded to itself, or to nothing if it has already been expanded.

In the example below, if the module pool.h was not previously included, then the macro _POOL_ is defined and the header file is expanded. Otherwise, _POOL_ 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) 1989  Microsoft Corporation

Module Name:

    pool.h
```

```
Abstract:

    This module defines the NT OS/2 pool data structures and
    function prototypes.

Author:

    Mark Lucovsky (markl) 16-Feb-1989

Revision History:

--*/

#ifndef _POOL_
#define _POOL_

#include "ntdef.h"
#include "list.h"
#include "process.h"

typedef enum _POOL_TYPE {
    NonPagedPool,
    PagedPool
    } POOL_TYPE;

#endif // _POOL_
```

Note that if module list.h were shown, the conditional would appear as follows:

```
   #ifndef _LIST_
   #define _LIST_

   //
   // body
   //

   #endif // _LIST_
```

Naming

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

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 does not use the Hungarian naming convention used in some of the other Microsoft products.

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
```

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
```

Data Type Names

A set of primitive data types for use in the NT OS/2 system is defined in the file \nt\sdk\inc\ntdef.h. All NT OS/2 software must declare variables using these defined types rather than standard C types, where appropriate. The following are some examples of NT OS/2 types:

```
VOID
PVOID
QUAD
UQUAD
STRING
TIME
```

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 to the type. The name for the pointer is the type name with a "P" prefix.

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

```
typedef struct _POOL_LIST_HEAD {
    ULONG CurrentFreeLength;
    ULONG TotalEverAllocated;
    LIST_ENTRY ListHead;
} POOL_LIST_HEAD, *PPOOL_LIST_HEAD;
```

The following example illustrates how to use typedef to create an enumerated type:

```
typedef enum _POOL_TYPE {
    NonPagedPool,
    PagedPool,
    MaxPoolType
    } POOL_TYPE;
```

Structure Field Names and Enumeration Constants

Structure field names should follow initial caps format. They should not have field name prefixes tied to a type. The following is a sample structure:

```
typedef struct _POOL_LIST_HEAD {
    ULONG CurrentFreeLength;
    ULONG TotalEverAllocated;
    LIST_ENTRY ListHead;
} POOL_LIST_HEAD, *PPOOL_LIST_HEAD;
```

As illustrated in the previous section, enumeration constants should also follow initial caps format.

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 macro and manifest constant names:

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

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

Indentation and Placement of Braces

The following skeletal statements illustrate the proper indentation and placement of braces for C control structures. In all cases, indentations consist of four spaces each.
All control structures should routinely use braces even if there is only a single statement that will be executed.
<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
//
//    - All if statements should use braces
//

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;

    }
}
```

Constructs to Avoid

NT OS/2 is written in portable, ANSI C. Due to differences in C compilers, there are a number of coding constructs that need to be avoided in order to promote portability.

Left Hand Side Typecasts

Some C compilers allow the cast operator on the left hand side of an assignment. This is not allowed by standard C and must be avoided in NT OS/2.

Zero Length Arrays in Structures

Zero length arrays embedded in structure definitions are not handled uniformly by all C compilers. They should not be used in NT OS/2.

Revision History

Original Draft 1.0, February 21, 1989 - ml
Revision 1.1, February 23, 1989 - ml
Revision 1.2, May 5, 1989 - hkc

1. Extracted coding guidelines from exec.txt and converted text to Word.
2. Added text regarding primitive data type definitions.
3. Added text and example describing OPTIONAL arguments.
4. Added text regarding the inclusion of header files in implementation modules.
5. Style edit.

Revision 1.3, May 11, 1989 - Incorporated group comments. hkc
Revision 1.5, January 21, 1991 tonye

1. Emphasized that all control structures must use braces.