Additional Considerations

When porting your code, consider the following points:

- The following assumption is no longer valid:

```
#ifdef _WIN32 // Win32 code
...
#else         // Win16 code
...
#endif
```

  However, the 64-bit compiler defines _WIN32 for backward compatibility.

- The following assumption is no longer valid:

```
#ifdef _WIN16 // Win16 code
...
#else         // Win32 code
...
#endif
```

  In this case, the else clause can represent _WIN32 or _WIN64.

- Be careful with data-type alignment. The **TYPE_ALIGNMENT** macro returns the alignment requirements of a data type. For example: `TYPE_ALIGNMENT( KFLOATING_SAVE ) == 4` on x86, 8 on Intel Itanium processor`TYPE_ALIGNMENT( UCHAR ) == 1` everywhere
  As an example, kernel code that currently looks like this:

```
ProbeForRead( UserBuffer, UserBufferLength, sizeof(ULONG) );
```

  should probably be changed to:

```
ProbeForRead( UserBuffer, UserBufferLength, TYPE_ALIGNMENT(IOCTL_STRUC) );
```

Automatic fixes of kernel-mode alignment exceptions are disabled for Intel Itanium systems.

- Be careful with NOT operations. Consider the following:

```
UINT_PTR a;
ULONG b;
a = a & ~(b - 1);
```

The problem is that ~(b–1) produces "0x0000 0000 xxxx xxxx" and not "0xFFFF FFFF xxxx xxxx". The compiler will not detect this. To fix this, change the code as follows:

```
a = a & ~((UINT_PTR)b - 1);
```

- Be careful performing unsigned and signed operations. Consider the following:

```
LONG a;
ULONG b;
LONG c;

a = -10;
b = 2;
c = a / b;
```

The result is unexpectedly large. The rule is that if either operand is unsigned, the result is unsigned. In the preceding example, a is converted to an unsigned value, divided by b, and the result stored in c. The conversion involves no numeric manipulation.

As another example, consider the following:

```
ULONG x;
LONG y;
```

```
LONG *pVar1;
LONG *pVar2;

pVar2 = pVar1 + y * (x - 1);
```

The problem arises because x is unsigned, which makes the entire expression unsigned. This works fine unless y is negative. In this case, y is converted to an unsigned value, the expression is evaluated using 32-bit precision, scaled, and added to pVar1. A 32-bit unsigned negative number becomes a large 64-bit positive number, which gives the wrong result. To fix this problem, declare x as a signed value or explicitly typecast it to **LONG** in the expression.

- Be careful when making piecemeal size allocations. For example:

```
struct xx {
   DWORD NumberOfPointers;
   PVOID Pointers[100];
};
```

The following code is wrong because the compiler will pad the structure with an additional 4 bytes to make the 8-byte alignment:

```
malloc(sizeof(DWORD) + 100*sizeof(PVOID));
```

The following code is correct:

```
malloc(offsetof(struct xx, Pointers) + 100*sizeof(PVOID));
```

- Do not pass `(HANDLE)0xFFFFFFFF` to functions such as **CreateFileMapping**. Instead, use **INVALID_HANDLE_VALUE**.
- Use the proper format specifiers when printing a string. Use %p to print pointers in hexadecimal. This is the best choice for printing pointers. Microsoft Visual C++ supports %I to print polymorphic data. Visual C++ also supports %I64 to print values that are 64 bits.