Bionic Buffalo Tech Note #10:

Portable Object Code

last revised Sunday 19 October 1997

What is Portable Object Code?

Portable object code is actually source code, but it is difficult to read and understand.

Unlike "normal" object code, portable object code can be recompiled for different target environments. It is not restricted to run on any specific processor architecture.

Formerly, portable object code was called source code in our documentation, but the expectation of intelligibility caused confusion. Therefore, we have begun to call it "portable object code" instead.

This Tech Note explains the purpose of portable object code, and shows examples.

The Strict Meaning of "Source Code"

The most common source code languages are well defined in various specifications. For example, ANSI X3.159:1989 defines the language commonly called Standard C. There are similar specifications for other languages. When these specifications are used as definitions, then portable object code is, in fact, source code.

Bionic Buffalo adheres to the formal definitions of these programming languages when creating portable object code. This allows compatibility with standard compilers and other tools, thereby improving portability.

The specifications allow a lot of freedom in chosing variable names, organizing nested loops, and using *go to* statements and other constructs. No attempt is made to create portable object code which is readible or sensible to humans. However, compilers (which are oblivious to the fine points of style) are very happy to digest portable object code.

WHY NOT SENSIBLE SOURCE CODE?

Our portable object code comes from two sources:

- Some portable object code is made by source code generators, which are programs that take a higher-level representation of a program and generate a lower-level realization in one of the standard programming languages. For example, we have tools which generate menus and forms for interactive data entry and queries, and these tools may emit obscure source code.

- Some code is deliberately obscured to hinder modification or reverse engineering.

Bionic Buffalo sometimes distributes portable object code as an alternative to standard object code. By providing obscure source, we can distribute programs which can be recompiled to run in alternative environments or on alternative platforms.

In some cases (when the code is created automatically by compilers and other tools), intelligible source code is not possible. In other cases, it is not available from us for the same reason that most software vendors do not offer source code: we are protecting our proprietary technology.

AN EXAMPLE

This section contains portions of an example program in one portable object code format. This example is from a version of the CORBA library. It is a routine which tests the type of a given object.

```
CORBA_boolean CORBA_Object_is_a ( CORBA_Object _gmx_40621,
CORBA_char * _gmx_302049,
CORBA_Environment *_gmx_234811 )
{
_gmx_148457 * _gmx_377941 ;
char * _gmx_305598 ;
_gmx_377941 = (_gmx_148457 *) _gmx_234811 ;
_gmx_21574 ( _gmx_377941 ) ;
_gmx_305598 = _gmx_305995 ( _gmx_377941, _gmx_40621 ) ;
if ( _gmx_377941 -> _gmx_286484 != CORBA_NO_EXCEPTION )
return 0 ;
if ( _gmx_305598 == NULL ) goto _gmx_231095;
goto _gmx_15736;
_gmx_231095:
if ( _gmx_302049 == NULL ) return 1 ;
if ( ! strcmp ( _gmx_302049, "" ) ) return 1 ;
return 0 ;
_gmx_15736:
if ( ! strcmp ( _gmx_305598, "" ) ) goto _gmx_131669;
goto _gmx_282251;
_gmx_131669:
```

```
free ( _gmx_305598 ) ;
if ( _gmx_302049 == NULL ) return 1 ;
if ( ! strcmp ( _gmx_302049, "" ) ) return 1 ;
return 0 ;
_gmx_282251:
if ( strcmp ( _gmx_302049, _gmx_305598 ) ) goto _gmx_197896;
goto _gmx_391337;
_gmx_197896:
free ( _gmx_305598 ) ;
return 0 ;
_gmx_391337:
free ( _gmx_305598 ) ;
return 1 ;
}
```

This code is difficult to understand. However, there are several important points to be noticed:

- The code is valid standard C, and is acceptable to any standard C compiler.
- Well-known external names (such as the name of the routine itself) are not modified.
- Some library routines (such as ***strcmp()***) which might cause in-line code generation by some compilers are not changed.

The last two points means that standard public routines may be replaced. For example, this routine might be replaced with a user-supplied version which performs differently, or a user-supplied preamble might be placed in front of the obscure source.

Other Forms

Although the example above is typical, there are other forms of portable object code to be found in Bionic Buffalo products. The form depends on which tool was used to create the program: some forms are more readable than this example, and some are less readable. In some cases, the line boundaries do not correspond to natural source-language statement boundaries, and in others there is extensive use of macros.

The main point is: **portable object code is not intended to be intelligible to humans**.

Elements of Portability

In spite of its appearance, portable object code is highly portable. We use ***lint*** and other tools extensively to verify our sources, whether or not generated by automated tools. Most important, we maintain a common code base across a wide variety of platforms.

Portable object code will compile and run properly on machines with varying word sizes, byte orders, and data representations. It is also tested against numerous operating systems. The same code is used for all machines: we do not maintain a different set of code for each environment.

When it is impossible to create a portable routine, then the non-portable portions are isolated to a separate library called a *portability adaptor*. Portability adaptors are written for each runtime environment, and contain the procedures which are not portable to all other supported environments.

As an example, record locking is handled differently among different operating systems. Because of this, record locking routines with a standardized API are implemented separately in the portability adaptors. Because the API is standardized, the portable object code is the same from one platform to the next.

It is relatively easy to write a portability adaptor for a new environment, especially if an existing one from a similar environment is used as a model.