Building Portable NEXTSTEP Applications

The operating system, client processes, development tools, and software libraries that comprise NEXTSTEP are standard for all computers on which NEXTSTEP runs. This means that, in general, when you port your NEXTSTEP application to a new computer, you won't have to redesign your code to achieve expected behavior. All the pieces are there, and as they work on one machine, so will they work on every other.

If you follow the NEXTSTEP user interface guidelines and avoid hard-wired data values, then your application will probably be portable: It should run properly on all configurations of a given architecture and should need only to be recompiled to run on a new architecture. But few applications are perfect. Yours might fall prey to the differences between computers, requiring a bit of fine-tuning before it will work with a new configuration or on a new architecture. It's anticipated that all such necessary changes will be of the type that generalize your codeÐyou should rarely need to ªspecial-caseº your code to adapt to a particular computer.

This paper describes some of the differences between computers that can run NEXTSTEP and suggests ways to avoid configuration- or architecture-specific code that could make your application non-portable. It's divided into two parts:

- The first part discusses differences between hardware configurations, such as differences in screen size and color capability, or between types of keyboards.
- The second part deals with differences in data representation between computer architectures. Almost all problems that arise in this arena can be cured by adhering to the tenets of good programming.

Hardware Considerations

Every computer on which NEXTSTEP runs will certainly possess the three hardware pieces that have come to be regarded as obligatory: a screen, a keyboard, and a mouse. However, the attributes of these devices aren't the same on all computers. The sound capabilities of computers also vary widely. The following sections describe the facilities that allow your application to query a computer for the attributes of its hardware devices, and warn against assumptions about the computer's configuration that can make your application less portable.

The Screen

Not all screens are the same size, nor do they provide the same color support. Therefore, a portable application shouldn't depend on a particular screen size or color capability.

To guarantee that windows appear on-screen in an appropriate manner, your application should always place them relative to the edges of the screen, rather than in absolute positions. Interface

Builder's Size Inspector can be used to set the position of a window relative to the edges of a screen, so when the window is displayed, it automatically appears in the expected place. If a window or panel must have a minimum size, try to keep it at a reasonable value (computer screens are usually at least 640 × 480 pixels). Non-resizable windows and panels should be given similar consideration about their sizes.

The Window class guarantees that windows are displayed in such a way that the user can manipulate them. If a window's position would result in its title bar being off-screen, the window will move itself enough so that the title bar does appear. Similarly, if a window is so tall that its resize bar would be below the bottom of the screen, the window will change its height to show the resize bar. All of Window's placement and movement methods perform this forcing to the screen, so if you need a standard window that's not visible, you should use the **orderOut:** method to remove it from the screen list, rather than trying to position the window out of the screen's bounds.

To handle the different color capabilities of screens, your application should use the NXImage class for bitmapped images. NXImage automatically uses the most appropriate image representation for a given screen. You should also make use the View method **shouldDrawColor**. This method lets application choose appropriate grayscale equivalents of colors (to avoid dithering on a grayscale screen, for example).

If you need more control over screen information than that provided by the above facilities, you can use the NXScreen structure, which represents the attributes of a screen. This structure is declared in **appkit/screens.h**. The Application Kit's Application and Window classes provide methods that return the NXScreen structures that represent the screens that are available to your application.

For more information on using the Application, Window, View, and NXImage classes in handling different screen configurations, see the specifications of those classes in **/NextLibrary/Documentation/NextDev/GeneralRef/02_ApplicationKit**. For more information on using Interface Builder, see *NEXTSTEP Development Tools and Techniques*.

The Mouse

All mice have at least one buttonÐsome have two. If your application was designed for a NeXT Computer, you may have used the second button that all NeXT mice have. This obviously subverts portability to a configuration that has a one-button mouse.

You should never depend on having a two-button mouse; the NeXT user interface guidelines urge you to ignore the second button on a NeXT mouse. However, a slightly less strict reading of the rule has it that at the very least, you must make sure that all second-button operations can be performed through some other method. For example, if you use the second button to create a ªspecialº selection, you might provide a menu item that acts on the current selection to turn it into such a selection, or use the Shift or Alternate key to signal this behavior on a mouse click.

If your application needs to know the type of mouse (or other pointing device, such as a tablet) that's attached to the computer, the **NXEventSystemInfo()** function can be used. This function describes the computer's input devices, including the type of mouse (see **/NextLibrary/Documentation/NextDev/ReleaseNotes/EventStatusDriver.rtf** and the header file **bsd/dev/ev_types.h** for more information). Unfortunately, NEXTSTEP doesn't provide any functions or methods through which you can specifically query for the number of mouse buttons.

The Keyboard

The keyboard, like the mouse, can't be queried for its attributes. However, you generally don't need to know how a keyboard is laid outÐfor example, whether it has a number pad as well as number keys. What you do need to know is what character was generated when the user pressed a key.

The NXEventData structure, defined in **dpsclient/event.h**, describes a keyboard event in its **key** substructure. The description is twofold:

- The *key code* describes the key that was pressed.
- The *character code* describes the character that was generated.

These two attributes, which sound similar, aren't necessarily the same. For example, the ª1º key that's typically found in the top row of the keyboard generates the same character code as the ª1º in the number pad, but they have two different key codes, since they are, physically, two different keys.

A keyboard event's key code is described in a single field, **key.keyCode**. The event's character code is a combination of two fields in the **key** substructure: **key.charSet**, which identifies a set of characters (such as ASCII or Symbol), and **key.charCode**, which indicates the character in the set. For portability, you should never use the **keyCode** field since, by its nature, it's keyboard-dependent. For example, the key code for the letter ªaº on one keyboard might be different from that on another. However, when the user presses ªaº, the same character code (in other words, the same **charSet** and **charCode** combination) will be generated regardless of the type of keyboard.

The set of character codes doesn't necessarily distinguish all key codes; as demonstrated in the number pad example above, there may be two key codes that are represented by the same character code. Thus, by using only character codes you may lose some keyboard-specific precision, but you gain portability. There are keyboard-independent ways to get certain information, though. For example, you can check for a key on the numeric keypad by masking the event record's **flags** field with the NX_NUMERICPADMASK mask.

As with the mouse, the **NXEventSystemInfo()** function can be used to determine what type of keyboard is attached to the computer (see the online release note **/NextLibrary/Documentation/NextDev/ReleaseNotes/EventStatusDriver.rtf** and the header file **bsd/dev/ev_types.h** for more information). If your application requires keyboard-specific information (as some terminal emulators do, for example), contact NeXT Developer Support.

Sound

The sound capabilities of different computers vary considerably. You can't assume that a particular computer will be able to play a sound created on another computer. If NEXTSTEP's sound software can't play a sound, the function or method will simply return an error code; the inability to play a sound should never cause your application to crash.

For information about determining the sound capabilities of a host machine at run time, see the online release notes.

Data Representation Considerations

Beyond concern with a computer's configuration, you must consider platform-specific differences when recompiling your application for a new architecture. One of the most fundamental differences between computer architectures is how data is represented. These differences fall into four arenas: datum size, byte alignment, byte order, and argument passing. The following paragraphs describe these properties and suggest ways to avoid the simple problems that arise from their differences. More complicated situations are examined in ªExternal Dataº and ªInternal Data,º below.

Certain NEXTSTEP kits require special consideration; the Indexing Kit, for example, maps file-based data directly into memory, which causes problems when the in-memory representations of data vary among computer architectures. If a kit has its own idiomatic portability issues, there will be a notice in the introduction to that kit's reference material, and the specific methods and functions

requiring special care will have notes about how to use them in a portable manner.

Datum Size

Datum sizes, or the amounts of memory that are devoted to single items of the various data types, aren't the same for all computers. Although almost every computer represents (as examples) a **char** in one byte, a **short** in two, and **int**s and **float**s in four, these sizes aren't mandated. Thus, you should never assume how much memory is needed to store the data that you allocate. In other words, you should never use hard-wired values in a call to a memory allocation function (such as **malloc()**); instead, use the **sizeof** operator to programmatically discover the size of a datum. This also applies to structures and unions, since different alignment restrictions (see below) can force them to be different sizes on different machines.

Byte Alignment

Some computers demand that the starting address (the first byte) of a value fall on a particular boundary. For example, a computer that uses *natural boundaries* expects the address of a value be divisible by the number of bytes that it takes to represent the value: the address of a two-byte value must be divisible by two, the address of a four-byte value must be divisible by four, and so on. In general, this isn't a concern to the programmer because the compiler and C allocation routines guarantee that all memory allocations, whether static or dynamic, will be on appropriate boundaries. However, there are two situations in which the compiler and C functions can't help you:

- By casting the data type of a pointer, you can write data into an illegal location. This is explained further in the section ªInternal Data,º below; briefly, you can avoid this error by never casting a data pointer to which you're writing data.
- If you redefine the memory allocation functions, you're on your own. Most visible of theseÐand most typically reimplementedÐis **malloc()** (and **realloc()**, and so on), but also included are the zone-allocation functions that make up the NXZone facility.

Byte Order

A datum of a type that can't be represented in a single byte is given as a series of consecutive bytes. If the most significant byte is given first, then the computer is said to be ªbig-endianº; if the least significant byte is first, then it's a ªlittle-endianº machine. Byte order is only a concern when you're reading or writing ªexternalº data. The data that your application creates and uses whiles it's running will by nature be ordered correctly.

If you use the data-reading and -writing mechanisms described in the section ªExternal Data,º below, your application may never need to know whether its running on a big- or little-endian machine. However, there are some situations in which this determination is essential. For this, the C preprocessor macros _ _BIG_ENDIAN_ _ and _ _LITTLE_ENDIAN_ _ can be examined:

```
#ifdef __BIG_ENDIAN__
/* do something for big-endian data */
#else
/* do something for little-endian data */
#endif
```

Datum Format

Like alignment restrictions, the form that a data type is given on one architecture may vary from that given on another. The general rule with regard to the internal format of any datum is: Never rely on it. *Always* use field names for structures and unions, and don't assume that you can pick apart a **float**'s mantissa and exponent directly (there are library functions to do this).

The format of structure bitfields is particularly variable from architecture to architecture. You should use bitfields only for data items that will remain entirely internal. If a structure is going to be written to or read from a file, you should avoid using bitfields unless absolutely necessary.

Argument Passing

The data that you pass as arguments to a function must be put somewhere so the function can retrieve it. Some computers place argument data (contiguously) on the stack, while others put arguments in CPU registers, for which there is no notion of contiguity. As long as a function always refers to its arguments symbolically, the difference between the stack and register approaches is inconsequential. However, a function that steps through its arguments by incrementing (or decrementing) a data pointerÐthus assuming that the arguments are being passed on the stackÐwon't be portable.

Functions that take a determinate number of arguments should never need to use the data-pointer approach. But if you're designing a function that takes a variable number of argumentsÐa function in the style of **printf()**, for exampleÐyou may be tempted to read the arguments by setting and moving a data pointer. The correct approach to reading an indeterminate number of arguments is to use the **stdarg** macros provided by the standard C library. These macros are included in your program by importing the file **stdarg.h**; they're described under **varargs** in section 3 of the UNIX manual.

External Data

The problems of external dataÐdata that's read from an external sourceÐarise primarily from differences in byte-order: Data written on a little-endian machine will be swapped when it's read on a big-endian machine (and vice versa). All NEXTSTEP data-communication mechanisms (such as those provided by the Application Kit's pasteboard and data link objects, and the distributed objects paradigm) automatically transform data that's transmitted between applications to the correct byte-order; thus, inter-application communication is taken care of.

What you need to be concerned with is data that your application reads and writes directly. The rule is simple: If you're reading and writing multi-byte data, you should *always* use typed streams. The typed stream functions recognize the byte-order differences between machines and can identify the sort of machine on which a particular stream of data was written. Typed streams are comprehensively described in Chapter 3, ªCommon Classes and Functions,º of the *NEXTSTEP General Reference* (**/NextLibrary/Documentation/NextDev/GeneralRef/03_Common**); the typed stream functions are declared in the header file **objc/typedstream.h**.

Note: Previous versions of NEXTSTEP documentation for the **NXRead()** and **NXWrite()** functions give an example which ignores this portability issue, and uses these functions to store a multi-byte structure. That example should not be considered a proper use for **NXRead()** and **NXWrite()**.

Given typed streams as the rule, then, you should be aware that there are a few exceptions:

- One-byte data (such as ASCII strings) can be written and read through any of the usual C functions, such as **read()** and **write()**. Writing ASCII through a typed stream isn't wrong, but it is somewhat inefficient. Of course, this slackening of the typed stream rule depends on the immutability of one-byte data on different machines (in other words, you must be sure that one-byte data on one machine isn't going to be two-byte data on another).

- Bitfields in a structure, even when written through a typed stream, will be improperly represented. The use of bitfields in a file format is strongly discouraged.

- File formats that already exist and that weren't written through typed streams can't be read

through typed streams.

The following sections describe solutions to the bitfield and existing file format problems.

Reading and Writing Structure Bitfields

Structure bitfields can help you conserve memory when your application is running. However, they're a poor choice with regard to storing data in a file, since neither the compiler nor typed streams resolve the order of contiguous bitfields to match the endian-ness of the computer. If you don't do anything to correct this situation, a series of bitfields that are written on a big-endian machine will be ordered differently when read on a little-endian machine, and vice-versa. The best solution to this problem is to avoid it altogether. If, for efficiency or compatibility reasons, your application must be able to read and write structure bitfields, you have two general options: modify the routines that read and write the structure, or redefine the structure itself.

Approach 1: Modify the Read and Write Routines

By modifying the routines that read and write particular structures, you can change the way the structure is represented externally. Thus, you can make sure that the external representation is portable. For example, consider the following class declaration:

```
@interface Dog : Mammal
{
    char *name;
    short age;
    struct _dogFlags {
        unsigned int canWalk:1;
        unsigned int canTalk:1;
        unsigned int whiskerCount:10;
        unsigned int PAD:20;
    } dogFlags;
}

/* ... */

@end
```

The **name** and **age** instance variables will be written and read correctly by the typed stream functions, but the bitfields won't be. To write the object and ensure its portability, you can define the **write:** method to ªexpandº the bitfields into full-byte values and then write the expanded data:

```
- write:(NXTypedStream *stream)
{
    [super write:stream];

    /* Create variables for the "expanded" bitfield data. */
    unsigned char canWalkHolder = dogFlags.canWalk;
    unsigned char canTalkHolder = dogFlags.canTalk;
    unsigned int whiskerCountHolder = dogFlags.whiskerCount;

    /* Write the data. */
    NXWriteTypes(stream, "*sCCI", name, &age, &canWalkHolder,
                 &canTalkHolder, &whiskerCountHolder);
    return self;
}
```

You would, of course, have to create an analogous **read:** method. Note that the data type of the variables that hold the expanded bitfield data must be big enough to represent the values that the bitfields contain.

By writing expanded bitfield data, the external representation of an object or structure may waste some space (compared to the internal representation), but unless you're writing thousands of items the waste is insignificant. Also, this is the only approach that's guaranteed to be portable with

regard to bitfield layout, whatever machine the application is run on.

Approach 2: Redefine the Structure

A quicker but less elegant (and discouraged) solution is to redefine the structure to accommodate the endian-ness of the machine. This means predicating the order of the bitfields according to the machine's byte order:

```
@interface Dog : Mammal
{
    char *name;
    short age;
    struct _dogFlags {
#ifdef __BIG_ENDIAN__
        unsigned int canWalk:1;
        unsigned int canTalk:1;
        unsigned int whiskerCount:10;
        unsigned int PAD:20;
#else
        unsigned int PAD:20;
        unsigned int whiskerCount:10;
        unsigned int canTalk:1;
        unsigned int canWalk:1;
#endif
    } dogFlags;
}

/* ... */

@end
```

This approach works transparently when used in the following manner:

- The bitfield structure is always read or written with the typed streams mechanism.
- The bitfield structure is designed so that its total size in bits is equal to one of the standard unsigned integer types: 8 (**unsigned char**), 16 (**unsigned short int**), 32 (**unsigned int** and **unsigned long int**). This involves using pad fields to fill out the structure, and never using zero-width fields to force alignment.
- A multi-byte bitfield structure is treated as its size-equivalent integral type when reading or writing it, instead of as an array of **char**s. This allows the typed streams mechanism to perform byte-swapping if needed.

For example, the **dogFlags** bitfield structure above is 32 bits, and would be written with this function call in Dog's **write:** method:

```
/*
 * The proper way to write a bitfield structure. Since dogFlags
 * is written as an unsigned int, NXWriteTypes() swaps it
 * automatically if needed.
 */
NXWriteTypes(stream, "*sI", &name, &age, &dogFlags);
```

If you don't use typed streams, you need to use the byte-swapping functions described later in this paper to swap the structure before writing and after reading it. In either case, you should always make sure that your bitfields are of a total size in bits equal to one of the standard unsigned integer types.

Although reversing the bitfield declaration is a quick way to solve the bitfield problem, you should be warned that it may not be a permanent solution: some future architecture may define a new way of representing bitfields, and you may have to revisit your code to add another branch to the endian predicate. In general, you should use the first approach (modifying the archival routines) rather than reversing bitfields.

Reading Existing Files

If your application defines its own (non-ASCII) file format, but doesn't use typed streams to read and write these files, you may have to rewrite the file-reading and -writing routines to accommodate the endian-ness of the machine that your application is running on. NEXTSTEP provides a suite of byte-swapping functions that convert individual data items. These functions come in two varietiesÐthose that always swap, and those that only swap if neededÐfor integer and floating-point data types. These functions may be used by including **architecture/byte_order.h**.

An ªalways-swapº function for an integer value takes an integer, swaps the order of the bytes it comprises, and returns the swapped value. There are four such functions, one for each integer type:

short **NXSwapShort(**short *x***)**
int **NXSwapInt(**int *x***)**
long int **NXSwapLong(**long int *x***)**
long long int **NXSwapLongLong(**long long int *x***)**

These functions each take a single argument and return a single value. The value that you pass as the argument can be used to store the value that's returned, as shown in the following example:

```
/* Swap the order of the bytes in a given int. */
givenInt = NXSwapInt(givenInt);
```

The ªalways-swapº functions for floating-point values are slightly more complex. Some processors modify the value of a floating-point number if that value is invalid, so the swap functions for floating-point values can't simply return **float** or **double**. To get around this problem, the types **NXSwappedFloat** and **NXSwappedDouble** are defined and used by the floating-point swap functions. The following functions are used to convert floating-point values to and from the corresponding swapped types:

NXSwappedFloat **NXConvertHostFloatToSwapped(**float *f***)**
NXSwappedDouble **NXConvertHostDoubleToSwapped(**double *d***)**
float **NXConvertSwappedFloatToHost(**NXSwappedFloat *sf***)**
double **NXConvertSwappedDoubleToHost(**NXSwappedDouble *sd***)**

The floating-point swap functions, then, take a single argument of type **NXSwappedFloat** or **NXSwappedDouble**, and return a value of the same type:

NXSwappedFloat **NXSwapFloat(**NXSwappedFloat *sf***)**
NXSwappedDouble **NXSwapDouble(**NXSwappedDouble *sd***)**

A second set of functions (called *predicated* functions) also take single data items of a particular type, but they're defined to swap the byte order only if the endian-ness indicated in the function name doesn't match the endian-ness of the machine the code is being compiled for. As explained in a later section, determining the endian-ness of data is up to you. There are four groups of these functions:

NXSwapBig*Type***ToHost()**
NXSwapLittle*Type***ToHost()**
NXSwapHost*Type***ToBig()**
NXSwapHost*Type***ToLittle()**

where *Type* is one of the six multi-byte data types (there are 24 of these functions in all). **NXSwapBigIntToHost()**, for example, would swap on a little-endian (i386 family) machine, but would do nothing on a big-endian (MC68000 family) machine. For floating-point types, the functions in the first pair each take an argument of type **NXSwappedFloat** or **NXSwappedDouble** and return a **float** or **double**, respectively. The second pair reverses this, each taking an argument of type **float** or **double** and returning an **NXSwappedFloat** or **NXSwappedDouble**.

How to Use the Byte-Swapping Functions

Regardless of which set of byte-swapping functions you use, you must determine the endian-ness of the data that you want to convert. To use the always-swap functions, you must also know the endian-ness of the host computer; you would use these functions only if the format of the given data and that of the host aren't the same. The predicated functions determine the host format for you and swap if the format indicated by the function's name doesn't match that of the host.

In the following example of the always-swap functions, it's been determined (by one of the methods given in the next section) that the data being read is in big-endian format. The bytes are swapped if the host is little-endian:

```
#define COUNT 1024
int buf[COUNT];
int byteCount, itemCount;

/* aStream is open to a file that contains big-endian integer data. */

byteCount = NXRead(aStream, (void *)buf, sizeof(buf)))
itemCount = byteCount / sizeof(int);

#ifdef __LITTLE_ENDIAN__

/* Swap if this is a little-endian machine. */
while (itemCount--) {
    buf[itemCount] = NXSwapInt(buf[itemCount]);
}

#endif
```

How to Determine Endian-ness of External Data

As mentioned above, you need to know the endian-ness of a datum whether you're using the always-swap functions or the predicated functions. You can't simply ask a datum for its byte-order, so how do you determine which format it's in?

One approach is to assume that files created by a certain application are of one endian-ness. For example, if you're reading existing data that was written by a NEXTSTEP application prior to the release of NEXTSTEP 3.1, you can be sure that it's in big-endian format. This is because NEXTSTEP, until now, only ran on NeXT Computers, and all existing NeXT Computers are big-endian. However, if you accept the guarantee that a file format is always of one endian-ness and not the other, then you must stick with that endian-ness when you write the data back to a file (so it can be read again). Thus, for example, you would use the **NXSwapBig***Type***ToHost()** functions to swap data that you've just read, and convert it back through the **NXSwapHost***Type***ToBig()** functions just before you write it. For applications running on a host with an endian-ness opposite that of the file format being used, this results in a performance penalty for both reading and writing that file format.

You don't have to adhere to the assumed-big-endian rule if your application inserts a ªmagic numberº in the files that it writes. Magic numbers are used to confirm the identity, format, or version of a file, and can also be used to determine whether the file as it lies on disk is in the same or the opposite endian-ness as the host machine. For example:

```
/* MY_MAGIC is the first long int in the file. If the value read
 * from the file doesn't match, swap it and try again. The magic
 * number shouldn't be byte-symmetric; for example, it shouldn't be
 * 0x50404050, as swapping results in the same number.
 */
#define MY_MAGIC 0x50ab40cd
#define COUNT 1024

BOOL fileNeedsSwapping;
long int magicNumber;
```

```
int buf[COUNT];
int byteCount, itemCount;

/* Assuming the file is opened onto aStream */

byteCount = NXRead(aStream, &magicNumber, sizeof(magicNumber));
if (sizeof(magicNumber) != byteCount)
    /* error */

if (MY_MAGIC == magicNumber) fileNeedSwapping = NO;
else {
    magicNumber = NXSwapLong(magicNumber);
    if (MY_MAGIC == magicNumber) fileNeedsSwapping = YES;
    else /* bad file? */
}

/* Now read the rest of the data. */
byteCount = NXRead(aStream, (void *)buf, sizeof(buf));
itemCount = byteCount / sizeof(int);

if (fileNeedsSwapping) {
    while (itemCount--)
        buf[itemCount] = NXSwapInt(buf[itemCount]);
}
```

Checking for endian-ness mismatch allows the routine to work for hosts of either endian-ness reading the file. This approach permits maximum performance in all possible cases: writing an entire file is always at normal speed regardless of host endian-ness, but reading is only slower when there's an endian-ness mismatch between the file format and the host. The only complication is that if an application writes into an existing file, it must remember the original endian-ness of that file and alter its output accordingly.

Since magic numbers are used to store several kinds of information, care should be taken in choosing a number for a particular version of a file format. Magic numbers should also never be byte-symmetric or mirror images of other magic numbers. If your application will be reading files created by other applications, you'll need to check what magic numbers they use. You should choose a magic number far from the range for file formats you intend to support. When file formats are revised, their magic numbers are often merely incremented instead of re-assigned; if you've chosen a magic number 1 greater than the file format's, you could come into conflict with that format's magic number when it's updated.

Internal Data

Internal dataÐdata that your application creates and uses while it's runningÐshouldn't be a problem as long as you adhere to a few principles:

- Always refer to the elements in a data structure by name. Because of possible byte-alignment padding, the distance between contiguous elements in a data structure (in other words, the elements in a **struct**, or the instance variables in an object) may be different on different computers. You should never try to access these elements by moving a pointer inside the structure.

- Be scrupulous about pointer types: use character pointers to point to characters, integer pointers to point to integers, and so on. For example, if you've allocated an integer array and then read the elements of the array through a character pointer, the data that you read may differ as the computer is big-endian or little-endian. If you must manipulate data of an unknown type through a pointer, use a pointer to **void** instead of a pointer to **char**.

- Never write the ªwrongº type of data by recasting a pointer. As a demonstration, the following code will break an application running on a computer that expects data on natural boundaries:

```
/* Create a character array and a pointer to the array. */
char buffer[6];
char *bufptr = buffer;

/* Write a character into the array and increment the pointer. */
*bufptr++ = 'd';

/* Write an integer into the array; THE PROGRAM WILL CRASH. */
*((int *)bufptr) = 10;
```

The example is trying to use the character array as a data structure. A better approach is to create a **struct** to store the data:

```
/* Create a struct that contains a character and an integer. */
struct shoeSize {
    char width;
    int length;
} aShoe;

aShoe.width = 'd';
aShoe.length = 10;
```

Memory-mapped Data

NEXTSTEP's Mach operating system allows files to be mapped directly into the address space of a process, turning external data directly into internal data. For performance reasons, you may want or need your application to access file-based data by mapping the file. If you do this, there are two things you should do to make your file format portable:

- Always use a magic number to record the endian-ness of the file's data.
- To skirt your way around alignment restrictions, always pad data elements so they lie on natural alignment boundaries.

The first point has been well covered in the previous section. The second, however, deserves some explanation. As an example, let's assume you have the following structure declaration for use with mapped files:

```
typedef struct _mappedFile {
    unsigned long int magicNumber;
    unsigned long int numRecords;
    addressRecord addresses[0];
} mappedFile;

mappedFile *myFile;  /* a pointer should align on a 32-bit boundary */
```

The idea is that the application will map a file into **myFile**, directly accessing the file's data in memory (swapping each datum upon access if needed). In order to avoid any alignment restriction problems, the as-yet-undefined **addressRecord** type should declare all of its fields on the most natural alignment boundaries. For example, 32-bit **int**s should lie on 4-byte boundaries, 64-bit **double**s should be on 8-byte boundaries, and so on. When using character arrays, it's best to declare them in multiples of 4 or 8 bytes, to avoid having to keep track of running offsets. Bitfields should be avoided altogether if possible, as using them requires detailed knowledge of how the compiler lays them outÐwhich may differ between processor architectures.

Here, then, is the **addressRecord** type:

```
typedef _addressRecord {
    char lastName[32];    /* multiple of 4 chars */
    char firstName[32];
    char street[32];
    char city[32];
```

```
        char state[2];
        char PAD[2];          /* forces alignment to unsigned long int */

        struct _phone {
            unsigned long int area;   /* kept apart to allow convenient */
            unsigned long int prefix; /* access to each part */
            unsigned long int phone;
        } phone;
    } addressRecord;
```

Note the use of the **PAD** field after **state**, which forces the next structure field to be aligned on a natural **unsigned long int** (32-bit) boundary. The phone number is stored in separate **long int**s, even though each could fit into a **short int**. Although groups of two **short int**s would each make 32 bits, keeping all fields (and the entire structure) on 32-bit boundaries guarantees that there will be no alignment restriction problems when this file is memory-mapped on some new architecture. The phone number could also be stored as an array of characters, or as a single **unsigned long int**; the form chosen depends on space considerations and on how the data will be used.

Note also that the entire structure fits into a multiple of 4 bytes, so that the following structure will begin on a natural boundary for most basic datum sizes. If this structure contained any **double**s, it would be better declared as fitting into a multiple of 8 bytes. Keeping alignment at its most general at every level of declaration within a mapped file guarantees that the file format will be maximally portable.