Exception Handling

NeXT Technical Support Notes

Written by Matt Morse

Exception Handling

Exception Handling

An exceptional condition is one that interrupts the normal flow of program execution. Each application can interpret different types of conditions as exceptional. For example, one application might view as exceptional the attempt to save a file in a directory that's write-protected. In this sense, an exceptional condition can be equivalent to an error. Another application might interpret the user's keypress as an exceptional condition: an indication that a long-running process should be aborted.

A robust application must be able to respond appropriately to exceptional conditions. Depending on the context, this might mean:

- Terminating normal processing
- Freeing dynamically allocated memory
- Closing files
- Restarting execution at some other point in the program

Exceptional conditions can occur deep within a calling sequence—within a function that's called by another function that's called by yet another function, and so on. Responding to the condition might entail backing out of each of these functions in the reverse order that they were called, cleaning up as necessary at each level along the way. This process is known as *unwinding the call stack.*

Traditionally, exceptional conditions are announced through a function's return value. This system has several disadvantages. Only limited information about the condition can be coded in a single return value. For example, the UNIX® manual page for **fopen()** states that the function returns NULL if it's unable to open a file, if too many files are already open, or if other needed resources can't be allocated. In addition, for the notification of the exceptional condition to propagate up the call stack, each function along the way must check return values and respond appropriately. Failing this, information about the exceptional condition can be lost. Finally, providing for exceptional cases can obscure the normal pathways through your code:

```
if ((returnValue1 = function1()) == NULL) goto onError;
if ((returnValue2 = function2()) == NULL) goto onError;
if ((returnValue3 = function3()) == NULL) goto onError;

onError:
    /* Check where the error occurred and
       take remedial action */
```


A good exception-handling system must provide the programmer with a unified and organized approach for responding to exceptional conditions once they've been identified at any level within an application. The following sections describe the system used in NextStep® and available for use in applications built with NextStep.

Note: The exception-handling system described here is distinct from the one used within the Mach operating system. See "Mach Exception Handling" in Chapter 16 of the preliminary 1.0 *NeXT™ System Reference Manual* for information on that system.

Detecting Exceptional Conditions

Before an exceptional condition can be responded to, it must be detected. Typically, an exceptional condition is discovered through the return value of a function or method, especially those that access data from the file system. An example is reading a file into memory, as this program excerpt illustrates:

```
NXStream *stream;
char *theFile = "/me/filename", *buffer = NULL;
int length, maxlength;

if ((stream = NXMapFile(theFile, NX_READONLY)) != NULL) {
    NXGetMemoryBuffer(stream, &buffer, &length, &maxlength);
    NXClose(stream);
} else {
    /* exceptional condition has been detected */
}
```

If **NXMapFile()** is unable to map the file into memory, it returns NULL, indicating an exceptional condition. Routines that write data generally have a similar system of reporting such conditions.

Even if the data can be read or written, it may not be usable to the program. Applications that run consistency checks on data may also use the exception-handling system in case of data inconsistency.

Another situation where the exception-handling system can be used is when the user wants to interrupt a long-running operation. In the following example, the application displays an attention panel to alert the user of its current state and to give the user the choice of aborting the operation:

```
id alert;
NXModalSession session;
int runState;
BOOL done;

alert = NXGetAlertPanel(NULL, "Doing something time-consuming.
```

```
        Please wait . . . ", "Stop", NULL, NULL);
    [NXApp beginModalSession: &session for:alert];
    runState = NX_RUNCONTINUES;
    done = NO;
    while (!done && runState == NX_RUNCONTINUES) {
        runState = [NXApp runModalSession: &session];
        /* Do small portion of lengthy process. */
        /* Set done == YES when the process is finished. */
    }
    [NXApp endModalSession: &session];
    [alert orderOut:self];
    NXFreeAlertPanel(alert);
    if (runState == NX_ALERTDEFAULT) {
        /* exceptional condition has been detected */
    }
```

Program execution continues in the **while** loop until either the incremental processing finishes

```
done == YES
```

or until the user clicks the panel's Stop button

```
runState != NX_RUNCONTINUES
```

If the user has interrupted processing, the statements in the body of the **if** construction are executed and the exception is detected. (See "Modal Sessions" in Chapter 7 of the preliminary 1.0 *System Reference Manual* for more information.)

Raising an Exception

Once an exceptional condition is detected, it must be propagated to the routine or routines that will handle it, a process referred to as *raising an exception*. In the NeXT exception-handling system, exceptions are raised by calling the macro **NX_RAISE**(), as defined in the header file **streams/error.h**. This routine takes three arguments:

```
void NX_RAISE(int code, const void *data1, const void *data2)
```

The first, *code*, is an integer that identifies the exception. As described in the section "Exception Codes" below, some code ranges are reserved for the Application Kit™, the Display PostScript® client library, and other software modules; you can define codes for exceptional conditions that might occur in your application.

The second two arguments are pointers to arbitrary data about the exception. For example, if a function's return value initiated the call to **NX_RAISE()**, you could use **data1** to pass the return value to the exception handler. Or, if the exception handler displays a panel in response to the exception, you could use **data2** to pass the text string to be displayed in the panel.

NX_RAISE() works by calling a function that's registered as the exception raiser; for applications using the Application Kit, this function is **NXDefaultExceptionRaiser()**. **NXSetExceptionRaiser()** and **NXGetExceptionRaiser()** give you access to the exception raiser, although it's unlikely that you'll find a need to alter it.

Handling an Exception

Calling **NX_RAISE()** initiates the propagation of the exception and passes data about it. Where and how the exception is handled depends on where you make the call to **NX_RAISE()**. Let's first look at a simple case.

In general, **NX_RAISE()** is called within the domain of an *exception handler*. An exception handler is a control structure created by the macros NX_DURING, NX_HANDLER, and NX_ENDHANDLER, as shown in Figure 1.

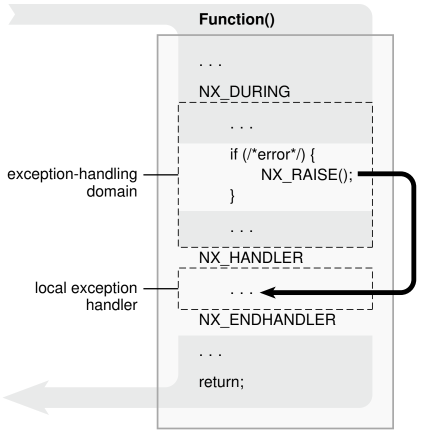

Figure 1. Flow of Control in an Exception Handler

The section of code between NX_DURING and NX_HANDLER is the *exception-handling domain*; the section between NX_HANDLER and NX_ENDHANDLER is the *local exception handler*. The normal flow of program execution is marked by the gray arrow; the code within the local exception handler is executed only if **NX_RAISE()** is called. A call to **NX_RAISE()** causes program control to jump to the first executable line following NX_HANDLER, as indicated by the black arrow.

Although you can call **NX_RAISE()** directly within the exception-handling domain, it's more often called indirectly within one of the procedures called from the domain. No matter how deeply in a call sequence the call to **NX_RAISE()** is made, execution jumps to the local exception handler (assuming there are no intervening exception handlers, as discussed in the next section). In this way, exceptions raised at a low level can be caught at a high level.

Besides transferring execution to the local exception handler, a call to **NX_RAISE()** initializes the variable **NXLocalHandler** of type NXHandler, as defined in **streams/errors.h**. This variable is defined only within the local exception handler and contains those structure members that correspond to the arguments passed to **NX_RAISE()**: **NXLocalHandler.code**, **NXLocalHandler.data1**, and **NXLocalHandler.data2**. These members transfer information about the exception to the code within the local exception handler.

For example, in the following program excerpt, the local exception handler displays an attention panel after detecting an exception having the code **error_one** (see "Exception Codes" below for information on defining exception codes):

```
. . .
NX_HANDLER
    switch(NXLocalHandler.code) {
        case error_one:
            NXRunAlertPanel ("Error Panel",
                NXLocalHandler.data1, "OK", NULL, NULL);
                break;
        case error_two:
            . . .
    }
NX_ENDHANDLER
```

If an exception of type **error_one** is raised, an attention panel appears and displays the text referred to by **NXLocalHandler.data1**.

Calling **NX_RAISE()** is one way for program execution to leave the exception-handling domain; three other ways are permitted:

- "Falling off the end"
- Calling **NX_VALRETURN()**
- Calling **NX_VOIDRETURN**

"Falling off the end" is simply the normal execution pathway introduced above. After all appropriate statements within the domain are executed (and no exception is raised), execution continues on the line following NX_ENDHANDLER. Alternatively, you can return control to the caller from within the domain by calling **NX_VALRETURN()** or **NX_VOIDRETURN**, depending on whether you need to return a value.

You can't use **goto** or **return()** to exit an exception-handling domain—errors will result. Nor can you use **setjmp()** and **longjmp()** if the jump entails crossing a NX_DURING statement. Since in many cases you won't know if the NextStep code that your program calls has exception-handling domains within it, it's generally not recommended that you use **setjmp()** and **longjmp()** in your application.

If an exception is raised and execution begins within the local exception handler, it either continues until all appropriate statements are executed (falling off the end of the local exception handler), or the exception is raised again to invoke the services of an encompassing exception handler, as described in the next section.

Nested Exception Handlers

Exception handlers can be nested so that an exception raised in an inner domain can be treated by the local exception handler and any number of encompassing exception handlers. This hierarchy of exception handlers is accessed with the macro NX_RERAISE, as illustrated in Figure 2.

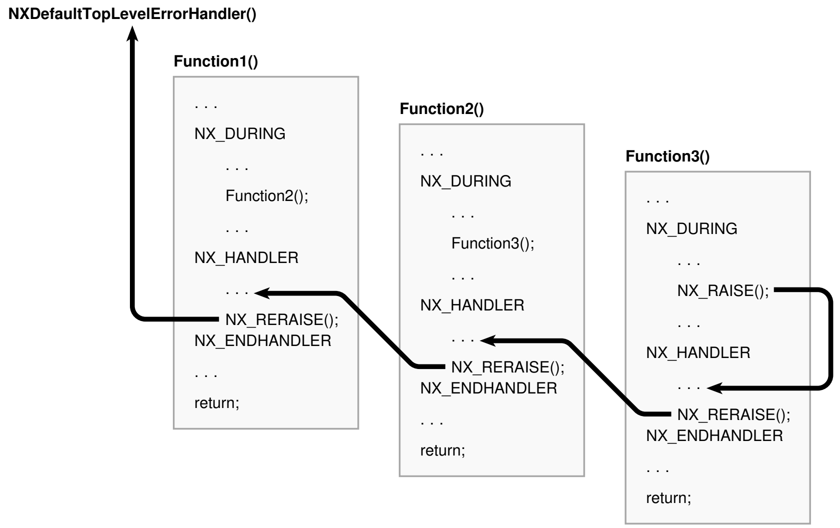

Figure 2. Nested Exception Handlers

An exception raised within **Function3()**'s domain causes execution to jump to its local exception handler. In a typical application, this exception handler checks the values contained in **NXLocalHandler** to determine the nature of the exception. For exception types that it recognizes, the local handler responds and then calls **NX_RERAISE()** to pass notification of the exception to the handler above it, in this case, the handler in **Function2()**. **Function2()**'s exception handler does the same and then reraises the exception to **Function1()**'s handler. Finally, **Function1()**'s handler reraises the exception. Since there's no exception-handling domain above **Function1()**, the exception is transferred to the default top-level error handler, as discussed below.

An exception that's reraised appears to the next higher handler just as if **NX_RAISE()** had been called within its own exception-handling domain. (**NX_RERAISE()** is in effect a cover for **NX_RAISE()**, transferring the values of **NXLocalHandler**'s **code**, **data1**, and **data2** members to the next higher exception handler.)

For applications based on the Application Kit, exceptions that are reraised within the highest-level local exception handler are sent to **NXDefaultTopLevelErrorHandler()**. Through a call to **NXReportError()**, this function prints a message about the exception. (See "Reporting Errors" below for more information.) If an application's connection to the Window Server becomes corrupt or dies, or if the application is unable to form a connection to the Server, **NXDefaultTopLevelErrorHandler()** terminates the application by calling **exit()** with a status code of −1.

NXSetTopLevelErrorHandler() lets you change the function used as the top-level handler; **NXTopLevelErrorHandler()** returns a pointer to the current top-level handler. If you substitute your own function for **NXDefaultTopLevelErrorHandler()**, you should probably call **NXDefaultTopLevelErrorHandler()** as part of its implementation. In this way, your function can give special handling to certain exceptions, passing all others to **NXDefaultTopLevelErrorHandler()**.

Raising an Exception Outside of an Exception Handler

If an exception is raised outside of any exception handler, it's intercepted by the *uncaught exception handler*, a function set by **NXSetUncaughtExceptionHandler()** and returned by **NXGetUncaughtExceptionHandler()**. The default uncaught exception handler for Application Kit programs writes the message "An uncaught exception was raised." to the Workspace Manager's console window (if the application was launched by the Workspace Manager™) or to a Shell or Terminal window (if the application was launched from either of those applications). It then calls the top-level exception handler, passing it the information contained in the arguments to the **NX_RAISE()** call that originally raised the exception.

You can change the way uncaught exceptions are handled by using **NXSetUncaughtExceptionHandler()** to establish a different procedure as the handler. However, because of the design of the Application Kit, it's rare for an exception to be raised outside of an exception-handling domain. The Application object's event loop itself is within an exception-handling domain. On each cycle of the loop, the Application object retrieves an event and sends an event message to the appropriate object in the application. Thus, the code you write for custom objects (as well as the code for Application Kit objects) is executed within the context of the event loop's exception handler. To customize the Application Kit's highest-level response to exceptions, modify the top-level exception handler.

Exception Codes

Each of the software modules provided by NeXT is assigned a range of exception-code values. The lowest value within the range is represented by a constant, as listed in the following table.

Exception Category	Base Constant
Client Library, Adobe	DPS_ERRORBASE
Client Library, NeXT Extensions	DPS_NEXTERRORBASE
Application Kit	NX_APPKITERRBASE
Streams	NX_STREAMERRBASE
Typed Streams	*<<NX_TYPEDSTREAMERRBASE>>*
Music Kit	MK_ERRORBASE
DSP C Library	DSP_ERRORS

<<The base constant for typed streams is currently TYPEDSTREAM_ERROR_RBASE. In the next release, it will be renamed NX_TYPEDSTREAMERRBASE.>>

Except for the first two categories, each range spans 1000 exception codes. The first two categories share a range of 1000: The first 100 codes are reserved for exceptions generated by Adobe's client library routines; the remaining 900 codes are reserved for the NeXT client library.

If, within an exception handler, you want to catch exceptions from one of these modules, you could use code such as this:

```
NX_HANDLER
    if (NXLocalHandler.code > = DSP_ERRORS &&
        NXLocalHandler.code < DSP_ERRORS +1000) {
        /* code for custom handling of DSP exceptions */
    } else
    NX_RERAISE();
NX_ENDHANDLER
```

Defining Codes for Your Application

The Application Kit defines one additional range of exception codes: the range for applications built upon the Application Kit. This range extends upward from the base constant NX_APPBASE, as defined in the header file **appkit/errors.h**. For example, you could could use this constant in an enumeration of exception types for a database application:

```
enum databaseExceptions {
    DB_invalidSearchKey = NX_APPBASE,
    DB_corruptRecord,
    DB_corruptIndex,
    DB_compactionError,
    DB_sortError
};
```

By initializing exception code in this way, you can be sure that they won't conflict with those assigned to other software modules.

Associating Messages with Codes

In general, you'll want to associate a message with each exception code you define for an application. One convenient way to centralize this list of messages is to declare an array of pointers to the messages that correspond to the application-specific exception codes:

```
char *dbErrorMessages[] = {
    /* DB_invalidSearchKey */   "Invalid search key",
    /* DB_corruptRecord */      "Error reading record",
    /* DB_corruptIndex */       "Error opening index",
    /* DB_compactionError */        "Error during compaction",
    /* DB_sortError */          "Error during sort operation"
};
```

Using this technique, a call to **NX_RAISE()** might look like this:

```
NX_RAISE(DB_corruptIndex,
    dbErrorMessages[DB_corruptIndex -NX_APPBASE], NULL);
```

The first argument is the exception code, and the second is a pointer to the string,“Error opening index”.

Besides centralizing your application's error messages, associating exception codes and messages in this way makes it easier for your application to work with the Application Kit's error reporter, as described in the next section.

Reporting Errors

The Application Kit lets you register a function as the error reporter for a range of exception codes. An error reporter typically logs information about the error by writing a message in the Workspace Manager's Console window. An application can have many error reporters, each responsible for a specific range of exception codes.

Once the reporters are registered (as described below), calling **NXReportError()** causes the Application Kit to search for the reporter responsible for the specific exception code. If it finds one, the reporter is called and passed data about the exception. If it can't find one, it logs the exception code and a message stating that an unknown exception code was reported.

An error reporter for the database application example introduced above might look like this:

```
void
DBErrorReporter(NXHandler *errorState)
{
    if (errorState->code == DB_invalidSearchKey)
        return;
    _NXLogError("DB error: %s\n",
        dbErrorMessages[errorState->code -NX_APPBASE]);
    return;
}
```

*<<Note: **_NXLogError()** was a private function in the 1.0 release. In future releases it will be made public, and the underscore will be removed.>>*

This reporter ignores exceptions of type **DB_invalidSearchKey** (presumably because they are recoverable errors) but logs messages concerning all other codes within its range.

The function **_NXLogError**() is much like **printf**(): It lets you write a formatted string to the Console, Shell, or Terminal window, depending on where the application was launched. **_NXLogError**(), however, calls **syslog()**, which marks the message with the time of occurrence and the application's process identification number. See the UNIX manual page for **syslog**() for more information.

If your application defines a range of exception codes as its own, it should also register an error reporter. This is because the default top-level exception handler calls **NXReportError()** for all exceptions raised to its level. If your application hasn't registered an error reporter for its range, then **NXReportError()** won't be able to print anything more informative than the exception code for the error.

Registering Error Reporters

You register an error reporter for a range of exception codes by calling **NXRegisterErrorReporter()**. You might, for example, place code such as the following in the **initialize** method of one of your application's custom classes:

```
+ initialize
{
        NXRegisterErrorReporter(NX_APPBASE, NX_APPBASE + 999,
                DBErrorReporter);
        return self;
}
```

The first two arguments to **NXRegisterErrorReporter()** represent the minimum and maximum values for exception codes sent to the reporter referred to by the third argument. The call above specifies the error reporter described in the previous section. Once an error reporter is registered for a specific range, no new reporter can be set for any part of that range until the existing reporter is removed.

You remove an error reporter by calling **NXRemoveErrorReporter()**. This function takes one argument, the minimum exception code value of the existing error reporter's range. For example, to remove the reporter registered in the preceding code excerpt, you'd make this call:

```
NXRemoveErrorReporter(NX_APPBASE);
```

Handling PostScript Errors

To draw on the screen, your application must send PostScript code to the Window Server, where it's executed by the PostScript interpreter. Most of the code that an application sends is generated by user-interface objects defined in the Application Kit. Other code is generated by routines you write for your application's custom classes. In some applications (for example, a PostScript language previewer such as Yap—see **/NextDeveloper/Examples/Yap**), the code is entered by the user or imported from a file. Careful debugging should ensure that PostScript code generated by Application Kit or custom objects won't generate exceptions at runtime. However, applications that import PostScript code must be prepared to handle exceptions raised by the PostScript interpreter at runtime.

In the following example, a PostScript program is executed from a file. Before reading the file, however, the application stores an image of its own PostScript virtual memory (VM) in the PostScript dictionary **userdict**. It then begins importing and executing the code. If the code is faulty, the Display PostScript client library raises an exception, transferring control to the local exception handler. After the handler has logged the exception, the contents of the application's PostScript VM are restored from **userdict**.

```
char *errorBuffer;
int streamLength, maxLength;
NXStream *errorStream;

/* save contents of PostScript VM */
PSuserdict();
DPSPrintf(DPSGetCurrentContext(), "/saveToken");
PSsave();
PSput();
```

```
NX_DURING
    PSrun(/* file path */);
    NXPing();   /* ensure all code is transmitted to Server */
NX_HANDLER
    switch (NXLocalHandler.code) {
        case dps_err_ps:
            errorStream = NXOpenMemory(NULL,0,NX_WRITEONLY);
            DPSPrintErrorToStream(errorStream,
                (DPSBinObjSeq)(NXLocalHandler.data2));
            NXFlush(errorStream);
            NXGetMemoryBuffer(errorStream, &errorBuffer,
                &streamLength, &maxLength);
            _NXLogError("%s\n",errorBuffer);
            NXCloseMemory (errorStream, NX_FREEBUFFER);
            /* restore contents of PostScript VM */
            break;
        . . .
        default:
            NX_RERAISE();
    }
NX_ENDHANDLER

/* restore contents of PostScript VM */
PSuserdict();
DPSPrintf(DPSGetCurrentContext(), "/saveToken");
PSget();
PSrestore();
```

*<<Note: **_NXLogError()** was a private function in the 1.0 release. In future releases it will be made public, and the underscore will be removed.>>*

The exception code **dps_err_ps** identifies errors reported by the PostScript language interpreter in the Window Server. (See **dpsclient/dpsclient.h** for a complete list of exception codes raised by the Display PostScript client library.) The local exception handler treats exceptions of this type by calling **DPSPrintErrorToStream()** to extract the error message from the binary object sequence returned from the Window Server. It then prints the message to a stream. **NXGetMemoryBuffer()** gives a pointer to this message, which **_NXLogError()** uses to log the error message, as described earlier. By default, all other exception types are reraised to upper-level handlers. If the default top-level exception handler receives a PostScript exception, it logs the corresponding error message in much the same way as illustrated here.

This excerpt contains a number of additional points of interest. First, note that **NXPing()** is called after the PostScript code is sent to the Window Server. Since an application and the Window Server are separate processes that execute asynchronously, notification of an error might not be received by the application until after control has passed from the exception-handling domain. Calling **NXPing()** keeps the two processes synchronized, ensuring that exceptions generated by the PostScript code are raised within the exception-handling domain.

Second, using **PSrun()** assumes that the file containing the PostScript program has the same path on all machines that run this application—a perilous assumption! **PSrun()** was used for simplicity in this example; a better choice for sending PostScript code to the Server is **DPSWriteData()**. (See the *Client Library Reference Manual* by Adobe Systems, Inc. for more information on **DPSWriteData()**).

Finally, the method of protecting an application's VM illustrated here isn't foolproof. In particular, **restore** doesn't affect the contents of the dictionary stack. If an imported PostScript program removes dictionaries that your application placed on the stack, they won't be restored. Perhaps the best way to protect your application from errors caused by an imported PostScript code is to execute the code in a context separate from that of your application. Then, if the code generates errors, your application can respond by calling **DPSDestroyContext()**.

Managing Exception Data

The macro **NX_RAISE()** takes three arguments: an exception code and two pointers to data that you might pass to the exception handler. So far in this discussion, we've used one of these pointers to pass a error message that's associated with the exception (see "Associating Messages with Codes" above). You might, in some circumstances, need to pass additional data along to the exception handler. The functions **NXAllocErrorData()** and **NXResetErrorData()** help you manage the memory that you might allocate for this additional data.

NXAllocErrorData() controls a buffer for error data, allocating the amount of memory you request; **NXResetErrorData()** frees this memory. The Application Kit calls **NXResetErrorData()** each time through the event loop, making it unnecessary for you to call the function directly. (However, if your application doesn't use the Application object's event loop, be sure to call **NXResetErrorData()** after getting each event in order to free this memory.) Thus, by using **NXAllocErrorData()** whenever you need to allocate memory within your exception handler, you don't have to be concerned about freeing this memory when it's no longer needed.

NXAllocErrorData() takes two arguments: One specifies the amount of memory to allocate, and the other refers to a pointer to this memory. This code excerpt illustrates its use:

```
CheckSearchKey()
{
    char *errorData;
    char theKey[1024];
```

```
    NX_DURING
        /* get search string and initialize theKey */
        if (/* invalid search key */) {
            NXAllocErrorData(strlen(theKey+1),
                &(void *)errorData);
            strcpy(errorData, theKey);
            NX_RAISE(DB_invalidSearchKey,
                dbErrorMessages[DB_ invalidSearchKey -
                    NX_APPBASE], errorData);
        }
        . . .
    NX_HANDLER
        switch(NXLocalHandler.code) {
            case DB_invalidSearchKey:
                NXRunAlertPanel (NXLocalHandler.data1,
                    NXLocalHandler.data2, "Restart",
                    NULL, NULL);
                NX_RERAISE();
            . . .
            default:
                NX_RERAISE();
        }
    NX_ENDHANDLER
    return;
}
```

In this example, if an exception occurs **NXAllocErrorData()** allocates memory for the search key that caused the exception. The key is then copied into the newly allocated memory. **NX_RAISE()** passes this data to the local exception handler where it's used as part of the text of an attention panel displayed to the user. From there, the exception is raised to the next higher-level handler.

We might have avoided allocating the storage represented by **errorData** by simply using **theKey** as the third argument to **NX_RAISE()**. However, **theKey** is only defined within the scope of this function; reraising the exception within the local exception handler exits this block, and so **theKey** would be undefined for higher-level exception handlers. Copying the search key into memory allocated with **NXAllocErrorData()** not only makes it available to the higher-level handler, but ensures that the memory will be freed when it's no longer needed.