ГЛАВА 1

Обработка ошибок

Прежде чем изучать функции, предлагаемые Microsoft Windows, посмотрим, как в них устроена обработка ошибок.

Когда вы вызываете функцию Windows, она проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если вы передали недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, она возвращает значение, свидетельствующее об ошибке. В таблице 1-1 показаны типы данных для возвращаемых значений большинства функций Windows.

Табл. 1-1. Стандартные типы значений, возвращаемых функциями Windows

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно. Таких функций в Windows очень мало
BOOL	Если вызов функции заканчивается неудачно, возвращается 0; в остальных случаях возвращаемое значение отлично от 0. (Не пытайтесь проверять его на соответствие TRUE, лучше проверить его на соответствие FALSE.)
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL; в остальных случаях HANDLE идентифицирует объект, которым Вы можете манипулировать. Будьте осторожны: некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным –1. В документации Platform SDK для каждой функции четко указывается, что именно она возвращает при ошибке — NULL или INVALID_HANDLE_VALUE
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL; в остальных случаях PVOID сообщает адрес блока данных в памяти

Табл. 1-1. (*окончание*)

Тип данных	Значение, свидетельствующее об ошибке
LONG	Это значение — «крепкий орешек». Функции, которые сообщают или DWORD значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не сумела сосчитать то, что вы хотели, она обычно возвращает 0 или –1 (все зависит от конкретной функции). Если вы используете одну из таких функций, проверьте по документации Platform SDK, каким именно значением она уведомляет об ошибке

При возникновении ошибки вы должны разобраться, почему вызов данной функции оказался неудачен. За каждой ошибкой закреплен свой код — 32-битное число.

Функция Windows, обнаружив ошибку, через механизм локальной памяти потока сопоставляет соответствующий код ошибки с вызывающим потоком. (Локальная память потока рассматривается в главе 21.) Это позволяет потокам работать независимо друг от друга, не вмешиваясь в чужие ошибки. Когда функция вернет вам управление, ее возвращаемое значение будет указывать на то, что произошла какая-то ошибка. Какая именно — вы узнаете, вызвав функцию *GetLastError*:

```
DWORD GetLastError();
```

Она просто возвращает 32-битный код ошибки для данного потока.

Теперь, когда у вас есть код ошибки, вам нужно обменять его на что-нибудь более внятное. Список кодов ошибок, определенных Майкрософт, содержится в заголовочном файле WinError.h. Я приведу здесь его небольшую часть, чтобы вы представляли, на что он похож:

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
//
//  The operation completed successfully.
//
#define ERROR_SUCCESS                    0L

#define NO_ERROR 0L                             // dderror
#define SEC_E_OK                        ((HRESULT)0x00000000L)

//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
//  Incorrect function.
```

```
//
#define ERROR_INVALID_FUNCTION           1L    // dderror

//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND             2L

//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the path specified.
//
#define ERROR_PATH_NOT_FOUND             3L

//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
//  The system cannot open the file.
//
#define ERROR_TOO_MANY_OPEN_FILES        4L

//
// MessageId: ERROR_ACCESS_DENIED
//
// MessageText:
//
//  Access is denied.
//
#define ERROR_ACCESS_DENIED              5L
```

Как видите, с каждой ошибкой связаны идентификатор сообщения (его можно использовать в исходном коде для сравнения со значением, возвращаемым GetLastError), текст сообщения (описание ошибки на нормальном языке) и номер (вместо него лучше использовать идентификатор). Учтите, что я показал лишь крошечную часть файла WinError.h; на самом деле в нем более 39 000 строк!

Функцию *GetLastError* нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян (перезаписан

кодом ошибки другой функции или кодом ERROR_SUCCESS в случае успешного завершения функции).

Некоторые функции Windows всегда завершаются успешно, но по разным причинам. Например, попытка создать объект ядра «событие» с определенным именем может быть успешна либо потому, что вы действительно создали его, либо потому, что такой объект уже есть. Но иногда нужно знать причину успеха. Для возврата этой информации Майкрософт предпочла использовать механизм установки кода последней ошибки. Так что и при успешном выполнении некоторых функций вы можете вызывать *GetLastError* и получать дополнительную информацию. К числу таких функций относится, например, *CreateEvent*. Сведения о других функциях и примеры возврата ERROR_ALREADY_EXISTS в случае, если именованное событие существует, см. в Platform SDK.

На мой взгляд, особенно полезно отслеживать код последней ошибки в процессе отладки. Кстати, отладчик в Microsoft Visual Studio позволяет настраивать окно Watch так, чтобы оно всегда показывало код и описание последней ошибки в текущем потоке. Для этого надо выбрать какую-нибудь строку в окне Watch и ввести «@err,hr». Теперь посмотрите на рис. 1-1. Видите, я вызвал функцию *CreateFile*. Она вернула значение INVALID_HANDLE_VALUE (–1) типа HANDLE, свидетельствующее о том, что ей не удалось открыть заданный файл. Но окно Watch показывает нам код последней ошибки (который вернула бы функция *GetLastError*, если бы я ее вызвал), равный 0x00000002, и описание «The system cannot find the file specified» («Система не может найти указанный файл»). Именно эта строка и определена в заголовочном файле WinError.h для ошибки с кодом 2.

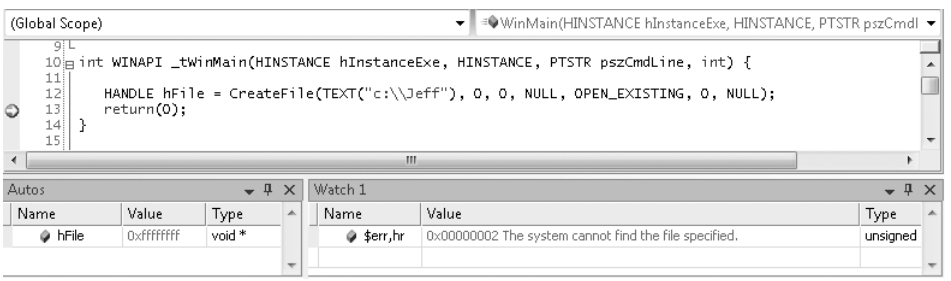
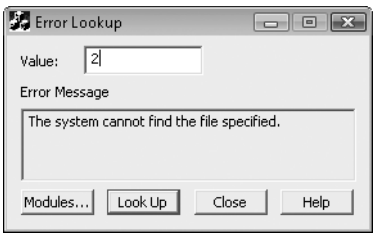


Рис. 1-1. *Используя «@err,hr» в окне Watch среды Visual Studio, вы можете просматривать код последней ошибки в текущем потоке*

С Visual Studio поставляется небольшая утилита Error Lookup, которая позволяет получать описание ошибки по ее коду.

Если приложение обнаруживает какую-нибудь ошибку, то, как правило, сообщает о ней пользователю, выводя на экран ее описание. В Windows для этого есть специальная функция, которая «конвертирует» код ошибки в ее описание, — *FormatMessage*:

```
DWORD FormatMessage(
   DWORD dwFlags,
   LPCVOID pSource,
   DWORD dwMessageId,
   DWORD dwLanguageId,
   PTSTR pszBuffer,
   DWORD nSize,
   va_list *Arguments);
```



FormatMessage — весьма богатая по своим возможностям функция, и именно ее желательно применять при формировании всех строк, показываемых пользователю. Дело в том, что она позволяет легко работать с множеством языков. *FormatMessage* определяет, какой язык выбран в системе в качестве основного (этот параметр задается через апплет Regional Settings в Control Panel), и возвращает текст на соответствующем языке. Разумеется, сначала вы должны перевести строки на нужные языки и встроить этот ресурс в свой EXE- или DLL-модуль, зато потом функция будет автоматически выбирать требуемый язык. Программа-пример ErrorShow, приведенная в конце главы, демонстрирует, как вызывать эту функцию для получения текстового описания ошибки по ее коду, определенному Майкрософт.

Время от времени меня кто-нибудь да спрашивает, составит ли Майкрософт полный список кодов всех ошибок, возможных в каждой функции Windows. Ответ: увы, нет. Скажу больше, такого списка никогда не будет — слишком уж сложно его составлять и поддерживать для все новых и новых версий системы.

Проблема с подобным списком еще и в том, что вы вызываете одну API-функцию, а она может обратиться к другой, та — к третьей и т. д. Любая из этих функций может завершиться неудачно (и по самым разным причинам). Иногда функция более высокого уровня сама справляется с ошибкой в одной из вызванных ею функций и в конечном счете выполняет то, что вы от нее хотели. В общем, для создания такого списка Майкрософт пришлось бы проследить цепочки вызовов в каждой функции, что очень трудно. А с появлением новой версии системы эти цепочки нужно было бы пересматривать заново.

Вы тоже можете это сделать

Итак, я показал, как функции Windows сообщают об ошибках. Майкрософт позволяет вам использовать этот механизм и в собственных функциях. Допустим, вы пишете функцию, к которой будут обращаться другие программы. Вызов этой функции может по какой-либо причине завершиться неудачно, и вам тоже нужно сообщать об ошибках. С этой целью вы просто устанавливаете код последней ошибки в потоке и возвращаете значение FALSE, INVALID_HANDLE_VALUE, NULL или что-то другое, более подходящее в Вашем случае. Чтобы установить код последней ошибки в потоке, вы вызываете *SetLastError*

```
VOID SetLastError(DWORD dwErrCode);
```

и передаете ей нужное 32-битное число. Я стараюсь использовать коды, уже определенные в WinError.h, — при условии, что они подходят под те ошибки, о которых могут сообщать мои функции. Если вы считаете, что ни один из кодов в WinError.h не годится для ошибки, возможной в вашей функции, определите свой код. Он представляет собой 32-битное значение, которое разбито на поля, показанные в следующей таблице.

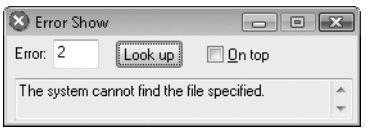
Табл. 1-2. Поля кода ошибки

Биты	31–30	29	28	27–16	15–0
Содержимое:	Код степени «тяжести» (severity)	Кем определен — Майкрософт или пользователем	Зарезервирован	Код подсистемы (facility code)	Код исключения
Значение:	0 = успех 1 = информация 2 = предупреждение 3 = ошибка	0 = Майкрософт 1 = пользователь	Должен быть 0	Первые 256 значений определяются Майкрософт	Определяется Майкрософт или пользователем

Подробнее об этих полях я рассказываю в главе 24. На данный момент единственное важное для вас поле — бит 29. Майкрософт обещает, что все коды ошибок, генерируемые ее функциями, будут содержать 0 в этом бите. Если вы определяете собственный код ошибки, запишите сюда 1. Тогда у вас будет гарантия, что ваш код ошибки не войдет в конфликт с кодом, определенным Майкрософт, — ни сейчас, ни в будущем. Заметьте, что код Facility может принимать 4096 возможных значений, из которых первые 256 Майкрософт зарезервировала для собственных нужд, а остальные доступны для использования в приложениях всем желающим.

Программа-пример ErrorShow

Эта программа, «01 ErrorShow.exe» (см. листинг на рис. 1-2), демонстрирует, как получить текстовое описание ошибки по ее коду. Файлы исходного кода и ресурсов программы находятся в каталоге 01-ErrorShow на компакт-диске, прилагаемом к книге, а также на сайте *http://wintellect.com/Books.aspx*. Программа ErrorShow в основном предназначена для того, чтобы вы увидели, как работают окно Watch отладчика и утилита Error Lookup. После запуска ErrorShow открывается следующее окно.



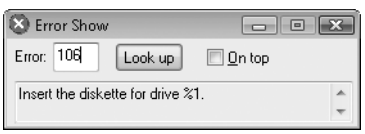
В поле Error можно ввести любой код ошибки. Когда вы щелкнете кнопку Look Up, внизу, в прокручиваемом окне появится текст с описанием данной ошибки. Единственная интересная особенность программы заключается в том, как она обращается к функции *FormatMessage*. Я использую эту функцию так:

```
// получаем код ошибки
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

HLOCAL hlocal = NULL;   // буфер для строки с описанием ошибки

// Мы ищем сообщения Windows, поэтому используем системные
// региональные параметры по умолчанию
// Примечание. Эта комбинация MAKELANGID имеет значение 0
DWORD systemLocale = MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL);

// получаем описание ошибки по коду
BOOL fOk = FormatMessage(
   FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
   FORMAT_MESSAGE_ALLOCATE_BUFFER,
   NULL, dwError, systemLocale,
   (PTSTR) &hlocal, 0, NULL);

if (!fOk) {
   // Это ошибка, связанная с сетью?
   HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
      DONT_RESOLVE_DLL_REFERENCES);

   if (hDll != NULL) {
      fOk = FormatMessage(
         FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_IGNORE_INSERTS |
         FORMAT_MESSAGE_ALLOCATE_BUFFER,
```

```
         hDll, dwError, systemLocale,
         (PTSTR) &hlocal, 0, NULL);
      FreeLibrary(hDll);
   }
}

if (fOk && (hlocal != NULL)) {
   SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
   LocalFree(hlocal);
} else {
   SetDlgItemText(hwnd, IDC_ERRORTEXT,
      TEXT("No text found for this error number."));
```

Первая строка считывает код ошибки из текстового поля. Далее я создаю экземпляр описателя (handle) блока памяти и инициализирую его значением NULL. Функция *FormatMessage* сама выделяет нужный блок памяти и возвращает нам его описатель.

Вызывая *FormatMessage*, я передаю флаг FORMAT_MESSAGE_FROM_SYSTEM. Он сообщает функции, что мне нужна строка, соответствующая коду ошибки, определенному в системе. Кроме того, я передаю флаг FORMAT_MESSAGE_ALLOCATE_BUFFER, чтобы функция выделила соответствующий блок памяти для хранения текста. Описатель этого блока будет возвращен в переменной hlocal. Флаг FORMAT_MESSAGE_IGNORE_INSERTS позволяет заменять в сообщениях параметры, которые используются Windows для передачи более детальной контекстной информации, подстановочными знаками, как показано на следующем рисунке:



Без этого флага необходимо передать вместо подстановочных знаков значения в параметре **Arguments**, но в случае Error Show это невозможно, поскольку содержимое сообщений заранее не известно.

Третий параметр указывает код интересующей нас ошибки, а четвертый — язык, на котором мы хотим увидеть ее описание. Поскольку мы хотим получить сообщения, переданные Windows, идентификатор языка создается из пары определенных констант, в результате получается 0 — значение, соответствующее языку по умолчанию, заданному в операционной системе. Это пример ситуации, в которой невозможно «зашить» в код идентификатор языка, поскольку нельзя узнать заранее, какой язык используется в той копии операционной системы, где будет запущена программа ErrorShow.

Если выполнение *FormatMessage* заканчивается успешно, описание ошибки помещается в блок памяти, и я копирую его в прокручиваемое окно, расположенное в нижней части окна программы. А если вызов *FormatMessage*

оказывается неудачным, я пытаюсь найти код сообщения в модуле NetMsg.dll, чтобы выяснить, не связана ли ошибка с сетью (о поиске DLL на диске см. в главе 20). Используя описатель NetMsg.dll, я вновь вызываю *FormatMessage*. Дело в том, что у каждого DLL или EXE-модуля может быть собственный набор кодов ошибок, который включается в модуль с помощью Message Compiler (MC.exe). Как раз это и позволяет делать утилита Error Lookup через свое диалоговое окно Modules.