Zgłaszanie błędów

HRESULT

Każda metoda interfejsu COM zwraca informację o błędzie w postaci typu `HRESULT` (`long int`). Struktura reprezentacji bitowej `HRESULT` podzielona jest na 4 sekcje:

- **1 bit błędu**
- **4 bity zarezerwowane dla przyszłych zastosowań**
- **11 bitów określających źródło błędu**
- **16 bitów określających rodzaj błędu**

HRESULT
Bit błędu

Bit błędu informuje czy zwrócony kod sygnalizuje błędnie wykonaną operację, czy jedynie jej zwraca status:

- **`SEVERITY_SUCCESS` (0) – operacja powiodła się**
- **`SEVERITY_ERROR` (1) – operacja zakończyła się błędem**

Do sprawdzenia wartości bitu błędu służy para makrodefinicji `SUCCEEDED(hr)` i `FAILED(hr)`. Zwracają one wartość `TRUE` odpowiednio dla operacji zakończonej pomyślnie i operacji zakończonej błędem.
Istnieje też makrodefinicja `HRESULT_SEVERITY(hr)`, która zwraca wartość bitu błędu.

HRESULT
Źródło błędu

Pole źródła błędu definiuje, jaka usługa wywołała błąd. Zdefiniowane są obecnie następujące źródła błędów związane z COM:

- `FACILITY_DISPATCH` (2) – błędy wywołane w metodach interfejsu `IDispatch`
- `FACILITY_ITF` (4) – błędy w innych metodach interfejsu
- `FACILITY_NULL` (0) – niezdefiniowane źródło błędu
- `FACILITY_RPC` (1) – błędy związane z RPC
- `FACILITY_STORAGE` (3) – błędy związane z metodami `IStorage` i `IStream`
- `FACILITY_WIN32` (7) – błędy zwracane przez funkcje systemowe Win32
- `FACILITY_WINDOWS` (8) – inne błędy pochodzące od systemu Windows

Wartość pola źródła błędu można odczytać za pomocą makrodefinicji `HRESULT_FACILITY(hr)`.

HRESULT
Kod błędu

Kody błędów definiowane są niezależnie dla różnych źródeł błędów. Zakres kodów 0x0000 – 0x01FF zarezerwowany jest dla standardowych błędów operacji. Pozostały zakres 0x0200 – 0xFFFF może zostać wykorzystany do niestandardowych innych komunikatów. Wartość kodu błędu można odczytać makrodefinicją `HRESULT_CODE(hr)`.

Do konwersji błędów systemowych odczytywanych funkcją `GetLastError()` do postaci `HRESULT` służy makrodefinicja `HRESULT_FROM_WIN32(err)`.

Własny kod błędu można utworzyć za pomocą makrodefinicji `MAKE_HRESULT(sev,fac,code)`, gdzie:

- `sev` = `SEVERITY_SUCCESS` lub `SEVERITY_ERROR`
- `fac` = `FACILITY_ITF`
- `code` = wartość z zakresu 0x0200 – 0xFFFF

HRESULT

HRESULT	Value	Description
E_ABORT	0x80004004	The operation was aborted because of an unspecified error.
E_ACCESSDENIED	0x80070005	A general access-denied error.
E_FAIL	0x80004005	An unspecified failure has occurred.
E_HANDLE	0x80070006	An invalid handle was used.
E_INVALIDARG	0x80070057	One or more arguments are invalid.
E_NOINTERFACE	0x80004002	The **QueryInterface** method did not recognize the requested interface. The interface is not supported.
E_NOTIMPL	0x80004001	The method is not implemented.
E_OUTOFMEMORY	0x8007000E	The method failed to allocate necessary memory.
E_PENDING	0x8000000A	The data necessary to complete the operation is not yet available.
E_POINTER	0x80004003	An invalid pointer was used.
E_UNEXPECTED	0x8000FFFF	A catastrophic failure has occurred.
S_FALSE	0x00000001	The method succeeded and returned the boolean value FALSE.
S_OK	0x00000000	The method succeeded. If a boolean return value is expected, the returned value is TRUE.

HRESULT

Często stosowanym rozwiązaniem stosowanym do identyfikacji błędów są stałe wyliczeniowe:

```
#include <winerror.h>
#define MAKE_HRESULT_ERROR(nCode) \
 MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF, nCode)

typedef [uuid(39fb3ab1-de17-4a10-a3db-95236c055135),
         helpstring("Possible error return codes")]
enum
{
 [helpstring("The hardware does not support the high perf. counters.")]
    tmErrorHighPerfTimerNotSupported = MAKE_HRESULT_ERROR(0x0200),
 [helpstring("Start needs to be called before ElapsedTime.")]
    tmErrorStartNotCalled = MAKE_HRESULT_ERROR(0x0201),
} TmStopwatchError;

library TIMERSLib
{
...
  enum TMStopwatchError;
...
}
```

Rozszerzona informacja o błędzie

W celu umożliwienia klientowi odczytu rozszerzonej informacji o błędzie opracowano standardowe interfejsy `ISupportErrorInfo` i `IErrorInfo`.

`ISupportErrorInfo` zawiera tylko jedną metodę `InterfaceSupportErrorInfo()` informującą czy dla określonego interfejsu dostępna jest rozszerzona informacja o błędzie.

```
interface ISupportErrorInfo: IUnknown
{
  HRESULT InterfaceSupportsErrorInfo( [in]  REFIID riid );
}
```

Rozszerzona informacja o błędzie
ISupportErrorInfo

```
//Custom example
HRESULT CStopwatch::InterfaceSupportsErrorInfo(REFIID riid)
{
        if (riid == IID_IStopwatch)
         return S_OK;
        else
          return S_FALSE;
}
//ATL generated form
STDMETHODIMP CStopwatch::InterfaceSupportsErrorInfo(REFIID riid)
{
        static const IID* arr[] =
        {
                &IID_IStopwatch,
                &IID_IStopwatch2,
        };
        for (int i=0; i < sizeof(arr) / sizeof(arr[0]); i++)
        {
                if (InlineIsEqualGUID(*arr[i],riid))
                        return S_OK;
        }
        return S_FALSE;
}
```

Rozszerzona informacja o błędzie
ICreateErrorInfo

Komponent, gdy podczas wykonywania procedury wystąpił błąd, powinien utworzyć obiekt `ICreateErrorInfo` za pomocą funkcji `CreateErrorInfo()`:

```
ICreateErrorInfo* pCreateErrorInfo;
CreateErrorInfo(&pCreateErrorInfo);
```

Obiekt `ICreateErrorInfo` zawiera zestaw metod pozwalających na ustawienie rozszerzonej informacji o błędzie:

```
interface ICreateErrorInfo: IUnknown
{
  HRESULT SetGUID( [in] REFGUID rguid );
  HRESULT SetSource( [in] LPOLESTR szSource );
  HRESULT SetDescription( [in] LPOLESTR szDescription );
  HRESULT SetHelpFile( [in] LPOLESTR szHelpFile );
  HRESULT SetHelpContext( [in] DWORD dwHelpContext );
}
```

Rozszerzona informacja o błędzie
`ICreateErrorInfo`

- `SetDescription()` – definiuje opis zaistniałego błędu
- `SetSource()` – identyfikuje komponent, w którym wystąpił błąd
- `SetGUID()` – identyfikuje interfejs, dla którego wystąpił błąd.
- `SetHelpContext()` – definiuje identyfikator opisu błędu w pliku pomocy.
- `SetHelpFile()` – definiuje lokalizację i nazwę pliku pomocy.

Rozszerzona informacja o błędzie `ICreateErrorInfo`

Dla utworzonego obiektu `ICreateErrorInfo` należy wywołać metody `Set...()` i podać informacje dotyczące błędu. Następnie z tego obiektu należy pobrać interfejs `IErrorInfo` i ustawić bierzącą informację o błędzie funkcją `SetErrorInfo()`.

```
HRESULT CStopwatch::Start() {
  if ( QueryPerformanceCounter((LARGE_INTEGER*) &m_nStartTime) )
    return S_OK;
  else {
    ICreateErrorInfo* pCreateErrorInfo;
    IErrorInfo* pErrorInfo;

    CreateErrorInfo(&pCreateErrorInfo);

    pCreateErrorInfo->SetDescription("High perf. counters not supported");
    pCreateErrorInfo->SetGUID(IID_IStopwatch);
    pCreateErrorInfo->SetSource(L"Component.Stopwatch");

    pCreateErrorInfo->QueryInterface(IID_IErrorInfo,(void**)&pErrorInfo);
    SetErrorInfo(0, pErrorInfo);

    pErrorInfo->Release();
    pCreateErrorInfo->Release();

    return tmErrorHighPerformanceTimersNotSupported;
  }
}
```

Rozszerzona informacja o błędzie `Error()`

W celu uproszczenia procedury tworzenia opisu błędu klasa ATL `CComCoClass` będąca klasą bazową komponentów ATL COM, posiada zestaw funkcji `Error()`, które są odpowiedzialne za procedurę ustawiania informacji o błędzie:

```
HRESULT Error( LPCOLESTR lpszDesc, const IID& iid, HRESULT hRes );
HRESULT Error( LPCOLESTR lpszDesc, DWORD dwHelpID, LPCOLESTR lpszHelpFile,
               const IID& iid, HRESULT hRes );
HRESULT Error( LPCSTR lpszDesc, const IID& iid, HRESULT hRes );
HRESULT Error( LPCSTR lpszDesc, DWORD dwHelpID, LPCSTR lpszHelpFile,
               const IID& iid, HRESULT hRes );
HRESULT Error( UINT nID, const IID& iid, HRESULT hRes, HINSTANCE hInst );
HRESULT Error( UINT nID, DWORD dwHelpID, LPCOLESTR lpszHelpFile,
               const IID& iid, HRESULT hRes, HINSTANCE hInst );
```

```
STDMETHODIMP CStopwatch::Start()
{
  if ( QueryPerformanceCounter((LARGE_INTEGER*) &m_nStartTime) )
    return S_OK;
  else
    return Error("High perf. counters not supported",
                 IID_IStopwatch,
                 tmErrorHighPerformanceTimersNotSupported);
}
```

Rozszerzona informacja o błędzie `Error()`

Jeżeli obiekt nie dziedziczy od `CComCoClass` można użyć globalnych funkcji AtlReportError:

```
HRESULT AtlReportError( const CLSID& clsid, LPCOLESTR lpszDesc,
                        const IID& iid, HRESULT hRes );
HRESULT AtlReportError( const CLSID& clsid, LPCOLESTR lpszDesc,
                        DWORD dwHelpID, LPCOLESTR lpszHelpFile,
                        const IID& iid, HRESULT hRes );
HRESULT AtlReportError( const CLSID& clsid, LPCSTR lpszDesc,
                        const IID& iid, HRESULT hRes );
HRESULT AtlReportError( const CLSID& clsid, LPCSTR lpszDesc,
                        DWORD dwHelpID, LPCSTR lpszHelpFile,
                        const IID& iid, HRESULT hRes );
HRESULT AtlReportError( const CLSID& clsid, UINT nID, const IID& iid
                        HRESULT hRes, HINSTANCE hInst );
HRESULT AtlReportError( const CLSID& clsid, UINT nID,
                        DWORD dwHelpID, LPCOLESTR lpszHelpFile,
                        const IID& iid, HRESULT hRes, HINSTANCE hInst );
```

Rozszerzona informacja o błędzie `Error()`

Gdy błędy zdefiniowano w pliku IDL jako stałe wyliczeniowe, można odczytać opis błędu (`helpstring`) bezpośrednio z biblioteki typów:

```
#define SET_ERROR_INFO(hr) SerErrorInfo(hr, L#hr)

HRESULT SetErrorInfo(HRESULT hr, WCHAR* ErrorName)
{
  HRESULT hresult;
  CComPtr<ITypeLib> pTypeLib;
  CComPtr<ITypeInfo> pTypeInfo;
  USHORT NumberOfIds = 1;
  MEMBERID Id;
  unsigned long lHashVal = 0;

  CComBSTR name(ErrorName);
  CComBSTR err_msg;
  hresult = AtlModuleLoadTypeLib(&_Module, 0, &(CComBSTR)NULL, &pTypeLib);
  if (SUCCEEDED(hresult)) {
    hresult =pTypeLib->FindName(name, 0, &pTypeInfo, &Id, &NumberOfIds);
    if (SUCCEEDED(hresult) && (Id != MEMBERID_NIL)) {
       hresult=pTypeInfo->GetDocumentation(Id, NULL, & err_msg, NULL, NULL);
    }
  }
  if (SUCCEEDED(hresult))
    AtlReportError(CLSID_Stopwatch, err_msg, IID_IStopwatch, hr);
  return hresult;
}
```

Rozszerzona informacja o błędzie `IErrorInfo`

W celu odczytania rozszerzonej informacji o błędzie klient po sprawdzeniu czy użyty interfejs zwraca taką informację (`ISupportErrorInfo:: InterfaceSupportErrorInfo()`) może ją odczytać za pomocą funkcji `GetErrorInfo()` zwracającej wskaźnik do obiektu `IErrorInfo`.

```
interface IErrorInfo: IUnknown
{
   HRESULT GetGUID( [out] GUID * pGUID );
   HRESULT GetSource( [out] BSTR * pBstrSource );
   HRESULT GetDescription( [out] BSTR * pBstrDescription );
   HRESULT GetHelpFile( [out] BSTR * pBstrHelpFile );
   HRESULT GetHelpContext( [out] DWORD * pdwHelpContext );
}
```

Rozszerzona informacja o błędzie
IErrorInfo

```
//Raw interface error handling
hr = pStopwatch->ElapsedTime( &nElapsedTime );

if (SUCCEEDED(hr))
  std::cout << "The overhead time is "<< nElapsedTime << std::endl;
else
{
  ISupportErrorInfo* pSupportErrorInfo;
  if (SUCCEEDED(pStopwatch->QueryInterface(IID_ISupportErrorInfo,
    (void**)&pSupportErrorInfo)))
  {
    if (pSupportErrorInfo->InterfaceSupportErrorInfo(IID_IStopwatch)
        == S_OK)
    {
       IErrorInfo* pErrorInfo;
       GetErrorInfo(0,&pErrorInfo);
       BSTR description;
       pErrorInfo->GetDescription(&description);
       wprintf(L"HRESULT = %x, Description: %s\n", hr, description);
       SysFreeString(description);
       pErrorInfo->Release();
    }
    pSupportErrorInfo->Release();
  }
}
```

Rozszerzona informacja o błędzie
Smart pointers, `_com_error`

Gdy klient wywołuje metody za pomocą inteligentnych wskaźników, a komponent zwrócił błąd, to zgłoszony zostaje zostaje wyjątek typu `_com_error`. Klasa `_com_error` zawiera w sobie pola `HRESULT` i `IErrorInfo`.

```
// Constructors
_com_error(HRESULT hr,
           IErrorInfo* perrinfo = NULL,
           bool fAddRef = false) throw();
_com_error(const _com_error& that) throw();

// Destructor
virtual ~_com_error() throw();

  // Assignment operator
_com_error& operator=(const _com_error& that) throw();
```

Rozszerzona informacja o błędzie
_com_error

```
// Accessors
HRESULT Error() const throw();
IErrorInfo * ErrorInfo() const throw();

// IErrorInfo method accessors
_bstr_t Description() const throw(_com_error);
DWORD HelpContext() const throw();
_bstr_t HelpFile() const throw(_com_error);
_bstr_t Source() const throw(_com_error);
GUID GUID() const throw();

// FormatMessage accessors
const TCHAR * ErrorMessage() const throw();
```

Rozszerzona informacja o błędzie
Komunikaty podczas debuggowania

Do śledzenia działania programu zdefiniowano funkcję `AtlTrace(LPCTSTR lpszFormat, ...)` oraz makrodefinicje `ATLTRACE()` i `ATLTRACE2()`. Pozwalają one na wyświetlanie komunikatów w oknie debuggera. Podczas kompilacji w wersji *release* funkcje te nie podejmują żadnego działania.

```
...
catch (_com_error e)
{
  ATLTRACE(_T("ERROR(%d)", e.Description());
  std::cout<<"ERROR(" << e.Description() << ")";
}
```