Legal Information
Internet Information Server

Overview

The Microsoft Internet Application Programming Interface (ISAPI) is a high-performance alternative to Common Gateway Interface (CGI) executable files. See **Advantages of DLLs Over Executable Files**.

This document describes the basics for:

Writing an Internet Server Application

With this, you can create fast, run-time DLLS. These low-overhead applications consistently out-perform executables, both in speed and under load.

Writing an ISAPI Filter

These custom filter DLLs can be used to handle authentication, encryption, and many other applications flowing between the network connection and the HTTP Server.

This document also contains a section that describes the design of the **Internet Service Manager** (ISM) Application and the Internet Service Manager API set (ISMAPI). The Internet Service Manager is used to manage a set of configuration DLLs and perform configuration tasks. These DLLs are used to administer the Internet Server services.

Writing an Internet Server Application

The first section of this documentation describes how to write an Internet Server application and begins with an introduction to Common Gateway Interfaces. Application developers should also review the additional instructions contained in **Important Notes**.

This section deals specifically with writing Internet Server applications for the Microsoft® Windows NT® operating system. However, it can also be used to build a sharable image for any operating system, provided the operating system supports loadable, shared images. Process Software has built an OpenVMS-loadable image based on this documentation for a Web server running on OpenVMS.

Introduction to CGI

Common Gateway Interface (CGI) is an interface for running external programs or gateways under an information server. Currently, the only supported information servers are HTTP servers. What is referred to as gateways are actually programs that handle information requests and return the appropriate document or generate a document on the fly.

In using CGI, your server can access information in a form not readable by the client (such as an SQL database) and then act as a gateway between the two to produce information that the client can use.

With the ever-expanding services available through the Web, more and more CGI applications will be developed. This requires a closer examination of the existing server-executed CGI applications with a view to improving performance.

A server responds to a CGI execution request from a client browser by creating a new process and then passing the data received from the browser through the environment variables and stdin. Results gathered by the CGI application are expected on the stdout of the newly created process. The server creates as many processes as the number of requests received.

For more information on the Common Gateway Interface, refer to http://hoohoo.ncsa.uiuc.edu/cgi/.

Drawbacks with Current HTTP Servers

Existing HTTP servers create a separate process for each request received. The more concurrent requests there are, the more concurrent processes created by the server. However, creating a process for every request is time-consuming and requires large amounts of server RAM. In addition, this can restrict the resources available for sharing from the server application itself, slowing down performance, and increasing wait times on the Web.

One way to avoid this is to convert the current CGI executable file into a DLL. The server loads the DLL the first time a request is received and the DLL then stays in memory, ready to service other requests until the server decides it is no longer needed.

Advantages of DLLs Over Executable Files

In the Microsoft® Windows® operating system, dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a dynamic-link library (DLL), which contains one or more functions that are compiled, linked, and stored separately from the processes which use them. For example, the Microsoft® Win32® Application Programming Interface (API) is implemented as a set of dynamic-link libraries, so any process using the Win32 API uses dynamic linking.

There are two methods for calling a function in a DLL:

- *Load-time dynamic linking:* This occurs when an application's code makes an explicit call to a DLL function. This type of linking requires that the executable module of the application be built by linking with the DLL's import library, which supplies the information needed to locate the DLL function when the application starts.
- *Run-time dynamic linking:* This occurs when a program uses the **LoadLibrary** and **GetProcAddress** functions to retrieve the starting address of a DLL function. This type of linking eliminates the need to link with an import library.

This section focuses on the latter category of DLLs. These DLLs, also called ISAPI applications, are loaded at run time by the HTTP server and are called at the common entry points of **GetExtensionVersion** and **HttpExtensionProc**. For more information on this, see **DLL Entry Points**.

Unlike .EXE type, script-executable files, the ISAPI application DLLs are loaded in the same address space as the HTTP server. This means all the resources made available by the HTTP server process are also available to the ISAPI application DLLs. There is minimal overhead associated with executing these applications because there is no additional overhead for each request. Preliminary benchmark programs show that loading ISAPI application DLLs in process can perform considerably faster than loading them into a new process. In addition, in-process applications scale much better under heavy load.

Since an HTTP server knows the ISAPI application DLLs that are already in memory, it is possible for the server to unload the ISAPI application DLLs that have not been accessed in a configurable amount of time. By preloading an ISAPI application DLL, the server can speed up even the first request for that ISAPI application. In addition, unloading ISAPI application DLLs that have not been used for some time will free up system resources.

ISAPI Applications Architecture and CGI Architecture

Multiple ISAPI application DLLs can reside in the same process as the HTTP server, while the conventional CGI applications run in different processes.

Interaction between an HTTP server and an ISAPI application DLL is accomplished through extension control blocks (ECBs). These control blocks are explained in **Interaction Between the HTTP Server and the ISAPI Applications**. In the case of conventional CGI executable files, the server creates a separate process for each request and communicates with the created process through environment variables and stdin/stdout.

The following illustration explains how an ISAPI application DLL interacts with an HTTP server and shows the interaction of script-executable files with an HTTP server.

{ewc msdncd, EWGraphic, bsd23508 0 /a "SDK.WMF"}

The ISAPI application DLLs must be multithread-safe since multiple requests will be received simultaneously. For information on how to write multithread-safe DLLs, refer to the related articles on the Microsoft Development Library CD-ROM, or any of the books on Win32 programming. Similarly, for information on thread-safe DLLs and the scope of usage of C run-time routines in a DLL, see the articles on sharing data in a DLL on the Microsoft Development Library CD-ROM.

Converting Existing CGI Scripts to ISAPI Application DLLs

This section explains the basic requirements for converting an existing CGI script-executable file to an ISAPI application DLL. As with other DLLs, Web server applications should be thread-safe. More than one client will be executing the same function at the same time, so the code should follow safety procedures in modifying a global or static variable.

By using appropriate synchronization techniques, such as critical sections and semaphores, this issue can be handled properly. For additional information on writing thread-safe DLLs, see the documentation in the Win32 SDK and in the Microsoft Development Library.

The primary differences between an ISAPI application DLL and a CGI executable file include the following:

- An ISAPI application receives most of its data through the *lpbData* member of the ECB as opposed to reading it from stdin. For any additional data, the extension uses the **ReadClient** callback function.
- The common CGI variables are provided in the ECB. For other variables, call **GetServerVariable**. In a CGI executable file, these are retrieved from the environment table using **getenv**.
- When sending data back to the client, use the **WriteClient** callback function instead of writing to stdout.
- When specifying a completion status, instead of sending a "Status: NNN xxxxx..." to stdout, send either the header directly using the **WriteClient** callback function, or use the HSE_REQ_SEND_RESPONSE_HEADER, **ServerSupportFunction**.
- When specifying a redirect with the "Location:" or "URI:" header, instead of writing the header to stdout, use the HSE_REQ_SEND_URL if the URL is local. However, if the URL is remote or unknown, use the HSE_REQ_SEND_URL_REDIRECT_RESP, **ServerSupportFunction** callback function.

Interaction Between the HTTP Server and the ISAPI Applications

The HTTP server communicates with the ISAPI applications through a data structure called the EXTENSION_CONTROL_BLOCK (ECB). A client uses an ISAPI application just like its CGI counterpart except, instead of referencing "http://scripts/foo.exe?Param1+Param2" in the CGI, the following form would be used:

"http://scripts/foo.dll?Param1+Param2"

In addition to identifying the files with extensions .EXE and .BAT as CGI executable files, the server will identify a file with a .DLL extension as a script to execute and, in the currrent version, will recognize an .ISA in place of a .DLL. When the server loads the .DLL, it calls the .DLL at the entry point of **GetExtensionVersion** to retrieve the version number of the document on which the extension is based, and a short readable description for server administrators. For every client request, the **HttpExtensionProc** entry point is called.

The extension receives commonly needed information such as the query string, path information, method name, and the translated path. (Retrieving data sent by the client browser is explained in detail later in this section.) The server communicates with the extension DLL through the EXTENSION_CONTROL_BLOCK data structure.

```
typedef struct _EXTENSION_CONTROL_BLOCK {
    DWORD     cbSize;                 // Size of this structure.
    DWORD     dwVersion;              // Version information of
                                      // this spec.
    HCONN     ConnID;                 // Context number not to
                                      // be modified!
    DWORD     dwHttpStatusCode;       // HTTP Status code
    CHAR      lpszLogData[HSE_LOG_BUFFER_LEN];// null-terminated log
                                              // information
                                              // specific to this
                                              // Extension DLL.
    LPSTR     lpszMethod;             // REQUEST_METHOD
    LPSTR     lpszQueryString;        // QUERY_STRING
    LPSTR     lpszPathInfo;           // PATH_INFO
    LPSTR     lpszPathTranslated;     // PATH_TRANSLATED
    DWORD     cbTotalBytes;           // Total bytes indicated from
                                      //  client
    DWORD     cbAvailable;            // Available number of bytes
    LPBYTE    lpbData;                // Pointer to cbAvailable
                                      //  bytes
    LPSTR     lpszContentType;        // Content type of client data
    BOOL (WINAPI * GetServerVariable) ( HCONN       hConn,
                                        LPSTR       lpszVariableName,
                                        LPVOID      lpvBuffer,
                                        LPDWORD     lpdwSizeofBuffer );
    BOOL (WINAPI * WriteClient)    ( HCONN     ConnID,
                                     LPVOID    Buffer,
                                     LPDWORD   lpdwBytes,
                                     DWORD     dwReserved );
    BOOL (WINAPI * ReadClient)     ( HCONN     ConnID,
                                    LPVOID     lpvBuffer,
                                    LPDWORD    lpdwSize );
    BOOL (WINAPI * ServerSupportFunction)( HCONN      hConn,
```

```
                                        DWORD       dwHSERRequest,
                                        LPVOID      lpvBuffer,
                                        LPDWORD     lpdwSize,
                                        LPDWORD     lpdwDataType );
}
```

This control block contains the following fields:

Field	Remarks
cbSize (IN)	The size of this structure.
dwVersion (IN)	The version information of this document. The HIWORD has the major version number and the LOWORD has the minor version number.
connID (IN)	A unique number assigned by the HTTP server and which should *not* be modified.
dwHttpStatusCode (OUT)	The status of the current transaction when the request is completed.
lpszLogData (OUT)	Buffer of size HSE_LOG_BUFFER_LEN. Contains a null-terminated log information string, specific to the ISAPI applications, of the current transaction. This log information will be entered in the HTTP server log. Maintaining a single log file with both HTTP server and ISAPI application transactions is very useful for administration purposes.
lpszMethod (IN)	The method with which the request was made. This is equivalent to the CGI variable REQUEST_METHOD.
lpszQueryString (IN)	A null-terminated string containing the query information. This is equivalent to the CGI variable QUERY_STRING.
lpszPathInfo (IN)	A null-terminated string containing extra path information given by the client. This is equivalent to the CGI variable PATH_INFO.
lpszPathTranslated	A null-terminated string

(IN)	containing the translated path. This is equivalent to the CGI variable PATH_TRANSLATED.
cbTotalBytes (IN)	The total number of bytes to be received from the client. This is equivalent to the CGI variable CONTENT_LENGTH. If this value is 0xffffffff, then there are 4 gigabytes or more of available data. In this case, **ReadClient** should be called until no more data is returned.
cbAvailable (IN)	The available number of bytes (out of a total of *cbTotalBytes*) in the buffer pointed to by *lpbData*. If *cbTotalBytes* is the same as *cbAvailable*, the *lpbData* variable will point to a buffer that contains all the data as sent by the client. Otherwise, *cbTotalBytes* will contain the total number of bytes of data received. The ISAPI applications will then need to use the callback function **ReadClient** to read the rest of the data (beginning from an offset of *cbAvailable*).
lpbData (IN)	This points to a buffer of size *cbAvailable* that has the data sent by the client. You will get the first 48k of data.
lpszContentType (IN)	A null-terminated string containing the content type of the data sent by the client. This is equivalent to the CGI variable CONTENT_TYPE.

Entry Points for Internet Web Server Applications

All DLLs written as Internet Web Server applications are required to export two entry points: **GetExtensionVersion** and **HttpExtensionProc (TerminateExtension** is optional).

When the HTTP server loads an ISAPI application for the first time after loading the DLL, it calls the **GetExtensionVersion** function. If this function does not exist, the call to load the ISAPI application will fail. The recommended implementation of this function is:

```
BOOL WINAPI GetExtensionVersion( HSE_VERSION_INFO  *pVer )
{
```

```
    pVer->dwExtensionVersion = MAKELONG( HSE_VERSION_MINOR,
                                         HSE_VERSION_MAJOR );
    lstrcpyn( pVer->lpszExtensionDesc,
              "This is a sample Web Server Application",
               HSE_MAX_EXT_DLL_NAME_LEN );
    return TRUE;
}
```

The second required entry point is:

```
DWORD  HttpExtensionProc( LPEXTENSION_CONTROL_BLOCK  *lpEcb );
```

This entry point is similar to the **main** function and uses the callback functions to read client data and decide on the action to be taken. Before returning to the server, a properly formatted response must be sent to the client either through **WriteClient** or **ServerSupportFunction**.

Return Values

These are the possible return values.

Value	Meaning
HSE_STATUS_SUCCESS	The ISAPI application has finished processing. The server can disconnect and free up allocated resources.
HSE_STATUS_SUCCESS_AND_KEEP_CONN	The ISAPI application has finished processing and the server should wait for the next HTTP request if the client supports persistent connections. The application should return this only if it was able to send the correct content length header to the client. The server is not required to keep the session open. The application should return this value only if it has sent a connection: a keep-alive header to the client.
HSE_STATUS_PENDING	The ISAPI application has queued the request for processing and will notify the server when it has finished. See HSE_REQ_DONE_WITH_SESSION under the **ServerSupportFunction** function.
HSE_STATUS_ERROR	The ISAPI application has encountered an error while processing the request. The server can disconnect and free up allocated resources.

ISAPI applications can expose an optional export which is similar to **GetExtensionVersion** and is called

TerminateExtension. This function is new and is called just before the server unloads the application. It provides a safe way to clean up threads and complete other shutdown type activities. The prototype is:

```
BOOL  WINAPI   TerminateExtension( DWORD dwFlags );
```

Where *dwFlags* is a bitfield consisting of the following values:

Bit Flags for **TerminateExtension**

Value	Meaning
HSE_TERM_ADVISORY_UNLOAD	The server wants to unload the extension. The extension can return TRUE if OK, or FALSE if the server should not unload the extension.
HSE_TERM_MUST_UNLOAD	The server is indicating the extension is about to be unloaded, the extension cannot refuse.

```
//

#define HSE_TERM_ADVISORY_UNLOAD                0x00000001
#define HSE_TERM_MUST_UNLOAD                    0x00000002
```

Important Notes

The application is called at **HttpExtensionProc** and receives a pointer to the extension control block (ECB) structure. The application then determines what needs to be done by reading the client input (calling the functions **GetServerVariable** and, if necessary, **ReadClient**). This is similar to setting up environment variables and reading stdin.

The ISAPI application DLL is loaded in the same process as the HTTP server. Consequently, an access violation by ISAPI applications can crash some HTTP servers. Therefore, you should thoroughly test the ISAPI applications to ensure integrity. This is important since malfunctioning ISAPI applications can corrupt the server's memory space, or cause memory or resource leaks, if the application fails to clean up after itself.

To alleviate this problem, many HTTP servers wrap the entry points for the ISAPI applications in a __try/__except clause so access violations or other exceptions will not directly affect the server. For more information on the __try/__except clause, refer to the Win32 API documentation.

The main entry point in the ISAPI applications, **HttpExtensionProc**, takes only one input parameter: a pointer to the structure type EXTENSION_CONTROL_BLOCK. Application developers are not expected to change the following fields in the ECB structure: *cbSize, dwVersion,* and *connID*.

However, developers are encouraged to initialize their DLL automatically by defining an entry-point function for the DLL (for example, **DllMain**). The operating system calls this entry point function by default, the first time a **LoadLibrary** call or the last time a **FreeLibrary** call is made for that DLL, or when a new thread is created or destroyed in the process.

Developers are also encouraged to maintain statistical information, or any information pertaining to the DLL, within the DLL itself. By creating appropriate forms, you can measure the usage/performance of a DLL remotely. Also, this information can be reviewed through the performance functions for integration with PerfMon. The *lpszLogData* field of the ECB can also be used to log data to the server log.

Writing an ISAPI Filter

This section of the documentation describes how to write an Internet Server API (ISAPI) filter for the Microsoft® Internet Server.

ISAPI Filter Overview

An Internet Server API (ISAPI) filter is a replaceable, dynamic-link library (DLL) which the server calls whenever there is an HTTP (Hypertext Transfer Protocol) request. When the filter is first loaded, it communicates to the server what sort of notifications will be accepted. After that, whenever a selected event occurs, the filter is called upon to process the event.

ISAPI filters are very powerful and can be used to facilitate a number of different applications, including the following:

- Custom authentication schemes
- Compression
- Encryption
- Logging
- Traffic analysis or other request analyses (for example, looking for requests to "..\..\etc\password")

Multiple filters can be installed. The notification order is based on the priority specified by the filter and, after that, the load order in the registry to resolve any ties.

Note Once a filter has signaled interest in a request, it will receive that data regardless of whether the request is for a file, a CGI (Common Gateway Interface) application, or an ISAPI application. ISAPI filters can be used to enhance the Microsoft Internet Server with custom features such as enhanced logging of HTTP requests, custom encryption, compression schemes, or new authentication methods. The filter applications sit between the network connection to the clients and the HTTP server.

Depending on the options chosen, the filter application can act on several server operations. This includes reading raw data from the client, processing headers, enabling communications over a secure port (for example, PCT−Private Communication Technology, and SSL−Secure Sockets Layer), as well as other stages in the processing of the HTTP request.

The setup program that installs the ISAPI filter should add it to the registry under "HKEY_LOCAL_MACHINE\ System\ CurrentControlSet\ Services\ W3Svc\ Parameters\ Filter DLLs." This is a comma-separated list and the filter should be added to the end of the list. When the filter is deinstalled, care should be taken not to disturb the other strings (other ISAPI filters that may have been added or removed since the DLL in question was installed).

DLL Entry Points

Every filter is contained in a separate DLL with two common entry points: **GetFilterVersion** and **HttpFilterProc**. When the DLL is loaded, **GetFilterVersion** is called. This lets the filter know the version of the server and also lets the filter tell the server the filter version and the events that the filter is interested in.

After this, the server will call the filter's **HttpFilterProc** entry point with appropriate notifications. Note that filters should register only for notifications that the filter needs to see – some filter notifications are very expensive in terms of CPU resources and I/O throughput, and can have a significant effect on the speed and scalability of the Microsoft Internet Server.

```
BOOL WINAPI GetFilterVersion(
        HTTP_FILTER_VERSION *pVer
       );

DWORD WINAPI HttpFilterProc(
        HTTP_FILTER_CONTEXT *pfc,
        DWORD notificationType,
        VOID *pvNotification
       );
```

Using ISAPI Filter Functions

When the server starts up, it reads the values and loads the DLLs listed. It then calls the **GetFilterVersion** entry point to exchange version information and determine the requested notifications and the order in which to deliver them. As events occur, the server will notify each filter application registered for an event in the priority order requested by **GetFilterVersion.** This is accomplished by calling the **HttpFilterProc** entry point. In the event of a tie, the order listed in the registry is used.

Every ISAPI filter DLL must export at least two entry points: the **GetFilterVersion** and the **HttpFilterProc**.

When the **GetFilterVersion** entry point is called, an HTTP_FILTER_VERSION structure is presented which must be filled out with the version information, the requested events, and a priority level. ISAPI filter applications should register only for the events that are immediately required, because registering for unnecessary events can have a significant, negative impact on performance and scalability.

After this first exchange, every time the server processes one of the available events, it will call any filters that have been registered for that event. The order in which the server calls the filters depends first on the priority specified in the *dwFlags* member of HTTP_FILTER_VERSION by **GetFilterVersion**. In the event that two or more different filters have registered for the same event at the same priority, the order that the filters were loaded from the registry determines the order in which they will be called.

When the **HttpFilterProc** entry point is called, the filter will typically perform a switch on the *notificationType* parameter to determine what action to take. For example, an encryption or compression filter will probably register for reading and writing raw data, while a logging filter will probably only register for the log event. Most filters will also register for the end of the net session event. This event is an opportune time to recycle any buffers used by that client request.

For performance reasons, most filters usually keep a pool of filter buffers and only allocate or free them when the pool becomes empty or too large to save on the overhead of the memory management. One useful callback is the **AllocMem** callback in the HTTP_FILTER_CONTEXT structure. This allocates memory that is automatically freed when the communication with the client is terminated. As noted, this can have a negative impact on performance, but with careful use it can be a valuable tool.

Filters

The following filters are used by the Internet Server.

GetFilterVersion

HttpFilterProc

GetFilterVersion

The **GetFilterVersion** function is the first entry point called by the Internet Server.

```
BOOL WINAPI GetFilterVersion(
  PHTTP_FILTER_VERSION pVer
);
```

Parameters

pVer

The HTTP_FILTER_VERSION structure pointed to by this parameter contains both the version information for the server and the fields for the client to indicate version number, notifications, and priority desired. There is also a space for the filter application to register a description of itself.

Return Values

The return code indicates whether the filter was properly loaded. If the filter returns FALSE, the filter application will be unloaded and will not receive any notifications.

Remarks

The **GetFilterVersion** function, implemented in the ISAPI filter application, is the first entry point called by the Internet Server. It is important to specify only the necessary notifications in the *pVer->dwFlags* member. Some notifications can have a strong impact on performance and scalability.

In addition to the notification flags described under the **HttpFilterProc** function, there are also priority flags to specify the order in which to call the filter:

Value	Meaning
SF_NOTIFY_ORDER_DEFAULT	Load the filter at the default priority (recommended).
SF_NOTIFY_ORDER_LOW	Load the filter at a low priority.
SF_NOTIFY_ORDER_MEDIUM	Load the filter at a medium priority.
SF_NOTIFY_ORDER_HIGH	Load the filter at a high priority.

Note Every filter is contained in a separate DLL with two common entry points: **GetFilterVersion** and **HttpFilterProc**. When the DLL is loaded, **GetFilterVersion** is called. This lets the filter know the server version and also lets the filter tell the server the filter version and the events that the filter is interested in.

After this, the server calls the filter's **HttpFilterProc** entry point with appropriate notifications. Note that filters should only register for notifications the filter needs to see – some filter notifications are very expensive in terms of CPU resources and I/O throughput, and can have a significant effect on the speed and scalability of the Microsoft Internet Server.

```
BOOL WINAPI GetFilterVersion(
        HTTP_FILTER_VERSION *pVer
      );

DWORD WINAPI HttpFilterProc(
        HTTP_FILTER_CONTEXT *pfc,
        DWORD notificationType,
        VOID *pvNotification
      );
```

See Also

HttpFilterProc, HTTP_FILTER_VERSION

HttpFilterProc

The **HttpFilterProc** records notification of events from the server.

```
DWORD WINAPI HttpFilterProc(
  PHTTP_FILTER_CONTEXT pfc,
  DWORD notificationType,
  LPVOID pvNotification
);
```

Parameters

pfc

The HTTP_FILTER_CONTEXT structure pointed to by this parameter contains context information. The *pFilterContext* member can be used by the filter to associate any context information with the HTTP request. The SF_NOTIFY_END_OF_NET_SESSION notification can be used to release any such context information.

notificationType

This indicates the type of event being processed. Valid types are:

SF_NOTIFY_SECURE_PORT
Notify application only for sessions over a secure port.

SF_NOTIFY_NONSECURE_PORT
Notify application only for sessions over a nonsecure port.

SF_NOTIFY_READ_RAW_DATA
Allow the application to see the raw data. The data returned will contain both headers and data.

SF_NOTIFY_PREPROC_HEADERS
The server has preprocessed the headers.

SF_NOTIFY_AUTHENTICATION
The server is authenticating the client.

SF_NOTIFY_URL_MAP
The server is mapping a logical URL to a physical path.

SF_NOTIFY_SEND_RAW_DATA
The server is sending raw data back to the client.

SF_NOTIFY_LOG
The server is writing information to the server log.

SF_NOTIFY_END_OF_NET_SESSION
The session with the client is ending.

SF_NOTIFY_ACCESS_DENIED
Allows an ISAPI Filter to be notified any time the server is about to return a "401 Access Denied". This lets the Filter analyze the failure and return a custom message.

pvNotification

The notification-specific structure.

Notification Type	***pvNotification* points to**
SF_NOTIFY_READ_RAW_DATA	HTTP_FILTER_RAW_DATA
SF_NOTIFY_SEND_RAW_DATA	HTTP_FILTER_RAW_DATA
SF_NOTIFY_PREPROC_HEADERS	HTTP_FILTER_PREPROC_HEADERS

SF_NOTIFY_AUTHENTICATION	HTTP_FILTER_AUTHENT
SF_NOTIFY_URL_MAP	HTTP_FILTER_URL_MAP
SF_NOTIFY_LOG	HTTP_FILTER_LOG
SF_NOTIFY_ACCESS_DENIED	HTTP_FILTER_ACCESS_DENIED

Return Values

The return codes indicate how the application handled the event. Possible return codes are:

SF_STATUS_REQ_FINISHED
The filter has handled the HTTP request. The server should disconnect the session.

SF_STATUS_REQ_FINISHED_KEEP_CONN
This is the same as SF_STATUS_REQ_FINISHED, except that the server should keep the TCP session open if the option was negotiated.

SF_STATUS_REQ_NEXT_NOTIFICATION
The next filter in the notification chain should be called.

SF_STATUS_REQ_HANDLED_NOTIFICATION
This filter handled the notification. No other handlers should be called for this particular notification type.

SF_STATUS_REQ_ERROR
An error occurred. The server should call **GetLastError** and indicate the error to the client.

SF_STATUS_REQ_READ_NEXT
The filter is an opaque stream filter and the session parameters are being negotiated. This is valid only for raw-read notification.

Remarks

This is where the core work of the ISAPI filter applications is done. The various structures pointed to by *pvNotification* contain data and function pointers specific to these operations. See the structure details for more information.

Note Every filter is contained in a separate DLL with two common entry points: **GetFilterVersion** and **HttpFilterProc**. When the DLL is loaded, **GetFilterVersion** is called. This lets the filter know the server version and also lets the filter tell the server the filter version and the events that the filter is interested in.

After this, the server calls the filter's **HttpFilterProc** entry point with appropriate notifications. Note that filters should only register for notifications the filter needs to see – some filter notifications are very expensive in terms of CPU resources and I/O throughput, and can have a significant effect on the speed and scalability of the Microsoft Internet Server.

```
BOOL WINAPI GetFilterVersion(
      HTTP_FILTER_VERSION *pVer
     );

DWORD WINAPI HttpFilterProc(
      HTTP_FILTER_CONTEXT *pfc,
      DWORD notificationType,
      VOID *pvNotification
     );
```

See Also

HTTP_FILTER_CONTEXT, HTTP_FILTER_RAW_DATA, HTTP_FILTER_PREPROC_HEADERS, HTTP_FILTER_AUTHENT, HTTP_FILTER_URL_MAP, HTTP_FILTER_LOG

Functions

The following functions are used by the Internet Server.

GetServerVariable

ReadClient

ServerSupportFunction

WriteClient

GetServerVariable

The **GetServerVariable** function retrieves information about a connection or about the server itself.

```
BOOL WINAPI GetServerVariable(
    HCONN hConn,
    LPSTR lpszVariableName,
    LPVOID lpvBuffer,
    LPDWORD lpdwSizeofBuffer
  );
```

Parameters

hConn
[in] The connection handle.

lpszVariableName
[in] A null-terminated string indicating which variable is being requested.

lpvBuffer
[out] A pointer to the buffer to receive the requested information.

lpdwSizeofBuffer
[in/out] A pointer to DWORD indicating the size of the buffer pointed to by *lpvBuffer*. On successful completion, the DWORD contains the size of bytes transferred into the buffer, including the null-terminating byte.

Return Values

If the function is successful, a return value of TRUE is returned. If the function fails, a return value of FALSE is returned. The Win32 **GetLastError** function can be used to determine why the call failed.

Possible error values include:

Value	Meaning
ERROR_INVALID_PARAMETER	Bad connection handle.
ERROR_INVALID_INDEX	Bad or unsupported variable identifier.
ERROR_INSUFFICIENT_BUFFER	Buffer too small. The required buffer size is *lpdwSize*.
ERROR_MORE_DATA	Buffer too small. Only part of the data is returned. The total size of the data is not known.
ERROR_NO_DATA	The data requested is not available.

Remarks

The **GetServerVariable** function copies information into a buffer supplied by the caller. The information can include CGI variables and information relating to an HTTP connection or to the server itself.

Possible *lpszVariableNames* include:

AUTH_TYPE	This contains the type of authentication used. For example, the string will be "basic" if basic authentication is used, and it will be "NTLM" for Challenge-response. Other authentication schemes will have other strings. Since new authentication types can be added to the Internet Server, it is not possible to list all the string possibilities. If the string is empty, then no authentication is used.
CONTENT_LENGTH	The number of bytes which the script can expect to receive from the client.
CONTENT_TYPE	The content type of the information supplied in the body of a POST request.
PATH_INFO	Additional path information, as given by the client. This consists of the trailing part of the URL after the script name, but before the query string, if any.
PATH_TRANSLATED	This is the value of PATH_INFO, but with any virtual path name expanded into a directory specification.
QUERY_STRING	The information which follows the "?" in the URL that referenced this script.
REMOTE_ADDR	The IP address of the client or agent of the client (for example, gateway or firewall) that sent the request.
REMOTE_HOST	The host name of the client or agent of the client (for example, gateway or firewall) that sent the request.
REMOTE_USER	This contains the user name supplied by the client and authenticated by the server. This comes back as an empty string when the user is anonymous (but authenticated).
UNMAPPED_REMOTE_USER	This is the user name before any ISAPI ApplicationsPI Filter mapped the user making the request to an NT

	user account (which appears as REMOTE_USER).
REQUEST_METHOD	The HTTP request method.
SCRIPT_NAME	The name of the script program being executed.
SERVER_NAME	The server's host name, or IP address, as it should appear in self-referencing URLs.
SERVER_PORT	The TCP/IP port on which the request was received.
SERVER_PORT_SECURE	A string of either zero or 1. If the request is being handled on the secure port, then this will be 1. Otherwise, it will be zero.
SERVER_PROTOCOL	The name and version of the information retrieval protocol relating to this request. This is usually HTTP/1.0.
SERVER_SOFTWARE	The name and version of the Web server under which the ISAPI ApplicationsPI DLL program is running.
ALL_HTTP	All HTTP headers that were not already parsed into one of the previous variables. These variables are of the form HTTP_<header field name>. The headers consist of a null-terminated string with the individual headers separated by line feeds.
HTTP_ACCEPT	Special-case HTTP header. Values of the Accept: fields are concatenated, and separated by a comma (","). For example, if the following lines are part of the HTTP header: accept: */*; q=0.1 accept: text/html accept: image/jpeg the HTTP_ACCEPT variable will have a value of: */*; q=0.1, text/html, image/jpeg
URL (new for version 2.0)	Gives the base portion of the URL.

Note In respect to Auth_Type, if the string is not empty it does not mean the user was authenticated, (if the authentication scheme is not "basic" or "NTLM"). The server allows authentication schemes it does not natively understand since an ISAPI ApplicationsPI Filter may be

able to handle that particular scheme.

ReadClient

The **ReadClient** function reads data from the body of the client's HTTP request.

```
BOOL ReadClient(
    HCONN hConn,
    LPVOID lpvBuffer,
    LPDWORD lpdwSize
  );
```

Parameters

hConn
: [in] A connection handle.

lpvBuffer
: [out] A pointer to the buffer area to receive the requested information.

lpdwSize
: [in/out] A pointer to DWORD indicating the number of bytes available in the buffer. On return, *lpdwSize* will contain the number of bytes actually transferred into the buffer.

Return Values

If the function is successful, a value of TRUE is returned. If an error occurs, a value of FALSE is returned. The **GetLastError** function can be called to determine the cause of the error.

Remarks

The **ReadClient** function reads information from the body of the Web client's HTTP request into the buffer supplied by the caller. Thus, the call can be used to read data from an HTML form that uses the POST method. If more than *lpdwSize* bytes are immediately available to be read, **ReadClient** will return after transferring that amount of data into the buffer. Otherwise, it will block and wait for data to become available. If the socket on which the server is listening to the client is closed, it will return TRUE, but with zero bytes read.

ServerSupportFunction

The **ServerSupportFunction** function is a callback function supplied in the EXTENSION_CONTROL_BLOCK (ECB). It supports several auxiliary functions not covered by other callback functions in the ECB.

```
BOOL ServerSupportFunction(
  HCONN ConnID,
  DWORD dwHSERequest,
  LPVOID lpvBuffer,
  LPDWORD lpdwSize,
  LPDWORD lpdwDataType
);
```

Parameters

ConnID

A logical connection identifier for identifying the client to whom the response data should be sent.

dwHSERequest

A DWORD containing the HTTP Server Extension Request type, which indicates the requested functions by the ISAPI application. The various values are:

HSE_REQ_SEND_URL_REDIRECT_RESP

This sends a 302 (URL Redirect) message to the client. No further processing is needed after the call. This operation is similar to specifying "URI: <URL>" in a CGI script header. The *lpvBuffer* variable should point to a null-terminated string of URL. The variable *lpdwSize* should have the size of *lpvBuffer*. The variable *lpdwDataType* is ignored.

HSE_REQ_SEND_URL

This sends the data specified by the URL to the client as if the client had requested that URL. The null-terminated URL pointed to by *lpvBuffer* must be on the server and must *not* specify protocol information (that is, it must begin with a "/"). No further processing is required after this call. The parameter *lpdwSize* points to a DWORD holding the size of *lpvBuffer*. The parameter *lpdwDataType* is ignored.

HSE_REQ_SEND_RESPONSE_HEADER

This sends a complete HTTP server response header including the status, server version, message time, and MIME version. The ISAPI application should append other HTTP headers such as the content type and content length, followed by an extra "\r\n". This function only takes textual data, up to the first '\0' terminator

HSE_REQ_MAP_URL_TO_PATH

The *lpvBuffer* parameter is a pointer to the buffer that contains the logical path on entry and the physical path on exit. The *lpdwSize* parameter is a pointer to the DWORD containing the size of the buffer passed in *lpvBuffer* on entry, and the number of bytes placed in the buffer on exit. The *lpdwDataType* parameter is ignored.

HSE_REQ_DONE_WITH_SESSION

If the server extension wants to hold on to the session because of extended processing requirements, it needs to tell the server when the session is finished so the server can close it and free the related structures. The parameters *lpvBuffer*, *lpdwSize*, and *lpdwDataType* are all ignored.

lpvBuffer

A pointer to the buffer that contains the primary argument required for the support function requested.

lpdwSize

A pointer to a DWORD that contains the size of data that is passed in the buffer pointed to by *lpvBuffer*. This parameter may be unused for several of the support function options.

lpdwDataType

A pointer to a DWORD containing a pointer, flags, or secondary arguments required for the specified support function.

Return Values

If the function is successful, a value of TRUE is returned. If an error occurs, a value of FALSE is returned. The **GetLastError** function can be called to determine the cause of the error.

Remarks

The **ServerSupportFunction** function provides several support functions that are not covered directly by ECB or the standard callback function. The following list describes the various requests that can be made using the **ServerSupportFunction** function:

1. *dwHSERRequest*: HSE_REQ_IO_COMPLETION

   lpvBuffer
   A pointer to callback function *PFN_HSE_IO_COMPLETION.*

   lpdwSize
   This is ignored.

   LpdwDataType
   A pointer of type (PVOID) which is used as the context.

This option lets the ISAPI application set a callback function and context to use for handling asynchronous I/O operations. The callback function is used only if the function pointer passed is non-NULL. Any context value is allowed. If multiple calls are made for this option, the values used in the most recent call to **ServerSupportFunction** will be used (all old values will be lost). The callback function is defined as:

```
typedef VOID  (WINAPI * PFN_HSE_IO_COMPLETION)(
    IN EXTENSION_CONTROL_BLOCK * pECB,
    IN PVOID    pContext,
    IN DWORD    cbIO,
    IN DWORD    dwError );
```

In case of errors during asynchronous I/O processing, the server makes a single call to the callback function. It is the responsibility of the ISAPI application to do the cleanup during the call.

2. *dwHSERRequest*: HSE_REQ_TRANSMIT_FILE

   lpvBuffer
   A pointer to the HSE_TF_INFO object.

   LpdwSize
   This is ignored.

   LpdwDataType
   This is ignored.

This option lets the ISAPI application use **TransmitFile** to send a file (faster) to the client. The server performs this operation asynchronously. The application should either submit a callback function (and context) by setting the values in the HSE_TF_INFO object passed in or should call the **ServerSupportFucntion** function with HSE_REQ_IO_COMPLETION.

If none of the above is done, this call will fail and the server will return an error (ERROR_INVALID_PARAMETER). Also, the application should specify the file handle in the structure that is passed in.

If all parameters are present, the server submits this operation to its internal asynchronous I/O queue and returns to the caller. At this point, the ISAPI application can return HSE_STATUS_PENDING (if returning from **HttpExtenionProc**) and later use HSE_DONE_WITH_SESSION when the callback occurs indicating that the I/O was completed.

```
typedef struct _HSE_TF_INFO  {
```

```
    //
    // callback and context information
    // the callback function will be called when IO is completed.
    // the context specified will be used during such callback.
    //
    // These values (if non-NULL) will override the one set by calling
    //  ServerSupportFunction() with HSE_REQ_IO_COMPLETION
    //
    PFN_HSE_IO_COMPLETION   pfnHseIO;
    PVOID  pContext;

    // file should have been opened with FILE_FLAG_SEQUENTIAL_SCAN
    HANDLE hFile;

    //
    // HTTP header and status code
    // These fields are used only if HSE_IO_SEND_HEADERS
    //  is present in dwFlags
    //
    LPCSTR pszStatusCode; // HTTP Status Code  eg: "200 OK"

    DWORD  BytesToWrite; // value of "0" means send entire file
    DWORD  Offset;         // offset value within file to start from

    PVOID  pHead;        // pointer to Headers to be sent before file
    DWORD  HeadLength;   // header length
    PVOID  pTail;        // pointer to Tail to be sent after file data
    DWORD  TailLength;   // tail length

    DWORD  dwFlags;      // includes HSE_IO_DISCONNET_AFTER_SEND, ...

} HSE_TF_INFO, * LPHSE_TF_INFO;
```

A special flag, HSE_IO_DISCONNECT_AFTER_SEND, can be used if the application is mainly concerned with transmitting the file and closing the connection. This flag enables the server to optimally reuse its internal buffers and sockets for future connections. Thus, it improves the perceived latency by optimally using system features. However, the application will not be able to do any further data transmission if this flag is indicated, because the session to the client will be torn down.

Another special flag, HSE_IO_SEND_HEADER, can be used for transmitting custom header information to the client. The ISAPI application can store the status code (for example, "200 OK") in the *pszStatusCode* member of the HSE_TF_INFO structure. When the header flag is turned on, the Internet server will automatically construct the appropriate HTTP header and send it to the client along with the file contents.

If this flag is used for header transmission, then the ISAPI application should not send its own header using the **ServerSupportFunction**(*HSE_REQ_SEND_RESPONSE_HEADERS*) function. In addition to the status code, the application can also include a special head buffer for each data chunk that it transmits from a file.

The file handle specified, *hFile,* should be opened using the Win32 **CreateFile** function with the FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_OVERLAPPED turned on. The head and tail buffers specified by *pHead* and *pTail* are optional.

Asynchronous I/O Support

Previous versions of this documentation have supported synchronous I/O operations using the callback functions, **ReadClient** and **WriteClient**. However, the ability to support asynchronous operations is important because it frees up a server pool thread from being blocked in completing the I/O operation. In addition, the Internet server engine already has built-in support to manage asynchronous I/O operations using the completion ports and server thread pool.

Support for asynchronous I/O operations were initially planned for a later version of this documentation. However, because of the need to support asynchronous I/O operations, a Microsoft-specific extension has been provided in this documentation. Thus, asynchronous write operations are supported in this documentation using the existing callback function **WriteClient**, with a special flag indicating that the operation has to be performed asynchronously. In addition, this documentation also provides a mechanism for requesting that the server transmit a file using the **TransmitFile** function. This function is a Win32 function that supports fast transmission of a file-from-file system over any stream sockets. It is supported in the Microsoft Windows Sockets implementation in Windows NT 3.51 and later.

WriteClient

The **WriteClient** function is a callback function supplied in the EXTENSION_CONTROL_BLOCK (ECB) for a request sent to the ISAPI application. This function sends data present in the given buffer to the client that made the request.

```
BOOL WriteClient(
    HCONN ConnID,
    LPVOID Buffer,
    LPDWORD lpdwBytesr,
    DWORD dwReserved
  );
```

Parameters

ConnID

[in] A logical connection identifier for identifying the client to whom the response data should be sent.

Buffer

[in] A pointer to the buffer containing the data to be sent.

lpdwBytes

[in/out] A pointer to a DWORD that contains the number of bytes in the buffer that need to be sent out when the call is made, and contains the number of bytes of data successfully sent out for synchronous write operations. For asynchronous write operations, the returned value has no meaning.

dwReserved

A DWORD containing flags indicating how the I/O operation should be handled. The following flags are supported:

Value	Meaning
HSE_IO_SYNC	This indicates that the I/O operation should be done synchronously.
HSE_IO_ASYNC	This indicates that the I/O operation should be done asynchronously. The ISAPI application should have made a call to the **ServerSupportFunction**(HSE_REQ_IO_COMPLETION) function and submitted a callback function and context value for handling completion of asynchronous operations.

Return Values

If the function is successful, a value of TRUE is returned. If an error occurs, a value of FALSE is returned. The **GetLastError** function can be called to determine the cause of the error.

Remarks

The **WriteClient** function attempts to write the data in the supplied buffer to the socket in which the client request came. For synchronous writes, it attempts to the write in the called thread and the I/O may block trying to send the data to the client. On completion, the **WriteClient** function returns the number of bytes sent in *lpdwBytes*.

For asynchronous writes, the **WriteClient** function submits the write operation to the asynchronous thread queue and returns from the call immediately. At this point, the ISAPI application can choose to do more

background processing or return from the **HttpExtensionProc** function with a HSE_STATUS_PENDING. When the I/O operation is completed, the server calls the callback function submitted by the ISAPI application with the ECB, context value, number of bytes sent, and error codes (if there are any errors).

It is the responsibility of the ISAPI application to do further processing and to use the **ServerSupportFunction**(HSE_DONE_WITH_SESSION) function to notify the server when it is done processing the request.

Only one outstanding asynchronous I/O operation is permitted per request. This includes asynchronous **WriteClient**, asychronous **TransmitFile**, or a **ServerSupportFunction** call with HSE_REQ_SEND_URL.

Asynchronous I/O Support

Previous versions of this documentation have supported synchronous I/O operations using the callback functions, **ReadClient** and **WriteClient**. However, the ability to support asynchronous operations is important because it frees up a server pool thread from being blocked in completing the I/O operation. In addition, the Internet server engine already has built-in support to manage asynchronous I/O operations using the completion ports and server thread pool.

Support for asynchronous I/O operations were initially planned for a later version of this documentation. However, because of the need to support asynchronous I/O operations, a Microsoft-specific extension has been provided in this documentation. Thus, asynchronous write operations are supported in this documentation using the existing callback function **WriteClient**, with a special flag indicating that the operation has to be performed asynchronously. In addition, this documentation also provides a mechanism for requesting that the server transmit a file using the **TransmitFile** function. This function is a Win32 function that supports fast transmission of a file-from-file system over any stream sockets. It is supported in the Microsoft Windows Sockets implementation in Windows NT 3.51 and later.

Modules

The following module is used by the Internet Server.

Http Server Extension Interface

HTTP Server Extension Interface

Module Name: HttpExt.h

The following section contains structure definitions and prototypes for version 1.0 of the HTTP Server Extension interface. See specifically **Converting Existing CGI Scripts to ISAPI Application DLLs** and **Important Notes**.

Structures

The following structures are used by the Internet Server.

HTTP_FILTER_ACCESS_DENIED

HTTP_FILTER_AUTHENT

HTTP_FILTER_CONTEXT

HTTP_FILTER_LOG

HTTP_FILTER_PREPROC_HEADERS

HTTP_FILTER_RAW_DATA

HTTP_FILTER_VERSION

HTTP_FILTER_URL_MAP

HTTP_FILTER_ACCESS_DENIED

Bitfield indicating the requested resource has been denied by the server due to a logon failure, an ACL on a resource, an ISAPI Filter, or an ISAPI application/CGI application.

SF_DENIED_BY_CONFIG can appear with SF_DENIED_LOGON if the server configuration did not allow the user to log on.

```
#define SF_DENIED_LOGON             0x00000001
#define SF_DENIED_RESOURCE          0x00000002
#define SF_DENIED_FILTER            0x00000004
#define SF_DENIED_APPLICATION       0x00000008

#define SF_DENIED_BY_CONFIG         0x00010000
```

typedef struct _HTTP_FILTER_ACCESS_DENIED (added in Version 2.0)

```
{
const CHAR * pszURL;           Requesting URL
const CHAR * pszPhysicalPath;  Physical path of resource
DWORD        dwReason;         Bitfield of SF_DENIED flags
}
```

HTTP_FILTER_ACCESS_DENIED, *PHTTP_FILTER_ACCESS_DENIED

The server will automatically include the supported authentication schemes when an ISAPI application, Filter or CGI Script returns a "401 Access Denied".

HTTP_FILTER_AUTHENT

```
typedef struct _HTTP_FILTER_AUTHENT
{
    CHAR *    pszUser;
    DWORD     cbUserBuff;
    CHAR *    pszPassword;
    DWORD     cbPasswordBuff;
} HTTP_FILTER_AUTHENT, *PHTTP_FILTER_AUTHENT;
```

Members

pszUser
[in/out] A pointer to a string containing the user name for this request. An empty string indicates an anonymous user.

cbUserBuff
[in] The size of the buffer pointed to by *pszUser*. This is guaranteed to be at least SF_MAX_USERNAME.

pszPassword
[in/out] A pointer to a string containing the password for this request.

cbPasswordBuff
[in] The size of the buffer pointed to by *pszPassword*. This is guaranteed to be at least SF_MAX_PASSWORD.

Remarks

When the server is about to authenticate the client, this structure is pointed to by the *pvNotification* in the **HttpFilterProc** when *notificationType* is SF_NOTIFY_AUTHENTICATION. This can be used to implement a different authentication scheme.

See Also

HttpFilterProc

HTTP_FILTER_CONTEXT

```
typedef struct _HTTP_FILTER_CONTEXT
{
    DWORD    cbSize;
    DWORD    Revision;
    PVOID    ServerContext;
    DWORD    ulReserved;
    BOOL     fIsSecurePort;
    PVOID    pFilterContext;
    BOOL    (WINAPI * GetServerVariable) (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    LPSTR      lpszVariableName,
    LPVOID     lpvBuffer,
    LPDWORD    lpdwSize
    );
BOOL    (WINAPI * AddResponseHeaders) (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    LPSTR    lpszHeaders,
    DWORD    dwReserved
    );
BOOL    (WINAPI * WriteClient)  (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    LPVOID     Buffer,
    LPDWORD    lpdwBytes,
    DWORD      dwReserved
    );
VOID *     (WINAPI * AllocMem) (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    DWORD      cbSize,
    DWORD      dwReserved
    );
BOOL    (WINAPI * ServerSupportFunction) (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    enum SF_REQ_TYPE    sfReq,
    PVOID       pData,
    DWORD       ul1,
    DWORD       ul2
    );
} HTTP_FILTER_CONTEXT, *PHTTP_FILTER_CONTEXT;
```

Members

cbSize

[in] The size of this structure, in bytes.

Revision

[in] The revision level of this structure. This is less than or equal to the version of the document, HTTP_FILTER_REVISION.

ServerContext

[in] Reserved for server use.

ulReserved

[in] Reserved for server use.

fIsSecurePort

[in] A value of TRUE indicates that this event is over a secure port.

pFilterContext
[in/out] A pointer to be used by the filter for any context information that the filter wants to associate with this request. Any memory associated with this request can be safely freed during the SF_NOTIFY_END_OF_NET_SESSION notification.

```
BOOL      (WINAPI * GetServerVariable) (
          struct _HTTP_FILTER_CONTEXT *pfc,
          LPSTR lpszVariableName,
          LPVOID lpvBuffer,
          LPDWORD lpdwSize
         );
```

A pointer to a function to retrieve information about the server and this connection. See **GetServerVariable** for more information.

Parameters

pfc
The *pfc* passed to **HttpFilterProc**.

lpszVariableName
The server variable to retrieve.

lpvBuffer
The buffer in which to store the value of the variable.

lpdwSize
The size of the buffer pointed to by *lpvBuffer.*

```
BOOL      (WINAPI * AddResponseHeaders) (
          struct _HTTP_FILTER_CONTEXT *pfc,
          LPSTR lpszHeaders,
          DWORD dwReserved
         );
```

A pointer to a function that adds a header to the HTTP response. See **ServerSupportFunction**, HSE_SEND_RESPONSE_HEADER, for more information.

Parameters

pfc
The *pfc* passed to **HttpFilterProc**.

lpszHeaders
A pointer string containing the headers to add.

dwReserved
Reserved for future use. This must be zero.

```
BOOL      (WINAPI * )    (
          struct _HTTP_FILTER_CONTEXT *pfc,
     .    LPVOID buffer,
          LPDWORD lpdwBytes,
          DWORD dwReserved
         );
```

A pointer to a function that sends raw data back to the client. See **WriteClient** for more information.

Parameters

pfc
The *pfc* passed to **HttpFilterProc**.

buffer
A buffer containing data to send to the client.

lpdwBytes
The size of the buffer pointed to by *buffer*.

dwReserved
Reserved for future use.

```
VOID *       (WINAPI * AllocMem) (
    struct _HTTP_FILTER_CONTEXT *pfc,
        DWORD cbSize,
        DWORD dwReserved
       );
```

A pointer to a function used to allocate memory. Any memory allocated with this function will automatically be freed when the request is completed.

Parameters

pfc
The *pfc* passed to **HttpFilterProc**.

cbSize
The size of the buffer to allocate.

dwReserved
Reserved for future use.

```
BOOL     (WINAPI * ServerSupportFunction) (
    struct _HTTP_FILTER_CONTEXT *pfc,
        enum SF_REQ_TYPE sfReq,
        PVOID pData,
        DWORD ul1,
        DWORD ul2
        );
```

A pointer to a function used to extend the ISAPI filter functions. Parameters are specific to the extensions. Possible values for *sfReq* are SF_REQ_SEND_RESPONSE_HEADER, SF_REQ_ADD_HEADERS_ON_DENIAL, and SF_REQ_SET_NEXT_READ_SIZE.

SF_REQ_SEND_RESPONSE_HEADER
This sends a complete HTTP server response header including the status, server version, message time, and MIME version. Server extensions should append other information, such as content type and content length, followed by an extra "\r\n".

Parameters

pData
A zero-terminated string pointing to optional status string (for example, "401 Access Denied") or NULL for the default response of "200 OK".

ul1
A zero-terminated string pointing to optional data to be appended and sent with the header. If NULL, the header will be terminated with an empty line.

SF_REQ_ADD_HEADERS_ON_DENIAL
If the server denies the HTTP request, add the specified headers to the server error response. This allows an authentication filter to advertise its services without filtering every request. Generally, the headers will be WWW-Authenticate headers with custom authentication schemes. However, no restriction is placed on which headers can be specified.

Parameters

pData
: A zero-terminated string pointing to one or more header lines with terminating "\r\n."

SF_REQ_SET_NEXT_READ_SIZE
: This is used only by raw data filters that return SF_STATUS_READ_NEXT.

Parameters

ul1
: The size in bytes for the next read.

HTTP_FILTER_LOG

```
typedef struct _HTTP_FILTER_LOG
{
    const CHAR *    pszClientHostName;
    const CHAR *    pszClientUserName;
    const CHAR *    pszServerName;
    const CHAR *    pszOperation;
    const CHAR *    pszTarget;
    const CHAR *    pszParameters;
    DWORD        dwHttpStatus;
    DWORD        dwWin32Status;
} HTTP_FILTER_LOG, *PHTTP_FILTER_LOG;
```

Members

pszClientHostName
[in/out] The client's host name.

pszClientUserName
[in/out] The client's user name.

pszServerName
[in/out] The name of the server the client is connected to.

pszOperation
[in/out] The HTTP command.

pszTarget
[in/out] The target of the HTTP command.

pszParameters
[in/out] The parameters passed to the HTTP command.

dwHttpStatus
[in/out] The HTTP return status.

dwWin32Status
[in/out] The Win32 error code.

Remarks

When the server is about to log information to the server log file, this structure is pointed to by the *pvNotification* in the **HttpFilterProc** when *notificationType* is SF_NOTIFY_LOG. The strings cannot be changed, but pointers can be replaced. If string pointers are changed, the memory they point to must have been allocated by the **AllocMem** callback function in the HTTP_FILTER_CONTEXT structure.

HTTP_FILTER_PREPROC_HEADERS

```
typedef struct _HTTP_FILTER_PREPROC_HEADERS
{
BOOL    (WINAPI * GetHeader) (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    LPSTR    lpszName,
    LPVOID   lpvBuffer,
    LPDWORD  lpdwSize
    );
BOOL    (WINAPI * SetHeader) (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    LPSTR    lpszName,
    LPSTR    lpszValue
    );
BOOL    (WINAPI * AddHeader) (
    struct _HTTP_FILTER_CONTEXT *    pfc,
    LPSTR    lpszName,
    LPSTR    lpszValue
    );
DWORD    dwReserved;
} HTTP_FILTER_PREPROC_HEADERS, *PHTTP_FILTER_PREPROC_HEADERS;
```

Members

BOOL (WINAPI * GetHeader) (
struct _HTTP_FILTER_CONTEXT * *pfc,*
LPSTR *lpszName,*
LPVOID *lpvBuffer,*
LPDWORD *lpdwSizeofBuffer*
);

A pointer to a function that retrieves the specified header value. Header names should include the trailing colon (":"). The special values "method", "url", and "version" can be used to retrieve the individual portions of the request line.

Parameters

pfc
The filter context for this request from the *pfc* passed to the **HttpFilterProc**.

lpszName
The name of the header to retrieve.

lpvBuffer
A pointer to a buffer of size *lpdwSizeofBuffer* where the value of the header will be stored.

lpdwSizeofBuffer
This should be set to the size of the buffer *lpvBuffer*, for example, sizeof(*lpvBuffer*). After the call, it contains the number of bytes retrieved including the null terminator. Therefore, for retrieved strings it is equal to strlen(lpvBuffer)+1.

BOOL (WINAPI * SetHeader) (
struct _HTTP_FILTER_CONTEXT **pfc*,
LPSTR *lpszName*,
LPSTR *lpszValue*
);

A pointer to a function used to change or delete the value of a header.

Parameters

pfc
The filter context for this request from the *pfc* passed to the **HttpFilterProc**.

lpszName
A pointer to the name of the header to change or delete.

lpszValue
A pointer to the string to change the header to, or a pointer to "\0" to delete the header.

```
BOOL (WINAPI * AddHeader) (
        struct _HTTP_FILTER_CONTEXT *pfc,
        LPSTR lpszName,
       LPSTR lpszValue
       );
```

A pointer to a function to add a header.

Parameters

pfc
The filter context for this request from the *pfc* passed to the **HttpFilterProc**.

lpszName
A pointer to the name of the header to change or delete.

lpszValue
A pointer to the string to change the header to, or a pointer to "\0" to delete the header.

Remarks

When the server is about to process the client headers, this structure is pointed to by the *pvNotification* in the **HttpFilterProc** when *notificationType* is SF_NOTIFY_PREPROC_HEADERS.

See Also

HttpFilterProc

HTTP_FILTER_RAW_DATA

```
typedef struct _HTTP_FILTER_RAW_DATA
{
    PVOID pvInData;
    DWORD cbInData;
    DWORD cbInBuffer;
    DWORD dwReserved;
} HTTP_FILTER_RAW_DATA, *PHTTP_FILTER_RAW_DATA;
```

Members

pvInData
[in] A pointer to the data buffer (input or output).

cbInData
[in] The amount of data in the buffer pointed to by *pvInData*.

cbInBuffer
[in] The size of the buffer pointed to by *pvInData*.

dwReserved
[in] Reserved for future use.

Remarks

This structure is passed to the SF_NOTIFY_READ_RAW_DATA and SF_NOTIFY_SEND_RAW_DATA notification routines.

See Also

HttpFilterProc

HTTP_FILTER_URL_MAP

```
typedef struct _HTTP_FILTER_URL_MAP
{
    const CHAR *    pszURL;
    CHAR *          pszPhysicalPath;
    DWORD           cbPathBuff;
} HTTP_FILTER_URL_MAP, *PHTTP_FILTER_URL_MAP;
```

Members

pszURL
[in] A pointer to the URL that is being mapped to a physical path.

pszPhysicalPath
[in/out] A pointer to the buffer where the physical path is stored.

cbPathBuff
[in] The size of the buffer pointed to by *pszPhysicalPath*.

Remarks

When the server is about to map the specified URL to a physical path, this structure is pointed to by the *pvNotification* in the **HttpFilterProc** when *notificationType* is SF_NOTIFY_URL_MAP. Filters can modify the physical path in place.

See Also

HttpFilterProc

HTTP_FILTER_VERSION

```
typedef struct _HTTP_FILTER_VERSION
{
    DWORD     dwServerFilterVersion;
    DWORD     dwFilterVersion;
    CHAR      lpszFilterDesc[SF_MAX_FILTER_DESC_LEN+1];
    DWORD     dwFlags;
} HTTP_FILTER_VERSION, *PHTTP_FILTER_VERSION;
```

Members

dwServerFilterVersion

[in] The version of the document used by the server. The version of the current header file is HTTP_FILTER_REVISION.

dwFilterVersion

[out] The version of the document used by the server. The version of the current header file is HTTP_FILTER_REVISION.

lpszFilterDesc

[out] The location in which to store a short string description of the ISAPI filter application.

dwFlags

[out] The combination of SF_NOTIFY_* flags to specify which events this application is interested in. See **HttpFilterProc** for a list of valid flags.

Remarks

This structure is passed to the application's **HttpFilterProc** entry point by the server.

See Also

HttpFilterProc

Internet Service Manager

The following section describes the design of the Internet Service Manager (ISM) application and the Internet Service Manager API set (ISMAPI), which the ISM uses to communicate with the ISM Configuration DLLs.

The Internet Service Manager is designed to be a dynamic, extendable remote and local administration tool, which performs no configuration itself, but instead manages a set of configuration DLLs (also referred to as extension DLLs) to perform configuration tasks. A service is deemed to be "configurable" by the ISM if a configuration DLL has been loaded by it.

Program Structure

The Internet Service Manager package is divided into two parts—the main application and a set of DLLs—one for each service. The main window of the ISM application is used to control the administration environment. The DLLs are set up for each service to perform configuration, discovery, and service status functions. In the first release of the application, we will provide three different DLLs to administer the Microsoft Internet Server services, namely FTP, WWW, and Gopher.

Internet Service Manager (ISM)

The main application is used to control the loading of each configuration DLL. When the application first starts, the ISM performs the following tasks:

1. Reads the registry to load preference settings and discover which extension DLLs it needs to load.
2. Loads and initializes each configuration DLL. The configuration DLL is queried on what type of service it is and what actions it supports.
3. Discovers what configurable services are installed on the local machine and adds them to the default view.
4. Awaits the user's action. Optionally, the ISM may discover available configurable services on the network.

Configuration DLLs

Configuration DLLs (also referred to as extension DLLs) are the bridge between the ISM application and configurable services. A configuration DLL must be able to perform the following tasks for the ISM application:

1. ***Return Service Information:*** Each successfully loaded service DLL tells the manager the name of the service it supports and the actions it supports. It also provides a bitmap to be shown in the toolbar and in the configuration views.
2. ***Configuration:*** When the user double clicks the server item(s) or selects the **Configure** menu item in the main window, the main window issues a call to the related service DLL with the selected computer names. At this point, the configuration DLL takes over and presents the configuration dialogs and returns when configuration is completed. This is done modally.
3. ***Start/Stop/Pause service:*** When the user selects the Start/Stop/Pause action in the menu bar, the main window calls the configuration DLL to issue a command to perform the proper action to the selected servers.
4. ***Discovery:*** There are two types of available discovery supported by the ISM application. The first is INETSLOC discovery, which is supported by all the Internet Servers. If a configuration DLL manages a type of service that is discoverable in this way, it need not provide a discovery API, but only a INETSLOC mask. The ISM application then uses this for discovery. Those services not discoverable in this fashion must provide their own discovery API.

Internet Service Manager Application

The ISM application stores information about the discovered computer and the configurable services that are available on them. This information can be presented in several different ways, as shown in the **Report View**, the **Servers View**, and the **Services View** for the Internet Service Manager. In each of these views, specific services can be included or excluded in the view by selecting the service types buttons in the toolbar, or selecting them in the **View** menu. Only selected services will be shown in the view.

Report View

The Report view presents an alphabetically sorted list of computers and the services running on them. The list can be sorted by computer name, service name, state, or comment.

{ewc msdncd, EWGraphic, bsd23509 0 /a "SDK.WMF"}

Servers View

The Servers view is a pivot of the services view. It is a tree view which has the computer names as top-level items, with the configurable services found on that server as child items.

{ewc msdncd, EWGraphic, bsd23509 1 /a "SDK.WMF"}

Services View

The Services view presents the configurable services as the top-level item in a tree view, with each known server that runs the service enumerated underneath the service.

{ewc msdncd, EWGraphic, bsd23509 2 /a "SDK.WMF"}

Service Features

These are the specific features that are available:

Connecting to a single server

When connecting to a single server, each service on the connected machine that can be administered is enumerated and added to the list.

Discovery

Discovery can be used to find all configurable services on the network.

Service Administration

When a configurable service (or services) is selected, the **Configure** menu item and toolbar button are enabled. Selecting Configure calls the configure API in the service DLL and passes control to it.

Changing Service State (Start/Stop/Pause)

When a controllable service (or services) is selected, the start, stop, and pause menu items and toolbar buttons are enabled in a manner consistent with the current state of the selected service. Selecting any of these menu commands calls upon the appropriate configuration DLL to start, stop, pause, or continue the appropriate service.

Extension DLLs

The Internet Service Manager is designed to support future extension. When a new Internet Server Service is created, the server service may be added to the administration tool using the following steps:

1. The extension service should add itself to the registry key, \\LocalMachine\Software\InetMgr\AddOnServices, as described under **Registry Layout**.
2. The extension service must provide a configuration DLL that contains the five DLL entry points. Optionally, a help file may be supplied, which should differ in name from the extension DLL only in that it should have the file extension .hlp.

Once the registry value entry has been added, the Internet Service Manager will automatically load and initialize the extension DLL when it starts up. It will then be able to administer the new extension service.

Internet Service Manager API (ISMAPI) Interface

The following sections contain the Definitions, Structures, and Functions for the ISMAPI interface.

Definitions

Functions

Structures

Definitions

```
enum
{
    //
    // the service has invoked de-registration or
    // the service has never called registration.
    //
    INetServiceStopped,
    //
    // the service is running.
    //
    INetServiceRunning,
    //
    //  the service is paused.
    //
    INetServicePaused,

};

#define INetServiceUnknown      INetServicePaused + 1

//
// Maximum length of some members in characters
//
#define MAX_SERVERNAME_LEN      256                 // We allow hostnames
#define MAX_COMMENT_LEN         MAXCOMMENTSZ

//
// Dimensions of the toolbar bitmaps
//
#define TOOLBAR_BMP_CX          17
#define TOOLBAR_BMP_CY          17
```

The INetService* definitions are taken straight from INETSLOC, and are provided here only for services that do not support INETSLOC discovery. TOOLBAR_BMP_CX and TOOLBAR_BMP_CY specify the ideal dimensions of the toolbar bitmap the configuration DLL provides.

```
//
// Service information flags
//
#define ISMI_INETSLOCDISCOVER   0x00000001  // Use INETSLOC for discovery
#define ISMI_CANCONTROLSERVICE  0x00000002  // Service state can be changed
#define ISMI_CANPAUSESERVICE    0x00000004  // Service is pausable.
#define ISMI_NORMALTBMAPPING    0x00000100  // Use normal toolbar color
mapping
#define ISMI_VIRTUALHOSTS       0x00000200  // Service supports virtual
hosts
#define ISMI_VIRTUALROOTS       0x00000400  // Service supports virtual
roots

#define MAX_SNLEN               20          // Maximum short name length
#define MAX_LNLEN               48          // Maximum long name length
```

The service information bits are used in the ISMSERVICEINFO structure to specify supported features.

Functions

The following section contains the functions for the ISMAPI interface.

ISMChangeServiceState

ISMConfigureServers

ISMDiscoverServers

ISMQueryServerInfo

ISMQueryServiceInfo

ISMChangeServiceState

DWORD ISMChangeServiceState(
int *nNewState*,
int * *pnCurrentState,*
DWORD *dwReserved*,
LPCTSTR *lpstrServers*
);

Parameters:

nNewState
The requested state.

pnCurrentState
A pointer to the new state after the function call.

dwReserved
Reserved. Must be zero.

lpstrServers
A double null-terminated list of servers.

Return Values

The return value is a WIN32 error code. Additionally, the *pnCurrentState* value returns the current state of the service.

Remarks

This function is called to change the service state (running, stopped, paused) of the selected servers, provided service control is available. The *nNewState* parameter can be INetServiceStarted, INetServiceStopped, or INetServicePaused. The *pnCurrentState* parameter returns the current state of the service after the function returns, which may or may not be the same as the state requested. The *dwReserved* parameter is reserved for future use and must be zero. The *lpstrServers* parameter is a double null-terminated list of servers that the new state is to be applied to.

ISMConfigureServers

DWORD ISMConfigureServers(
HWND *hWnd*,
DWORD *dwReserved*,
LPCTSTR *lpstrServers*
);

Parameters

hWnd
Handle to main window.

dwReserved
Reserved. Must be zero.

lpstrServers
A double null-terminated list of servers.

Return Values

The return value is a WIN32 error code.

Remarks

The **ISMConfigureServers** function is called with a double null-terminated list of servers that are to be administered. It is the responsibility of the configuration DLL to provide the dialogs that the user uses to interact with the service, and it is also the responsibility of the configuration DLL to present multiple-server configuration to the user. The *hWnd* parameter is a window handle to the owning application (typically, the main frame window of the ISM application). The *dwReserved* parameter is reserved for future expansion and should be zero. The *lpstrServers* parameter is a double null-terminated list of server names.

ISMDiscoverServers

DWORD ISMDiscoverServers(
ISMSERVERINFO **psi*,
DWORD **pdwBufferSize*,
int **cServers*
);

Parameters

psi
A pointer to a buffer of ISMSERVERINFO structures.

pdwBufferSize
A pointer to the size of the buffer.

cServers
A pointer to the number of the discovered server.

Return Values

The return value is a WIN32 error code.

Remarks

This function is used only for those configuration DLLs which are not configurable with INETSLOC. The calling program (for example, ISM) calls this function first with a buffer size of zero. It should then return the actual size required for the ISMSERVERINFO structures (for example, one structure per discovered service). The calling program then allocates sufficient storage for this and calls the function again with a pointer to this buffer. The function fills in this buffer with ISMSERVERINFO structures, one per discovered service, and returns the number of discovered servers in *cServers*.

ISMQueryServerInfo

DWORD ISMQueryServerInfo(
LPCTSTR *lpstrServerName*,
ISMSERVERINFO **psi*
);

Parameters

lpstrServerName
A double null-terminated list of servers.

ps
A pointer to a ISMSERVERINFO structure.

Return Values

The return value is a WIN32 error code.

Remarks

This function is called to provide information about the service in respect to a specific server, which may or may not be running the service. The calling program calls this function to determine if the specified host is running the service and, if so, the function then fills in the ISMSERVERINFO structure with the computer name, the comment, and the current state of the service. If the service is not running on the specified computer, this is reflected in the error return code. As with all ISM API calls, it is the responsibility of the calling program to fill in the **dwSize** member of the ISMSERVERINFO structure prior to calling the function, and it is the responsibility of the configuration DLL to provide version control based on this size element.

ISMQueryServiceInfo

DWORD ISMQueryServiceInfo(
ISMSERVICEINFO * *psi*
);

Parameters

psi
A pointer to the SERVICEINFO structure.

Return Values

The return value is a WIN32 error code.

Remarks

This function fills in a ISMSERVICEINFO structure with information about the service and is called immediately after the configuration DLL has been successfully loaded. The calling program (ISM presumably) fills in the **dwSize** element of the ISMSERVICEINFO structure with the size of the structure. It is the responsibility of the configuration DLL to use this size for version control.

Structures

Important: Many of the following structures have a **dwSize** member. Functions that fill in these structures require that this member be filled in before the function is called.

ISMSERVERINFO

ISMSERVICEINFO

ISMSERVERINFO

```
//
// Standard Server information structure.
//
typedef struct tagISMSERVERINFO
{
    DWORD dwSize;                                  // Structure size
    TCHAR atchServerName[ MAX_SERVERNAME_LEN + 1]; // Server name
    TCHAR atchComment[ MAX_COMMENT_LEN + 1 ];      // Server Comment
    int   nState;                                  // Service State
} ISMSERVERINFO, *PISMSERVERINFO;
```

The ISMSERVERINFO structure returns information about a specific computer. The **nState** member can specify INetServiceStarted, INetServiceStopped, INetServicePaused, or INetServiceUnknown. This last value should be used by all services that do not support controllable services.

ISMSERVICEINFO

```
//
// Standard service configuration information structure
//
typedef struct tagISMSERVICEINFO
{
    DWORD dwSize;                    // Structure size
    DWORD dwVersion;                 // Version information
    DWORD flServiceInfoFlags;        // ISMI_ flags
    ULONGLONG ullDiscoveryMask;      // InetSloc mask (if necessary)
    COLORREF rgbButtonBkMask;        // Toolbar button bitmap background
mask
    UINT nButtonBitmapID;            // Toolbar button bitmap resource ID
    COLORREF rgbServiceBkMask;       // Service bitmap background mask
    UINT nServiceBitmapID;           // Service bitmap resource ID
    TCHAR atchShortName[MAX_SNLEN+1]; // The name as it appears in the menu
    TCHAR atchLongName[MAX_LNLEN+1];  // The name as it appears in tool tips
} ISMSERVICEINFO, *PISMSERVICEINFO;
```

The ISMSERVICEINFO structure specifies information about the service, including its name and a description text, and bitmap identifiers and background masks for the toolbar button and service view bitmaps. These two bitmaps can be the same bitmap resource. The background masks refer to the color in the bitmap that is designated as the transparent color. The **flServiceInfoFlags** is made up of bits which specify the supported features. (See the ISMI_ flag definitions above). The ullDiscoveryMask is the mask used for INETSLOC discovery, if supported, and should be zero, if INETSLOC discovery is not supported.

Help Files

Each Internet Service Manager Extension DLL is expected to have its own help file. The name of the help file is generated by the Internet Service Manager at run time and is expected to have the same base name as the configuration DLL, with the file name extension .hlp. When the configuration function is called, this help file is set as the default help file by the ISM application. When the configuration function returns, the old help file name is restored.

Registry Layout

This section describes the registry layout that is used by the Internet Service Manager.

Local_Machine\Software\Microsoft\InetMgr\Parameters

Various preference settings, such as the size of the ISM window and the current view, are stored here.

Local_Machine\Software\Microsoft\InetMgr\Parameters\AddOnServices

This entry consists of a series of REG_SZ entries, the *name* which specifies the name of the service, and *value* which specifies the configuration DLL name. For example, FTP has the following registry entry:

FTP: REG_SZ: fscfg.dll

Local_Machine\Software\Microsoft\InetMgr\Parameters\AddOnTools

This section specifies the add-on tools supported by the ISM application. An add-on tool is a free-standing application, which provides some additional functionality. Add-on tools are run synchronously and can be configured to receive information about the currently selected servers and their services through command line parameters.

A toolbar button is added for each add-on tool consisting of the small icon of the executable, and a menu item is added underneath the **Tools** menu as well. This registry entry consists of a series of REG_SZ entries, the *name* which specifies the menu item associated with it underneath the **Tools** menu (an accelerator may be used), and the *value* which has the following format:

application; tooltips text; selected parameters; nonselected parameters application:
: Executable or batch file associated with the menu item/toolbar button. If this is an executable, its icon will be added to the toolbar.

tooltips text:
: The text as it will appear in the tooltips, when the cursor is over the toolbar button.

selected parameters:
: The command line parameters passed when an item is selected in the current view. There are three escape sequences that can be used here:

  $C Will be replaced with the selected computer name

  $S Will be replaced with the selected service name

  $$ Will be replaced by a single $

nonselected parameters:
: The command line parameters when no item is currently selected.

Any of these items, with the exception of *application*, are optional and can be omitted. For example:
&Sample: REG_SZ: c:\sample.exe;Sample Application;/c $c /s $s;

Local_Machine\Software\Microsoft\InetMgr\Parameters\AddOnHelp

Similar to add-on tools, additional help topics can be added to the **Help** menu. This is a series of registry entries, the *name* which specifies the **Help** menu item (as with add-on tools, an accelerator can be used), and the *value* which specifies an executable or document name to be associated with the menu item.