A Specification For Writing Internet Server Applications

A High Performance alternative to CGI Executables

Introduction to CGI

CGI (Common Gateway Interface) is an interface for running external programs or gateways under an information server. Currently the only supported information servers are HTTP servers. What we refer to gateways are really programs which handle information requests and return the appropriate document or generate a document on the fly. With CGI, your server can serve information which is not in a form readable by the client (such as an SQL database), and act as a gateway between the two to produce something which the client can use.

With the ever expanding services being made available through the Web, more and more CGI applications will be developed. This calls for a closer look at the existing method of server executing CGI applications and possible ways of improving its performance.

The way a server responds to a request for CGI execution from a client browser is to create a new process and pass the data received from the client browser through environment variables and stdin and expects the results gathered by the CGI application, on stdout of the newly created process. The server creates as many processes as the number of requests received, one per each request.

For more information on the CGI specification please refer to http://hoohoo.ncsa.uiuc.edu/cgi/

Drawbacks with current implementations

As explained above, the existing HTTP servers create a separate process for each request received. The more concurrent requests, the more concurrent processes created by the server. Creating a process for every request is time consuming, expensive in terms of server RAM and it can be restrictive as far as sharing the resources of the server application itself.

One way to circumvent creating processes is to convert the current CGI executable into a DLL which the server loads the first time a request is received for that DLL. Then the DLL stays in memory ready to service other requests until the server decides that it's no longer needed.

Note Even though this specification talks specifically about writing Internet Server Applications for Windows NT, the same specification could 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 specification for a web server running on OpenVMS.

Advantages of DLLs over Executables

In Microsoft® Windows™, 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), containing one or more functions that are compiled, linked, and stored separately from the processes using 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 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 occurs when a program uses the LoadLibrary and GetProcAddress functions to get the starting address of a DLL function. This type of linking eliminates the need to link with an import library.

This specification is aimed at are of the latter category of DLLs. These DLLs (also called Internet Server Applications or ISAs) are loaded at run time by the HTTP server. They are called at common entrypoints **GetExtensionVersion()** and **HttpExtensionProc**(). Details of this interaction are explained more in detail in the following chapters.

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

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

The following picture explains how a ISA DLL interacts with an HTTP server with respect to the interaction of script executables with an HTTP server.

ISA architecture and CGI Architecture

As you can see in the above picture, all of the ISA DLLs reside in the same process as the HTTP server, while the conventional CGI apps run in different processes. Interaction between an HTTP server and ISA DLL is via Extension Control Blocks(ECB). The ECB is explained in detail in the following chapter. As the picture above shows, multiple ISA DLLs can co-exist in the same process as the server. In case of conventional CGI executables, the server creates a separate process for each request and server communicates with the created process via environment variables and stdin/stdout.

ISA DLLs need to be multithread safe since multiple requests will be received simultaneously. For more information on how to write multi-thread safe DLLs please refer to any of the several articles on writing multithreaded applications on MSDN or any of the several books on Win32 Programming.

Similarly, for information on thread-safe DLLs and the scope of usage of C-Runtime routines in a DLL, please refer to any of the several articles on sharing data in a DLL on the MSDN CD-ROM.

Detailed Interaction between the HTTP server and ISA

The HTTP server communicates with the ISA via a data structure called Extension Control Block. This data structure is explained in detail in the following section. A client uses an ISA just like its CGI counterpart, except rather then referencing:

"http://scripts/foo.exe?Param1+Param2" in the CGI instance, the following form would be used:

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

This means that in addition to identifying the files with extensions .EXE and .BAT as CGI executables, the server will also identify a file with .DLL extension as a script to execute. When the server loads the .DLL, it calls the DLL at the entry point **GetExtensionVersion**() to get the version number of the specification the Extension is based on and a short human readable description for server administrators. For every client request, the **HttpExtensionProc()** entry point is called. The extension receives the commonly needed information such as the Query String, Path Info, Method Name and the translated path. Subsequent sections of this document explain in detail how to retrieve the data sent by the client browser. The way the server communicates with the Extension DLL is via a data structure called the **EXTENSION_CONTROL_BLOCK**.

The Extension Control Block contains the following fields

cbSize (IN)	The size of this structure.
dwVersion (IN)	The version information of this specification. 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 **NOT** to 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 ISA, of the current transaction. This log information will be entered in the HTTP server log. Maintaining a single log file which with both HTTP server and ISA 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)	Null terminated string containing the query information. This is equivalent to the CGI variable **QUERY_STRING**.
LpszPathInfo (IN)	Null terminated string containing extra path information given by the client. This is equivalent to the CGI variable **PATH_INFO**.
lpszPathTranslated (IN)	Null terminated string containing the translated path. This is equivalent to the CGI variable

	PATH_TRANSLATED.
cbTotalBytes (IN)	The total number bytes to be received from the client. This is equivalent to the CGI variable **CONTENT_LENGTH**. If this value is 0xffffffff then there is four 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 variable **lpbData** will point to a buffer which contains all the data as sent by the client. Otherwise **cbTotalBytes** will contain the total number of bytes of data received. The ISA will then need to use the callback function **ReadClient()** to read the rest of the data (starting from an offset of **cbAvailable)**.
lpbData (IN)	Points to a buffer of size **cbAvailable** that has the data sent by the client.
lpszContentType (IN)	Null terminated string containing the content type of the data sent by the client. This is equivalent to the CGI variable **CONTENT_TYPE**.

Mandatory Entry Points For Internet Web Server Applications

All the DLLs written as Internet Web Server Applications must export the following two entry points. They are:

GetExtensionVersion().When the HTTP server loads a ISA for the first time, after loading the DLL, it calls the **GetExtensionVersion()** function. If this function does not exist, the call to load the ISA 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 a **main**() function a the script executable. This entry point would use the callback functions to read client data and decide on what action to be taken. Before returning back to the server, a properly formatted response must be sent to the client via either the **WriteClient()** or the **ServerSupportFunction()** APIs.

Return Values

HSE_STATUS_SUCCESS
The ISA has finished processing and the server can disconnect and free up allocated resources.

HSE_STATUS_SUCCESS_AND_KEEP_CONN
The ISA has finished processing and the server should wait for the next HTTP request if the client supports persistent connections. The application should only return this if it was able to send the correct Content-Length header to the client. The server is not required to keep the session open.

HSE_STATUS_PENDING
The ISA has queued the request for processing and will notify the server when it has finished. See **HSE_REQ_DONE_WITH_SESSION** under **ServerSupportFunction**.

HSE_STATUS_ERROR
The ISA has encountered an error while processing the request and the server can disconnect and free up allocated resources.

GetServerVariable

Get information about a connection or about the server itself.

```
Prototype:
BOOL WINAPI GetServerVariable(;
HCONN  hConn,
LPSTR    lpszVariableName,
LPVOID  lpvBuffer,
LPDWORD  lpdwSize );
```

Parameters

hConn (IN)
Connection handle.

lpszVariableName (IN)
Null terminated string indicating which variable is being requested. Variable names are as defined in the CGI specification located at http://hoohoo.ncsa.uiuc.edu/cgi/env.htm.

lpvBuffer (OUT)
Pointer to buffer to receive the requested information.

lpdwSize (IN/OUT)
Pointer to DWORD indicating the number of bytes available in the buffer. On successful completion the DWORD contains the number of bytes transferred into the buffer (including the null terminating byte).

Return Value

TRUE if successful, or FALSE if error. The Win32 API call GetLastError can be used to determine why the call failed. Possible error values include:

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

Description: This function copies information (including CGI variables) relating to an HTTP connection, or to the server itself, into a buffer supplied by the caller. Possible **lpszVariableNames** include:

AUTH_TYPE	The type of authorization in use. If the username has been authenticated by the server, this will contain Basic.Otherwise, it will not be present.
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.
GATEWAY_INTERFACE	The revision of the CGI specification to which this server complies. The

	current version is CGI/1.1.
PATH_INFO	Additional path information, as given by the client. This comprises 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 which referenced this script.
REMOTE_ADDR	The IP address of the client.
REMOTE_HOST	The hostname of the client.
REMOTE_USER	This contains the username supplied by the client and authenticated by the server. **REQUEST_METHOD** String The HTTP request method.
SCRIPT_NAME	The name of the script program being executed.
SERVER_NAME	The server's hostname (or IP address) as it should appear in self-referencing URLs.
SERVER_PORT	The TCP/IP port on which the request was received.
SERVER_PROTOCOL	The name and version of the information retrieval protocol relating to this request. Normally HTTP/1.0.
SERVER_SOFTWARE	The name and version of the Professional Web Server under which the CGI program is running.
AUTH_PASS	This will retrieve the the password corresponding to **REMOTE_USER** as supplied by the client. It will be a null terminated string.
ALL_HTTP	All HTTP headers that were not already parsed into one of the above variables. These variables are of the form **HTTP_<header field name>**.
HTTP_ACCEPT	Special case HTTP header. Values of the Accept: fields are concatenated, separated by ", ". E.g. if the following lines are part of the HTTP header: accept: */*; q=0.1 accept: text/html accept: image/jpeg then the **HTTP_ACCEPT** variable will have a value of: */*; q=0.1, text/html, image/jpeg

ReadClient

Read data from the body of the client's HTTP request.

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

Parameters

hConn (IN)
Connection handle.

lpvBuffer (OUT)
Pointer to buffer area to receive the requested information.

lpdwSize (IN/OUT)
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 Value

TRUE on success; FALSE if error. If the call fails, the Win32 API **GetLastError** may be called to determine the cause of the error.

Description: This function reads information from the body of the Web client's HTTP request into the buffer supplied by the caller. Thus, the call might be used to read data from an HTML form which 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 waiting for data to become available. If the socket, on which the server is listening to client, is closed, it will return TRUE but with zero bytes read.

WriteClient

Write data to the client.

```
Prototype:
BOOL WriteClient(;
HCONN hConn,
LPVOID lpvBuffer,
LPDWORD lpdwSize,
DWORD dwReserved );
```

Parameters

hConn (IN)
Connection handle.

lpvBuffer (OUT)
Pointer to the data to be written.

LpdwSize (IN/OUT)
Pointer to the number of bytes in the buffer. On return this will be updated to the number of bytes actually sent on this call. Only if an error has occurred will this be less than the number of bytes in the buffer.

dwReserved
Reserved for future use.

Return Value

TRUE on success; FALSE if error. If the call fails, the Win32 API **GetLastError** may be called to determine the cause of the error.

Description: This function sends information to the HTTP client from the buffer supplied by the caller.

ServerSupportFunction

To provide the ISAs with some general purpose functions as well as functions that are specific to HTTP server implementation.

```
Prototype:
BOOL ServerSupportFunction(;
HCONN hConn,
DWORD dwHSERequest,
LPVOID lpvBuffer,
LPDWORD lpdwSize,
LPDWORD lpdwDataType );
```

Note General purpose functions should have a dwHSERequest value larger than 1000. Values up to 1000 are reserved for mandatory ServerSupportFunctions and should not be used.

The following section describes the various defined values for **dwHSERequest**.

HSE_REQ_SEND_URL_REDIRECT_RESP
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 variable lpvBuffer should point to a null terminated string of URL. Variable **lpdwSize** should have the size of **lpvBuffer**. Variable **lpdwDataType** is ignored.

HSE_REQ_SEND_URL
Sends the data specified by the URL to the client as if the client had requested that URL. The Null terminated URL pointed by **lpvBuffer**, **MUST be on the server and must not specify protocol information(i.e. it must begin with a '/')**.

No further processing is required after this call. Variable **lpdwSize** points to a DWORD holding the size of **lpvBuffer**. Variable **lpdwDataType** is ignored.

HSE_REQ_SEND_RESPONSE_HEADER
Sends a complete HTTP server response header including the status, server version, message time and MIME version. The ISA should append other HTTP Headers at the end such as the Content-Type, Content-Length etc followed by an extra '\r\n'.

lpvBuffer
Points to a null terminated optional status string(i.e 401 Access Denied". If this buffer is null, a default response of "200 Ok" will be sent by this function.

lpdwDataType
This is a zero terminated string pointing to optional headers or data to be appended and sent with the header. If this is NULL, the header will be terminated by a '\r\n' pair.

lpdwSize
Points to the size of the buffer **lpdwDataType**.

HSE_REQ_DONE_WITH_SESSION
If the server Extension wants to hold onto the session because they have extended processing requirements, they need to tell the server when the session is finished so that the server can close it and free the related structures. Variables **lpvBuffer**, **lpdwSize** and **lpdwDataType** are all ignored.

lpvBuffer
Points to a DWORD indicating the status code of the request.

Header File Associated with this Specification

Module Name : HttpExt.h

Abstract :

This module contains the structure definitions and prototypes for the version 1.0 HTTP Server Extension interface.

```
#ifndef _HTTPEXT_H
#define _HTTPEXT_H

#include <windows.h>

#define   HSE_VERSION_MAJOR           1      // major version of this spec
#define   HSE_VERSION_MINOR           0      // minor version of this spec
#define   HSE_LOG_BUFFER_LEN         80
#define   HSE_MAX_EXT_DLL_NAME_LEN  256

typedef   LPVOID  HCONN;

// the following are the status codes returned by the Extension DLL

#define   HSE_STATUS_SUCCESS                      1
#define   HSE_STATUS_SUCCESS_AND_KEEP_CONN        2
#define   HSE_STATUS_PENDING                      3
#define   HSE_STATUS_ERROR                        4

// The following are the values to request services with the
ServerSupportFunction.
//  Values from 0 to 1000 are reserved for future versions of the interface

#define   HSE_REQ_BASE                            0
#define   HSE_REQ_SEND_URL_REDIRECT_RESP          ( HSE_REQ_BASE + 1 )
#define   HSE_REQ_SEND_URL                        ( HSE_REQ_BASE + 2 )
#define   HSE_REQ_SEND_RESPONSE_HEADER            ( HSE_REQ_BASE + 3 )
#define   HSE_REQ_DONE_WITH_SESSION               ( HSE_REQ_BASE + 4 )
#define   HSE_REQ_END_RESERVED                    1000

//
// passed to GetExtensionVersion
//

typedef struct   _HSE_VERSION_INFO {

    DWORD  dwExtensionVersion;
    CHAR   lpszExtensionDesc[HSE_MAX_EXT_DLL_NAME_LEN];

} HSE_VERSION_INFO, *LPHSE_VERSION_INFO;

//
// passed to extension procedure on a new request
//
```

```
typedef struct _EXTENSION_CONTROL_BLOCK {

    DWORD     cbSize;                 // size of this struct.
    DWORD     dwVersion;              // version info 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 info
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     lpdwSize );

    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 );

} EXTENSION_CONTROL_BLOCK, *LPEXTENSION_CONTROL_BLOCK;

//
//  these are the prototypes that must be exported from the extension DLL
//

BOOL  WINAPI   GetExtensionVersion( HSE_VERSION_INFO  *pVer );
DWORD WINAPI   HttpExtensionProc(  EXTENSION_CONTROL_BLOCK *pECB );

// the following type declarations is for the server side

typedef BOOL  (WINAPI * PFN_GETEXTENSIONVERSION)( HSE_VERSION_INFO  *pVer );
typedef DWORD (WINAPI * PFN_HTTPEXTENSIONPROC )( EXTENSION_CONTROL_BLOCK
*pECB );
```

```
#endif  // end definition _HTTPEXT_H
```

Notes to Application Developers

The application will get called at **HttpExtensionProc**() and will be passed a pointer to ECB structure. The application will then decide on what exactly needs to be done, by reading the client input (by calling the functions **GetServerVariable()** and, if necessary, **ReadClient()**). This is similar to setting up environment variables and reading stdin.

Since the ISA DLL is loaded in the same process as the HTTP server, an access violation by the ISA may crash some HTTP servers. As a result you should ensure the integrity of the ISA by testing it thoroughly. ISAs that misbehave may also corrupt the server's memory space or may result in memory or resource leaks if they fail to properly cleanup after themselves. To take help with this problem, many HTTP servers will wrap the ISA entry points in a __try/__except clause so that access violations (or other exceptions) will not directly affect the server. For more information on __try/__except clause please refer to the Win32 API documentation

The main entry point in the ISA, **HttpExtensionProc**()takes only one input parameter - a pointer to structure of type **EXTENSION_CONTROL_BLOCK**. Application developers are not expected to change the following fields in ECB structure: **cbSize**, **dwVersion** & **ConnID**.

Application developers are encouraged to initialize their DLL automatically by defining an entry point function for the DLL (e.g **DllMain**()). The OS will call 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.

Application developers are encouraged to maintain statistical information or any information pertaining to the DLL within the DLL itself. By creating appropriate forms one could measure the usage/performance of a DLL remotely. Also, this information could be exposed via the Performance APIs for integration with PerfMon. The **lpszLogData** field of the ECB can also be used to log data to the Windows NT event viewer.

Steps to convert existing CGI scripts to ISA DLLs

This chapter explains the basic requirements for converting an existing CGI script executable to a ISA DLL. Like any other DLL, WEB Server Applications should be thread safe. This means 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 properly handled. For more explanation on writing thread safe DLLs please refer to the documentation in the Win32 SDK and on MSDN.

The primary differences between a ISA DLL and a CGI executable include the following:

- An ISA will receive most of its data through the **lpbData** member of the **ECB** as opposed to reading it from stdin. For any additional data the extension will need to use the **ReadClient()** callback function.
- The common CGI variables are provided in the **ECB**. For other variables, call **GetServerVariable()**. In a CGI executable these are retrieved from the environment table using **getenv()**.
- When sending data back to the client use the **WriteClient()** callback functions instead of writing to stdout.
- When specifying a completion status, rather then sending a "Status: NNN xxxxx..." to stdout, either send 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 rather then writing the header to stdout use the **HSE_REQ_SEND_URL** (if the URL is local) or **HSE_REQ_SEND_URL_REDIRECT_RESP** (if the URL is remote or unknown) **ServerSupportFunction** callback function.

Conclusions, Acknowledgments and Contact Information

The above proposal by Process Software Corporation is aimed at helping the third party script executable developers to optimize their application and improve the performance. We welcome any suggestions or concerns that you may have and please send them directly to me at the email address ramanathan@process.com.

I would appreciate your feedback especially on the **ServerSupportFunction**. If you have any specific server variable to be returned to an Extension DLL, please drop me a mail and we'll consider including that.

I would like to acknowledge the valuable suggestions made by Chris Adie of EMWAC, UK and I thank him for reviewing the document as it was being written.