Internet Apps With ISAPI

by Steve Troxell

In past issues, you've read about how to use the Common Gateway Interface (CGI) to build Delphi web server applications which respond to user input via a web browser and return dynamically generated HTML pages in response. Now I'd like to show you a different approach for accomplishing similar tasks.

Nearly all web server systems provide a native server API which allows you to write applications to handle certain types of HTTP requests. Netscape provides NSAPI, Microsoft provides ISAPI and other vendors have their own APIs. Each API is proprietary to the web server software it was developed for, so details of their usage will vary. In this issue, we're going to focus on ISAPI, the native server API provided with Microsoft's Internet Information Server (IIS) and Personal Web Server (PWS) products. This article assumes familiarity with the basic issues of developing CGI applications. For more information on CGI, see *Developing Dynamic Web Pages* in Issue 16, and Bob Swart's *Under Construction* column in this issue, which describes his CGI Debugger.

Server Extensions

ISAPI applications are just DLLs with specific entry points which the web server calls in response to a request from a web browser. The DLL is loaded into the same process space as the WWW publishing service itself. Because of this, these DLLs are sometimes referred to as server extensions.

ISAPI apps are called from an HTML page in a browser in the same manner as CGI programs. That is, the relative URL and filename of the program to execute is identified in the HTML, usually via the `ACTION` parameter of a `FORM` tag. The only difference is that you'll reference a DLL rather than an EXE. Beyond that there is no change in how you set up the client-side HTML page. ISAPI DLLs are not specifically 'installed': the web server simply loads whatever DLL is identified in the HTML. Like CGI programs, the DLLs must be in a directory on the web server to which your Internet users have read and execute permissions.

Since an ISAPI app is loaded into server memory when it is first requested, subsequent requests by other web users are handled much faster because there is no additional loading overhead involved. This is the primary advantage of using a native server API over CGI. However, you have to take the good with the bad. Since these DLLs are loaded into the same process space as the web server software, a buggy ISAPI app could bring down the whole server. Also, you'll more than likely have to shut down your WWW publishing service to get it to unload your DLL before you can copy an updated version over it. Further, since the DLL can be called by any number of clients simultaneously, you must program it with 'thread-safe' practices in mind, such as avoiding global variables.

You can see that developing an ISAPI app is a bit more complicated than CGI. You must be extremely careful when developing and debugging any DLL-based web application. You'll save yourself and your site manager a lot of grief if you use a local web server to develop and debug your ISAPI app before installing it on a public server.

The web server communicates with your DLL by passing specific data structures through specific entry points. Fortunately, Delphi 2 ships with a unit called ISAPI.PAS which defines most of the structures, datatypes and constants you need (see Listing 1). An ISAPI DLL must export the two functions shown at the bottom of Listing 1. When the server first loads the DLL, it queries it through the `GetExtensionVersion` function, which simply returns a version number and DLL description (see Listing 2). Whenever the server receives a client request to execute the ISAPI app, the server calls the `HttpExtensionProc` function. This is where to place all the code which constitutes your application.

The Extension Control Block

All communication between the browser client and your program takes place via an Extension Control Block (ECB). If you recall, web servers recognize HTTP requests to execute CGI applications and place the user input and session information for that request in environment variables which the CGI program then reads as input. In the same vein, the web server recognizes a request to execute an ISAPI DLL, allocates and fills an Extension Control Block, loads the DLL and runs the application by calling the `HttpExtensionProc` function and passing a pointer to the ECB.

Listing 1 shows the ECB in detail; let's take a closer look.

The `ConnID` field is a connection handle which the server uses to keep track of multiple web clients. As our DLL interacts with the server to process the request (for example, to send the output back to the client), we'll be passing the `ConnID` into various server callback functions so the server knows which of its clients to interact with. You should never change the value of `ConnID` in your DLL or the server will lose track of the requests.

The `lpszMethod` field points to a string of either `GET` or `POST` depending on how the application was called in the HTML page. As you may recall from Issue 16, this mimicks the behavior of the CGI variable `REQUEST_METHOD`. Likewise, the `lpszQueryString` field parallels the CGI variable `QUERY_STRING` and contains the URL-encoded user data from the client if the request method is `GET`. If the request

method is POST, the user data is found in the buffer pointed to by lpbData. Note that lpbData does not point to a null-terminated string, but rather a series of bytes. The number of bytes found in this buffer is given in the cbAvailable field.

Server Callback Functions

As you can see, the ECB provides essentially the same information that we had to work with in CGI – we just get it from a different place. In addition, the ECB defines function pointers to four callback functions implemented within the web server software: GetServerVariable, WriteClient, ReadClient, and ServerSupportFunction. The prototypes for each of these functions are shown in Listing 1. These functions allow us to interact directly with the web server to get more information and perform the actions that will ultimately affect what the user sees in their browser.

You may have noticed that not all of the session information we have available to us with CGI is represented by fields in the ECB. Is ISAPI shortchanging us? Not at all. All the standard CGI variables can be accessed via the GetServerVariable callback function. By passing in the name of the variable we are interested in and, as with all the server callbacks, the ConnID from the ECB, we get back a pointer to a buffer containing the value of that variable.

We need the ReadClient function for extremely large volumes of input data. The lpbData buffer only contains a maximum of 48Kb of user data. In the rare event that more than this amount is inbound from the client browser, lpbData contains the first 48Kb worth and the rest is retrieved through the ReadClient function. We can determine when it is necessary to use ReadClient because cbTotalBytes will be greater than cbAvailable (which will equal 48Kb).

To use ReadClient we just create a buffer to receive the data and pass a pointer to the buffer and its size into ReadClient. ReadClient writes the additional client data into our buffer and changes the Size parameter to indicate how much data it wrote. We simply keep calling ReadClient until it tells us there is no more data to read (see Listing 2).

Returning A Response Page

Most ISAPI apps will need to create an HTML page to send back to the browser. With CGI, we simply wrote strings to the standard output device (standard CGI) or to a designated temporary file (WinCGI). With ISAPI, we simply create a buffer containing the HTML-formatted text we want to send back and call the WriteClient function. We pass into it a pointer to our buffer of output text, the size of the buffer, and a flag value of 1, indicating synchronous I/O. Synchronous I/O basically means that the callback function won't return until the server has sent all of the output data back to the client browser (we won't be covering asynchronous I/O in this issue).

➤ *Listing 2*

```
var
  Buffer:  array[0..1023] of Char;
  Result: BOOL;
  BufSize: Integer;
with ECB do begin
  if cbTotalBytes > cbAvailable then begin
    repeat
      BufSize := SizeOf(Buffer);
      Result := ReadClient(ConnID, @Buffer, BufSize);
      { process data }
    until not Result or (BufSize = 0);
  end;
end;
```

➤ *Listing 1*

```
{ Abbreviated listing of ISAPI.PAS provided with Delphi }
unit isapi;
interface
uses Windows;
const
  HSE_VERSION_MAJOR     =   1;  // major version of this spec
  HSE_VERSION_MINOR     =   0;  // minor version of this spec
  HSE_LOG_BUFFER_LEN    =  80;
  HSE_MAX_EXT_DLL_NAME_LEN  = 256;
type
  HCONN = THandle;
// the following are the status codes returned by the Extension DLL
const
  HSE_STATUS_SUCCESS                      = 1;
  HSE_STATUS_SUCCESS_AND_KEEP_CONN        = 2;
  HSE_STATUS_PENDING                      = 3;
  HSE_STATUS_ERROR                        = 4;
// passed to GetExtensionVersion
type
  PHSE_VERSION_INFO = ^THSE_VERSION_INFO;
  THSE_VERSION_INFO = packed record
    dwExtensionVersion: DWORD;
    lpszExtensionDesc:
      array [0..HSE_MAX_EXT_DLL_NAME_LEN-1] of Char;
  end;
type
  TGetServerVariableProc = function ( hConn: HCONN;
    VariableName: PChar; Buffer: Pointer; var Size: DWORD ):
    BOOL stdcall;
  TWriteClientProc = function ( ConnID: HCONN; Buffer:
    Pointer; var Bytes: DWORD; dwReserved: DWORD ):
    BOOL stdcall;
  TReadClientProc  = function ( ConnID: HCONN; Buffer:
    Pointer; var Size: DWORD ): BOOL stdcall;
  TServerSupportFunctionProc = function ( hConn: HCONN;
    HSERRequest: DWORD; Buffer: Pointer; var Size: DWORD;
    var DataType: DWORD ): BOOL stdcall;
// passed to extension procedure on a new request
type
  PEXTENSION_CONTROL_BLOCK = ^TEXTENSION_CONTROL_BLOCK;
  TEXTENSION_CONTROL_BLOCK = packed record
    cbSize: DWORD;              // size of this struct.
    dwVersion: DWORD;           // version info of this spec
    ConnID: HCONN;      // Context number not to be modified!
    dwHttpStatusCode: DWORD;          // HTTP Status code
    //null terminated log info specific to this Extension DLL
    lpszLogData: array [0..HSE_LOG_BUFFER_LEN-1] of Char;
    lpszMethod: PChar;                // REQUEST_METHOD
    lpszQueryString: PChar;           // QUERY_STRING
    lpszPathInfo: PChar;              // PATH_INFO
    lpszPathTranslated: PChar;        // PATH_TRANSLATED
    cbTotalBytes: DWORD;  //Total bytes indicated from client
    cbAvailable: DWORD;          // Available number of bytes
    lpbData: Pointer;         // pointer to cbAvailable bytes
    lpszContentType: PChar;   // Content type of client data
    GetServerVariable: TGetServerVariableProc;
    WriteClient: TWriteClientProc;
    ReadClient: TReadClientProc;
    ServerSupportFunction: TServerSupportFunctionProc;
  end;
// these are the prototypes that must be exported
// from the extension DLL
//  function GetExtensionVersion(var Ver:
//    THSE_VERSION_INFO): BOOL; stdcall;
//  function HttpExtensionProc(var ECB:
//    TEXTENSION_CONTROL_BLOCK): DWORD; stdcall;
implementation
end.
```

Putting It All Together

Let's demonstrate all we've learned through an example. Listing 3 shows the HTML for the test query page shown in Figure 1. This page calls our application by two means: by hyperlink and by data entry form. You should recognize the data entry form approach from the discussions of CGI in past issues. The only difference here is that our `ACTION` parameter refers to a DLL rather than an EXE.

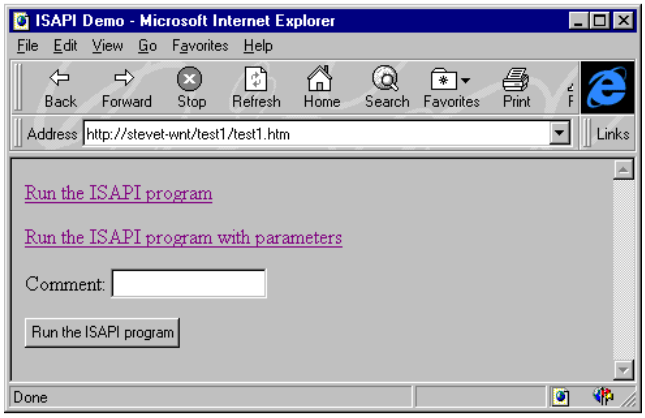
The hyperlinks on this page show an alternative way of calling an ISAPI app directly without using an HTML data entry form. Rather than link to an HTML page, you link to the program to execute (you can do this with CGI as well). In addition, you can pass data into the program simply by including it after the program name separated by a question mark. Any data after the question mark must be URL encoded. For example, spaces must be coded as plus signs. This type of request is translated as a `GET` and all the data after the question mark appears in the `lpszQueryString` field of the ECB (the `QUERY STRING` variable in CGI).

➤ *Listing 3*

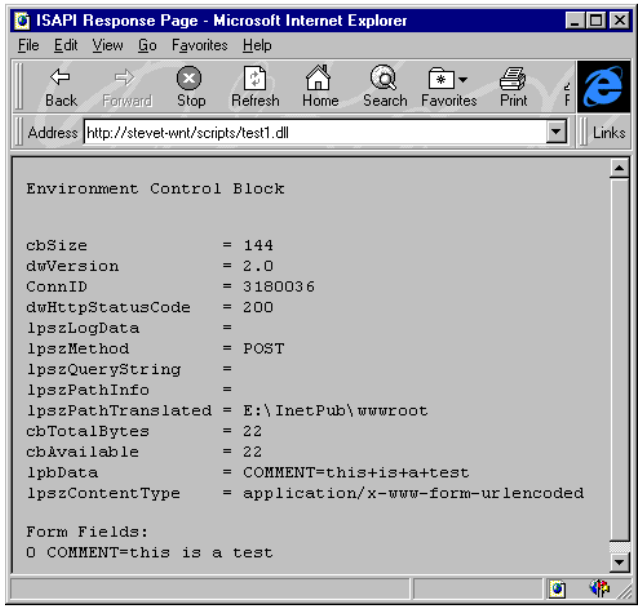
```
<HTML>
<HEAD><TITLE>ISAPI Demo</TITLE></HEAD>
<BODY>
<A HREF="http://stevet-wnt/scripts/test1.dll">Run the ISAPI program</A>
<BR>
<BR>
<A HREF="http://stevet-wnt/scripts/test1.dll?PARAM=Hi+There">Run the ISAPI
program with parameters</A>
<BR>
<BR>
<FORM ACTION="http://stevet-wnt/scripts/test1.dll" METHOD="POST">
Comment: <INPUT TYPE="TEXT" NAME="COMMENT"><BR><BR>
<INPUT TYPE="SUBMIT" VALUE="Run the ISAPI program">
</FORM>
</BODY>
</HTML>
```

➤ *Listing 4*

```
library Test1;
uses SysUtils, Classes, Windows, Isapi;
const
  HSE_IO_SYNC      = 1;
  HSE_IO_ASYNC     = 2;
function GetExtensionVersion(Ver: THSE_VERSION_INFO):
  BOOL; stdcall;
begin
  Ver.dwExtensionVersion :=
    MakeLong(HSE_VERSION_MINOR, HSE_VERSION_MAJOR);
  StrLCopy(Ver.lpszExtensionDesc,
    'Internet Server Application, Example #1',
    HSE_MAX_EXT_DLL_NAME_LEN);
  Result := True;
end;
function HttpExtensionProc(
  var ECB: TEXTENSION_CONTROL_BLOCK): DWORD; stdcall;
  procedure UnpackURLString(S: PChar; List: TStringList);
  { Parses and decodes a URL-encoded string. Copies variable
    values into List. See details in Issue #16. }
  var LabelStr, ValueStr: ShortString;
  begin
    LabelStr := '';
    ValueStr := '';
    while S^ <> #0 do begin
      case S^ of
        '+' : ValueStr := ValueStr + ' ';
        '%' : begin
                ValueStr := ValueStr + Chr(StrToInt('$' +
                  (S + 1)^ + (S + 2)^));
                Inc(S, 2);
              end;
        '=' : if LabelStr = '' then begin
                LabelStr := ValueStr;
                ValueStr := '';
              end;
        '&' : begin
                List.Values[LabelStr] := ValueStr;
                ValueStr := '';
                LabelStr := '';
              end;
        else ValueStr := ValueStr + S^;
      end;
      Inc(S);
    end;
    if ValueStr <> '' then
      List.Values[LabelStr] := ValueStr;
  end;
  function ISAWriteLn(Msg: string): Boolean;
  { Encapsulate the WriteClient callback into something
    more manageable. }
  var NBytes: DWORD;
      Buffer: PChar;
  begin
    Buffer := StrAlloc(Length(Msg) + 3);
    try
      StrPCopy(Buffer, Msg);
      StrCat(Buffer, #13#10);
      nBytes := StrLen(Buffer);
      Result := ECB.WriteClient(ECB.ConnID,
        Buffer, NBytes, HSE_IO_SYNC);
    finally
      StrDispose(Buffer);
    end;
  end;
var FormFields: TStringList;
    I: Integer;
    PostData: PChar;
begin
  FormFields := TStringList.Create;
  try
    with ECB do begin
      if StrPas(lpszMethod) = 'GET' then
        UnpackURLString(lpszQueryString, FormFields)
      else begin
        if Assigned(ECB.lpbData) then begin
          PostData := StrAlloc(cbAvailable + 1);
          StrMove(PostData, ECB.lpbData, cbAvailable);
          UnpackURLString(PostData, FormFields);
        end;
      end;
      ISAWriteLn('<HTML><HEAD>');
      ISAWriteLn('<TITLE>ISAPI Response Page</TITLE>');
      ISAWriteLn('</HEAD><BODY>');
      ISAWriteLn('<PRE>Environment Control Block');
      ISAWriteLn('<BR>');
      ISAWriteLn('cbSize             = ' + IntToStr(cbSize));
      ISAWriteLn('dwVersion          = ' +
        IntToStr(dwVersion shr 16) + '.' +
        IntToStr(dwVersion and $FFFF));
      ISAWriteLn('ConnID             = ' + IntToStr(ConnID));
      ISAWriteLn('dwHttpStatusCode   = ' +
        IntToStr(dwHttpStatusCode));
      ISAWriteLn('lpszLogData        = ' + lpszLogData);
      ISAWriteLn('lpszMethod         = ' +
        StrPas(lpszMethod));
      ISAWriteLn('lpszQueryString    = ' +
        StrPas(lpszQueryString));
      ISAWriteLn('lpszPathInfo       = ' +
        StrPas(lpszPathInfo));
      ISAWriteLn('lpszPathTranslated = ' +
        StrPas(lpszPathTranslated));
      ISAWriteLn('cbTotalBytes       = ' +
        IntToStr(cbTotalBytes));
      ISAWriteLn('cbAvailable        = ' +
        IntToStr(cbAvailable));
      if not Assigned(lpbData) then
        ISAWriteLn('lpbData            = nil')
      else
        ISAWriteLn('lpbData            = ' +
          StrPas(PostData));
      ISAWriteLn('lpszContentType    = ' +
          StrPas(lpszContentType));
      ISAWriteLn('');
      ISAWriteLn('Form Fields:');
      for I := 0 to FormFields.Count - 1 do
        ISAWriteLn(IntToStr(I) + ' ' + FormFields[I]);
      ISAWriteln('</PRE>');
      ISAWriteLn('</BODY></HTML>');
    end;
  finally
    FormFields.Free;
    StrDispose(PostData);
    Result := HSE_STATUS_SUCCESS;
  end;
end;
exports
  GetExtensionVersion,
  HttpExtensionProc;
begin
end.
```



➤ *Above: Figure 1*



➤ *Right: Figure 2*

All our application is going to do is reflect back to us the contents of the ECB, as shown in Figure 2. The ISAPI DLL which accomplishes this is shown in Listing 4. We've borrowed the URL decoding algorithm from our CGI work in Issue 16.

Conclusion

Although they require a bit more care to develop, ISAPI applications outperform CGI for high-volume data processing on a web server. With the material presented here, you should easily be able to convert any CGI application into an ISAPI DLL. You can get more detailed information on ISAPI at Microsoft's web site:

```
http://www.microsoft.com/win32dev/apiext/isapimrg.html
```

You may also care to visit the ISAPI developers site at:

```
http://www.genusa.com/isapi/
```

(thanks to John Steventon for the info on this site).

Steve Troxell is a Senior Software Engineer with TurboPower Software. He can be reached by email at stevet@tpower.com or on CompuServe at 74071,2207