

USING CGI WITH LABVIEW

The secret of achievement is not to let what you're doing get to you before you get to it.
—Lloyd Cory

Overview of CGI

■ What Is CGI?

When you're surfing the Web, you probably come across sites that make you wonder, "How did they do that?" These documents could consist of, among other things, forms that ask for feedback or registration information, imagemaps that allow you to click on various parts of the image, counters that display the number of users that accessed the document, and utilities that allow you to search databases for particular information. In most cases, you'll find that these effects were achieved using the Common Gateway Interface, commonly known as CGI. You can put the power of CGI to work for LabVIEW as well.

CGI is one of the "older" Web technologies, but still widely used and often very convenient to implement for an interactive Web site. Although several other technologies now exist for interactive sites (some of which were described in Chapter 8,

Advanced Web Technologies: An Overview), CGI was the first such standard for the Web that allowed a user to call external routines remotely in a platform-independent manner.

CGI is a standard specification that lets HTTP servers run other programs on the server machine. It provides a mechanism for passing parameters to those programs and sending their output (usually an HTML page) back to the Web browser. CGI is useful anytime a user attribute or input is used in determining what content the Web server will send to the user. For example, using CGI you could design an e-commerce application that processes merchandise orders through a Web browser. The CGI application would take the user's credit card information, send the data to an external credit card verification program, and then post the approval and shipping information back to the Web browser, confirming the transaction. The end user never "sees" the external credit card verification program, but can still send it data and get results back through a standard Web browser. In some senses, a CGI application is just like a function: It takes input and produces output. By having the ability to call external routines through CGI, you can build powerful, platform-independent client-server Web applications, since now you are not limited to HTTP commands. (See Figure 9-1.)

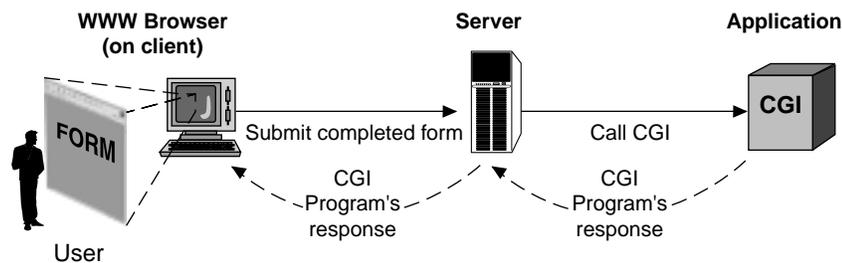


Figure 9-1

The CGI specification does not define what language can be used for the external programs. In fact, CGI does not care at all what the external programs look like; it only defines the interface between the HTTP server and the external code. You can write code to handle CGI applications in C, C++, Visual Basic, Perl, and . . . LabVIEW, of course! Most commercial-application CGI programs are written in Perl or C. Perl is an interpreted, text-based language that is freeware and very popular with Web developers. Perl is well

sued for handling and parsing strings in ways that would be much more complicated to do with other languages. You can find more information about Perl at <http://www.perl.org>. But, of course, in this book, we will show you how to write CGI applications with LabVIEW.

It's fair to say that CGI is much more complicated than the Web technologies we've seen so far. An in-depth treatment on writing CGI applications requires another whole book in itself (check C, *References*, for some suggestions). However, you don't need to become an expert in CGI and HTTP to build simple applications, especially with LabVIEW. This chapter will show you how to get started in building an interactive Web system with LabVIEW—just be aware that it does not cover everything about CGI.

■ Why CGI?

So far, the LabVIEW-Web technologies we have examined have all been restricted to *monitoring* a VI. That works fine if all you need is to retrieve information from LabVIEW, but what if you need to send data back to LabVIEW without leaving the Web browser? With CGI, you will be able to create applications where you can *control*, as well as monitor, your VI or LabVIEW itself through a Web browser interface. CGI is one of three choices for Web-based control of VIs (virtual instruments); in later chapters, we will examine how you can also use Java and ActiveX technologies (see Figure 9-2).



The Internet Toolkit for LabVIEW provides a complete set of CGI VIs that can be used to create CGI applications (see Figure 9-3).

With these **CGI VIs**, you can do things such as:

- Allow a user to load and run a VI dynamically from a Web browser
- Fill out an HTML form that will feed the inputs to VI, and publish the results back to the Web browser
- Set up security schemes for access to VIs (user authorization and authentication)
- Have LabVIEW create dynamic HTML pages on the fly (e.g., a Web counter)

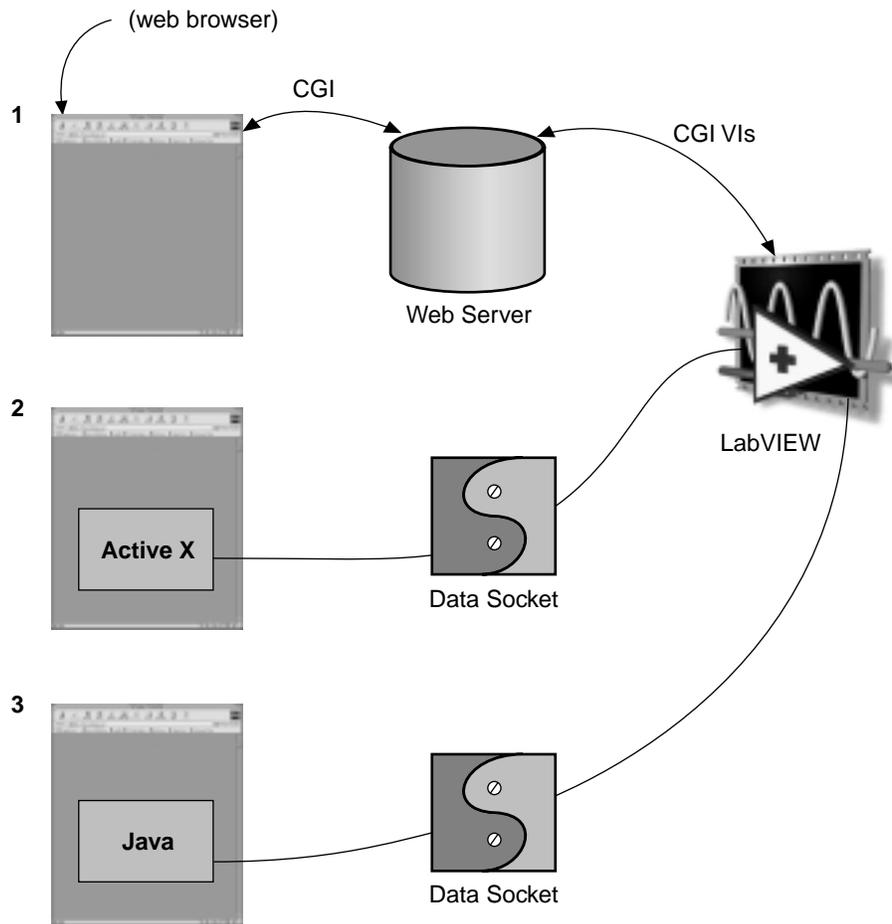


Figure 9-2

The three methods for Web control of LabVIEW

- Use imagemaps to let users click on different parts of a VI's panel image, simulating the behavior of a VI as if it were inside the Web browser
- Publish static and animated images of a VI
- Keep state information about a users who have visited the server by using cookies

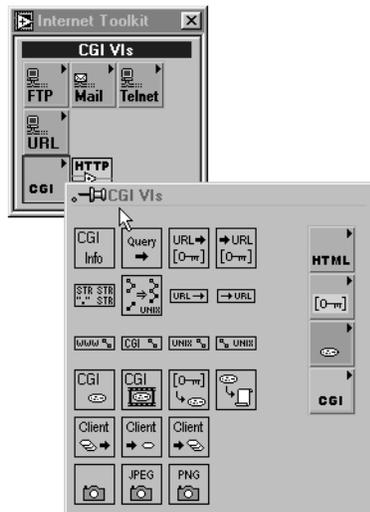


Figure 9-3
The CGI VIs palette

Some More Background on CGI

■ How CGI Works

First, let's understand how CGI works in general; and second, we will see how it is specifically implemented in LabVIEW with the Internet Toolkit.

When you provide a URL to a Web server, requesting a static document, the server simply returns the document's contents. However, when a user wants to provide data *to* the Web server, he or she must be able to enter or click some kind of data on the Web browser. The most common way of entering data on a Web page is through an **HTML form**. An HTML form is simply an HTML document that contains special HTML tags that, when displayed in the browser, show up as textboxes, checkboxes, radio buttons, buttons, selection boxes, and so forth, that a user can fill out. Users can enter data in other ways, such as clicking on an **imagemap** or even clicking on a link; we'll talk about these scenarios as well.

When a Web browser asks for a URL that points to a CGI application (such as submitting an HTML form), three steps occur:

1. **The Web browser sends an HTTP request to the Web server.** This request includes the name and location of the CGI application, and all the input parameters to the application.
2. **The HTTP server then calls the external CGI application, passing it the input parameters.** The server also makes *environment variables* available to the program. HTTP environment variables are part of the CGI specification, and contain data like the time and date of the request, the browser version, the IP address of the client, and so on.
3. **When the CGI application finishes executing, it returns its output to the Web browser.** This output is treated by the Web browser like any other HTML document; the browser cannot tell the difference between this generated output and static data.



Many Web servers let you specify a special directory where the CGI application program should reside. This directory is often called "cgi-bin" (for CGI binaries, or executables). In reality, it doesn't matter too much where the CGI application resides. On the Internet Toolkit, the HTTP server installation places a directory called "cgi-bin" in the Web root directory. So, any VI that acts as your CGI application should normally reside inside this directory.

Although CGI is a general specification, the implementation details are specific to the HTTP server you are using. That is, you must understand how your Web server works and how to configure it for CGI programs, where to place them, and so on. So, if you need to call LabVIEW VIs through CGI, you should use the Internet Toolkit's HTTP server since it readily handles CGI VIs; using another Web server instead would be tricky, to say the least. (Conversely, if you decide to write your CGI apps in C or Perl, you would not want to use the G Web server).

■ Important Terms and Concepts You Need to Know to Use CGI

At the very beginning, learning CGI is sometimes a challenge because it's hard to talk about a concept without making reference to another concept



*You should install and test that your Internet Toolkit Web server is working properly, since all of the CGI examples in this chapter require that you are running the Internet Toolkit HTTP server. Refer to your documentation for help. This also means you should **not** be running the built-in Web server in LabVIEW at the same time. When you run the Internet Toolkit's HTTP server, make sure the LabVIEW Web server is disabled (in the **Edit>>Preferences--Web Server:Configuration** dialog). Otherwise, the Internet Toolkit HTTP server and the examples will not work correctly.*

that has yet to be defined. In this section, I'll briefly go over some of the important terms and concepts that will be the background for building a CGI application. Don't worry if all of it doesn't make sense at first; after reading this section, study an example and then come back for references. Or, feel free to skip ahead to the CGI processing forms example, and come back to here later.

We'll take a look at:

- Parameter strings
- Keyed arrays
- Environment variables
- Cookies

Then we'll spend some time looking at the interfaces normally used on the Web browser for CGI:

- Forms
- Imagemaps

Parameter Strings

A CGI application starts with an HTTP **request** from the browser, and usually ends with an HTTP **response** from the server. When the browser makes an HTTP request, it sends several pieces of data to the server. In particular, the server receives a set of **parameters**. Each parameter has a **variable** and a **value** associated with it. For example, a CGI application may be expecting as input three parameters: the name of a person, his or her age, and his or her sex.

Table 9–1 CGI Parameters

Variable	Value
Name	John Smith
Age	25
Sex	Male

The parameters are passed to the CGI as one whole string in the HTTP request, in a format known as a **URL-encoded parameter** string, or just parameter string for short. A parameter string is a list of ampersand (&)-separated `variable = value` parameter pairs with proper URL encoding (see Chapter 6, *How the World Wide Web Works*, for information on URL encoding); for example (remember “%20” means a space):

```
Name=John%20Smith&Age=25&Sex=Male
```

Sometimes the parameters are also referred to as **form data** or **form parameters**, since they often come from an HTML form that a user fills out.

How do the form parameters get sent from the Web browser to the Web server? One of the most common ways, known as the GET method, is to place the parameter string in the **query string** section of a URL. Remember that a URL can optionally have a “?” character after its host name and path. The string that immediately follows the “?” character is the **query string**. For example:

```
http://my.lv.server/cgi-bin/data.vi?  
Name=John%20Smith&Age=25&Sex=Male
```

Notice that this URL points to a CGI application (“data.vi”) and includes a query string that has all the parameters we’ve been exemplifying. Obviously, it would be impractical for users to know and remember the correct syntax and type this URL directly into a Web browser— this is almost never done; we’ll see how HTML forms can generate the query strings automatically.



A query string in a URL does not necessarily mean a CGI application is being called. The query string is more generic: it is an optional component of a URL. For example, consider how the G Web server parses query strings that start with “?.monitor” in a special way to create a front panel image, without any CGI being involved. However, CGI applications do often make heavy use of query strings.

Activity 9.1

To see how query strings work, and how a CGI application can run them, do the following:

1. Make sure the Internet Toolkit's Web server is running and configured in its default way (i.e., the root Web directory points to the `/internet/home` in the LabVIEW directory so that you can view the ITK's online examples).
2. From your Web browser, open

```
http://127.0.0.1/ examples/get_sgl.htm
```

and

```
http://127.0.0.1/ examples/get_mlt.htm
```

which will take you to some CGI Query examples as part of the Internet Toolkit.

3. Try the three links and observe the URL textbox in your browser and the results displayed, as shown in Figure 9-4.

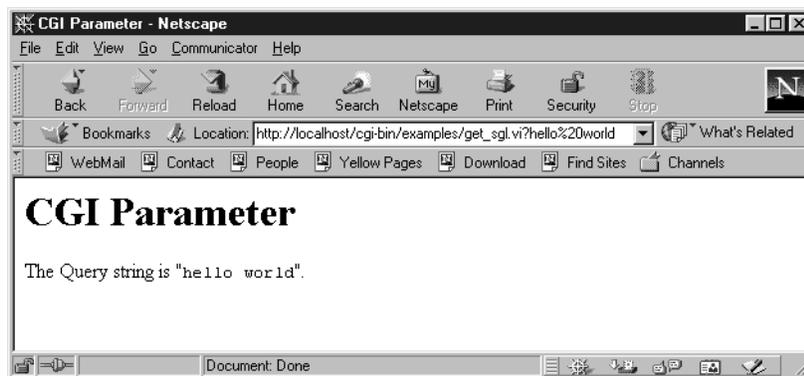


Figure 9-4

Keyed Arrays

A **keyed array** is a special data structure that isn't very common to languages like LabVIEW. Keyed arrays contains key-value pairs*. An element of the array is indexed by its key (a string type), rather than a numeric index. For example, in a "normal" array of strings:

```
a = ["Polo", "Columbus", "Armstrong"]
```

you reference the values of the array by the numeric index. For example:

```
a[0] = "Polo"
a[1] = "Columbus" ...
```

In a keyed array, you instead assign a string **key** to reference the **value** of every element. For example:

```
k = ["Marco", "Polo", "Christopher", "Columbus", "Neil", "Armstrong"]
```

so that you can reference a value by its key:

```
k["Marco"] = "Polo"
k["Christopher"] = "Columbus"
```

Keyed arrays are very handy for dealing with lists of data, as they allow you to store data in an array and use meaningful keys to be associated with their values. Keyed arrays are frequently used in CGI programs, as a method of organizing and working with form parameters. For example, the query string we looked at earlier could be represented on the server program as a keyed array.

Table 9-2

Key	Value
Name	John Smith
Age	25
Sex	Male

In the CGI palette, you can find a whole subpalette for working with keyed arrays in **Internet Toolkit>>CGI VIs>>Keyed Array VIs**. The keyed arrays

* In Perl, they are called **associative arrays**.



The concept of an “order” for keyed arrays is not relevant, like it is for normal arrays. In particular, never rely on your key-value pairs being sorted in a certain order within a keyed array, as the application software will arrange them internally to its liking. You can only access the array in terms of its keys and values, and not in an auto-indexing manner.

are represented internally as an array of clusters; each cluster contains a “key” and a “value” string. There are functions for adding, retrieving, and deleting values from a keyed array, comparing two keyed arrays for equality, and others. Refer to the online documentation for the specifics of how each function works (see Figure 9–5).

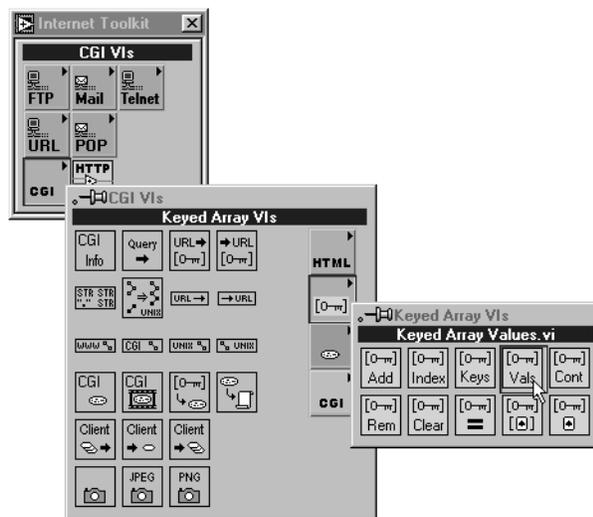


Figure 9–5
Keyed array VIs in the Internet Toolkit

CGI Environment Variables

CGI environment variables are simply global variables describing the HTTP environment and CGI session that the server program can access, such as the client’s IP address, the server name, the server software version, the query

string, and so forth. Every server provides environment variables, and the Internet Toolkit Server is no exception. In the Internet Toolkit, the environment variables are stored in an array called “**env**”.

When a CGI application is called in LabVIEW, the server passes it environmental information in the **env** array. This array contains information about the server application, the browser application, server and browser addresses, protocol version, and so forth. The **env** array also contains query information, if any, that was sent with the request. Table 9–3 is the list of CGI environment variables your VIs can access.

Table 9–3

<i>Environment Variable</i>	<i>Description</i>
GATEWAY_INTERFACE	Version of the interface, currently CGI/1.1.
SERVER_SOFTWARE	Name and version of the G Web Server.
SERVER_NAME	Name of the computer running the G Web Server as configured or determined by the server.
SERVER_PORT	TCP port at which the server listens for requests.
DOCUMENT_ROOT	Root directory, in Unix format, of your server documents.
REMOTE_HOST	Domain name or IP address of the remote system the client uses to connect.
REMOTE_ADDR	IP address of the remote system the client connects from.
SCRIPT_NAME	Virtual path to the CGI VI.
REQUEST_METHOD	Method by which you invoke the CGI, either GET or POST.
SERVER_PROTOCOL	Protocol over which the client communicates with the server, currently HTTP/1.0.
HTTP_REFERER	URL of document that contains the link that invoked this CGI.
HTTP_USER_AGENT	Browser software the remote client uses.
HTTP_ACCEPT	List of MIME-like types that the browser understands.
QUERY_STRING	URL-encoded parameters sent to this CGI.
REMOTE_USER	Username of client.
REMOTE_IDENT	User password of client.

Activity 9.2

Here you'll examine the values of the environment variables, by running the online examples in the Internet Toolkit. With the Internet Toolkit's Web server running as in the previous exercise, point your browser to

```
http://127.0.0.1/examples/environ.htm
```

and try the different links. As shown in Figure 9–6, you'll see what the values of the `env` variable look like.

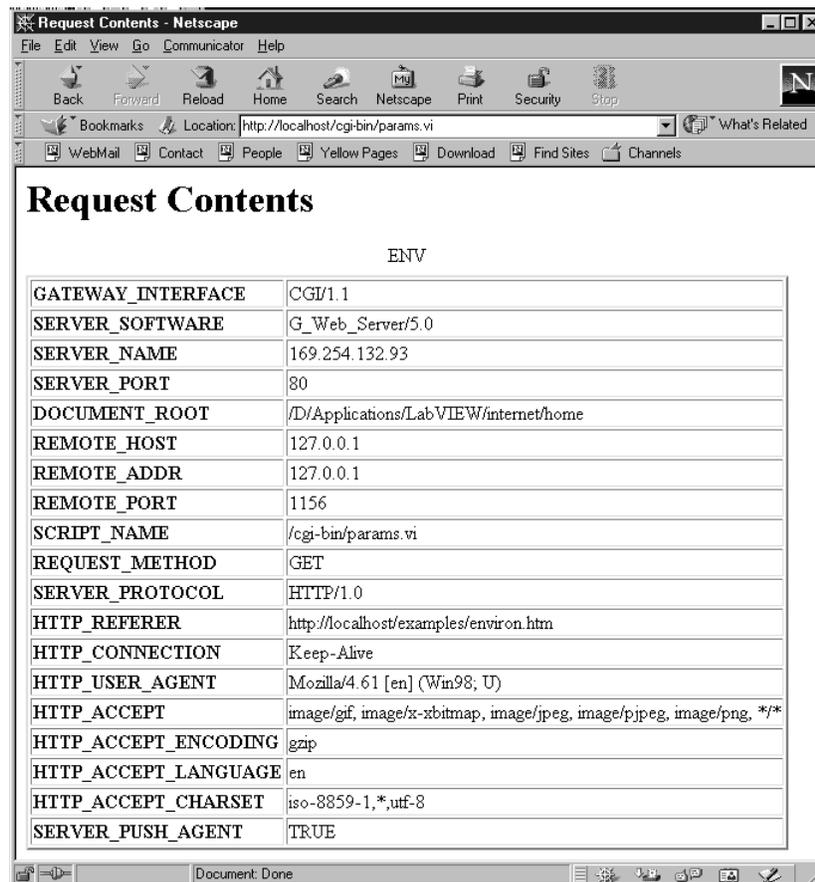


Figure 9–6

Cookies

One of the complications for some CGI programs is how to maintain “state”; that is, how to remember what parameters had what values as a user navigates through a site. This problem arises because HTTP is a **stateless** protocol. Each time a client wants a document from a server, the client must establish a new connection and send a request. The server receives the request, returns a reply, and closes the connection. The server does not maintain state information between individual connections. The HTTP server has no idea if you just filled out a form and are visiting the site for the 14th time, or if it is your first visit. In addition, HTTP can’t “remember” anything if the page reloads or the user is directed to another page.

Often, it is useful to maintain state information across several connections. For example, imagine you are an online vendor. When a user browses through your online catalog, he or she can add items to a shopping cart. When the user finishes shopping, he or she can choose to purchase the items in the cart. You must maintain the information about the items until the user finishes the purchase. You can store information, such as what items the user chose, in a simple data file. However, this does not work if more than one user is shopping at a time. Because you want to work with multiple users, you must find an alternative information storage method. You can choose from several approaches to maintain client-state information across multiple accesses. For example, you can insert information you collect into hidden fields of an HTML form, or you can use the infamous **cookie**.

A **cookie** is a token that uniquely identifies some information. **Client-side cookies** allow you to store a small amount of data (a 4K limit imposed by the browser) on the client’s browser. **Server-side cookies** keep this same information on the server; however, server-side cookies automatically expire after a given time.

In the shopping cart example, you could use a cookie to record all the items the user has put into the shopping cart. Using a cookie, you can maintain your information on the client side or on the server side.

Not all browsers support client-side cookies, and some users don’t like using them and will disable them. In general, it’s best to avoid client-side cookies if you can, unless you want to be able to store persistent data on all your users’ browsers.



Cookies have probably gotten a lot more bad press than they deserve. Many people feel that they infringe on their privacy or present a security hole on a commercial site. While this can very occasionally be the case, most of the time, cookies are very small, harmless bits of data that are simply used to facilitate a CGI script.

With LabVIEW, you can use Internet Toolkit VIs to create, write to, and read from both client- and server-side cookies. The **server-side cookie** functions are in **Internet Toolkit>>CGI VIs>>Cookie VIs** (see Figure 9–7).

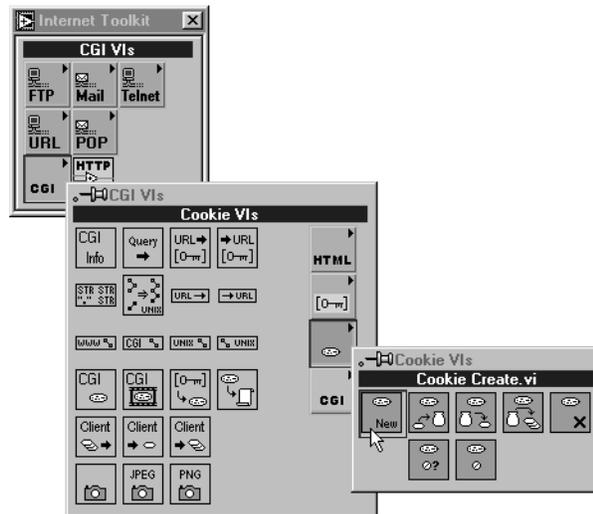


Figure 9–7
Cookie VIs palette

For each server-side cookie you create, you must add data entries as key-value pairs. The information in server-side cookies is stored in a keyed array.

Client-side cookies can be created, written to, and read from, using the VIs from **Internet Toolkit>>CGI VIs**, in the fourth and fifth rows (see Figure 9–8).

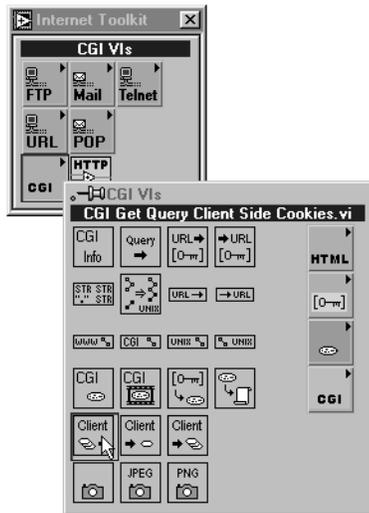


Figure 9-8
Client-side Cookie VIs

Activity 9.3

To see an example of maintaining state with server-side cookies, run the following Internet Toolkit example. With the ITK Web server running, point your browser to:

```
http://localhost/examples/ck_clnt.htm
```

and follow the example.

To see an example of maintaining state with client-side cookies, do the following:

1. In your browser, change your preferences to be “Warn me before accepting a cookie.” In Netscape 4.x, go to **Edit>>Preferences>>Advanced**. In Internet Explorer 5.x, go to **Tools>>Internet Options>>Security>>Custom Level...>>“Allow Cookies That Are Stored On Your Computer”** and choose “Prompt.”

You can revert these settings later, but this will allow you to see when the cookie is sent to the browser.

2. Point your browser to:

`http://localhost/examples/custom.htm`

and follow the example. (See Figure 9–9.)

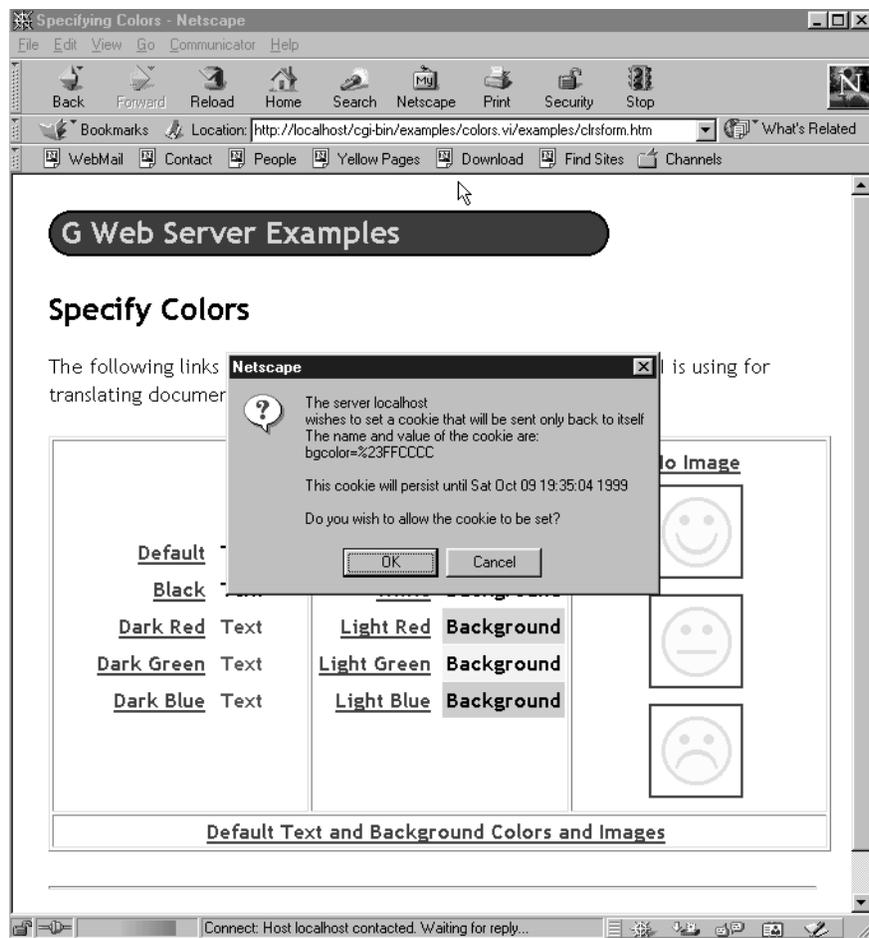


Figure 9–9

■ Forms and Imagemaps

Forms

HTML Forms are the most prevalent way to gather input for a CGI application. On the client side, forms are built exclusively with HTML and are comprised of text input boxes, radio buttons, checkboxes, pull-down menus, and clickable images, all nested inside a `<FORM>` tag. The CGI application on the server side doesn't know or care what the form looks like on the browser; all it needs is the form **parameters** to process the request. So, to work with forms, we'll need to expand on the knowledge of HTML tags we began with in Chapter 6.

First, let's look at an example the HTML tag that encloses a form, the `<FORM>` tag:

```
<form name="MyForm" action="/cgi-bin/process_form.vi" method="GET">  
...  
</form>
```

The form tag uses an optional **name** attribute, a required **action** attribute and a required **method** attribute:

- The **name** is any string name you want to give it.
- The **action** is a URL (absolute or relative) that tells the Web server what CGI program it should call when the form is submitted.
- The **method** specifies how the data is sent to the server. There are two methods: GET and POST.

When using LabVIEW CGIs, the "action" attribute will point to the top-level CGI VI you designed to handle the form. The form data can be passed to the CGI application by the GET method or the POST method. GET and POST refer to how data are passed in the HTTP session. With GET, the form parameters are automatically URL-encoded into the query string. With POST, the form parameters are passed in the header information of the HTTP request. To see the difference between GET and POST, do the following activity.

Activity 9.4

With the Internet Toolkit Web server running, open your browser to the following URL:

```
http://localhost/examples/post_mlt.htm
```

There are two forms on this page. One uses the GET method, and the other the POST method. Notice that the final result is exactly the same on the returned Web page; however, the GET method shows the URL-encoded query string in the browser's location bar.



So when should you use GET or POST? As a very general rule of thumb, your VIs will normally be easier to program if you use the GET method, since the CGI VIs have some utilities for easily parsing the query string that contains the parameters when the GET data is sent.

A situation when you should use POST instead, however, is if you are passing a password or other sensitive information to a CGI. The reason for this is that if you were to use GET, the password would be part of the URL, which may end up being stored in the cache or history of the browser. This leaves open a potential security hazard since someone could discover a password by examining the browser's history URLs.

Now let's create an actual form in HTML that you will later use.

Activity 9.5 Creating an HTML Form

You will create an HTML form in this activity. Do the following:

1. In a text editor, type the following HTML code. Pay special attention to the syntax.

```

<HTML>
<HEAD>
<TITLE>Calculator Form</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1 ALIGN=CENTER>A Web calculator</H1>
  <FORM METHOD="POST" ACTION="/cgi-bin/calculator.vi">
    <INPUT TYPE="text" SIZE="8" VALUE="12.5" NAME="T1">
    <SELECT NAME="operation">
      <OPTION VALUE="+" SELECTED>+
      <OPTION VALUE="-">-
      <OPTION VALUE="*">*
      <OPTION VALUE="/">/
    </SELECT>
    <INPUT TYPE="text" SIZE="8" VALUE="3.2" NAME="T2">
    <P>
    Base Notation:<BR>
    <INPUT TYPE="radio" NAME="base" VALUE="Decimal"
      CHECKED>Decimal<BR>
    <INPUT TYPE="radio" NAME="base" VALUE="Hex">Hex<BR>
    <P>
    <INPUT TYPE="checkbox" NAME="errorcheck">Perform error
      checking.
    <P>
    <INPUT TYPE="hidden" NAME="Heh-heh" VALUE="This is our
      little secret">
    <P>
    <INPUT TYPE="submit" VALUE="Calculate Result">
    <BR>
    <INPUT TYPE="reset" >
  </FORM>
</BODY>
</HTML>

```

2. Save your document as **Calculator.html**.
3. With a Web browser, open this HTML document. It should look like Figure 9–10.
4. Play around with the input controls. Notice what each one does.

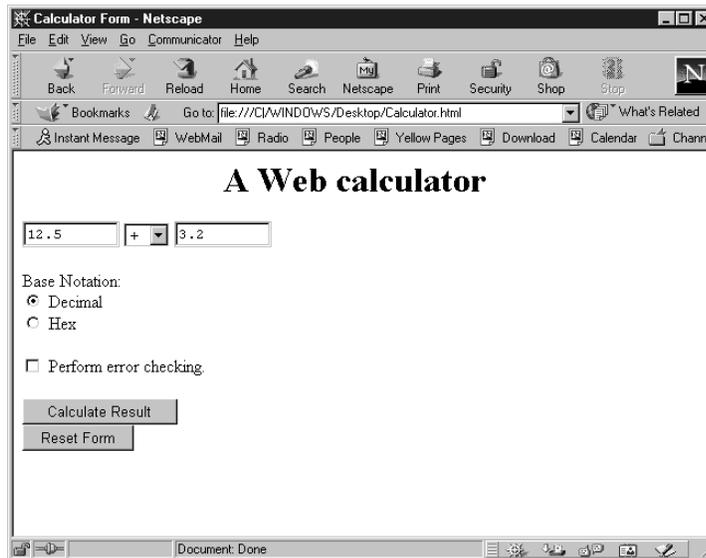


Figure 9-10

Notice several things about this form you just created:

- Each input element has a “name” attribute. This is very important, as it will be used by the CGI program.
- You’ll get an error if you press “Calculate Result”—this is because all we have made is our form interface, but we have not written the back-end to do any calculations.
- The Reset button sets everything back to the default value the page had when it was loaded.
- You didn’t need any server programming to create an HTML interface like this one.

Creating a form is like creating a VI’s front panel—it’s the interface, not the code. The “block diagram” of this HTML form will be a `calculator.vi`, which we will create later on in this chapter.

Table 9-4 lists some of the form elements you can create in HTML.

Table 9-4

Form Element	HTML Sample	Notes
Button	<code><INPUT TYPE="button" NAME=name VALUE=value></code>	A button. The <i>value</i> string is what shows up as the button's label.
Text-box (single-line)	<code><INPUT TYPE="text" NAME=name></code>	A single-line text entry box. Optional attributes for text inputs are: size=" <i>display_width</i> " maxlength=" <i>string_length</i> "
Password	<code><INPUT TYPE="password" NAME=name></code>	Just like the text input, but characters are hidden as bullets or asterisks when the user types.
Checkbox	<code><INPUT TYPE="checkbox" NAME=name [checked]></code>	A Boolean checkbox. "checked" is a standalone attribute that specifies the checkbox be checked by default.
Radio buttons	<code><INPUT TYPE="radio" NAME="r1"> Red <INPUT TYPE="radio" NAME="r1"> Yellow <INPUT TYPE="radio" NAME="r1" checked> Green ...</code>	Radio button set. A group of radio buttons that have mutually exclusive checked options must all have the same NAME attribute. The checked attribute also specifies which radio button is selected by default.
Hidden	<code><INPUT TYPE="hidden" NAME=name VALUE=value></code>	A hidden input does not show up on the Web browser. It is used to pass information to a CGI program as part of the form, and the value cannot be modified by the user.
Submit	<code><INPUT TYPE="submit" NAME=name VALUE=value></code>	This shows up as a button on the Web browser, but it has the effect of submitting the form to the URL specified in the ACTION attribute of the <FORM> tag.
Reset	<code><INPUT TYPE="reset" NAME=name VALUE=value></code>	This also shows up as a button. Clicking it has the effect of resetting all the form elements back to the default values in the page.

Besides the <INPUT> tags, there are also tags like <TEXTAREA> for multiline textboxes, and <SELECT> for drop-down combo boxes. The details of these form tags are fairly intricate and have many nuances, which precludes me from including a detailed reference for them here. Instead, I highly recommend a good reference such as *HTML: The Definitive Guide* (O'Reilly, 1999) if you want to effectively design and debug your own HTML forms.

Imagemaps

One of the more useful form inputs in CGI-LabVIEW applications is to use a clickable image. The image could be a snapshot of a VI's front panel, or it could be some other graphical interface. In any case, you can define an image to be clickable and send information to a CGI application in two ways:

1. An **image input**: <INPUT TYPE=image>. With the image type of <input> form element, you create a custom button, one that is a “clickable” image. It's a special button made out of your specified image that, when clicked by the user, tells the browser to submit the form to the server, and includes the **x,y coordinates** of the mouse pointer in the form's parameter list. Image buttons require a “src” attribute with the URL of the image file.
2. Define an image as an **imagemap**, and define a hyperlink for “hotspots” on different pieces of the images. An imagemap is created by defining arbitrary pieces of an image that are clickable, and associating a URL with clicking each piece. For example, you could create an imagemap of a world map, and define each hotspot to be a different country, allowing a user to click on a country and send the data back to the server.

The first method, using an image input, can be useful if you need (x,y) coordinate information about where a user clicked. The (x,y) coordinates are the number of pixels relative to the top-left corner of the image. When a user clicks on the image, the (x,y) coordinates are passed to the CGI application as parameters. This is the *only* information from the image that can be sent back to the server. It's then up to the server to interpret or decide what to do with the (x,y) coordinates.

The second method is probably more useful in most LabVIEW applications. With an imagemap, you can “chop up” an image like a puzzle and define what parameters get sent for by clicking on different pieces of an

image. There are two ways to create imagemaps, known as **server-side** and **client-side image maps**. The former, enabled by the `ismap` attribute for the `` tag, requires access to a server and related imagemap processing applications. The latter is created with the `usemap` attribute for the `` tag, along with corresponding `<map>` and `<area>` tags. We will only show an example of a client-side imagemap, since in general, they are easier to create and work with, and generally are more widely used than server-side image maps.

The following HTML code is an example of a client-side imagemap:

```
<!-- This HTML code defines the areas for the imagemap and the associated
links. Note how each link points to the same CGI VI, but provides different
parameters in the query string -->
<MAP Name="buttons">
  <AREA Shape="Rect" coords = "18,26,35,47"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Frequency&value=up">
  <AREA Shape="Rect" coords = "18,48,35,71"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Frequency&value=down">
  <AREA Shape="Rect" coords = "18,90,35,111"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Noise&value=up">
  <AREA Shape="Rect" coords = "18,112,35,133"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Noise&value=down">
  <AREA Shape="Rect" coords = "12,154,111,180"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Pause&value=switch">
  <AREA Shape="Rect" coords = "12,189,111,215"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Stop&value=switch">
</MAP>

<!-- This is the image itself. Note the "usemap" attribute that refers to
the map defined earlier-->

```

Figure 9–11 shows a screenshot of how the imagemap was created in Macromedia Dreamweaver’s Web editor.



Creating client-side imagemaps is one of the places where you really need a professional HTML editor like Macromedia Dreamweaver or Microsoft FrontPage. These editors allow you to “draw” the shapes on your images and will create the `<MAP>` tags automatically for you. Most freeware editors (e.g., FrontPage Express) don’t have support for graphically creating imagemaps.

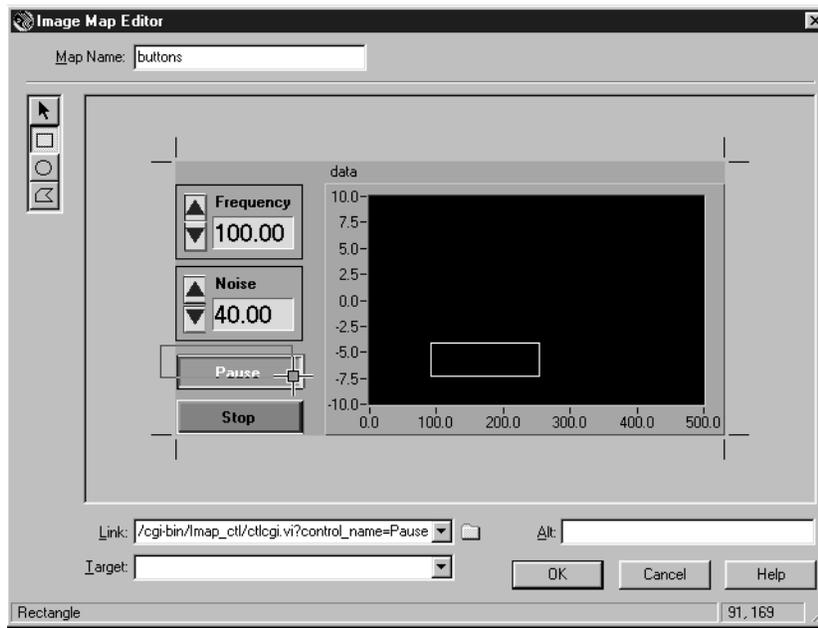


Figure 9-11

Using an HTML Editor for defining an imagemap

The best way to study imagemaps is to do the activity examples in the next section, where you will see how the preceding HTML code is created and used in conjunction with a VI to provide control over the Web.

**Requires
Internet
Toolkit**

LabVIEW and CGI Applications

Okay, you've just read more facts about CGI than the average person would want for breakfast. But the best way to understand CGI is by example. This section guides you step by step through some simple examples you can build on for your applications. All of the code is on the CD, of course. We will detail two HTML form examples and one imagemap example that use CGI VIs as the back-end.

■ Processing Forms in LabVIEW



Activity 9.6 The Simple Calculator CGI

We'll start with an example of a CGI application, written in LabVIEW, that can simulate a simple calculator. This CGI application will let the user input data through a Web browser, will perform the calculations in LabVIEW, and send the results back to the Web browser. It is similar to the HTML form you created in Activity 9.5, but with a simpler interface.

Start by just running the example, and then we'll go through it step by step to see how it works. To run this example, do the following:

1. Copy the files **calculator.vi** and **calculator.llb** from the CD to the CGI directory of your Internet Toolkit's HTTP server (this is usually the directory [*path to LabVIEW*]\LabVIEW\internet\home\cgi-bin\).
2. From LabVIEW, run the Internet Toolkit HTTP server (you must have the Internet Toolkit installed) by selecting the menu option **Project>>Internet Toolkit>>Start HTTP Server**. Make sure the built-in LabVIEW Web server or any other Web servers are not running at this time.
3. With your Web browser, open the URL that points to the VI **calculator.vi**. If you are opening the browser on your local machine, this should look like **http://localhost/cgi-bin/calculator.vi**. Make sure you are opening this via an http:// URL, not a File:///URL.
4. You should see the simple calculator as shown in Figure 9-12. Try out some operations and press the "=" button to see the results displayed.

Notice that the end user of this application never "sees" LabVIEW. In fact, there would be no clue at all what kind of program is behind this Web-based calculator, except that if you look closely, you'll see that the URL points to a VI instead of a static HTML file.

By using LabVIEW and the CGI capabilities of the Internet Toolkit, you effectively can put LabVIEW VIs to work and allow remote access without the user needing anything special beyond the Web browser. This example,

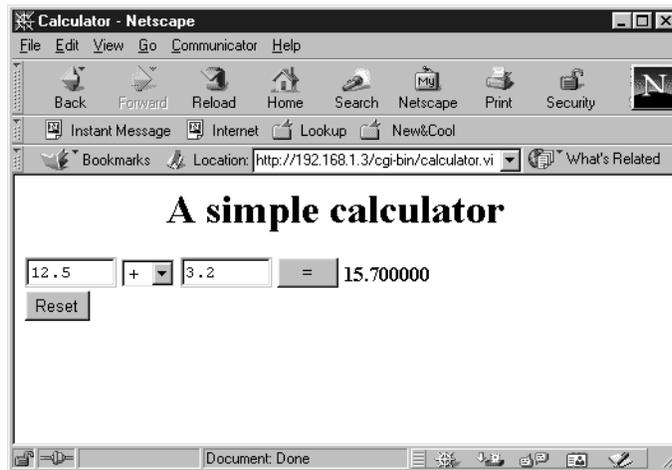


Figure 9-12
The Calculator.vi CGI application

although trivial, shows how a user can remotely run computations on LabVIEW without needing to install LabVIEW or even know how to use it.

Dissecting the Simple Calculator CGI Example

Now let's take a look "behind the scenes" to see how this CGI works.

Calculator CGI: Top-Level Calculator VI

Open the VI **calculator.vi** in LabVIEW and study the block diagram, shown in Figure 9-13.



Be sure that you copied this example VI to the cgi-bin directory of the ITK Web server, and you do not have another copy of this same VI open (e.g., from the CD). Otherwise the CGI call from the browser will fail because the LabVIEW will try to run the VI in memory instead of the VI in the correct server location.

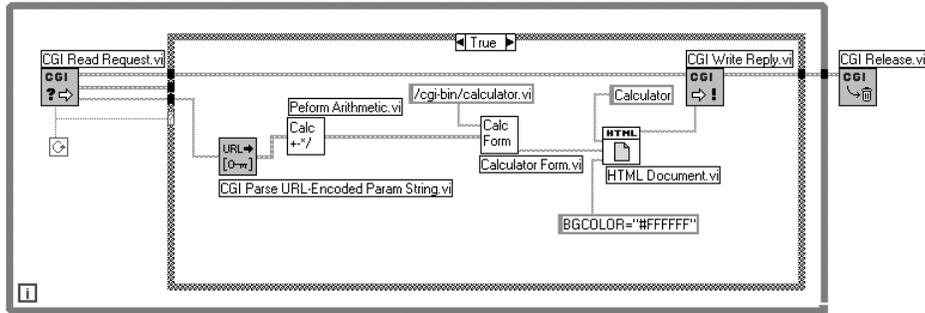


Figure 9-13
The calculator VI block diagram. The front panel has nothing on it.

The functions **CGI Read Request**, **CGI Write Reply**, and **CGI Release VI** are VIs available on the **CGI** palette of the Internet Toolkit. These VIs are always used as part of a CGI application. The functions **Perform Arithmetic.vi** and **Calculator Form.vi** are specific to this calculator example (they are in **calculator.llb**). (See Table 9-5.)

Table 9-5

<p>CGI Read Request.vi</p>	<p>Waits timeout seconds for a request before timing out. If a valid request arrives, valid request? is set to TRUE and the calling VI can use the content and the associated environment variables (env) to build an appropriate reply that must be returned with a call to the CGI Write Reply VI. The cgi connection info is a cluster that identifies a particular CGI session.</p>
<p>CGI Write Reply.vi</p>	<p>Writes a reply to the HTTP connections specified by cgi connection info. The content string is the data (usually HTML) to be sent back to the browser; the header info is optional.</p>
<p>CGI Release.vi</p>	<p>Informs the server that the CGI has finished processing requests and can be unloaded from memory. Call this VI when the CGI Read Request VI returns FALSE in its valid request? parameter.</p>



Since the VIs *CGI Read Request*, *CGI Write Reply*, and *CGI Release VI* are always required for a CGI application in LabVIEW, you can save time by using a CGI template. When building a new VI, use the *CGI Template.vit* from the CGI palette (see Figure 9–14).

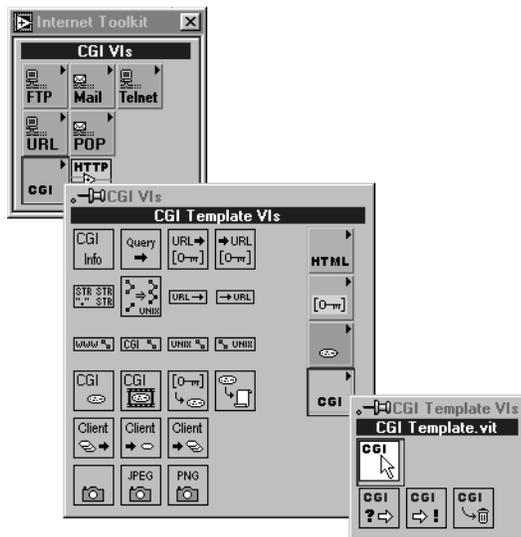


Figure 9–14
CGI Template.vit

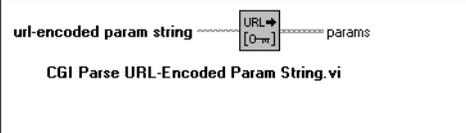
When the browser requests the URL that points to **calculator.vi**, the Internet Toolkit's HTTP server figures out that the client is requesting to run a CGI executable instead of static document. So, the server automatically runs the calculator VI. Once the VI runs, **CGI Read Request** loops repeatedly until it receives a valid and complete HTTP request. A valid HTTP request means that someone pressed the “=” button on the HTML form, sending the HTTP request through the browser. Once the valid request is received, the **cgi-connection info** cluster is passed out, as well as the **content** string. The **content** string contains all the parameters passed through the HTML form; that is, the numeric fields the user filled out and the value of the drop-down combo box (+, -, * or /). This information is processed by our subVI **Perform Arithmetic**, and then the new HTML data is created by **Calculator Form**. This HTML string is formatted by **Build HTML Document** and finally is

passed back to the browser with **CGI Write Reply**. Once the response is complete, we free the CGI resources by calling **CGI Release**.

Does this seem complicated? Hang in there, you're going to see how it works! As I mentioned before, CGI is more complex than the Web technologies we have seen so far because it requires understanding the interaction of HTML forms, HTTP, Web servers, and the programming language they are written in. However, I find that working through and understanding an example is the best way to start writing my own CGIs. So let's press on.

Returning back to the output of **CGI Read Request**: the content string is passed to a function **CGI Parse URL-Encoded Param String.vi**, which is available on the **CGI** palette. It converts the URL-encoded parameter string that is returned from the browser into a keyed array (see Table 9-6).

Table 9-6

	<p>Returns the contents of a URL-encoded parameter list string in a keyed array. You can use this VI to parse the contents of an HTML form (POST) request.</p>
--	--

Let's now get a picture of what is on the client side: the HTML source. If you select **View Source** on your Web browser from the calculator form, you should see something like the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Draft//EN">
<HTML>
<!-- Constructed with the G Web Server -->
<HEAD>
<TITLE>Calculator</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1 ALIGN=CENTER>A simple calculator</H1>
<FORM METHOD="POST" ACTION="/cgi-bin/calculator.vi">
<INPUT TYPE="text" SIZE="8" VALUE="12.5" NAME="T1">
<SELECT NAME="operation">
<OPTION VALUE="+" SELECTED>+
<OPTION VALUE="-">-
<OPTION VALUE="*">*
<OPTION VALUE="/">/
</SELECT>
<INPUT TYPE="text" SIZE="8" VALUE="3.2" NAME="T2">
<INPUT TYPE="submit" VALUE=" = ">
<B>15.700000</B><BR>
```

```
<INPUT TYPE="reset" >
</FORM>
</BODY>
</HTML>
```

Notice the `<FORM>` tag: It specifies that we use the `POST` method to pass the form parameters, and it specifies the CGI application to call when the form is submitted: `/cgi-bin/calculator.vi`. The action value must be a path relative to the Web root or relative to the HTML document.

Our form inputs consist of two textboxes, “T1” and “T2”, a combo-box “operation” with four valid selections (+, -, *, or /), a Submit button, and a Reset button. When we press the Submit button, the URL-encoded parameter string sent to the CGI application will look something like:

```
T1=12.5&operation=%2B&T2=3.2
```

The `%2B` is the URL encoding for the “+” character. After we convert this to a keyed array, the key-value pairs will be:

Table 9–7

Key	Value
T1	12.5
T2	3.2
Operation	+



Remember we discussed *URL-encoded strings* and *keyed arrays* earlier in the chapter; refer back to the definitions if you're confused about what these are.

Calculator CGI: Perform Arithmetic VI

Let's go back to LabVIEW. This keyed array of the form parameters is used by the subVI **Perform Arithmetic.vi**. Open this VI to see its block diagram (the VI is in **calculator.llb**) (see Figure 9–15).

The **params in** is the keyed array of the form parameters. The VI uses the **Keyed Array Index** function to get the values for “operation”, “T1”, and “T2”. It

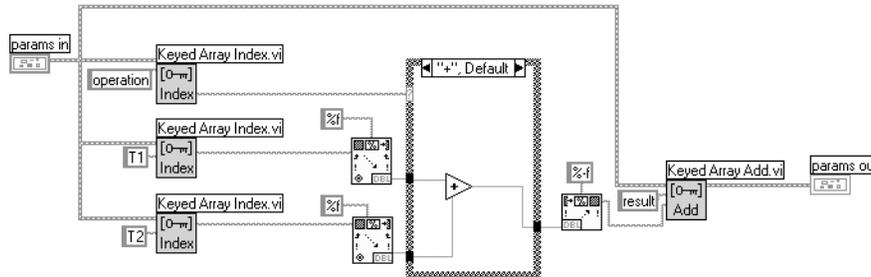


Figure 9-15
The Perform Arithmetic VI

does the string-to-number conversion, gets the appropriate arithmetic result, and adds the result as a new element (indexed by “result” key) using the **Keyed Array Add**. Remember, the keyed array VIs are part of the Internet Toolkit and are located on the **Internet Toolkit>>CGI>>Keyed Array VIs** palette. The two functions we just used are described in Table 9-8.

Table 9-8

<p>Keyed Array Add.vi</p>	<p>Keyed Array Add adds a new element identified by key and containing value to array in. If an element with the same key already exists in array in, the value is returned in prev value, and replaced returns TRUE.</p>
<p>Keyed Array Index.vi</p>	<p>Keyed Array Index returns the value of the element identified by key.</p>

Calculator CGI: Calculator Form VI

Now that we have done the calculation that is the heart of this CGI application, we need to send everything back to the browser. This step involves first dynamically creating an HTML string that sends the same calculator form plus the result. Open the **Calculator Form VI**'s block diagram (see Figure 9-16).

This VI makes use of the HTML construction VIs (in **Internet Toolkit>>CGI>>HTML VIs**). If you study the block diagram, you'll see that we index our keyed array to get the values for the parameters and the result. The HTML VIs build the page and the form we saw in the browser.

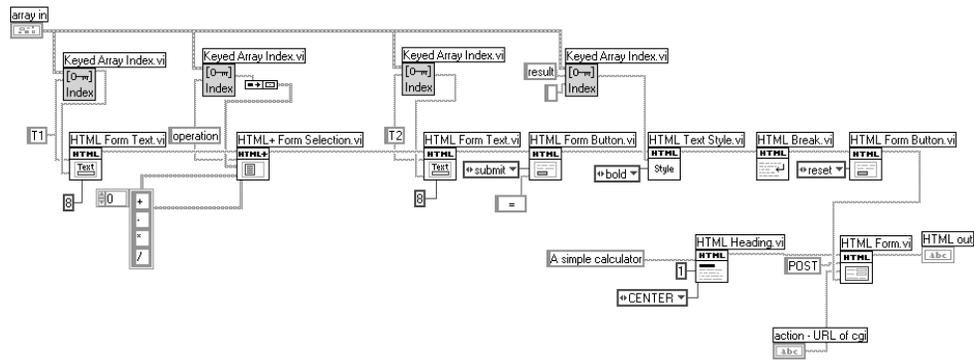


Figure 9-16

Calculator Form VI builds the HTML to be sent back to the browser

Wait a minute . . . why are we *building* the same HTML form inside the VI? Isn't this VI *called* by our HTML form? The answer is yes, we are creating a self-referential CGI application on purpose, which is quite common with CGI applications. Think about it a minute: If we wanted to, we could have made our application respond with a simple HTML document that gave the calculation result and just said "Thank You." But then the user would have to press "Back" on his or her browser to get to the calculator form again. It's much more elegant to display the same form again, with the results *and* the input values the user just submitted.

So, the **Calculator Form VI** outputs a string, which is simply the HTML source that is fed back to the browser by the function **CGI Write Reply**.

You might be wondering how the "first" HTML form gets created, if the form is generated only after calling the CGI. There are two ways to run the CGI: pressing the Submit button on the form, or requesting the CGI's URL directly from the browser. When you type in the URL that pointed to **calculator.vi**, you actually call the CGI application, but do not pass it any parameters. Although the CGI application expects the form parameters (e.g., "T1", "T2", "operation"), it can still execute successfully without them, simply returning the form and leaving blank those sections it did not have information for (e.g., the results). This method is common for CGI applications, since it avoids the need for a "starter" HTML file.

Calculator CGI: Summary

To summarize the steps in our example:

1. The HTTP server is running when a Web client requests the **calculator.vi** CGI.
2. Inside **calculator.vi**, we use the **CGI Read Request** to read the browser's environment variables and the form parameters after the user presses the equal sign.
3. With some subVIs, we process the form parameters and calculate the arithmetic result with **Perform Arithmetic**.
4. The HTML containing the form and the result is built with **Calculator Form**. This HTML "on-the-fly" document is passed back to the browser with **CGI Write Reply**.
5. The browser shows the form with the latest result to the right of the equal sign.

Hopefully, this dissection of an example has given you a good understanding of how a CGI application in LabVIEW works.

Challenge Activity 9.7

Modify the previous calculator CGI so that it allows the user to choose decimal or hex base for the numbers.

Now let's look at another example: a guestbook form that allows you to type text into the browser and see the results of a dynamically created HTML page.

Activity 9.8 Guestbook Form



The following "guest book" example is provided on the CD. To run this example, do the following:

1. From LabVIEW, run the Internet Toolkit HTTP server (you must have the Internet Toolkit installed).
2. Copy the file `guestbook.html` (located on the CD directory) to the root directory of your HTTP server (this is usually the directory `[path to LabVIEW]\LabVIEW\internet\home\`).

3. Copy the two files **guestbook.vi** and **guestbook.llb** from the CD (in the Ch9 directory) to the CGI directory of your HTTP server (this should normally be the directory [path to LabVIEW]\LabVIEW\internet\cgi-bin\).
4. With a Web browser such as Netscape Navigator, open this HTML page **guestbook.html** through HTTP (make sure you are not opening it as a file:/// URL). The URL will should look like `http://[your ip address]/guestbook.html`.
5. You should see the page shown in Figure 9–17.



Figure 9–17

6. This HTML document, called **guestbook.html**, contains a simple form that a user can fill out. Fill out the fields and press the Submit button.

7. If everything was set up properly, you should see a new HTML page that says “Thank you for visiting,” along with your name and a history of people who have run the CGI (you can go back and resubmit new names to see how the history works).

When you filled out the fields “First Name” and “Last Name” and pressed the Submit button, the data in those fields is passed to a CGI application. The CGI application in this case happens to be the LabVIEW VI called **guest-book.vi**, which you copied from the CD (this VI, in turn, calls some subVIs from **guestbook.llb**). The **guestbook.vi** takes the data from the HTML form, adds it to a log file, and returns new HTML data with a welcome message, the name just entered, and the contents of the guestbook file. (See Figure 9–18.)

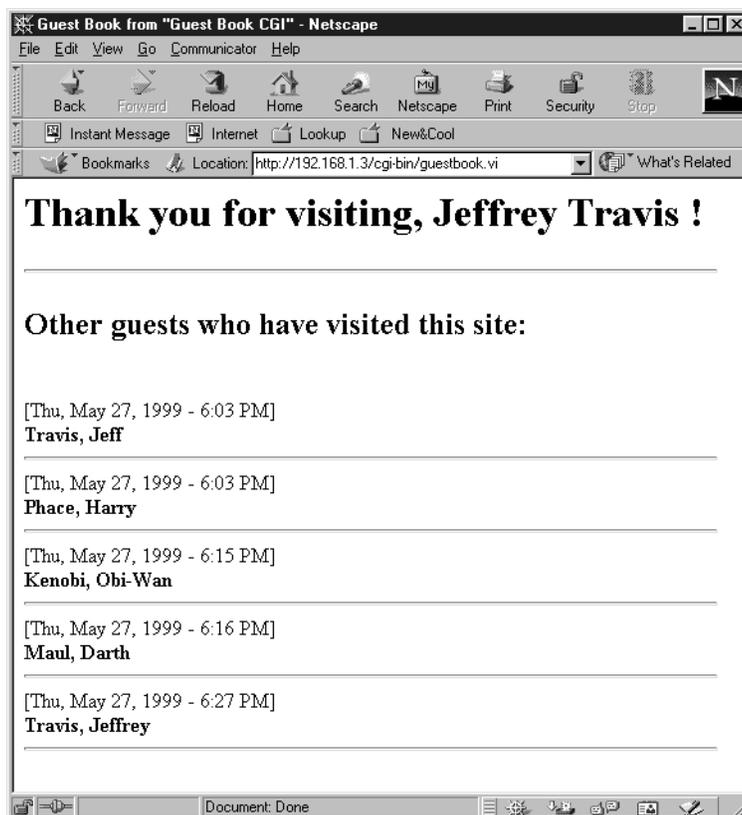


Figure 9–18

Notice that the resulting HTML page was generated on the fly, and was not a static document on the server somewhere. If you look closely at Figure 9–18, you’ll notice also that the URL in the address bar points to the guestbook VI, and not to an HTML document. This VI is the CGI application that just generated the document. The VI does all the “work” of saving the name, adding it to the log file, reading all the entries in the log file, and building a new HTML document. It also does some field validation (try this example, but leave one of the fields blank to see what happens).

Examine the HTML source code and the guestbook VIs to see how this example was made. Notice in particular how the form validation was done. The block diagram of **guestbook.vi** is shown in Figure 9–19.

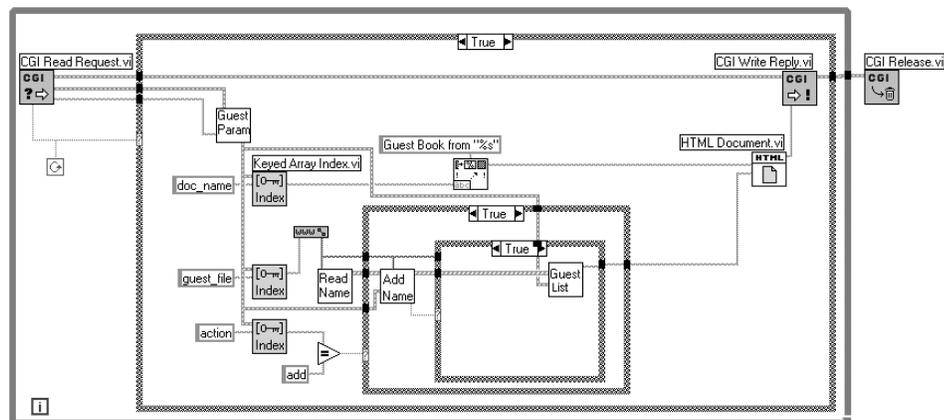


Figure 9–19
The top-level CGI guestbook.vi

■ Using Imagemaps to Control VIs

Now let’s look at an example of using graphical image controls on the Web. By using imagemaps, you can display front panel images on the Web that also can respond to user input. This is one of the ways you can **control** VIs over the Web.

Activity 9.9 Controlling a VI Over the Web with CGI and Imagemaps



This example is available also from the National Instrument's Web site. Do the following:

1. Inside the `cgi-bin` directory of the ITK Web server, create a new directory called `imap_ctl`.
2. From the CD, copy the contents of the "CGI Imagemap" directory to the `cgi-bin/imap_ctl/` directory.
3. Make sure the ITK Web server is running.
4. From your Web browser (preferably use Netscape for this example), open the URL:

```
http://127.0.0.1/cgi-bin/imap_ctl/index.htm
```

and follow the link.

5. You will see the image of a VI you can click on, as shown in Figure 9-20.

If you go to LabVIEW, you will see that the VI **panel.vi** is launched and running. This is the VI you are controlling (see Figure 9-21).

On the Web browser, click on the up and down arrows for "Frequency" and "Noise"; click on the Paused and Stop buttons. Notice also you can input data via form elements on the right.

Analyzing the CGI Imagemap Example

Now let's take a closer look at how the imagemap example works. Note that this example uses three VIs:

- **Panel.vi** is the VI we are controlling.
- **run_panl.vi** is a CGI VI that launches **panel.vi** and runs it.
- **ctlcgi.vi** is the CGI VI that handles all the user clicks on the imagemap and sends an updated image back to the browser.

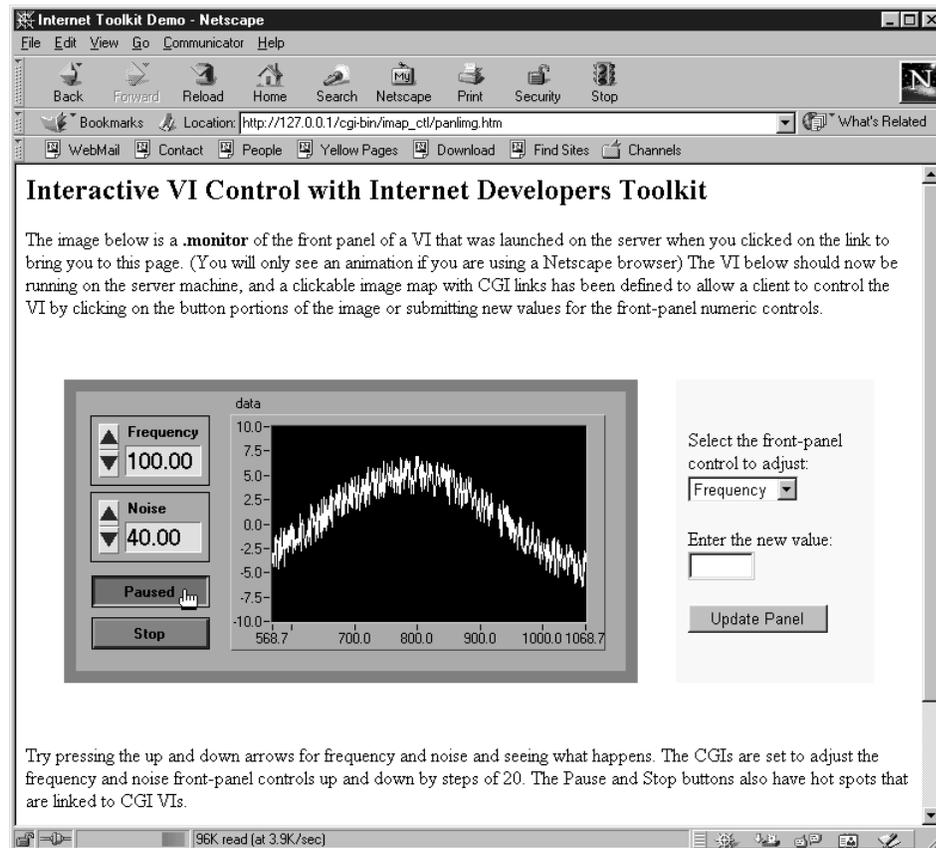


Figure 9-20

Controllable VI over the Web using CGI and imagemaps

Look at the HTML source of `panling.htm`, the HTML page that we interacted with. The HTML is shown as follows, with the important portions in **bold**.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
<HEAD>
  <TITLE>Internet Toolkit Demo</TITLE>
</HEAD>
<BODY>
<!-- The lines below define an image map for the panel of
panel.vi, creating clickable hot spots around the button
controls for freq/noise up/down, Pause, and Stop.
```

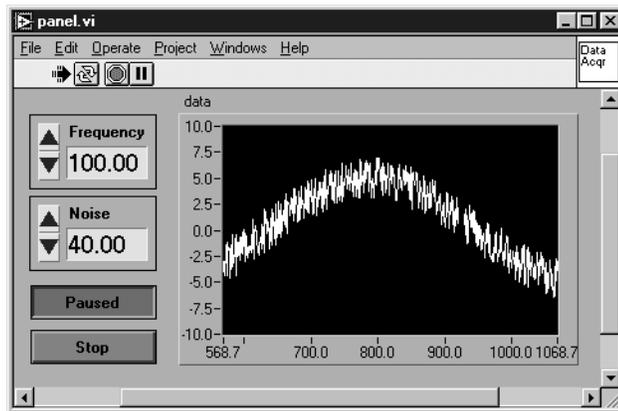


Figure 9-21

By associating the HREF values that it does with the hot spots, the map assures that the `ctlcgi.vi` callback VI will be invoked with the appropriate arguments when the user clicks on the various areas. -->

```
<MAP Name="buttons">
  <AREA Shape="Rect" coords = "18,26,35,47"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Fre-
    quency&value=up">
  <AREA Shape="Rect" coords = "18,48,35,71"
    HREF="/cgi-bin/Imap_ctl/ctlcgi.vi?control_name=Fre-
    quency&value=down">
  <AREA Shape="Rect" coords = "18,90,35,111"
    HREF="/cgi-bin/Imap_ctl/ctl-
    cgi.vi?control_name=Noise&value=up">
  <AREA Shape="Rect" coords = "18,112,35,133"
    HREF="/cgi-bin/Imap_ctl/ctl-
    cgi.vi?control_name=Noise&value=down">
  <AREA Shape="Rect" coords = "12,154,111,180"
    HREF="/cgi-bin/Imap_ctl/ctl-
    cgi.vi?control_name=Pause&value=switch">
  <AREA Shape="Rect" coords = "12,189,111,215"
    HREF="/cgi-bin/Imap_ctl/ctl-
    cgi.vi?control_name=Stop&value=switch">
</MAP>
<h2>
Interactive VI Control with Internet Developers Toolkit
</h2>
<p>
```

The image below is a **.monitor** of the front panel of a VI that was launched on the server when you clicked on the link to bring you to this page. (You will only see an animation if you are using a Netscape browser.) The VI below should now be running on the server machine, and a clickable image map with CGI links has been defined to allow a client to control the VI by clicking on the button portions of the image or submitting new values for the front-panel numeric controls.

<p>

```
<table cellspacing=30 cellpadding=10>
  <tr>
    <td bgcolor="gray">
      <p>
        <center></center>
        </td>
    <td width=150 bgcolor="yellow">
      <form ACTION="/cgi-bin/Imap_ctl/ctlcgi.vi"
        METHOD="GET" ENCTYPE="application/x-www-form-
        urlencoded">
        <p>
          Select the front-panel control to adjust:<br>
          <select name="control_name">
            <option value="Frequency">Frequency
            <option value="Noise">Noise
          </select>
          <p>
          Enter the new value:<br>
          <input name="value" type="text" size=6>
          <p>
          <INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Update
          Panel"></TD></TR>
        </form>
      </td>
  </tr>
```

</table>

<p>

Try pressing the up and down arrows for frequency and noise and seeing what happens. The CGIs are set to adjust the frequency and noise front-panel controls up and down by steps of 20. The Pause and Stop buttons also have hot spots that are linked to CGI VIs.

<p>

The form in the yellow table cell also invokes the same CGI when you press the "Update Panel" button, allowing you to interactively set the values of the front-panel frequency and noise controls.

```
</p>
```

```
<p>
```

Examine the HTML code for this page (panlimg.htm) and the CGI VIs that it uses to understand how it works.

```
</p>
```

```
</BODY>
```

```
</HTML>
```

Notice several things about the HTML code:

- It combines the use of the Web server's capability of creating front panel images on the fly with an imagemap. That is, in the `` tag, the `SRC` attribute is `".monitor?panel.vi"`, which is our VI's front panel, and it also references `usemap=#buttons`, which tells it to treat the image as an imagemap.
- The map is defined in the HTML within the `<MAP> ... </MAP>` tags.
- Notice that each hotspot on the map definition (in the `<AREA>` tag) points to our CGI VI, `"ctlcgi.vi"` but has a different query string; for example:

```
<AREA Shape="Rect" coords = "12,154,111,180" HREF="/cgi-bin/
Imap_ctl/ctlcgi.vi?control_name=Pause&value=switch">
```

is a rectangular shape whose coordinates enclose the Pause button on the image.

- From the URLs in each of the `<AREA>` tags, it's clear that we are always sending two parameters in the query string: **control_name** and **value**. The first parameter is the name of the "control" the user clicks on; the second parameter is the action or value of that control.
- In addition to the imagemap, there is a form with some inputs that also will send data to the CGI application if the user clicks on the Submit button.

Incidentally, clicking on an imagemap has the effect of **submitting** the CGI parameters, so there is no need to click on a separate Submit button.

Now let's look at the VIs that handle the click events on the imagemap. First, if you examine **panel.vi**'s block diagram, you'll see it doesn't even "know" about the Web interface, nor does it have any ITK VIs in the block diagram; this VI can run as a standalone. That's because the other two VIs control **panel.vi** through the user of VI Server functions.

Second, the **run_panl.vi** simply launches **panel.vi** when it is called through a CGI (this happened when you clicked on the link "click here" to start the program).

Third, the **ctlcgi.vi** is what does all the real work. Study its block diagram shown in Figure 9–22.

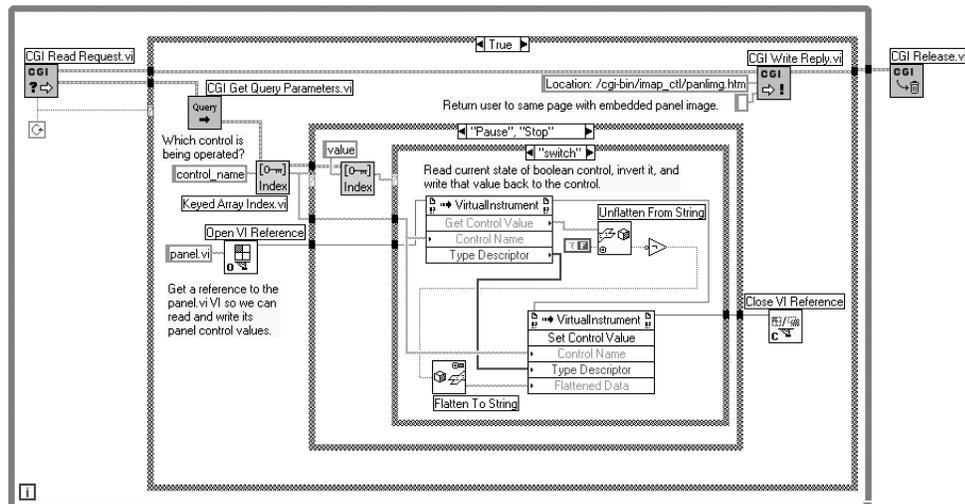


Figure 9–22

ctlcgi.vi handles the user's clicks on the imagemap and passes the changes to the VI "panel.vi" through a VI reference

Notice how the query parameters are parsed using **CGI Get Query Parameters.vi**. This function (part of the Internet Toolkit's CGI library) returns the query string portion of the URL. We know from the HTML code that the two query parameters are "control_name" and "value". Since these are passed as a keyed array, the **ctlcgi.vi** uses the **Keyed Array Index** to get the values of these two parameters.

Also note how the VI opened a VI reference to **panel.vi** (see Chapter 4, *The VI Server*, for information). It uses the VI Server methods "Get Control

Value” and “Set Control Value” from the VI class to set the values dynamically on **panel.vi**. The updated values come from “control_name” and “value”. Once the panel VI is updated, the whole HTML page is sent back to the user again and the VI reference is closed. Since the HTML pages use a “.monitor?” URL, it will get the latest animated image of **panel.vi** and reflect the changes made by the user’s click.

Activity 9.10 Modify the Imagemap Example

In this activity, you will modify the example we just looked at. You will add the capability for the user to change the plot color from the Web browser. Start by doing the following:

1. Make a copy of the VI and HTML files from the previous example, and rename them so they all have “2” after the name (e.g., **panel2.vi**, **index2.html**, etc.). Be sure these files are still in the `cgi-bin/imap_ctl/` directory.
2. Change **panel2.vi** so that it has a switch to modify the plot color, as shown in Figure 9–23.

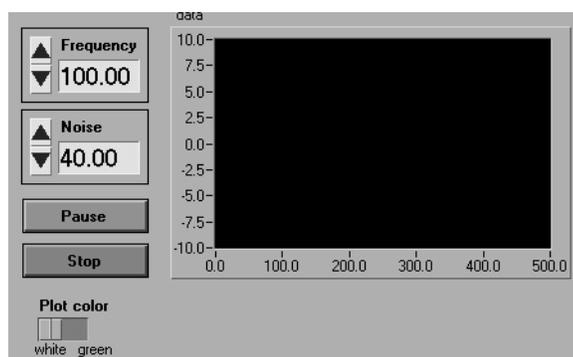


Figure 9–23

(Hint: on the block diagram, use a chart Attribute node to switch plot colors.)

3. Modify **index2.html** so that the link points to `/cgi-bin/imap_ctl/runpanl2.vi`.
4. Modify **panling2.html** so that every reference to “`cgictl.vi`” is updated to “`cgictl2.vi`”. Also update the `` tag so the source points to `.monitor?panel2.vi`.
5. Modify **run_panl2.vi** in the block diagram so that it references the new files.
6. Modify **cgictl2.vi** so that it returns the new HTML page, **panling2.html**.
7. Open `http://localhost/cgi-bin/imap_ctl/index2.html` in your Web browser and verify that everything works as in the old demo. The “plot color” hotspot won’t be active yet.
8. Now comes the only tricky part. You need to add an entry to the `imagemap` in the HTML file “`panling2.html`” so that users can click on the “plot color” switch. The best way to do this is with a professional HTML editor like Dreamweaver. First, you should create a PNG or JPEG image of “`panling2.html`” (with the **Print...** menu and using the HTML option). Then, with your HTML editor, open this image and define a hotspot for the “plot color” switch. You can then take the generated `<AREA>` tag with the appropriate coordinates and paste it into the “`panling2.html`” file.

Alternatively, you can try guessing the size and position of your switch (in pixels) and typing in the `<AREA>` tag. Either way, your last entry in the map should look something like this (the exact coordinates will vary, but you should type the HREF source exactly as shown):

```
<AREA shape="rect" coords="14,244,53,263" HREF="/cgi-bin/Imap_ctl/ctlcgi2.vi?control_name=Plot%20color&value=switch">
```

9. Now that you modified the map interface, you must be able to handle the “plot color” value in the “`control_name`” parameter. Because of the design architecture in **cgictl.vi**, this is surprisingly easy. Simply add the entry “Plot color” in the Case structure, as shown in Figure 9–24. That’s it!!

This works because the string “Plot color” is the same as the name of the control in **panel2.vi** and as the parameter value for “`control_name`” on the HTML code.

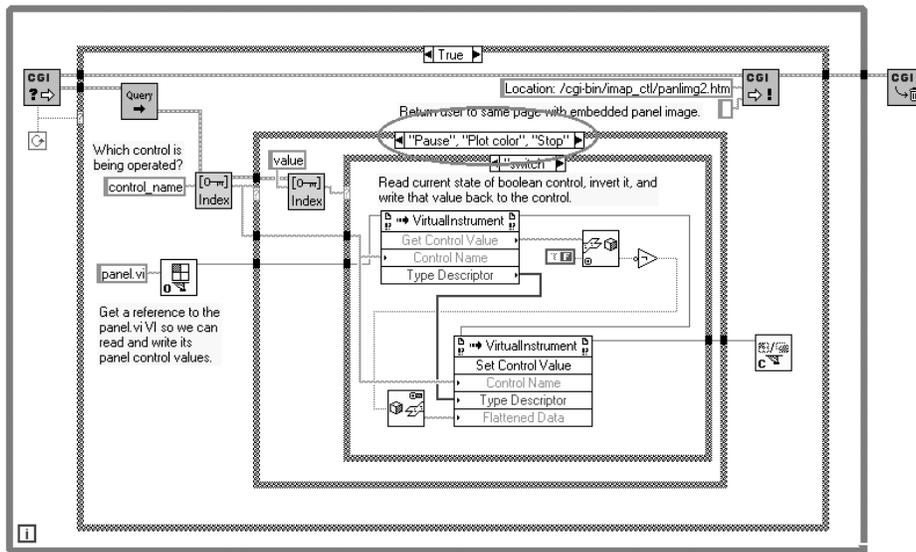


Figure 9-24

10. Run the program again by typing

```
http://localhost/cgi-bin/imap_ctl/index2.html
```

in your browser. Now you should be able to click on the hotspot and watch the plot color change (see Figure 9-25).

■ More CGI Examples

The best place for more CGI examples are the ones provided with the Internet Toolkit themselves. You can find examples on using CGI with:

- HTML generation
- Client-side imagemaps
- Server-side imagemaps
- Forms
- Cookies
- Sample applications

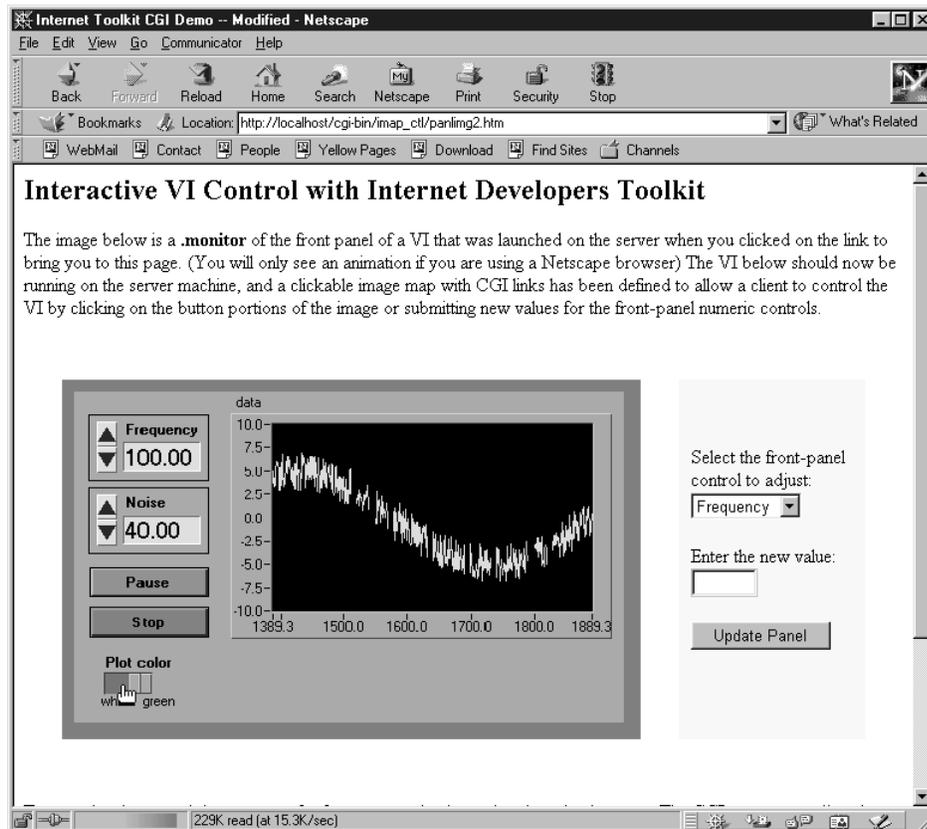


Figure 9-25

Just run the Internet Toolkit Web browser, open your browser to:

`http://localhost/examples/index.htm`

and explore!

FAQs

Where can I get more information about CGI specifications and programming?

Start by looking at these links:

World Wide Web consortium home: <http://www.w3.org>

NCSA HTTPD (HTTP Server) homepage: <http://hoohoo.ncsa.uiuc.edu>

A good reference and tutorial book on CGI is *CGI Programming on the World Wide Web* (O'Reilly, 1998).

Can I press the buttons and otherwise interact with the controls of my VI when I'm viewing it with a Web browser by using CGI?

Yes, in a somewhat limited manner as the example showed in this chapter. The image of the remote VI front panel is simply that—an image. It is not an ActiveX control or any other object that allows you to enter information directly into it. As we have seen, one method of providing VI interactivity for a remote client who is viewing a VI using a Web browser is to use a combination of the following:

1. An HTML image map to bind elements of the front-panel image to CGI VI links that change the corresponding front-panel values via VI Server. Any LabVIEW Boolean, slider, and—to some extent—knob element or numeric, can be “controlled” in this manner, given some tweaking of the imagemap and the callback CGI VI. No changes are required in the diagram of the VI being controlled.
2. HTML form elements, used in cases where simple clicks on the VI image cannot provide the control that is required. Examples are rings (which map to HTML form selection elements) and arbitrary numeric settings (which map to HTML form text entry boxes). Again, these form elements are also processed by CGI VIs that change the value of the front-panel elements via VI Server.

For greater flexibility in the Web browser's UI, or to simulate a front panel more accurately in a Web browser, you should use ActiveX or Java controls—see the next two chapters.

Can I use CGI scripts or executables written in languages other than G (Perl, Tcl, shell scripts, compiled code, etc.) to process client requests?

No, not unless you come up with a modification to the Server code to allow this. The problem here involves argument passing.

LabVIEW and the G Server do not have a convenient way to pass the query string from a client's browser to any CGI routine other than those written in G. There is also no convenient way to access the result of the non-LabVIEW script and pass it back to the client. Similar limitations are encountered when using the System Exec VI in LabVIEW.

Can other HTTP servers (httpd, etc.) make use of CGI routines written in G? How about compiled CGI VIs?

No, for reasons similar to those in the preceding explanation. Native CGI VIs would be completely unusable by other servers, and even with compiled version there is the problem of argument passing.

