eComet

and

eXtended Active Page (XAP)

Technology

Signature Systems, Inc.

Table of Contents

Introduction

Common Gateway Interface (CGI) is an industry-standard method for sending data from a web browser to a server. This document describes Signature Systems' implementation of CGI in the Comet system. This implementation is called **eXtended Active Page (XAP)** technology.

The following quotes provide the basic definition of Common Gateway Interface. The first quote comes from an introductory, college-level computer science textbook, and the second one comes from a book written "by working programmers for working programmers."

- "To interact with a Web-based database, it is necessary to pass requests from the browser to the database, then pass the results back to the browser. Programs written to the Common Gateway Interface (CGI) provide this capability. These programs can be written in programming languages such as Perl, C, and Visual Basic."

  Source: *Computer Concepts*
  by June Parsons and Dan Oja (Course Technology)

- "The CGI specification was created and documented by the main HTTP server authors: Tony Sanders, Ari Luotonen, George Phillips, and John Franks. These folks discovered that they didn't want to keep adding functionality to their HTTP servers every day, in keeping with needs of one particular Web server or another. They decided to build a clearly defined core or WWW server functionality and to provide a way to extend services and capabilities from there.

  "They needed an application programming interface (API) available to any Perl, C, or shell hacker willing to learn the appropriate interface details. Hence, the creation of CGI as a formal, rigorous specification that includes some nasty notation and grammar.

  "For more information about the CGI specification, please consult the following URL: http://hoohoo.ncsa.uiuc.edu/cgi/interface.html"

  Source: *Foundations of World Wide Web Programming with HTML & CGI*
  by Ed Tittle, Mark Gaither, Sebastian Hassinger, and Mike Erwin (IDG Books)

Signature Systems has implemented CGI via extensions to the Comet system and additions to the MoreThanBasic programming language. This document provides an overview of these features.

These features were announced to Signature's dealers and developers in February 2000. For an introductory period, these features are available to existing Comet98 licensees. At the end of this period, Signature Systems plans to market these features in the next major version of Comet under the product name "*eComet* Internet Application Server."

Signature's **eXtended Active Page** technology is patent pending.

Passing Requests from a Browser to a Server

As described above, Common Gateway Interface is a two-sided process. One of these sides sends data from a web browser (e.g., Netscape Navigator, Internet Explorer) to a server. Examples include:

- sending a URL to the server
- sending fields from a web-based data entry form to the server
- sending cookie information back to the originating server

A URL is a name that is contained within an HTML web page or typed on the "command line" of the web browser. The name may be a simple web page address, such as:

www.signature.net

The name may contain additional data intended for a CGI program on the server, such as:

http://signature.net:8080/XAP/sample?OPTION=READ&VALUE=123

The characters following the question mark in a URL are referred to as a "query string."

Web-based forms provide a way to send one or more fields of user-entered data from a web browser to a server. The most common example is a search engine, where the user enters data into a text box and clicks on a "submit" button. Other examples include more extensive forms with additional items such as radio buttons, check boxes, pull-down options, password fields, and hidden fields. In each of these examples, the data is sent from the browser to a CGI program on the server.

There are two methods for sending form data:

- METHOD GET builds a query string of all of the field names and data values and passes the resulting string to the server as a URL.
- METHOD POST sends all of the field names and data values to an awaiting CGI program without including them in a visible query string.

Cookies are strings of text that are sent to a browser from a CGI program running on a server. To complete the circle, the browser is able to send a cookie back to its originating server, thus allowing a CGI program to retrieve and process the information contained in a cookie (e.g., security information, a counter for measuring how many times the user has visited a certain web page, or simply sending application data from the browser back to the server).

eComet supports all of the above features. *eComet*, running as a server-based program, listens to a user-specified TCP/IP port. When *eComet* gets a request from a web browser, it performs the following actions:

- *eComet* receives all of the data that has been sent from the web browser
- If the URL contains a query string, *eComet* parses the query string and writes individual records to a Comet data file
- If form data has been sent via METHOD GET, *eComet* parses the query string and writes individual records to a Comet data file
- If form data has been send via METHOD POST, *eComet* parses the incoming data and writes individual records to a Comet data file
- *eComet* retrieves any cookies that it has previously sent to the browser, parses the "cookie string" and writes individual cookie records to a Comet data file
- *eComet* retrieves a group of CGI environment variables from the browser and writes individual records to a Comet data file

eComet then activates the object program that was specified in the original request from the web browser. For example, the following URL starts an object program running at Signature Systems' web server:

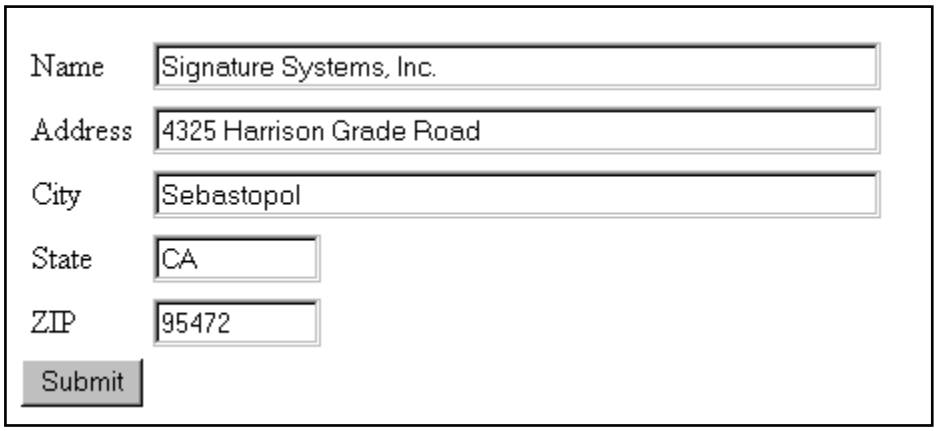
http://signature.net:8080/XAP/million

In the above example, the domain name is **signature.net**, the TCP/IP port is **8080**, the Comet directory name is **XAP**, and the object program name is **million**. (Note: Comet98 is running on this server, listening for CGI requests made on port 8080. When a request is made, *eComet* performs the actions listed above.)

As mentioned above, *eComet* parses the incoming data and writes individual records to a Comet data file, hereinafter called the **CGI Environment File**. This is a very powerful feature that provides a valuable service for the *eComet* programmer.

> Note: Such a feature is **not** provided in other CGI languages such as Perl. With those languages, the application program is responsible for parsing the incoming data and retrieving cookies, tasks that require extra code in each CGI program.

Here is an example that demonstrates the CGI Environment File feature. The following diagram contains a simple data entry form from a web page. There are five text boxes and one submit button. The submit button sends the data to *eComet*.



When *eComet* receives the above data, it parses the incoming data string and writes six records to the CGI Environment File. The CGI Environment File is a Comet keyed data file. Each key contains a maximum of 32 characters and each record contains a maximum of 1024 characters. The key and records contents for the above example are shown in the following chart:

Key	Record contents
Q.ADDRESS	4325 Harrison Grade Road
Q.BUTTON	Submit
Q.CITY	Sebastopol
Q.NAME	Signature Systems, Inc.
Q.STATE	CA
Q.ZIP	95472

Notes:

1. The letter "Q" at the beginning of each key signifies that these fields are **query** data. Other types of data in the CGI Environment File contain other prefixes and names (see below).

2. The field names (ADDRESS, BUTTON, CITY, etc.) are derived from the field names in the HTML document containing the data entry form. For the above example, here are the HTML statements that define the data entry fields:

```
<INPUT TYPE="TEXT" NAME="NAME" VALUE="" SIZE=50 MAXLENGTH=50>
<INPUT TYPE="TEXT" NAME="ADDRESS" VALUE="" SIZE=50 MAXLENGTH=50>
<INPUT TYPE="TEXT" NAME="CITY" VALUE="" SIZE=50 MAXLENGTH=50>
<INPUT TYPE="TEXT" NAME="STATE" VALUE="" SIZE=10 MAXLENGTH=10>
<INPUT TYPE="TEXT" NAME="ZIP" VALUE="" SIZE=10 MAXLENGTH=10>
<INPUT TYPE="SUBMIT" NAME="BUTTON" VALUE="Submit">
```

   eComet uses each field name from the HTML definition, converts the name to upper case and adds the letter "Q" as a prefix. Thus, the field named CITY results in a key named Q.CITY in the CGI Environment File.

eComet also writes the complete query string to the CGI Environment File. For the above example, the following record is written:

Key	Record contents
QUERY.STRING	NAME=Signature Systems, Inc.&ADDRESS=4325 Harrison Grade Road&CITY=Sebastopol&STATE=CA&ZIP=95472&BUTTON=Submit

Note: The *eComet* programmer will rarely need to use the complete query string, since *eComet* parses this data into individual records in the CGI Environment File.

The application program can retrieve the individual form fields by reading the appropriate record from the CGI Environment File. For example, to read the CITY field, the application program (written in MoreThanBasic) would use the following instructions:

```
     CGIDATA: FORMAT CGIDATA$                  ! data record
format
     .
     .
     .
     READ (1,CGIDATA) KEY='Q.CITY'             ! read CITY data
```

Likewise, to read the ADDRESS field, the program would use the following instruction:

```
     READ (1,CGIDATA) KEY='Q.ADDRESS'          ! read ADDRESS data
```

To restate the important point here, ***eComet* writes all incoming data to the CGI Environment File**. Here are some more examples:

1. Suppose the incoming data is a query string, such as:

   http://signature.net/XAP/sample?OPTION=READ&VALUE=123

   eComet writes the following data to the CGI Environment File:

Key	Record contents
Q.OPTION	READ
Q.VALUE	123
QUERY.STRING	OPTION=READ&VALUE=123

2. Using METHOD GET to send form data to the server results in the same type of URL as example 1 (i.e., one that includes field names and values in a visible query string). *eComet* parses the query string and writes individual records to the CGI Environment File.

3. Using METHOD POST to send form data to the server does not send a visible query string with the URL (i.e., you can't see the field names and values), but *eComet* still parses the incoming data and writes individual records to the CGI Environment File.

4. *eComet* retrieves any cookies that it has previously sent to the browser. This is an automatic process, handled by *eComet*, and does not require any code to be written in the application program. If one or more cookies is present on the browser, *eComet* retrieves them and performs the following actions:

   - Writes a master cookie string to the CGI Environment File
   - Parses the master cookie string into individual cookie strings
   - Writes individual cookie records to the CGI Environment File

   Examples:

   a. Suppose a single cookie has been previously written to the browser (by an *eComet* program running on the server). The cookie is named SHOPPINGCART and has a value of EMPTY. *eComet* retrieves the cookie string, parses it, and writes these two records to the CGI Environment File:

Key	Record contents
COOKIE	SHOPPINGCART=EMPTY
C.SHOPPINGCART	EMPTY

      Notes:

      1. The "COOKIE" key is the master cookie string.
      2. The letter "C" at the beginning of a key name signifies that this is an individual cookie.

   b. Suppose there are three cookies on the browser, as follows:

      SHOPPINGCART=EMPTY
      CUSTOMER=12345
      DATE=05/15/2000

      eComet retrieves these cookies and writes four records to the CGI Environment File:

Key	Record contents
COOKIE	SHOPPINGCART=EMPTY; CUSTOMER=12345; DATE=05/15/2000
C.CUSTOMER	12345
C.DATE	05/15/2000
C.SHOPPINGCART	EMPTY

To read an individual cookie from the CGI Environment File, the *eComet* program would use a single executable instruction. For example, here's how to read the DATE cookie:

```
READ (1,CGIDATA) KEY='C.DATE'       ! read DATE cookie
```

5. *eComet* also retrieves a group of CGI environment variables from the browser and writes individual records to the CGI Environment File. The CGI environment variables are defined in the **CGI specification**. Here are the variables that are currently implemented in *eComet*:

Key
CONTENT.LENGTH
CONTENT.TYPE
HTTP.ACCEPT
HTTP.REFERER
HTTP.USER.AGENT
PATH.INFO
REMOTE.ADDR
REMOTE.HOST
REQUEST.METHOD
SCRIPT.NAME
SERVER.NAME
SERVER.PORT
SERVER.PROTOCOL

6. Here is an example of a complete CGI Environment File containing cookies, query data, and CGI environment variables:

Key	Record
C.CUSTOMERNUMBER	123456789
C.DATE	11/18/1999
C.SHOPPINGCART	EMPTY
C.TIME	09:53:36.21
CONTENT.LENGTH	112
CONTENT.TYPE	application/x-www-form-urlencoded
COOKIE	CUSTOMERNUMBER=123456789; DATE=11/18/1999; TIME=09:53:36.21; SHOPPINGCART=EMPTY
HTTP.ACCEPT	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, */*
HTTP.USER.AGENT	Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)
PATH.INFO	
Q.ADDRESS	4325 Harrison Grade Road
Q.BUTTON	Submit
Q.CITY	Sebastopol
Q.NAME	Signature Systems, Inc.
Q.STATE	CA
Q.ZIP	95472
QUERY.STRING	NAME=Signature Systems, Inc.&ADDRESS=4325 Harrison Grade Road&CITY=Sebastopol &STATE=CA&ZIP=95472&BUTTON=Submit
REMOTE.ADDR	127.0.0.1
REQUEST.METHOD	POST
SCRIPT.NAME	/CGIFILE
SERVER.NAME	127.0.0.1
SERVER.PORT	80
SERVER.PROTOCOL	HTTP/1.1

Note: The Comet file system provides two ways for a MoreThanBasic application program to read records from a keyed file. The program can read a record by specifying an exact key match (as shown in the previous coding examples) or can read records in key-sequential order (increasing or decreasing ASCII order). The latter method gives a Comet program the ability to search through a keyed file without having to specify an exact key match. This is a general feature of the Comet system, and can facilitate the process of retrieving CGI data from the CGI Environment File.

Summary

To summarize this section, eXtended Active Page (XAP) technology in Comet facilitates the CGI process by receiving data from the browser, parsing the data, and writing records to a CGI Environment File. A Comet-based CGI application program is then able to access this data via READ statements using the appropriate keys to the CGI Environment File or reading through the CGI Environment File in key-sequential order (both methods are well understood by MoreThanBasic programmers).

Passing Results from the Server to the Browser

There are several ways that *eComet* sends information to a web browser. Some of these are similar to other CGI languages, while others are entirely unique to *eComet*.

Direct Printing

One method is to print HTML tags and/or text directly from the CGI program. With *eComet* and MoreThanBasic, this is accomplished with the PRINT statement. For example, the following statements send HTML tags and text (i.e., a web page) from *eComet* to a web browser:

```
Print '<HTML>'
Print '<HEAD>'
Print '<TITLE>'
Print 'Sample web page'
Print '</TITLE>'
Print '</HEAD>'
Print '<BODY>'
Print '<H1>'
Print 'This is a sample web page.'
Print '</H1>'
Print '</BODY>'
Print '</HTML>'
```

In CGI terms, this type of web page is called a "return page." The type of data is referred to as "text/html." Since the PRINT statements are controlled by the MoreThanBasic program, the content of the web page is completely dynamic. This powerful feature brings the full capability of the MoreThanBasic language and *eComet* to the CGI world.

If the above program was named SAMPLE and stored on Signature Systems' web server, the following browser command would activate the program:

http://signature.net:8080/XAP/sample

eComet can also send text (with no HTML tags) to the browser. This is referred to as "text/plain." The following MoreThanBasic statement initiates this process:

```
A$ = CONTROL(0,"TYPE 2")
```

Where:

A$ = an 80-byte string that stores the value returned by the CONTROL function. A plus sign (+) in the first position of this string indicates that the function was performed successfully, while a minus sign (-) in the first position indicates that the function failed. The addition 79 bytes contain descriptive information about the success/failure of the function.

0 = the logical unit number representing the XAP gateway

"TYPE 2" = the function to be performed (i.e., set "text/plain" mode)

A second version of this statement instructs *eComet* to add a line feed after each printed line:

```
A$ = CONTROL(0,"TYPE 3")
```

Here is a short *eComet* program that sends plain text to the browser and includes a line feed after each printed line:

```
A$ = CONTROL(0,"TYPE 3")
Print 'Payment      Payment      Interest     Principal   Balance'
Print 'Number       Amount       Paid         Paid        Due'
Print ''
FOR I = 1 TO N
      INTPAID=BALANCE*RATE
      PRINCPAID=PAYMENT-INTPAID
      BALANCE=BALANCE-PRINCPAID
      Print I;PAYMENT;INTPAID;PRINCPAID;BALANCE
NEXT I
```

Sending a Stored HTML Document

The above examples show how *eComet* programs can send dynamic web pages to a browser. However, not all web pages are dynamic. One of the unique features of *eComet* is its ability to send an entire stored HTML document to the browser using a single instruction. The MoreThanBasic instruction is:

```
String = CONTROL(0,"SEND html-file-name")
```

For example, here's the instruction to send an HTML file named **c:\comet\htm\sample.htm**:

```
A$ = CONTROL(0,"SEND c:\comet\htm\sample.htm")
```

When executing the SEND command, *eComet* retrieves the stored HTML file from disk and transmits it to the browser. There is no size limit for the HTML file sent using the SEND command.

Merging Data into a Stored HTML Document

One of the most powerful features of *eComet*'s eXtended Active Page technology is the MERGE instruction. The MERGE instruction performs two tasks:

- It merges "live" data from an *eComet* system into a stored HTML document, a feature similar to word processing "mail merge." The result is a dynamic web page containing data from a Comet data base.

- It sends sections of a stored HTML document to a browser, anything from a single character to 64K bytes

> Note: In and of itself, the "mail merge" capability is not unique. In fact, many web servers support a similar feature called "server side include." However, when combined with the ability to send *sections* of a stored HTML document, the *eComet* MERGE feature is a unique and powerful programming tool.

The following example demonstrates the point. Suppose you have a stored HTML document named **c:\comet\htm\customer.htm**. The purpose of this document is to display a list of customer names on a web browser. Assume that the customer data is stored in a Comet data file.

Using the HTML **comment tag**, the HTML document has been divided into three sections:

- The top section contains the HTML formatting tags and text that display the browser window title and page heading

- The middle section contains a merge field in *eComet* format. The "**" characters surround the field name.

- The bottom section contains the HTML tags that end the document

The sections are indicated by the <!--BEGIN--> and <!--END--> tags. To an HTML browser, these tags appear to be comments (i.e., they are not interpreted by the browser), but to *eComet*, these indicate the beginning and ending of sections within the document.

```
<HTML>
<HEAD>
<TITLE>
Customer list
</TITLE>
</HEAD>
<BODY>
<H1>
Customer list
</H1>
<P>
<!--END TOP-->


<!--BEGIN MIDDLE-->
**CUSTOMERNAME**
<BR>
<!--END MIDDLE-->

<!--BEGIN BOTTOM-->
</BODY>
```

c:\comet\htm\customer.htm

Notes:

- The first section does not require a <!--BEGIN--> tag, but the other sections do.
- All sections require an <!--END--> tag.
- Each section must have an alphanumeric section label (maximum of 28 characters per label). In this example, the labels are TOP, MIDDLE, and BOTTOM.

eComet sends HTML file to the browser as follows:

1. First, the MoreThanBasic program retrieves and sends the TOP section using a MERGE statement:

   ```
   A$=CONTROL(0,"MERGE c:\comet\htm\customer.htm SECTION TOP")
   ```

   The MERGE syntax is similar to the SEND statement, but allows for sending sections of the stored HTML document and merging data into the document.

2. Next, the MoreThanBasic program retrieves, merges and sends the MIDDLE section multiple times. Here's an outline of the process:

   a. The MoreThanBasic application program reads the next record from the Comet data file (i.e., the customer file)
   b. The MoreThanBasic application program writes the merge data to the CGI Environment File, using "**" as the prefix and suffix to the merge field name.

The key to the CGI Environment File must match the merge field name contained in the HTML document

c. *eComet* merges the CGI Environment File with the MIDDLE section of the HTML file

d. Repeat steps a through c until there are no more data records to process

In MoreThanBasic, this would be:

```
          OPEN (10) 'CUSTFILE'

ReadMore: READ (10,CUSTOMER) EXCP=AllDone
          CGIDATA$ = NAME$
          WRITE (1,CGIDATA) KEY='**CUSTOMERNAME**'
          A$ = CONTROL(0,"MERGE . SECTION MIDDLE")
          GOTO ReadMore

AllDone:  CLOSE (10)
```

Note: This version of the MERGE instruction shows a coding shortcut. The "." following the MERGE command represents the HTML file name merged in the TOP section.

3. Finally, a single instruction merges the BOTTOM section of the HTML document:

```
A$ = CONTROL(0,"MERGE . SECTION BOTTOM")
```

The result is a dynamic web page containing a simple heading and the complete list of customers from a Comet data file.

The MERGE feature can "search and replace" any part of a stored HTML document, including HTML tags, formatting codes, page titles, colors codes, etc. For example, suppose an HTML document contained the following tags:

```
<HTML>
<HEAD>
<TITLE>
**PAGETITLE**
</TITLE>
</HEAD>
<BODY BGCOLOR="**BACKGROUNDCOLOR**">
etc.
```

An MTB could write two records to the CGI Environment File (with keys of **PAGETITLE** and **BACKGROUNDCOLOR**), then instruct *eComet* to MERGE the CGI Environment File into the HTML document. The result would be a web page with a dynamic window title and background page color.

An observation:

The MoreThanBasic language includes a FORMAT statement that defines screen formats for existing Comet applications (i.e., programs running within the Comet window, not within a web browser). The MERGE feature provides a similar capability for *eComet* programs. As MoreThanBasic programmers gain experience with *eComet* and the MERGE feature, they will most likely start to use stored HTML documents in the same way their current applications use the FORMAT statement.

Sending Cookies to a Browser

eComet programs can send cookies to a web browser using the following statement:

```
String = CONTROL(0,"COOKIE name=value [,time-period]")
```

For example, the following statement sends a cookie named CUSTOMER with a value equal to 12345:

```
A$ = CONTROL(0,"COOKIE CUSTOMER=12345")
```

Since no time period was specified, this cookie will expire when the user exists from the browser program. This type of cookie is called a **dynamic cookie**.

To make a cookie remain in the browser for a longer period of time, use the time period parameter. The COOKIE instruction recognizes three units of time:

+*n***M** = *n* **minutes** into the future
+*n***H** = *n* **hours** into the future
+*n***D** = *n* **days** into the future

This type of cookie is called a **persistent cookie**. For example, to make the above cookie persist for 10 days, use the following statement:

```
A$ = CONTROL(0,"COOKIE CUSTOMER=12345,+10D")
```

This cookie will expire 10 days from the current date.

An *eComet* program can write multiple cookies to the browser, but note that cookies must be sent before any other data is sent from the *eComet* program.

> Note: As mentioned in an earlier section of this document, *eComet* automatically retrieves all cookies it has sent to the browser each time an *eComet* program is started. The cookie string is parsed and individual cookie records are written to the CGI Environment File.

Application notes:

Using cookies is a circular process. Initially, an *eComet* program writes a cookie to a browser. Subsequent *eComet* programs can READ the cookie and its value from the CGI Environment File, and (if desired) write a new cookie value to the browser.

One simple example is using a cookie to count the number of times a user has visited certain web page. On their initial visit, the *eComet* program could write a cookie with a value of 1. On subsequent visits to the web page, the program could read the user's cookie, increment the value, then re-send the cookie to the browser. At any given time (until the expiration of the cookie), the value of the cookie would equal the number of times the user had visited the web page.

A more complex example is an online shopping cart. The *eComet* program could keep track of the items in a user's shopping cart by writing a cookie to the browser each time the user added or removed an item from their cart. At any given moment, the cookie would contain data about the items in the cart. If the cookie was a persistent cookie, the user could even exit from the browser and return at a later time to find that the cookie still contained information about the items in the cart (until the expiration of the cookie, of course).

Persistent cookies always expire at some specified future time/date. However, if an *eComet* program sends a persistent cookie more than once, the most recent occurrence will determine the expiration date. Example: Suppose an *eComet* program writes a persistent cookie with a 10-day expiration date. And, suppose the user executes the same program five days later. At that point, the cookie will be rewritten with expiration 10 days from that date.

Summary

To summarize this section, eXtended Active Page (XAP) technology in Comet facilitates the Common Gateway Interface process by providing numerous methods for sending data to a browser. These methods include direct printing of html/text or plain/text, sending an entire static HTML document, merging "live" data into sections of a stored HTML document, sending cookies, and functioning as a web server.

The next section includes information about additional programming features, configuration, security, and application notes.

Additional Features and Commands

Automatically Preserving Program Context

One of the general problems that all CGI programmers face is how to maintain the continuity between a browser and the server. Each time a CGI program executes, it sends a web page to the browser and then stops executing. When (or if) the browser user responds, their response starts another CGI program or restarts the initial CGI program. In either case, the newly-started CGI program often has to re-establish the context from the initial CGI program.

There are several general solutions available to CGI programmers, regardless of the CGI environment being used:

- The CGI program can send **hidden fields** in forms. When the browser sends the form data back to the server, the hidden data accompanies it. This data can be used to re-establish the context from the initial program.

- The CGI program can send a **cookie** (or multiple cookies) to the browser. When the browser sends form data back to the server, the cookies are also returned. These can also be used to re-establish the context.

- The CGI program can store data on the server and identify it with the browser (i.e., the remote user) using the **REMOTE.ADDR** CGI Environment Variable containing the IP address of the remote user (a unique value). Then, when the browser sends a follow-up request, the CGI program can read the data from the file and continue processing.

The above general solutions require the CGI programmer to add instructions to each program that needs to re-establish context.

While these solutions are available in *eComet* applications, Comet itself provides a unique feature that automatically preserves CGI program data on the server. This data is stored under a unique name (a name that corresponds to the remote user's specific computer). When the remote user (browser) starts another *eComet* program on the Comet server, *eComet* reads and restores this data, thus automatically re-establishing the context of the initial program.

This feature facilitates continuity in a unique and expedient way. The *eComet* programmer does not need to write a single line of code to preserve the program's context.

Here is a detailed description of the process. When Comet compiles a MoreThanBasic source program, it creates a Comet object program. This object program is built according to a memory map that contains five sections.

- The first section contains "COMMON" data; program variables that are intended to be used in more than one program (as in a series of program "overlays" or in a series of multiple CGI programs).

- The second section contains “LOCAL” data; variables intended for the current program only.
- The third section contains the “constants” used by the program.
- The fourth section contains the “formats” that will be used by this program to control input/output of data (e.g., file layouts, screen layouts, etc.).
- The fifth section contains the “executable code” for the program.

Here’s a visual representation of a Comet object program:

COMMON data
LOCAL data
Constants
Formats
Executable code

With *eComet*, the COMMON data in a program is stored on the *eComet* server, and a special cookie is sent to the browser. When this special cookie is returned (i.e., the next time the web browser runs an *eComet* program on the same server), the COMMON data is restored.

Here's a step-by-step description of the process:

1. Every time an *eComet* program is invoked, *eComet* looks for a cookie called "*eComet*Context" (case sensitive). If this cookie is found, its value is interpreted as a unique file name in the /temp/ subdirectory under the Comet "COS" directory.
2. If the Comet program-to-be-launched has COMMON declared, after loading the program, *eComet* clears COMMON and attempts to read the above file into COMMON for the length of COMMON declared in it.
3. When *eComet* generates a web page from a program containing any COMMON storage, it will automatically generate a dynamic cookie with the name of the unique file which will contain all of the COMMON storage once the XAP gateway is closed.
4. When the XAP gateway is closed, all COMMON variables are written to the file and it is closed.
5. If there is no COMMON declared in a program, nothing happens. The web browser will retain the cookie, however, for the next program that has COMMON declared.
6. If there is COMMON declared in the program, but the file does not exist, or there is no cookie, a new file name will be created and COMMON will be automatically cleared.
7. Since the cookie is dynamic, the COMMON data will be lost once the browser is closed.

As mentioned above, this is an automatic feature of *eComet*. The MTB programmer does not need to write a single instruction to use this feature.

Redirecting to Another URL

eComet includes a statement to redirect an *eComet* program to another URL. The statement is:

```
String = CONTROL(0,"REDIR new-url")
```

For example, to redirect an *eComet* program to Signature Systems' web site, you would use the following instruction:

```
A$ = CONTROL(0,"REDIR www.signature.net")
```

Redirection, as a general CGI process, is typically used in conjunction with other processing steps. For example, a CGI program might read a cookie from the browser, use that cookie to retrieve and validate data from a data file (such as a customer number or security code), send a new cookie to the browser, then redirect to a new CGI program or web page.

Logging all CGI Output

eComet includes a statement that logs outbound data to a text file. This can help the MTB programmer debug a program that sends data to a browser. The statement is:

```
String = CONTROL(0,"LOGFILE path")
```

For example, to log outbound data to a file named c:\test\cgilog.txt, you would use the following statement:

```
A$ = CONTROL(0,"LOGFILE c:\test\cgilog.txt")
```

When the *eComet* program is activated, the log file is created. If a file by the same name already exists, it is erased and a new (empty) file is created. All outbound data sent by the program to the browser is also written to the log file. The log file is a text file that can be viewed by any text editor (e.g., Notepad, Wordpad).

Setting the MIME Type

Data sent from a server to a browser must contain a properly formatted header at the beginning of the message (before any program-generated data). One of the items in this header portion of the message is called the "response header" and includes specific information about the data being sent, plus whatever Multipurpose Internet Mail Extensions (MIME) declarations may be required to send the data.

eComet provides a statement that can change the MIME type. The statement, which was introduced earlier in this document, is:

```
String = CONTROL(0,"TYPE MIME-type")
```

The following table describes the MIME types available in *eComet*:

MIME type	Meaning
0	Raw output. The *eComet* program must send all of the required header data. This type should be used only by those who have knowledge of the exact syntax of response headers.
1	HTML output (text/html). The response header is generated by *eComet*. This is the default MIME type for *eComet*.
2	Output is text/plain with no line feeds after each printed line. The response header is generated by *eComet*.
3	Output is text/plain with a line feed sent after each printed line. The response header is generated by *eComet*.

Security

One of the concerns in running a CGI system is security. Typically, CGI programs are required to reside in a specific directory (such as *my-cgi-bin* on a Unix server). On an *eComet* server, CGI programs must reside in a Comet directory named **XAP**. This ensures that a browser will have access to only those Comet programs deliberately placed in a public directory.

Additional security can be achieved by isolating the Comet data files from the *eComet* programs. This can be done by placing the data files on another computer (a network file server) and establishing a firewall between the *eComet* computer and the network file server. Thus, the *eComet* machine becomes an Internet Application Server (only) whose data files reside on a network file server.

Configuring and Running the XAP Gateway

Here are the configuration requirements for *eComet*. The *eComet* machine must contain:

- a type 4 gateway
- background partitions where the *eComet* programs will be activated
- a Comet directory named XAP (for the *eComet* object programs)
- a Comet directory named COS (for temporary files)
- a subdirectory of COS named /temp (for COMMON context files)

In addition, the XAPMON startup program must be executed on the *eComet* machine. This program activates code in Comet to listen to a specified TCP/IP port (see below).

When a CGI request is received, Comet accepts, parses and stores the incoming data (as described above), then activates the requested *eComet* program in the next available background partition. The background partition inherits the following:

- Logical Unit Number (LUN) 0 is the web browser. Thus, any data printed to LUN (0) is sent to the browser.

- LUN (1) is the CGI Environment File. This LUN is open and available for reading and writing.

The *eComet* program (running in a background partition) carries out its assigned tasks, which presumably includes sending data back to the browser. When these tasks are complete, the background partition terminates itself using an instruction such as KILL PARTITION$.

If the browser user wishes to continue a transaction, the browser sends another request to *eComet*, and another background partition is activated. Continuity between the browser and server can be achieved using cookies, hidden fields within HTML forms, and/or the automatic context preservation feature in *eComet*.

Choosing the TCP/IP Port

All of the examples in this document have included port 8080 as the TCP/IP port for *eComet*. Here is the background information regarding port numbers.

All commercial web servers use a default port number of 80. Thus, when you specify a URL such as `http://www.signature.net`, the request is being sent to port 80 of the domain name signature.net, where the web server software is listening for data. In the case of signaure.net, the current web server software is O'Reilly's Website.

In order to run *eComet* on the same machine as another web server, you must specify a different port number. Signature Systems recommends port 8080 for these systems. The port number is a configurable value, though, so you can use any open port you choose.

For example, if your server is *not* running another web server, you could use port 80 for *eComet*, and avoid using the port number is the URL. Thus, requests made to your domain name would go to port 80 and be processed by *eComet*.

Miscellaneous Programming Notes

1. Logical unit numbers

   In an *eComet* program, Logical Unit Numbers 0 through 9 are reserved (i.e., the MoreThanBasic application may not use these LUNs).

   - LUN (0) represents data that the *eComet* program sends to the browser.
   - LUN (1) is the CGI Environment File
   - LUN (2) through LUN (9) are not currently used, but are reserved

   Note: When the MoreThanBasic program closes LUN (0), *eComet* starts sending data to the browser.

2. CGI Environment File

   When *eComet* receives incoming data from a browser, it parses the data and writes it to the CGI Environment File. This file is created in a Comet directory named **COS** (one of the standard directories on every Comet system). The file is a Comet keyed file, with 1024-byte records and 32-byte keys.

   The CGI Environment File name is **CGI_xxxx**, where **xxxx** is the Comet task identification number (e.g. 0001, 0002, 0003, 0004).

   The CGI Environment File remains on the disk until (1) it is overwritten by a subsequent *eComet* program using the same background partition, or (2) erased by some other Comet program.

3. Results returned by CONTROL statement

   Each time the CONTROL statement is used, Comet returns an 80-byte result string indicating the success or failure of the command. For example:

   ```
   A$ = CONTROL(0,control-function)
   ```

   - A successful command returns a plus sign (+) in the first byte of A$.
   - An unsuccessful command returns a minus sign (-) in the first byte of A$.

   The remainder of A$ (up to 79 additional bytes) contains a description of any error that occurred for an unsuccessful command.

4. Alternative syntax for the CONTROL statement

The syntax for the CONTROL statement is:

```
String = CONTROL(logical-unit-number, control-function
[,EXCP=exception-path])
```

For example, the following statement sets MIME type 3 on logical unit number 0, and specifies an exception path to statement 9999:

```
A$ = CONTROL(0,"TYPE 3",EXCP=9999)
```

The CONTROL function can also be accomplished with the FILE statement, as follows:

```
FILE (logical-unit-number) CTL="control-function"
[,EXCP=exception-path]
```

For example, here is the statement that sets MIME type 3:

```
FILE (0) CTL="TYPE 3",EXCP=9999
```

When this statement is executed, the result string is placed in the user's input buffer. To retrieve this value, the program must execute an INPUT statement immediately after executing the FILE statement, as follows:

```
FILE (0) CTL="TYPE 3",EXCP=9999
INPUT (0) A$
```

In this example, A$ is an 80-byte string that contains the result string.

The CONTROL statement and FILE statement produce the same object code when compiled, but Signature Systems recommends the CONTROL statement as it simplifies the interrogation of the result string (i.e., a separate INPUT statement is not required).