WASD Hypertext Services - Scripting Environment

January 2001

For version 7.1.1 release of the WASD VMS Hypertext Services HTTP server.

Supercedes:

```
November 2000
June 2000
December 1999
May 1999 (Technical Overview)
```

Abstract

This document is an overview of scripting for the WASD VMS HyperText Transport Daemon.

Also see "WASD Technical Overview" for information on using the WASD VMS Hypertext Services and "WASD Hypertext Environment" for information on using the WASD VMS Hypertext Services and "WASD Nuts and Bolts" for technical information on server design and implementation.

It is strongly suggested those using printed versions of this document also access the Hypertext version. It provides online access to some examples, etc.

Author

Mark G. Daniel
Surveillance Systems Division
Defence Science and Technology Organisation

For WASD-related email please use
Mark.Daniel@vsm.com.au

Should the above address present problems or provide no response for an extended period then use
Mark.Daniel@dsto.defence.gov.au

+61 (8) 82596189 (bus)
+61 (8) 82596673 (fax)

PO Box 1500
Salisbury
South Australia 5108

Online Search

online search

Online PostScript

This book is available in PostScript for access by suitable viewers (e.g. GhostScript). Use a browser to access a print menu in this same location. It can be reprinted from the location HT_ROOT:[DOC.SCRIPTING]SCRIPTING.PS

Online Demonstrations

Some of the online demonstrations may not work due to the local organisation of the Web environment differing from WASD where it was originally written.

WASD VMS Hypertext Services

Copyright © 1996-2001 Mark G. Daniel.

This package is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; version 2 of the License, or any later version.

This package is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

See HT_ROOT:[000000]GNU_GENERAL_PUBLIC_LICENSE.TXT

You should have received a copy of the GNU General Public License along with this package; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

The Apache Group

This product includes software developed by the Apache Group for use in the Apache HTTP server project (http://www.apache.org/).

```
Redistribution and use in source and binary forms, with or without
modification, are permitted ...
```

OpenSSL Project

This product *can* include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

```
Redistribution and use in source and binary forms, with or without
modification, are permitted ...
```

Eric A. Young

This package *can* include cryptographic software written by Eric Young (eay@cryptsoft.com) and Tim Hudson (tjh@cryptsoft.com).

```
This library is free for commercial and non-commercial use provided ...
Eric Young should be given attribution as the author ...
copyright notice is retained
```

Ohio State University

This package contains software provided with the OSU (DECthreads) HTTP server package, authored by David Jones:

```
Copyright 1994,1997 The Ohio State University.
The Ohio State University will not assert copyright with respect
to reproduction, distribution, performance and/or modification
of this program by any person or entity that ensures that all
copies made, controlled or distributed by or for him or it bear
appropriate acknowlegement of the developers of this program.
```

RSA Data Security

This software contains code derived in part from RSA Data Security, Inc:

```
permission granted to make and use derivative works provided that such
works are identified as "derived from the RSA Data Security, Inc.
MD5 Message-Digest Algorithm" in all material mentioning or referencing
the derived work.
```

Bailey Brown Jr.

LZW compression is implemented using code derived in part from the PBM suite. This code is copyright by the original author:

```
* GIF Image compression - LZW algorithm implemented with Tree type
*                         structure.
*                         Written by Bailey Brown, Jr.
*                         last change May 24, 1990
*                         file: compgif.c
*
*  You may use or modify this code as you wish, as long as you mention
*  my name in your documentation.
```

Other

OpenVMS, **Digital TCP/IP Services for OpenVMS**, **VAX C**, **DEC C**, **VAX** and **AXP** are registered trademarks of Compaq Computer Corporation.

MultiNet is a registered trademark of Cisco Systems, Inc.

Pathway is a registered trademark of Attachmate, Inc.

TCPware is a registered trademark of Process Software Corporation.

Contents

Chapter 1 Introduction

1.1 Scripting Processes 1–2
1.1.1 Detached Process Scripting 1–2
1.1.1.1 Non-Server Account Scripting 1–3
1.1.1.2 Restricting Non-Server Scripting 1–4
1.1.1.3 Process Priorities 1–5
1.1.2 Subprocess Scripting 1–5
1.2 Script Mapping 1–6
1.3 Script Run-Time 1–8
1.4 Scripting Logicals 1–9
1.5 Scripting Scratch Space 1–10
1.6 DCL Processing of Requests 1–11
1.7 Scripting Function Library 1–11
1.8 HTTP Persistant-State Cookies 1–12

Chapter 2 CGI

2.1 CGI Environment Variables 2–1
2.2 Script Output 2–4
2.2.1 CGI Compliant Output 2–5
2.2.2 Non-Parsed-Header Output 2–8
2.3 Raw HTTP Input (POST Processing) 2–10

Chapter 3 CGIplus

3.1 CGIplus Programming 3–1
3.2 Code Examples 3–3
3.3 Other Considerations 3–5

Chapter 4 Run-Time Environments

4.1 RTE Programming 4–2
4.2 Server Configuration 4–3

Chapter 5 CGI Callouts

5.1 Requests and Responses 5–2
5.2 Code Examples 5–3

Chapter 6 ISAPI

6.1 CGIsapi 6–1
6.2 Writing ISAPI Scripts 6–2
6.3 Server Configuration 6–3

Chapter 7 DECnet & OSU

7.1 Script System Environment 7–3
7.1.1 Proxy Access 7–3
7.1.2 DECnet Objects 7–4
7.1.3 Reducing Script Latency 7–5
7.1.4 DECnet/OSU Startup 7–5
7.2 CGI 7–5
7.3 OSU (DECthreads) Emulation 7–6
7.4 User Scripts 7–9

Chapter 8 Perl

8.1 Activating Perl 8–1
8.2 CGI Environment 8–2
8.3 POSTed Requests 8–3
8.4 Reducing Latency 8–3
8.4.1 CGIplus 8–4
8.4.2 Run-Time Environment 8–4
8.5 Requirements 8–5

Chapter 9 Java

9.1 Requirements 9–1

Chapter 10 Raw TCP/IP Socket

Chapter 1
Introduction

This document is **not a general tutorial on authoring scripts**, CGI or any other. A large number of references in the popular computing press covers all aspects of this technology, usually quite comprehensively. The information here is about the specifics of scripting in the WASD environment, which for CGI and ISAPI is generally very much like any other implementation. (Although there are always annoying idiosyncracies, see Section 1.7 for a partial solution to smoothing out some of these wrinkles.)

Scripts are mechanisms for creating simple HTTP services, sending data to (and sometimes receiving data from) a client, extending the capabilities of the basic HTTPd. Anything that can write to SYS$OUTPUT can be used to generate script output. A DCL procedure or an executable can be the basis for a script. Simply TYPE-ing a file can be provide script output. Scripts execute in processes separate from the actual HTTP server but under it's control and interacting with it.

WASD manages a script's process environment either as a dependent subprocess or independent detached process created by the HTTP server, or as a network process created using DECnet. By default it supports subprocess CGI scripts without further configuration.

WASD scripting can deployed in a number of environments. Other chapters cover the specifics of these.

- Chapter 2 - CGI
- Chapter 3 - CGIplus
- Chapter 4 - Run-Time Environments
- Chapter 5 - CGI Callouts
- Chapter 6 - ISAPI
- Chapter 7 - DECnet & OSU
- Chapter 8 - Perl
- Chapter 9 - Java
- Chapter 10 - Raw TCP/IP Socket

1.1 Scripting Processes

Process creation under the VMS operating system is notoriously slow and expensive. This is an inescapable overhead when scripting via child processes. An obvious strategy is to avoid, at least as much as possible, the creation of these processes. The only way to do this is to share processes between multiple scripts/requests, addressing the attendant complications of isolating potential interactions between requests. These could occur through changes made by any script to the process' enviroment. For VMS this involves symbol and logical name creation, and files opened at the DCL level. In reality few scripts need to make logical name changes and symbols are easily removed between uses. DCL-opened files are a little more problematic, but again, in reality most scripts doing file manipulation will be images.

A reasonable assumption is that for almost all environments scripts can quite safely share processes with great benefit to response latency and system impact (see ''Technical Overview, Performance'') for a table with some comparative performances). If the local environment requires absolute script isolation for some reason then this process-persistance may easily be disabled with a consequent trade-off on performance.

Zombies

The term *zombie* is used to describe processes when persisting between uses (the reason should be obvious, they are neither ''alive'' (processing a request) nor are they ''dead'' (deleted) :^) Zombie processes have a finite time to exist (*non*-life-time?) before they are automatically purged from the system (see see ''Technical Overview, Configuration''). This keeps process clutter on the system to a minimum.

1.1.1 Detached Process Scripting

With WASD it is possible to execute scripts in processes created completely independently of the server process itself. This offers a significant number of advantages over subprocesses

- pooled quotas are no longer a consideration
- cannot directly affect the server process
- can be created with more appropriate process priorities
- can be executed under accounts different to that of the server
  (available via the $PERSONA services of VMS V6.2 and later)
- allows secure but yet full-featured user scripting
- processes created through the full account login process

without too many disadvantages

- process management is a little more complex
  (particularly with server process deletion)
- processes created through the full account login process
  (yes, it's both an advantage and disadvantage ;^)

Creation of a detached process is slightly more expensive in terms of system resources and initial invocation response latency (particularly if extensive login procedures are required), but this quickly becomes negligable as most script processes are used multiple times for successive scripts and/or requests.

Enabling Detached Processes

By default the server uses subprocesses for scripting (also the historical method by which WASD executes scripts). The HTTPD$CONFIG directive [DclDetachProcess] when enabled has the server create (almost) completely independent detached processes to execute scripts.

```
[DclDetachProcess]  enabled
```

When using detached processes, during shutdown the server must explicitly ensure that each scripting process is removed from the system (with subprocesses the VMS *executive* provides this automatically). This is performed by the server exit handler. With VMS it is possible to bypass the exit handler (using a $DELPRC or it's equivalent $STOP/ID= for instance), making it possible for ''orphaned'' scripting processes to remain - and potentially accumulate on the system!

To address this possibility, during startup the server scans the system for candidate processes. These are identified by a *terminal* mailbox (SYS$COMMAND device), and then further that the mailbox has an ACL with two entries; the first identifying itself as a WASD HTTPd mailbox and the second allowing access to the account the script is being executed under. Such a device ACL looks like the following example.

```
Device MBA335:, device type local memory mailbox, is online, record-oriented
  device, shareable, mailbox device.

  Error count                    0    Operations completed                  0
  Owner process                 ""    Owner UIC             [WEB,HTTP$SERVER]
  Owner process ID        00000000    Dev Prot              S:RWPL,O:RWPL,G,W
  Reference count                1    Default buffer size                2048
  Device access control list:
    (IDENTIFIER=WASD_HTTPD_80,ACCESS=NONE)
    (IDENTIFIER=[WEB,HTTP$SERVER],ACCESS=READ+WRITE+PHYSICAL+LOGICAL)
```

This rights identifier is generated from the server process name and is therefore system-unique (so multiple autonomous servers will not accidentally cleanup the script processes of others), and **is created during server startup** if it does not already exist. For example, if the process name was ''HTTPd:80'' (the default for a standard service) the rights identifier name would be ''WASD_HTTPD_80'' (as shown in the example above).

1.1.1.1 Non-Server Account Scripting

Generally when a script executes it is within a process owned by the server account. There are often advantages to running a script under another account. The most obvious of these is the security isolation it offers with respect to the rest of the Web and server environment. It also means that the server account does not need to be resourced especially for any particularly demanding application.

Non-server account scripting requires detached processes be enabled, and the $PERSONA system services available with VMS V6.2 and later. Non-server account scripting is not available under earlier versions of VMS.

Enabling Non-Server Scripting

The $PERSONA functionality must be explicitly enabled at server startup using the /PERSONA qualifier (see ‘‘Technical Overview, Server Account and Environment’’). The ability for the server to be able to execute scripts under any user account is a very powerful (and potentially dangerous) capability, and so is designed that the site administrator must explicitly and deliberately enable the functionality. Configuration files need to be rigorously protected against unauthorized modification.

A specific script or directory of scripts can be designated for execution under a specified account using the HTTPD$MAP configuration file SET *script=as=* mapping rule. The following example illustrates the essentials.

```
# one script to be executed under the account
SET  /cgi-bin/a_big_script*  script=as=BIG_ACCOUNT
# all scripts in this area to be executed under this account
SET  /database-bin/*  script=as=DBACCNT
```

User Account Scripting

In some situations it may be desirable to allow the average Web user to experiment with or implement scripts. If the ‘‘script=as=’’ mapping rule specifies a circumflex character then for a user request the mapped SYSUAF username is substituted.

The following example shows the essentials of setting up a user environment where access to a subdirectory in the user’s home directory, [.WWW] with script’s located in a subdirectory of that, [.WWW.CGI-BIN].

```
SET   /~*/www/cgi-bin/*  script=as=~
UXEC  /~*/cgi-bin/*  /*/www/cgi-bin/*
USER  /~*/*  /*/www/*
REDIRECT  /~*  /~*/
PASS  /~*/*  /dka0/users/*/*
```

To enable user CGIplus scripting include something like

```
UXEC+  /~*/cgiplus-bin/*  /*/www/cgi-bin/*
```

Authenticated User Scripting

If the ‘‘script=as=’’ mapping rule specifies a dollar then a request that has been SYSUAF authenticated has the SYSUAF username substituted.

```
SET   /cgi-bin/cgi_process  script=as=$
```

1.1.1.2 Restricting Non-Server Scripting

By default, activating the /PERSONA server startup qualifier allows all the modes described above to be deployed using appropriate mapping rules. Of course there may be circumstances where such broad capabilities are inappropriate or otherwise undesirable. It is possible to control which user accounts are able to be used in this fashion with a rights identifier. Only those accounts granted the identifier can have scripts activated under them. This means all accounts . . . including the server account!

This is enabled by specifying the name of a rights identifier as a parameter to the /PERSONA qualifier. This may be any identifier but the one shown in the following example is probably as good as any.

```
$ HTTPD /PERSONA=WASD_SCRIPTING
```

This identifier could be created using the following commands

```
$ SET DEFAULT SYS$SYSTEM
$ MCR AUTHORIZE
UAF> ADD /IDENTIFIER WASD_SCRIPTING
```

and granted to accounts using

```
UAF> GRANT /IDENTIFIER WASD_SCRIPTING HTTP$SERVER
```

1.1.1.3 Process Priorities

When detached processes are created they can be assigned differing priorities depending on the origin and purpose. The objective is to give the server process a slight advantage when competing with scripts for system resources. This allows the server to respond to new requests more quickly (reducing latency) even if a script may then take some time to complete the request.

The allocation of base process priorities is determined from the HTTPD$CONFIG [DclDetach-ProcessPriority] configuration directive, which takes one or two (comma-separated) integers that determine how many priorities lower than the server scripting processes are created. The first integer determines server processes. A second, if supplied, determines user scripts. User scripts may never be a higher priority that server scripts. The following provides example directives.

```
[DclDetachProcessPriority]  1
[DclDetachProcessPriority]  0,1
[DclDetachProcessPriority]  1,2
```

Scripts executed under the server account, or those created using a mapped username (i.e. ‘‘script=as=*username*’’), have a process priority set by the first/only integer.

Scripts activated from user mappings (i.e. ‘‘script=as=~’’ or ‘‘script=as=$’’) have a process priority set by any second integer, or fall back to the priority of the first/only integer.

1.1.2 Subprocess Scripting

WASD’s default (and historical) scripting environment is with subprocesses created by the server.

With persistent subprocess scripting the pooled-resource BYTLM can become a particular issue. After the first subprocess-based script is executed the WATCH report provides some information on the BYTLM required to support both the desired number of incoming network connections and script subprocess IPC mailboxes. When using these numbers to resource the BYTLM quota of the server account keep in mind that as well as server-subprocess IPC consumption of BYTLM there may be additional requirements whatever processing is performed by the script.

For a standard configuration 15,000 bytes should be allowed for each possible script subprocess, 1,000 bytes for each potential client network connection, an additional 20,000 bytes overhead, plus any additional requirements for script processing, etc. Hence for a maximum of 30 scripts and 100 network clients, a BYTLM of approximately 260,000 minimum should be allowed.

Subprocess Environment

When the subprocess is spawned by the server none of the parent's environment is propagated. Hence the subprocess has no symbols, logical names, etc., created by the site's SYLOGIN.COM, the server account's LOGIN.COM, etc. This is done quite deliberately to provide a pristine and standard default environment for the script's execution. For this reason all scripts must provide all of their required environment to operate. In particular, if a verb is made available via a SY/LOGIN.COM this will not be available to the script. Verbs available via the DCLTABLES.EXE or DCL$PATH of course will be available.

There are two basic methods for supplying a script with a required environment.

- Create a DCL *wrapper* procedure that explicitly sets up the required environment, assigns required foreign verbs, etc. If a request does not specify a script type (i.e. a .EXE, .COM, .PL, etc.) the server always searches for a .COM first. Hence a script DCL wrapper procedure with the same name as the script itself will always be found and executed first. It can set up the required environment and then activate the actual script itself (see Wrapping Local or Third-Party Scripts).
- Use the HTTPD$LOGIN procedure to establish a standard environment for all scripts (see Section 1.4).

Caution!

When scripts are executed within unprivileged subprocesses created by the HTTP server, the processes are owned by the HTTP server account (HTTP$SERVER). Script actions could potentially affect server behaviour. For example it is possible for a script to issue an ''HTTPD/DO=EXIT=NOW'' command, or for subprocesses to create or modify logical name values in the JOB table (e.g. change the value of LNM$FILE_DEV altering the logical search path). Obviously these types of actions are undesirable. In addition scripts can access any WORLD-readable and modify any WORLD-writable resource in the system/cluster, opening a window for information leakage or mischievous/malicious actions (some might argue that anyone with important WORLD-accessable resources on their system deserves all that happens to them - but we know they're out there :^) Script authors should be aware of any potential side-effects of their scripts and Web administrators vigilant against possible malicious behaviours of scripts they do not author.

1.2 Script Mapping

Scripts are enabled using the *exec*/*uxec* or *script* rules in the mapping file (also see ''Technical Overview, Mapping Rules''). The script portion of the *result* must be a URL equivalent of the physical VMS procedure or executable specification.

All files in a directory may be mapped as scripts using the *exec* rule. For instance, in the HTTPD$MAP configuration file can be found a rule

```
exec /cgi-bin/* /cgi-bin/*
```

which results in request paths beginning ''/cgi-bin/'' having the following path component mapped as a script. Hence a path ''/cgi-bin/cgi_symbols.com'' will result in the server attempting to execute a file named CGI-BIN:[000000]CGI_SYMBOLS.COM.

Multiple such paths may be designated as *exec*utable, with their contents expected to be scripts, either directly executable by VMS (e.g. .EXEs and .COMs) or processable by a designated interpreter, etc., (e.g. .PLs, .CLASSes). See Section 1.3 below.

In addition individual files may be specified as scripts. This is done using the *script* rule. In the following example the request path ''/help'' activates the ''Conan The Librarian'' script.

```
script /help* /cgi-bin/conan*
```

Of course, multiple such rules may be used to map such abbreviated or self-explanatory script paths to the actual script providing the application.

Mapping Local or Third-Party Scripts

It is not necessary to move/copy scripts into the server directory structure to make them accessable. In fact there are probably good reasons for not doing so! For instance, it keeps a package together so that at the next upgrade there is no possibility of the ''server-instance'' of that application being overlooked.

To make scripts provided by third party packages available for server activation two requirements must be met.

- The server account must have read/execute access to any and all images and other resources required to use the application.
- Mapping rules must exist to make the script and any required resources accessable.

Most packages having such an interface for Web server access would provide details on mapping into the package directory. For illustration the following mapping rules provide access to a package's scripts (assuming it provides more than one) and also into a documentation area.

The hypothetical ''Application X'' directory locations are

```
APPLICATIONX_ROOT:[DOC]
APPLICATIONX_ROOT:[CGI-BIN]
```

The required mapping rules would be

```
pass /applicationX/* /applicationX_root/docs/*
exec /appX-bin/* /applicationX_root/cgi-bin/*
```

Access to X's scripts would be using a path such as

```
http://the.host.name/appx-bin/main_script?plus=some&query=string
```

"Wrapping" Local or Third-Party Scripts

Sometimes it may be necessary to provide a particular non-WASD, local, or third-party script with particular environment in which to execute. This can be provided by *wrapping* the script executable or interpreted script in a DCL procedure (of course, if the local or third-party script is already activated by a DCL procedure, then that may need to be directly modified). Simply create a DCL procedure, in the same directory as the script executable, containing the required environmental commands.

For example, the following DCL procedure defines a scratch directory and provides the location of the configuration file. It is assumed the script executable is APPLICATIONX_ROOT:[CGI-BIN]APPX.EXE and the script wrapper APPLICATIONX_ROOT:[CGI-BIN]APPX.COM.

```
$! wrapper for APPX CGI executable
$ SET DEFAULT APPLICATIONX_ROOT:[000000]
$ DEFINE /USER SYS$SCRATCH APPLICATIONX_ROOT:[SCRATCH]
$ APPX == "$APPLICATIONX_ROOT:[CGI-BIN]APPX"
$ APPX /CONFIG=APPLICATIONX_ROOT:[CONFIG]APPX.CONF
```

1.3 Script Run-Time

A script is merely an executed or interpreted file. Although by default VMS executables and DCL procedures can be used as scripts, other environments may also be configured. For example, scripts written for the Perl language may be transparently given to the Perl interpreter in a script subprocess. This type of script activation is based on a unique file type (extension following the file name), for the Perl example this is most commonly ''.PL'', or sometimes ''.CGI''. Both of these may be configured to automatically invoke the site's Perl interpreter, or any other for that matter.

This configuration is performed using the HTTPD$CONFIG [DclScriptRunTime] directive, where a file type is associated with a run-time interpreter. This parameter takes two components, the file extension and the run-time verb. The verb may be specified as a simple, globally-accessable verb (e.g. one embedded in the CLI tables), or in the format to construct a *foreign-verb*, providing reasonable versatility. Run-time parameters may also be appended to the verb if desired. The server ensures the verb is foreign-assigned if necessary, then used on a command line with the script file name as the final parameter to it.

The following is an example showing a Perl interpreter being specified. The first line assumes the ''Perl'' verb is globally accessable on the system (e.g. perhaps provided by the DCL$PATH logical) while the second (for the sake of illustration) shows the same Perl interpreter being configured for a different file type using the foreign verb syntax.

```
[DclScriptRunTime]
.PL PERL
.CGI $PERL_EXE:PERL
```

A file contain a Perl script then may be activated merely by specifying a path such as the following

```
/cgi-bin/example.pl
```

To add any required parameters just append them to the verb specified.

```
[DclScriptRunTime]
.XYZ XYZ_INTERPRETER -vms -verbose -etc
.XYZ $XYZ_EXE:XYZ_INTERPRETER /vms /verbose /etc
```

If a more complex run-time interpreter is required it may be necessary to *wrap* the script's execution in a DCL procedure.

Script File Extensions

The WASD server does not require a file type (extension) to be explicitly provided when activating a script. This can help hide the implementation detail of any script. If the script path does not contain a file type the server searches the script location for a file with one of the known file types, first ''.COM'' for a DCL procedure, then ''.EXE'' for an executable, then any file types specified using script run-time configuration directive, in the order specified.

For instance, the script activated in the Perl example above could have been specified as below and (provided there was no ''EXAMPLE.COM'' or ''EXAMPLE.EXE'' in the search) the same script would have been executed.

```
/cgi-bin/example
```

1.4 Scripting Logicals

Two logicals provide some control of and input to the DCL subprocess scripting environment (which includes standard CGI, CGIplus and ISAPI, DECnet-based CGI, but excludes DECnet-based OSU).

- **HTTPD$LOGIN -** Specifies the location of a command procedure that can be executed immediately before the script procedure/image/script-file is activated. This is intended for the provision of a common per-site environment, etc., but could be used for any purpose. Activate using a system-wide logical as in the following example.

  ```
  $ DEFINE /SYSTEM HTTPD$LOGIN HT_ROOT:[LOCAL]HTTPD$LOGIN.COM
  ```

  Note that each layer of execution added to the scripting environment increases both system overhead and response latency.

- **HTTPD$VERIFY -** Activates DCL verify for the DCL subprocess scripting environment. This shows the DCL commands used to support script activation. Intended for problem investigation.

  ```
  $ DEFINE /SYSTEM HTTPD$VERIFY 1
  ```

Note that most WASD scripts also contain logical names that can be set for debugging purposes. These are generally in the format *script_name*$DBUG and if exist activate debugging statements throughout the script.

1.5 Scripting Scratch Space

Scripts often require temporary file space during execution. Of course this can be located anywhere the scripting account (most often HTTP$SERVER) has appropriate access. The WASD package does provide a default area for such purposes with permissions set during startup to allow the server account full access. The default area is located in

```
HT_ROOT:[SCRATCH]
```

as is accessed by the server and scripts using the logical name

```
HT_SCRATCH:
```

The server provides for the routine clean-up of old files in HT_SCRATCH: left behind by aborted or misbehaving scripts (although as a matter of design all scripts should attempt to clean up after themselves). The HTTPD$CONFIG directives

```
[DclCleanupScratchMinutesMax]
[DclCleanupScratchMinutesOld]
```

control how frequently the clean-up scan occurs, and how old files need to be before being deleted. Whenever script processes are active the scratch area is scanned at the maximum period specified, or whenever the last script process is purged from the system by the server.

Of course there is always the potential for interaction between scripts using a common area for such purposes. At the most elemetary, care must be taken to ensure unique file name are generated. At worst there is the potential for malicious interaction and information leakage. Use such common areas with discretion.

Unique File Names - DCL

The ''UNIQUE_ID'' CGI variable provides a unique 19 character alpha-numeric string (see UNIQUE_ID Note) suitable for many uses including the type extension of temporary files. The following DCL illustrates the essentials of generating a script-unqiue file name. For mutliple file names add further text to the type, as shown below.

```
$ SCRATCH_DIR = "HT_SCRATCH:"
$ PROC_NAME = F$PARSE(F$ENVIRONMENT("PROCEDURE"),,,"NAME")
$ INFILE_NAME = SCRATCH_DIR + PROC_NAME + "." + WWW_UNIQUE_ID + "_IN"
$ OUTFILE_NAME = SCRATCH_DIR + PROC_NAME + "." + WWW_UNIQUE_ID + "_OUT"
```

Unique File Names - C Language

A similar approach can be used for script coded using the C language, with the useful capacity to mark the file for delete-on-close (of course this is only really useful if it's, say, only to be written, rewound and then re-read without closing first - but I'm sure you get the idea).

```
#define HT_SCRATCH "HT_SCRATCH:"
#define SCRIPT_NAME "EXAMPLE"

char  *unqiueId;
char  tmpFileName [256];
FILE  *tmpFile;
```

```
if ((uniqueId = getenv("WWW_UNIQUE_ID")) == NULL)
{
   printf ("Error: WWW_UNIQUE_ID absent!\n");
   exit (1);
}
sprintf (tmpFileName, HT_SCRATCH SCRIPT_NAME ".%s", uniqueId);

if ((tmpFile = fopen (tmpFileName, "w+", "fop=dlt")) == NULL)
   exit (vaxc$errno);
```

1.6 DCL Processing of Requests

To assist with the processing of request content and response generation from within DCL procedures the CGIUTL utility is available in HT_ROOT:[SRC.MISC]

Functionality includes

- decode a POSTed request body into DCL symbols
- write a POSTed request body to a file
- *massage* DCL symbol quotation characters
- generate HTTP response headers
- *binary* transfer of file contents

Most usefully it can read the request body, decoding form-URL-encoded contents into DCL symbols and/or a scratch file, allowing a DCL procedure to easily and effectively process this form of request.

1.7 Scripting Function Library

A source code collection of C language functions useful for processing the more vexing aspects of CGI and general script programming is available in CGILIB. This and an example implementation is available in HT_ROOT:[SRC.MISC]

Functionality includes

- detection and appropriate initialization of the scripting environment, including WASD, CGIplus, VMS Apache, OSU, Purveyor, and possibly other CGI (e.g. Cern, Netscape FastTrack)
- transparent access to CGI variables
- transparent access to the body of a POSTed request, both URL-encoded and MIME multipart/form-data (from <INPUT TYPE=file> upload tags)
- placing the contents of a form-URL-encoded or multipart/form-data body into CGI-like environment variables
- URL-decoding, URL-encoding and HTML-escaping a string
- CGIplus-specific functionality (including callouts)

The WASD scripts use this library extensively and may serve as example applications.

1.8 HTTP Persistant-State Cookies

The WASD server is *cookie-aware*. That is, if the client supplies a ‘‘Cookie:’’ request header line it is passed to a CGI script as ‘‘WWW_HTTP_COOKIE’’ CGI variable symbol. If a cookie is not part of the request this symbol does not exist. A script may use the ‘‘Set-Cookie:’’ response header line to set cookies.

Here is a small demonstration of cookie processing using a DCL script.

online demonstration

Chapter 2
CGI

The information in this chapter merely outlines the WASD implementation details, which are in general very much vanilla CGI and NCSA CGI (Common Gateway Interface) compliant, originally based the INTERNET-DRAFT authored by D.Robinson (drtr@ast.cam.ac.uk), 8 January 1996, and confirmed against the latest INTERNET-DRAFT authored by Ken A.L.Coar (drtr@etrade.co.uk), 12 June 1999.

2.1 CGI Environment Variables

With the standard CGI environment variables are provided to the script via DCL global symbols. Each CGI variable symbol name is prefixed with ''WWW_'' (by default, although this can be changed using the ''/CGI_PREFIX'' qualifier and the SET CGIPREFIX mapping rule, see ''Technical Overview'', this is not recommended if the WASD VMS scripts are to be used, as they expect CGI variable symbols to be prefixed in this manner).

In line with other CGI implementations, additional, non-compliant variables are provided to ease CGI interfacing. These provide the various components of the query string. A *keyword* query string and a *form* query string are parsed into separated variables, named

```
WWW_KEY_number
WWW_KEY_COUNT
WWW_FORM_form-element-name
```

plus a number of non-''standard'' CGI variables to assist in tailoring scripts for the WASD environment. Do not make your scripts dependent on any of these if portability is a goal. See the table below.

VMS Apache Compliance

WASD v7.0 had it's CGI environment tailored slightly to ease portability between VMS Apache and WASD. This included the provision of an APACHE$INPUT: stream and several Apache-specific CGI variables (see the table below). The CGILIB C function library (see Section 1.7) has also been made CSWS V1.0-1 (Apache 1.3.12) compliant.

VMS Limits

CGI variable values are limited to approximately 980 characters (regardless of the version of VMS in use). This should be sufficient for most circumstances (if not consider using CGIplus or ISAPI, extensions to CGI programming which remove this limitation). Why such an odd number and why a little rubbery? A DCL command line is limited to 255 characters so the symbols for larger variables are built up over successive DCL commands. The limit of 980 characters is set by *what can actually be achieved* this way (still, it's much more useful than a limit of 255 or less!).

The total length of all CGI variable names and values is determined by the value of [BufferSizeDclCommand] configuration directive, which determines the total buffer space of a mailbox providing the script's SYS$COMMAND. The default value of 4096 bytes will be ample for the typical CGI script request, however if it contains an extraordinary number of form fields, etc., it may be possible to exhaust this buffer space.

CGI Variables

Remember, all variables are prefixed by ''WWW_'', and not all variables will be present for all requests.

Name	Description	''Standard''
AUTH_ACCESS	''READ'' or ''READ+WRITE''	WASD
AUTH_AGENT	only present to and used by authorization agent	WASD
AUTH_GROUP	authentication group	WASD
AUTH_PASSWORD	plain-text password, only if EXTERNAL realm	WASD
AUTH_REALM	authentication realm	WASD
AUTH_REALM_DESCRIPTION	browser displayed string	WASD
AUTH_TYPE	authentication type (BASIC or DIGEST)	yes
AUTH_USER	details of authenticated user	WASD
CONTENT_LENGTH	''Content-Length:'' from request header	yes
CONTENT_TYPE	''Content-Type:'' from request header	yes
DOCUMENT_ROOT	always empty for WASD	Apache
FORM_field	query string ''&'' separated form elements	no
GATEWAY_BG	device name ''BG*nnnn*:'' of raw client socket	WASD
GATEWAY_INTERFACE	''CGI/1.1''	yes
GATEWAY_MRS	maximum record size of script SYS$OUTPUT	WASD
HTTP_ACCEPT	any list of browser-accepted content types	optional

Name	Description	''Standard''
HTTP_ACCEPT_CHARSET	any list of browser-accepted character sets	optional
HTTP_ACCEPT_LANGUAGE	any list of browser-accepted languages	optional
HTTP_AUTHORIZATION	any from request header	optional
HTTP_COOKIE	any cookie sent by the client	optional
HTTP_FORWARDED	any proxy/gateway hosts that forwarded the request	optional
HTTP_HOST	host and port request was sent to	optional
HTTP_IF_MODIFIED_SINCE	any last modified GMT time string	optional
HTTP_PRAGMA	any pragma directive of request header	optional
HTTP_REFERER	any source document URL for this request	optional
HTTP_USER_AGENT	client/browser identification string	optional
KEY_n	query string ''+'' separated elements	no
KEY_COUNT	number of ''+'' separated elements	no
PATH_INFO	virtual path of data requested in URL	yes
PATH_TRANSLATED	VMS file path of data requested in URL	yes
QUERY_STRING	un-URL-decoded string following ''?'' in URL	yes
REMOTE_ADDR	IP host address of HTTP client	yes
REMOTE_HOST	IP host name of HTTP client	yes
REMOTE_USER	authenticated remote user name (or empty)	yes
REQUEST_CHARSET	any server-determined request character set	WASD
REQUEST_METHOD	''GET'', ''PUT'', etc.	yes
REQUEST_SCHEME	''http:'' or ''https:''	WASD
REQUEST_TIME_GMT	GMT time request received	WASD
REQUEST_TIME_LOCAL	Local time request received	WASD
REQUEST_URI	full, unescaped request string	Apache
SCRIPT_FILENAME	script file name (e.g. ''CGI-BIN:[000000]QUERY.COM'')	Apache
SCRIPT_NAME	script being executed (e.g. ''/query'')	yes
SERVER_ADDR	IP host name of server system	WASD
SERVER_ADMIN	email address for server administration	Apache
SERVER_CHARSET	server default character set	WASD

Name	Description	‘‘Standard’’
SERVER_GMT	offset from GMT (e.g. ‘‘+09:30’’)	WASD
SERVER_NAME	IP host name of server	yes
SERVER_PROTOCOL	HTTP protocol version (always ‘‘HTTP/1.0’’)	yes
SERVER_PORT	IP port request was received on	yes
SERVER_SIGNATURE	server ID, host name and port	Apache
SERVER_SOFTWARE	software ID of HTTP server	yes
UNIQUE_ID	request-unique 19 character string	Apache

If the request path is set to provide them, there are also be variables providing information about a Secure Sockets Layer transported request’s SSL environment.

CGI Variable Demonstration

The basic CGI symbol names are demonstrated here with a call to a script that simply executes the following DCL code:

```
$ SHOW SYMBOL WWW_*
$ SHOW SYMBOL *
```

Note how the request components are represented for ‘‘ISINDEX’’-style searching (third item) and a forms-based query (fourth item).

online demonstration

UNIQUE_ID Note

The UNIQUE_ID variable is a mostly Apache-compliant implementation (the ‘‘_’’ has been substituted for the ‘‘@’’ to allow it’s use in file names), for each request generating a globally and temporally unique 19 character string that can be used where such a identifier might be needed. This string contains only ‘‘A’’-‘‘Z’’, ‘‘a’’-‘‘z’’, ‘‘0’’-‘‘9’’, ‘‘_’’ and ‘‘-’’ characters and is generated using a combination of time-stamp, host IP address, server system process identifier and counter, and is ‘‘guaranteed’’ to be unique in (Internet) space and time.

2.2 Script Output

This information applies to all subprocess-based scripting, CGI, CGIplus, RTE, ISAPI. For subprocess scripting WASD uses mailboxes for inter-process communication (IPC). These are efficient, versatile and allow direct output from all VMS environments and utilities. Like many VMS record-oriented devices however there are some things to consider when using them.

- **Record-Oriented**

The mailboxes are created record, not stream oriented. This means records output by standard VMS means (e.g. DCL, utilities, programming languages) are discretely identified and may be processed appropriately by the server as text or binary depending on the content-type.

- **Maximum Record Size**

  Being record oriented there is a maximum record size (MRS) that can be output. Records larger than this result in SYSTEM-F-MBTOOSML errors. The WASD default is 4096 bytes. This may be changed using the [BufferSizeDclOutput] configuration directive. This allocation consumes process BYTLM with each mailbox created so the account must be dimensioned sufficiently to supply demands for this quota. The maximum possible size for this is a VMS-limit of 60,000 bytes.

- **Buffer Space**

  When created the mailbox has it's buffer space set. With WASD IPC mailboxes this is the same as the MRS. The total data buffered may not exceed this without the script entering a wait state (for the mailbox contents to be cleared by the server). As mailboxes use a little of the buffer space to delimit records stored in it the amount of data is actually less than the total buffer space.

To determine the maximum record size and total capacity of the mailbox buffer between server and script WASD provides a CGI environment variable, GATEWAY_MRS, containing an integer with this value.

2.2.1 CGI Compliant Output

Script response may be CGI or NPH compliant (see Section 2.2.2). CGI compliance means the script's response must begin with a line containing one of the following fields.

- **Status:** an HTTP status code and associated explanation string
- **Content-Type:** the script body's MIME content-type
- **Location:** a redirection URL

Other HTTP-compliant response fields may follow, with the response header terminated and the response body begun by a single empty line. The following is an example of a CGI-compliant response.

```
Content-Type: text/html
Content-Length: 35

<HTML>
<B>Hello world!</B>
</HTML>
```

Strict CGI output compliance can be enabled and disabled using the [CgiStrictOutput] configuration directive. With it disabled the server will accept any output from the script, if not CGI or NPH compliant then it automatically generates plain-text header. When enabled, if not a CGI or NPH header the server returns a ''502 Bad Gateway'' error. For debugging scripts generating this error introduce a *plain-text* debug mode and header, or use the WATCH facility's CGI item (see the Technical Overview).

WASD Specifics

This section describes how WASD deals with some particular output issues.

- **Content-Type: text/ . . .**

  If the script response content-type is ‘‘text/ . . . ’’ (text document) WASD assumes that output will be line-oriented and requiring HTTP carriage-control (each record/line terminated by a line-feed), and will ensure each record it receives is correctly terminated before passing it to the client. In this way DCL procedure output (and the VMS environment in general) is supported transparently. Any other content-type is assumed to be *binary* and no carriage-control is enforced. This default behaviour may be modified as described below.

- **Carriage-Control**

  Carriage-control behaviour for any content-type may be explicitly set using either of two additional response header fields. The term *stream* is used to describe the server just transfering records, without additional processing, as they were received from the script. This is obviously necessary for binary/raw content such as images, octet-streams, etc. The term *record* describes the server ensuring each record it receives has correct carriage-control - a trailing newline. If not present one is added. This mode is useful for VMS textual streams (e.g. output from DCL and VMS utilities).

  Using the Apache Group’s proposed CGI/1.2 ‘‘Script-Control:’’ field. The WASD extension-directives *X-record-mode* and *X-stream-mode* sets the script output into each of the respective modes. See Script-Control:.

  Examples of usage this field:

  ```
  Script-Control: X-stream-mode
  Script-Control: X-record-mode
  ```

- **Script Output Buffering**

  By default WASD writes each record received from the script to the client as it is received. This can range from a single byte to a complete mailbox buffer full. WASD leaves it up to the script to determine the rate at which output flows back to the client.

  While this allows a certain flexibility it can be inefficient. There will be many instances where a script will be providing just a body of data to the client, and wish to do it as quickly and efficiently as possible. Using the proposed CGI/1.2 ‘‘Script-Control:’’ field with the WASD extension directive *X-buffer-records* a script can direct the server to buffer as many script output records as possible before transfering it to the client. The following should be added to the CGI response header.

  ```
  Script-Control: X-buffer-records
  ```

  While the above offers some significant improvements to efficiency and perceived throughput the best approach is for the script to provide records the same size as the mailbox (see Section 2.2 for detail on determining this size if required). The can be done explicitly by the script programming or if using the C language simply by changing *stdout* to a binary stream. With this environment the C-RTL will control output, automatically buffering as much as possible before writing it to the server.

```
if ((stdout = freopen ("SYS$OUTPUT", "w", stdout, "ctx=bin")) == NULL)
   exit (vaxc$errno);
```

Also see the section describing NPH C Script.

Script-Control:

The *Apache Group* has proposed a CGI/1.2 that includes a *Script-Control:* CGI response header field. WASD implements the one proposed directive, along with a number of WASD extensions (those beginning with the ''X-''). Note that by convention extensions unknown by an agent should be ignored, meaning that they can be freely included, only being meaningful to WASD and not significant to other implementations.

- **no-abort -** The server must not terminate the script during processing for either no output or no-progress timeouts. The script is to be left completely alone to control it's own termination. Caution, such scripts if problematic could easily accumulate and ''clog up'' a server or system.
- **X-buffer-records -** Buffer records written by the script until there is [BufferSizeDclOutput] bytes available then write it as a single block to the client.
- **X-crlf-mode -** The server should always ensure each record has trailing carriage-return then newline characters (0x0d, 0x0a). This is generally what VMS requires for carriage control on terminals, printers, etc.
- **X-lifetime=*value* -** The number of minutes before the idle script subprocess is deleted by the server. Zero sets it back to the default, ''none'' disables this functionality.
- **X-record-mode -** The server should always ensure each record has a trailing newline character (0x0a), regardless of whether the response is a *text/...* content-type or not. This is what is usually required by browsers for carriage-control in text documents.
- **X-stream-mode -** The server is not to adjust the carriage-control of records regardless of whether the response is a *text/...* content-type or not. What the script writes is exactly what the client is sent.
- **X-timeout-noprogress=*value* -** The number of minutes allowed where the script does not transfer any data to the server before the server deletes the process. Zero sets it back to the default, ''none'' disables this functionality.
- **X-timeout-output=*value* -** The number of minutes allowed before an active script is deleted by the server, regardless of it still processing the request. Zero sets it back to the default, ''none'' disables this functionality.

The following is a simple example response where the server is instructed not to delete the script process under any circumstances, and that the body does not require any carriage-control changes.

```
Content-Type: text/plain
Script-Control: no-abort; X-stream-mode

long, slowww script-output  . . .
```

Example DCL Scripts

A simple script to provide the system time might be:

```
$ say = "write sys$output"
$! the next two lines make it CGI-compliant
$ say "Content-Type: text/plain"
$ say ""
$! start of plain-text body
$ show time
```

A script to provide the system time more elaborately (using HTML):

```
$ say = "write sys$output"
$! the next two lines make it CGI-compliant
$ say "Content-Type: text/html"
$ say ""
$! start of HTML script output
$ say "<HTML>"
$ say "Hello ''WWW_REMOTE_HOST'"  !(CGI variable)
$ say "<P>"
$ say "System time on node ''f$getsyi("nodename")' is:"
$ say "<H1>''f$cvtime()'</H1>"
$ say "</HTML>"
```

2.2.2 Non-Parsed-Header Output

A script does not have to output a CGI-compliant data stream. If it begins with a HTTP header status line WASD assumes it will supply a **raw** HTTP data stream, containing all the HTTP requirements. This is the same or equivalent to the *non-parsed-header*, or ‘‘nph . . . ’’ scripts of many environments. This is an example of such a script response.

```
HTTP/1.0 200 Success
Content-Type: text/html
Content-Length: 35

<HTML>
<B>Hello world!</B>
</HTML>
```

Any such script must observe the HyperText Transfer Protocol, supplying a **full response header and body, including correct carriage-control**. Once the server detects the HTTP status header line it pays no more attention to any response header fields or body records, just transfering everything directly to the client. This can be very efficient, the server just a conduit between script and client, but does transfer the responsibility for a correct HTTP response onto the script.

NPH DCL Script

The following example shows a DCL script. Note the full HTTP header and each line explicitly terminated with a carriage-return and line-feed pair.

```
$ lf[0,8] = %x0a
$ crlf[0,16] = %x0d0a
$ say = "write sys$output"
$! the next line determines that it is raw HTTP stream
$ say "HTTP/1.0 200 Success" + crlf
$ say "Content-Type: text/html" + crlf
$! response header separating blank line
$ say crlf
$! start of HTML script output
$ say "<HTML>" + lf
$ say "Hello ''WWW_REMOTE_HOST'" + lf
$ say "<P>" + lf
$ say "Local time is ''WWW_REQUEST_TIME_LOCAL'" + lf
$ say "</HTML>" + lf
```

CGIUTL Utility

This assists with the generation of HTTP responses, including the transfer of binary content from files (copying a file back to the client as part of the request), and the processing of the contents of POSTed requests from DCL. See Section 1.6.

NPH C Script

When scripting using the C programming language there can be considerable efficiencies to be gained by providing a binary output stream from the script. This results in the C Run-Time Library (C-RTL) buffering output up to the maximum supported by the IPC mailbox. This may be simply enabled using a code construct similar to following to reopen *stdout* in binary mode.

```
if ((stdout = freopen ("SYS$OUTPUT", "w", stdout, "ctx=bin")) == NULL)
    exit (vaxc$errno);
```

This is used consistently in WASD scripts. Carriage-control must be supplied as part of the C standard output (no differently to any other C program). Output can be be explicitly sent to the client at any stage using the *fflush( )* standard library function. Note that if the *fwrite( )* function is used the current contents of the C-RTL buffer are automatically flushed along the the content of the fwrite( ).

```
    fprintf (stdout,
"HTTP/1.0 200 Success\r\n\
Content-Type: text/html\r\n\
\r\n\
<HTML>\n\
Hello %s\n\
<P>\n\
System time is %s\n\
</HTML>\n",
    getenv("WWW_REMOTE_HOST"),
    getenv("WWW_REQUEST_TIME_LOCAL"));
```

CGI Function Library

A source code collection of C language functions useful for processing the more vexing aspects of CGI/CGIplus programming. See Section 1.7.

2.3 Raw HTTP Input (POST Processing)

For POST and PUT HTTP methods (e.g. a POSTed HTML form) the body of the request may be read from the HTTP$INPUT stream. For executable image scripts requiring the body to be present on SYS$INPUT (the C language *stdin* stream) a user-mode logical may be defined immediately before invoking the image, as in the example.

```
$ EGSCRIPT = "$HT_EXE:EGSCRIPT.EXE"
$ DEFINE /USER SYS$INPUT HTTP$INPUT
$ EGSCRIPT
```

The HTTP$INPUT stream may be explicitly opened and read. Note that this is a raw stream, and HTTP *lines* (carriage-return/line-feed terminated sequences of characters) may have been blocked together for network transport. These would need to be explicity parsed by the program.

Chapter 3
CGIplus

Common Gateway Interface ... plus lower latency, plus greater efficiency, plus far less system impact!

I know, I know! The term CGIplus is a bit too cute but I had to call it something!

CGIplus attempts to eliminate the overhead associated with creating the subprocess and then executing the image of a CGI script. It does this by allowing the subprocess and any associated image/application to continue executing between uses, eliminating any startup overheads. This reduces both the load on the system and the request latency. In this sense these advantages parallel those offered by commercial HTTP server-integration APIs, such as Netscape NSAPI and Microsoft ISAPI, without the disadvantages of such proprietory interfaces, the API complexity, language dependency and server process integration.

Existing CGI scripts can rapidly and elegantly be modified to additionally support CGIplus. The capability of scripts to easily differentiate between and operate in both standard CGI and CGIplus environments with a minimum of code revision offers great versatility. Many WASD scripts operate in both environments.

CGIplus Performance

A simple performance evaluation indicates the advantage of CGIplus. See ‘‘Techncial Overview, Performance’’ for some test results comparing the CGI and CGIplus environments.

Without a doubt, the subjective difference in activating the same script within the standard CGI and CGIplus environments is quite startling!

3.1 CGIplus Programming

The script interface is still CGI, with all the usual environment variables and input/output streams available, which means a new API does not need to be learned and existing CGI scripts are simple to modify.

See examples in HT_ROOT:[SRC.CGIPLUS]

on-line hypertext link

Instead of having the CGI variables available from the environment (generally accessed via the C Language *getenv( )* standard library call) a CGIplus script must read the CGI variables from CGIPLUSIN. They are supplied as a series of records (lines) containing a CGI variable name (in upper-case), an equate symbol and then the variable value. The format may be easily parsed and as the value contains no encoded characters may be directly used. The quantity of characters in each record depends on the size of the variable name and the length of the associated value. The value can vary from zero, to tens, hundreds, even thousands of characters. It is limited by the size of the CGIPLUSIN mailbox, which is in turn set by the [BufferSizeDclCgiPlusIn] configuration directive.

Requirements when using:

- The read will block between subsequent requests and so may be used to coordinate the application.
- The first record read in any request can always be discarded. This is provided so that a script may be synchronized outside of the general CGIplus variable read loop (the DCL and Perl examples use this feature).
- The CGIplus variable stream should be completely read (up until the blank line, see below) before beginning any request processing.
- The CGIplus variable stream should ALWAYS BE COMPLETELY READ (up until the blank line, see below).
- An empty record (blank line) indicates the end of a single request's CGIplus variable stream. Reading MUST be halted at this stage. Request processing may then commence.

After processing, the CGIplus script can loop, waiting to read the details of the next request from CGIPLUSIN.

Request output (to the client) is written to SYS$OUTPUT (<stdout>) as per normal CGI behaviour. **End of output MUST be indicated by writing a special EOF record to the output stream.** A unique EOF sequence is generated for each use of DCL via a zombie or CGIplus subprocess. A non-repeating series of bits most unlikely to occur in normal output is employed . . . but there is still a very, very, very small chance of premature termination of output (one in 2^224 I think!) See HT_ROOT:[SRC.HTTPD]CGI.C for how the value is generated.

The CGIplus EOF string is obtained by the script from the logical name CGIPLUSEOF, defined in the script subprocess' process table, using the scripting language's equivalent of F$TRNLNM( ), SYS$TRNLNM( ), or a getenv( ) call (in the C standard library). This string will always contain less than 64 characters and comprise only printable characters. It must be written at the conclusion of a request's output to the output stream as a single record (line) but may also contain a <CR><LF> or just <LF> trailing carriage-control (to allow for programming language requirements). It only has to be evaluated once, as the processing begins, remaining the same for all requests over the life-time of that instance of the script.

HTTP input (raw request body stream) is still available to a CGIplus script.

CGI Function Library

A source code collection of C language functions useful for processing the more vexing aspects of CGI/CGIplus/RTE programming. See Section 1.7.

3.2 Code Examples

Of course a CGIplus script should only have a single exit point and should explicitly close files, free allocated memory, etc., after processing a request (i.e. not rely on image run-down to clean-up after itself). It is particularly important when modifying existing scripts to work in the CGIplus environment to ensure this requirement is met (who of us hasn't thought ''well, this file will close when the image exits anyway''?)

It is a simple task to design a script to modify it's behaviour according to the environment it is executing in. Detecting the presence or absence of the CGIPLUSEOF logical is sufficient indication. The following C code fragment shows simultaneously determining whether it is a standard or CGIplus environment (and setting an appropriate boolean), and getting the CGIplus EOF sequence (if it exists).

```
int  IsCgiPlus;
char  *CgiPlusEofPtr;

IsCgiPlus = ((CgiPlusEofPtr = getenv("CGIPLUSEOF")) != NULL);
```

The following C code fragment shows a basic CGIplus request loop, reading lines from CGIPLUSIN, and some basic processing to select required CGI variables for request processing.

```
if (IsCgiPlus)
{
   char  *cptr;
   char  Line [1024],
         RemoteHost [128];
   FILE  *CgiPlusIn;

   if ((CgiPlusIn = fopen (getenv("CGIPLUSIN"), "r")) == NULL)
   {
      perror ("CGIplus: fopen");
      exit (0);
   }

   for (;;)
   {
      /* will block waiting for subsequent requests */
      for (;;)
      {
         /* should never have a problem reading CGIPLUSIN, but */
         if (fgets (Line, sizeof(Line), CgiPlusIn) == NULL)
         {
            perror ("CGIplus: fgets");
            exit (0);
         }
         /* first empty line signals the end of CGIplus variables */
         if (Line[0] == '\n') break;
         /* remove the trailing newline */
         if ((cptr = strchr(Line, '\n')) != NULL) *cptr = '\0';
```

```
            /* process the CGI variable(s) we are interested in */
            if (!strncmp (Line, "WWW_REMOTE_HOST=", 16))
               strcpy (RemoteHost, Line+16);
         }

         (process request, signal end-of-output)
      }
   }
```

CGI scripts can write output in record (line-by-line) or binary mode (more efficient because of buffering by the C RTL). When in binary mode the output stream must be flushed immediately before and after writing the CGIplus EOF sequence (note that in binary a full HTTP stream must also be used). This code fragment shows placing a script output stream into binary mode and the flushing steps.

```
/* reopen output stream so that the '\r' and '\n' are not filtered */
if ((stdout = freopen ("SYS$OUTPUT", "w", stdout, "ctx=bin")) == NULL)
   exit (vaxc$errno);

do {

   (read request ...)

   /* HTTP response header */
   fprintf (stdout, "HTTP/1.0 200 Success\r\nContent-Type: text/html\r\n\r\n");

   (other output ...)

   if (IsCgiPlus)
   {
      /* the CGIplus EOF must be an independant I/O record */
      fflush (stdout);
      fprintf (stdout, "%s", CgiPlusEofPtr);
      fflush (stdout);
   }

} while (IsCgiPlus);
```

If the script output is not binary (using default <stdout>) it is only necessary to ensure the EOF string has a record-delimiting new-line.

```
fprintf (stdout, "%s\n", CgiPlusEofPtr);
```

Other languages may not have this same requirement. DCL procedures are quite capable of being used as CGIplus scripts.

See examples in HT_ROOT:[SRC.CGIPLUS]

on-line hypertext link

Hint!

Whenever developing CGIplus scripts/applications (unlike standard CGI) don't forget that after compiling, the old image must be purged from the server before trying out the new!!! (I've been caught a number of times :^)

Scripting subprocesses may be purged or deleted using (see ''Techncial Overview, Server Command Line Control''):

```
$ HTTPD /DO=DCL=DELETE
$ HTTPD /DO=DCL=PURGE
```

3.3 Other Considerations

Multiple CGIplus scripts may be executing in subprocesses at any one time. This includes multiple instances of any particular script. It is the server's task to track these, distributing appropriate requests to idle subprocesses, monitoring those currently processing requests, creating new instances if and when necessary, and deleting the least-used, idle CGIplus subprocesses when configurable thresholds are reached. Of course it is the script's job to maintain coherency if multiple instances may result in resource conflicts or race conditions, etc., between the scripts.

The CGIplus subprocess can be given a finite life-time set by configuration parameter (see ''Technical Overview, Server Configuration''). If this life-time is not set then the CGIplus will persist indefinitely (i.e. until purged due to soft-limits being reached, or explicitly purged/deleted). When a life-time has been set the CGIplus subprocess is automatically deleted after being idle for the specified period (i.e. not having processed a request). This can be useful in preventing sporadically used scripts from cluttering up the system indefinitely.

In addition, an idle CGIplus script can be terminated by the server at any time the subprocess soft-limit is reached (the subprocess SYS$DELPRC( )ed) so resources should be largely quiescent when not actually processing. Of course a CGIplus subprocesses may also be manually terminated from the command line (e.g. STOP/ID=).

Some CGIplus scripting information and management is available via the server administration menu, see ''Technical Overview, Server Reports''.

CGIplus Rule Mapping

CGIplus scripts are differentiated from standard CGI scripts in the mapping rule configuration file using the ''script+'' and ''exec+'' directives. See ''Technical Overview, Mapping Rules''.

Scripts capable of operating in both standard CGI and CGIplus environments may simply be accessed in either via rules such as

```
exec /cgi-bin/* /cgi-bin/*
exec+ /cgiplus-bin/* /cgi-bin/*
```

while specific scripts can be individually designated as CGIplus using

```
script+ /cgiplus_example* /cgi-bin/cgiplus_example*
```

Caution! If changing CGIplus script mapping it is advised to restart the server rather than reloading the rules. Some conflict is possible when using new rules while existing CGIplus scripts are executing.

Chapter 4
Run-Time Environments

A Run-Time Environment (RTE) is a persistant scripting environment with similar objectives to CGIplus . . . reducing script response time, increasing server throughput and reducing system impact. In fact the RTE environment is implemented using CGIplus! There is very little difference in the behaviour of CGIplus scripts and RTEs. Both are activated by the server, process multiple requests (reading the request CGI environment from a data stream supplied by the server), persist between requests in a quiescent state, and may be removed by the server if idle for a specified period or when it wishes to use the process for some other purpose. **Like CGIplus an RTE must be purpose-written for the environment!** What is the difference then?

With CGIplus the script itself persists between uses, retaining all of it's state. With an RTE the script does not persist or retain state, only the RTE itself.

A RTE is intended as an environment in which a script source is interpreted or otherwise processed, although it is not limited to that. The essential difference between an RTE and a CGIplus script is this script source. In CGIplus the SCRIPT_NAME and SCRIPT_FILENAME CGI variables reflect the script itself, and remain constant for each activation of the script, with PATH_INFO and PATH_TRANSLATED providing the additional ''location'' information for the script processing. With an RTE the SCRIPT_NAME and SCRIPT_FILENAME can vary with each activation. This allows the RTE to process multiple, successive different (or the same) scripts, each with it's own PATH_INFO and PATH_TRANSLATED. Hence, it is not unreasonable to consider the two environments to be the same, with a slight difference in the mapping of resources passed to them.

This might be best illustrated with examples.

CGIplus Example

Consider the mapping rule

```
exec+ /cgiplus-bin/* /cgi-bin/*
```

applied to the following CGIplus request

```
/cgiplus-bin/xyz/the/path/information?and=a&query=string
```

If the script was an executable it would be activated as

```
CGI-BIN:[000000]XYZ.EXE
```

with script CGI information

```
/cgiplus-bin/xyz
CGI-BIN:[000000]XYZ.EXE
```

and the request path information and query string supplied as

```
/the/path/information
THE:[PATH]INFORMATION
and=a&query=string
```

RTE Example

By contrast with a request to activate an RTE the following mapping rule

```
exec /xyz-bin/* (CGI-BIN:[000000]XYZ.EXE)/ht_root/src/xyz/*
```

(note the RTE executable specified inside parentheses) and request

```
/xyz-bin/an_example/the/path/information?and=a&query=string
```

would activate the scripting environment (perhaps interpreter)

```
CGI-BIN:[000000]XYZ.EXE
```

supplying it with per-request script name and file information

```
/xyz-bin/an_example.xyz
HT_ROOT:[SRC.XYZ]AN_EXAMPLE.XYZ
```

and path and query string information

```
/the/path/information
THE:[PATH]INFORMATION
and=a&query=string
```

Summary

As can be seen the script information is constant for each request to a CGIplus script, while with RTE the script information could vary with each request (although of course it would be the same if the same script is requested). In the case of CGIplus the *process what?* request information is provided only by path information, however with RTE both script and path information are used.

4.1 RTE Programming

The RTE interface is still CGI, with all the usual environment variables and input/output streams available, just in a CGIplus environment! Hence when coding a Run-Time Environment the same considerations involved in CGIplus programming apply. See Chapter 3.

In particular it is important a RTE should explicitly close files, free allocated memory, etc., after processing a request (of course it cannot rely on image run-down to clean-up after itself). It is particularly important that all traces of each script's processing are removed after it concludes. This does not mean for example that databases need to be completely closed, etc., which might defeat the purpose of using a persistant environment, just that great care must be exercised by the programmer to prevent one script interfering with another!

Examples

- An example RTE, HT_ROOT:[SRC.CGIPLUS]RTE_EXAMPLE.C.

  on-line hypertext link provides the basics of the environment.

- WASD's own Perl RTE, HT_ROOT:[SRC.PERL]PERLRTE.C/PERLRTENG.C, can also serve as an implementation example.

CGI Function Library

A source code collection of C language functions useful for processing the more vexing aspects of CGI/CGIplus/RTE programming. See Section 1.7. The example RTE implementations use this library.

4.2 Server Configuration

The following configuration information uses the supplied Perl RTE as the example.

The following rule in HTTPD$MAP maps the /pl-bin/ location to where the site wishes to locate it's CGI Perl scripts (not necessarily the same as in the example).

```
exec /pl-bin/* (CGI-BIN:[000000]PERLRTE.EXE)/ht_root/src/perl/*
```

With this rule Perl scripts may be accessed using

```
http://host.name.domain/pl-bin/an_example
```

A DCL procedure can be specified in place of an executable. Merely prefix the specification with a ''@''.

This HTTPD$CONFIG rule ensures Perl scripts could be activated via the Perl RTE even if the HTTPD$MAP rule did not exist (see Section 1.3).

```
[DclScriptRunTime]
.PL  (CGI-BIN:[000000]PERLRTE.EXE)
```

Note

The server makes no check of the RTE executable (or procedure) before attempting to activate it using DCL for processing the script. If it does not exist or is not accessable due to incorrent file protections the DCL of the subprocess will report the problem.

It does by default however, check that the file used as the script source exists as with other scripting environments. If it does not this is reported as a ''script not found''. For RTEs that wish to report on this themselves, or that possibly construct their own script specification via some internal processing, this server behaviour may be suppressed for the script activation path using the HTTPD$MAP path SETting ''script=nofind'' as in the following example.

```
set /xyz-bin/* script=nofind
exec /xyz-bin/* (CGI-BIN:[000000]XYZ.EXE)/ht_root/src/xyz/*
```

Chapter 5
CGI Callouts

During CGI or CGIplus processing (though not DECnet-based CGI) it is possible to suspend normal script output to the client and for the script to interact directly with the server, then resume output to the client. This may done more than once during processing. During the *callout* the script makes a request of the server and receives a response. These requests are to perform some server function, such as the mapping of a path to a file specification, on behalf of the script. Naturally, this makes the script no longer portable, but may be extrememly useful in some circumstances.

It is this general callout mechanism that allows specific authentication agents (see ‘‘Technical Overview, Authorization’’) to be implemented as standard CGIplus scripts.

The mechanism is quite simple.

1. The script suspends normal output by writing a record containing a unique *escape* sequence.
2. It then writes a record containing a formatted request. The server interprets the request, performs the action and returns a success or error response.
3. The script concludes the callout by writing a record containing a unique *end-of-text* sequence.
4. The script reads the server’s response and continues processing. In reality the response read could occur immediately after the request write (i.e. before the concluding end-of-text sequence).

This is a basic callout. Between the callout escape and end-of-text sequences multiple request/responses transactions may be undertaken.

5.1 Requests and Responses

The request record is plain text, comprising a request key-word (case-insensitive), full-colon, and following optional white-space and parameter(s). It is designed not to be implementation language specific.

The response record is also plain-text. It begins with a three-digit response code, with similar meanings and used for the same purpose as HTTP response codes. That is 200 is a success status, 500 a server error, 400 a request error, etc. Following the response code is white-space and the plain text result or error message. A response to any given callout request may be suppressed by providing a request record with a leading ''!'' or ''#''.

- **AUTH-FILE:** *file specification*

  If the specialized /PROFILE capability is enabled (see ''Technical Overview, Security Profile'') this can determine whether the specified file name is allowed access by the request's username.

- **GATEWAY-BEGIN:** *integer*

  When using the raw TCP/IP socket for output (see Chapter 10) this callout is used to notify the server that the script will be using the gateway device and the HTTP status code (e.g. 200, 302, 404, etc.)

- **GATEWAY-END:** *integer*

  When using the raw TCP/IP socket for output (see Chapter 10) this callout is used to notify the server of the quantity of data transfered directly to the client by the script.

- **LIFETIME:** *integer*

  Sets/resets a script subprocess' lifetime (in minutes). For instance, use to give frequently used CGIplus scripts an extended lifetime before being rundown by the server (override the [DclCgiPlusLifeTime] configuration parameter). Specifying ''none'' (or -1) gives it an *infinite* lifetime, zero resets to default.

- **MAP-FILE:** *file specification*

  Map the supplied file specification to it's URL-style path equivalent, and against the server's mapping rule. This does not check the file name is legal RMS syntax.

- **MAP-PATH:** *URL-style path*

  Map the supplied URL-style path against the server's rule database into a VMS file specification. Note that this does not verify the file name legaility or that the file actually exists.

- **NOOP:**

  No operation. Just return a success response.

- **TIMEOUT-OUTPUT:** *integer*

  Sets/resets a script request lifetime (in minutes, overrides the [TimeoutOutput] configuration parameter). Specifying ''none'' (or -1) gives it an *infinite* lifetime, zero resets to default.

- **TIMEOUT-NOPROGRESS:** *integer*

Sets/resets a script request no-progress (in minutes, overrides the [TimeoutNoProgress] configuration parameter). The *no-progress* period is the maximum number of seconds that there may be no output from the script before it is aborted. Specifying ‘‘none’’ (or -1) gives it an *infinite* lifetime, zero resets to default.

5.2 Code Examples

The record-oriented callout sequences and request/response makes implementation quite straight-forward. The following C language and DCL procedure code fragments illustrate the basics.

```
/* C language */
CgiPlusIn = fopen ("CGIPLUSIN:", "r");
printf ("%s\nMAP-FILE: %s\n%s\n",
        getenv("CGIPLUSESC"), FileNamePtr, getenv("CGIPLUSEOT"));
fgets (CalloutResponse, sizeof(CalloutResponse), CgiPlusIn);
```

```
$! DCL procedure
$ open /read CgiPlusIn CGIPLUSIN
$ write sys$output f$trnlnm("CGIPLUSESC")
$ write sys$output "MAP-PATH: " + PathPtr
$ read CgiPlusIn Response
$!(no need to read a response for this next request, it's suppressed)
$ write sys$output "#TIMEOUT-OUTPUT:10"
$ write sys$output f$trnlnm("CGIPLUSEOT")
```

Also see working examples in HT_ROOT:[SRC.CGIPLUS]

on-line hypertext link

Chapter 6
ISAPI

ISAPI (procounced *eye-sap-ee*) was developed by Process Software Corporation (the developer of Purveyor Encrypt Web Server available under VMS), Microsoft Corporation and a small number of other vendors. It has an software infrastructure similar to CGI but a different architecture. It is designed to eliminate the expensive process creation overheads of CGI (under Unix, let alone VMS), reduce latency for expensive-to-activate resources, and generally improve server throughput, particularly on busy sites.

Unlike standard CGI, which creates a child process to service each request, ISAPI is designed to load additional sharable code (DLLs, or Dynamic Link Libraries in MSWindows, shareable images under VMS) into the Web server's process space. These are known as server *extensions*. This radically reduces the overheads of subsequent request processing and makes possible server functionality that can maintain resources between requests (for instance keep open a large database), again contributing to reduced latency and increased throughput.

Of course there is a down-side! Loading foreign executable code into the server compromises it's integrity. Poorly written extensions can seriously impact server performance and in the worst-case even crash a server process. The other significant concern is the multi-threaded environment of most servers. Extensions must be carefully constructed so as not to impact the granularity of processing in a server and to be *thread-safe*, not creating circumstances where processing corruption or deadlock occurs.

6.1 CGIsapi

WASD provides an environment for executing ISAPI based extensions. Unlike classic ISAPI the DLLs are not loaded into server space but into separate subprocesses, in much the same way as CGIplus scripts are handled (see Chapter 3). This still offers significantly improved performance through the persistance of the ISAPI extension between requests. Measurements show a potential four-fold increase in throughput compared to an equivalent CGI script! This is comparable to reported performance differences between the two environments in the Microsoft IIS environment.

While the subprocess context does add more overhead than if the DLL was loaded directly into the server process space, it does have two significant advantages.

1. Buggy DLL code will generally not directly affect the integrity of the server process. At worst the subprocess may terminate.
2. Each subprocess services only the one request at a time. This eliminates the threading issues.

WASD implements the ISAPI environment as an instance of it's CGIplus environment. CGIplus shares two significant characteristics with ISAPI, persistance and a CGI-like environment. This allows a simple CGIplus *wrapper* script to be used to load and interface with the ISAPI DLL. After being loaded the ISAPI-compliant code cannot tell the difference between the WASD environment and any other vanilla ISAPI one!

This wrapper is known as **CGIsapi** (pronounced *see-gee-eye-sap-ee*).

Wrapping another layer does introduce overhead not present in the native CGIplus itself, however measurements indicate in *the real world* (tm) performance of the two is quite comparable. See ''Technical Overview, Performance'' for further information. The advantage of ISAPI over CGIplus is not performance but the fact it's a well documented interface. Writing a script to that specification may be an easier option, particularly for sites with a mixture or history of different Web servers, than learning the CGIplus interface (simple as CGIplus is).

6.2 Writing ISAPI Scripts

This section is by-no-means a tutorial on how to write for ISAPI.

First, get a book on ISAPI. Second, ignore most of it! Generally these tomes concentrate on the Microsoft environment. Still, information on the basic behaviour of ISAPI extensions and the Internet Server API is valuable. Other resources are available at no cost from the Microsoft and Process Software Corporation sites.

Have a look at the WASD example DLL and it's build procedure in HT_ROOT:[SRC.CGIPLUS].

The CGIsapi wrapper, HT_ROOT:[SRC.CGIPLUS]CGISAPI.C, is relatively straight-forward, relying on CGIplus for IPC with the parent server process. A brief description of the detail of the implementation is provided in the source code.

CGIsapi has a simple facility to assist with debugging DLLs. When enabled, information on passed parameters is output whenever a call is made to an ISAPI function. This debugging can be toggled on and off whenever desired. Once enabled DLL debugging remains active through multiple uses of a CGISAPI instance, or until disabled, or until the particular CGISAPI subprocess' lifetime expires. Check detail in the CGIsapi source code description.

CGIsapi Considerations

The wrapper is designed to be ISAPI 1.0 compliant. It should also be vanilla ISAPI 2.0 compliant (not the Microsoft WIN32 variety, so don't think you'll necessarily be able to grab all those IIS extensions and just recompile and use ;^)

With CGIsapi multiple instances of any one extension may be active on the one server (each in an autonomous subprocess, unlike a server-process-space loaded extension where only one would ever be active at any one time). Be aware this could present different concurrency issues than one multiple or single threaded instance.

When CGIplus subprocesses are idle they can be sys$delprc( )ed at any time by the server at expiry of lifetime or to free up required server resources. For this reason ISAPI extensions (scripts) should finalize the processing of transactions when finished, not leave anything in a state where it's unexpected demise might corrupt resources or otherwise cause problems (which is fairly good general advice anyway ;^) That is, when finished tidy up as much as is necessary.

CGIsapi loaded extensions can exit at any time they wish. The subprocess context allows this. Of course, normally a server-process-space loaded instance would not be able to do so!

For other technical detail refer to the description with the source code.

Hint!

Whenever developing ISAPI extensions don't forget that after compiling, the old version must be purged from the server before trying out the new!!!

Scripting subprocesses may be purged or deleted using (see ''Techncial Overview, Server Command Line Control''):

```
$ HTTPD /DO=DCL=DELETE
$ HTTPD /DO=DCL=PURGE
```

6.3 Server Configuration

Ensure the following are in the appropriate sections of HTTPD$CONFIG.

```
[DclScriptRunTime]
.DLL  $CGI-BIN:[000000]CGISAPI.EXE

[AddType]
.DLL  application/octet-stream  -  ISAPI extension DLL
```

Ensure this rule exists in the scripting section of HTTPD$MAP.

```
exec+ /isapi/* /cgi-bin/*
```

With this rule DLLs may be accessed using something like

```
http://host.name.domain/isapi/isapiexample.dll
```

Chapter 7
DECnet & OSU

''Imitation is the sincerest form of flattery'' - proverb

Please Note

WASD requires no additional configuration to support subprocess-based scripting. The following information applies only if DECnet-based scripting is desired.

By default WASD executes scripts within subprocesses, but can also provide scripting using DECnet for the process management. DECnet scripting is not provided to generally supplant the subprocess-based scripting but augment it for certain circumstances:

- To provide an environment within WASD where OSU-based scripts (both CGI and OSU-specific) may be employed without modification.
- To allow nodes without a full HTTP service to participate in providing resources via a well-known server, possibly resources that only they have access to.
- Load-sharing amongst cluster members for *high-impact* scripts or particularly busy sites.
- Provide user-account scripting.

DECnet Performance

Any DECnet based processing incurs some overheads.

- connection establishment
- NETSERVER image activation
- NETSERVER maintenance (such as logs, etc.)
- activation of DECnet object image or procedure
- DECnet object processing
- activation by object of image or procedure
- DECnet object run-down
- NETSERVER image reactivation on completion of object processing

As of version 5.2 WASD provides reuse of DECnet connections for both CGI and OSU scripting, in-line with OSU v3.3 which provided reuse for OSU scripts. This means multiple script requests can be made for the cost of a single DECnet connection establishment and task object activation. This functionality provides substantial performance improvements as indicated by ‘‘Technical Overview, Performance, DECnet Observations’’. Note that the OSU task procedure requires the definition of the logical name WWW_SCRIPT_MAX_REUSE representing the number of times a script may be reused. The WASD startup procedures can provide this.

In practice both the WASD CGI and OSU scripts seem to provide acceptable responsiveness.

Rule Mapping

DECnet-based scripts are mapped using the same rules as subprocess-based scripts, using the SCRIPT and EXEC rules (see ‘‘Technical Overview, Mapping User Directories’’ for general information on mapping rules). DECnet scripts have a DECnet node and *task specification string* as part of the mapping rule. There are minor variations within these to further identify it as a WASD or an OSU script. See Section 7.4 for information on mapping user scripting.

The specification string follows basic VMS file system syntax (RMS), preceding the file components of the specification. The following example illustrates declaring that paths beginning with FRODO will allow the execution of scripts from the CGI-BIN:[000000] directory on DECnet node FRODO.

```
exec /FRODO/* /FRODO::/cgi-bin/*
```

In similar fashion the following example illustrates a script ‘‘frodo_show’’ that might do a ‘‘SHOW SYSTEM’’ on node FRODO. Note that these rules are case-insensitive.

```
script /frodo-showsys /frodo::/cgi-bin/showsys.com
```

Both of the above examples would use the WASD CGI DECnet environment (the default if no task specification string is provided). By including task information other environments, in particular the OSU scripting enviroment, can be specified for the script to be executed within. The default task is named CGIWASD and can also be explicitly specified (although this behaviour would be the same as that in the first example)

```
exec /frodo/* /frodo::"task=cgiwasd"/cgi-bin/*
```

All task specification strings may also use zero as the task abbreviation.

```
exec /frodo/* /frodo::"0=cgiwasd"/cgi-bin/*
```

To execute a script within the OSU environment specify the standard OSU task executive WWWEXEC, as in the following example:

```
exec /osu/* /FRODO::"task=wwwexec"/cgi-bin/*
```

This would allow any URL beginning with ‘‘/osu/’’ to execute a script in the OSU environment.

Local System

To specify any script to execute on the same system as the HTTP server specify the node name as zero or SYS$NODE.

```
exec /decnet/* /0::"task=cgiwasd"/cgi-bin/*
exec /osu/* /sys$node::"task=wwwexec"/cgi-bin/*
```

Mapping rules are included in the examples (see HT_ROOT:[EXAMPLE]) providing this. After the DECnet environment has been started any CGI script may be executed on the local system via DECnet by substituting ‘‘/decnet/’’ for ‘‘/cgi-bin/’’ as the script path, and any OSU script available by using ‘‘/osu/’’. Behaviour is indeterminate, though it shouldn’t be catastrophic, if one is invoked using the incorrect path (i.e. an OSU script using /decnet/ or a CGI script using /osu/).

7.1 Script System Environment

The target system must have sufficient of the WASD server environment to support the required CGI script activation and activity. If the target system is actually the same system as the HTTP server then it already exists, or if part of the local system’s cluster, then providing this should be relatively straight-forward. If the target system has none of the server environment then at a minimum it must have the logical name CGI-BIN defined representing the directory containing the required DECnet object procedure and scripts. The following fragment illustrates this:

```
$ DEFINE /SYSTEM /TRANSLATION=(CONCEALED) CGI-BIN device:[dir.]
```

In this directory must be located the WASDCGI.COM and WWWEXEC.COM procedures required by the network task. Of course other parts of the environment may need to be provided depending on script requirements.

7.1.1 Proxy Access

The local system must have *proxy access* to each target scripting system (even if that ‘‘target’’ system is the same system as the HTTP server). This involves creating a proxy entry in each target hosts’s authorization database. The following example assumes the existance of a local HTTP$SERVER account. If it does not exist on the target node then one must be created with the same security profile as the HTTP server’s.

Caution!

If unsure of the security implications of this action consult the relevant VMS system management security documentation.

The zero represents the system the server is currently executing on.

```
$ SET DEFAULT SYS$SYSTEM
$ MCR AUTHORIZE
UAF> ADD /PROXY 0::HTTP$SERVER HTTP$SERVER /DEFAULT
```

It is necessary to ensure the account has permission to write into it’s home directory. A network process creates a NETSERVER.LOG (Phase-IV) or NET$SERVER.LOG (DECnet-Plus) file in the home directory, and will fail to start if it cannot!

7.1.2 DECnet Objects

To provide DECnet scripting DECnet *object(s)* must be specified for any system on which the scripts will be executed. The DECnet object is the program or procedure that is activated at the target system inside of a network-mode process to interact with the HTTP server.

DECnet-Plus (OSI/Phase-V)

DECnet-Plus uses the NCL utility to administer the network environment. The following NCL scripting shows the creation of a network application for the WASD CGI object:

```
$ MCR NCL
CREATE NODE 0 SESSION CONTROL APPLICATION CGIWASD
SET NODE 0 SESSION CONTROL APPLICATION CGIWASD ADDRESSES = {NAME=CGIWASD} -
,CLIENT =  -
,INCOMING ALIAS = TRUE -
,INCOMING PROXY = TRUE -
,OUTGOING ALIAS = FALSE -
,OUTGOING PROXY = TRUE -
,NODE SYNONYM = TRUE -
,IMAGE NAME = CGI-BIN:[000000]CGIWASD.COM -
,INCOMING OSI TSEL =
```

To create a DECnet-Plus OSU WWWEXEC object:

```
$ MCR NCL
SET NODE 0 SESSION CONTROL APPLICATION WWWEXEC ADDRESSES = {NAME=WWWEXEC} -
,CLIENT =  -
,INCOMING ALIAS = TRUE -
,INCOMING PROXY = TRUE -
,OUTGOING ALIAS = FALSE -
,OUTGOING PROXY = TRUE -
,NODE SYNONYM = TRUE -
,IMAGE NAME = CGI-BIN:[000000]WWWEXEC.COM -
,INCOMING OSI TSEL =
```

These must be executed at each system (or server) startup, and may be executed standalone, as illustrated, or incorporated in the NCL script SYS$STARTUP:NET$APPLICATION_STARTUP.NCL for automatic creation at each system startup. Examples may be found in HT_ROOT:[EXAMPLE].

Phase-IV

DECnet Phase-IV uses the NCP utility to administer the network environment. The following NCP commands may be used each time during server startup to create the required DECnet objects. With Phase-IV the SET verb may be replaced with a DEFINE verb and the commands issued just once to permanently create the objects (a SET must also be done that first time to create working instances of the DEFINEd objects).

To create a DECnet CGI object:

```
$ MCR NCP
SET OBJECT CGIWASD NUMBER 0 FILE CGI-BIN:[000000]CGIWASD.COM
```

To create a DECnet OSU WWWEXEC object:

```
$ MCR NCP
SET OBJECT WWWEXEC NUMBER 0 FILE CGI-BIN:[000000]WWWEXEC.COM
```

Examples may be found in HT_ROOT:[EXAMPLE].

7.1.3 Reducing Script Latency

Script system network process persistance may be configured using NETSERVER logical names. These can control the number and quiescent period of the server processes. These logical names must be defined in the LOGIN.COM of the HTTP server account on the target script system.

- **NETSERVER$SERVERS_*username* -** This logical controls the number of network server processes that are kept available at any one time. Defining this logical results in a minimum of the specified number of quiescent server processes maintained. This can improve script response latency by circumventing the need to create a process to service the request, at the cost of cluttering the system with NETSERVER processes.

  ```
  DEFINE /JOB NETSERVER$SERVERS_HTTP$SERVER 5
  ```

- **NETSERVER$TIMEOUT -** This logical controls the duration a quiescent network process persists before being deleted. The default period is five minutes. The following examples first show reducing that to thirty seconds, the second increasing it to one hour. Again, this can improve script response latency by circumventing the need to create a process to service the request, at least during the period a previously created process continues to exist.

  ```
  DEFINE /JOB NETSERVER$TIMEOUT "0 00:00:30"
  DEFINE /JOB NETSERVER$TIMEOUT "0 01:00:00"
  ```

7.1.4 DECnet/OSU Startup

The example STARTUP.COM and STARTUP_DECNET.COM procedures found in the HT_ROOT:[EXAMPLE] directory provide the essentials for DECnet/OSU scripting. If the INSTALL.COM startup environment is used setting the PROVIDE_DECNET symbol to 1 in STARTUP.COM will create the DECnet scripting environment during server startup.

7.2 CGI

CGI scripts that use HTTP GET (the default) may be transparently executed within the DECnet scripting environment. This means that the script is executed within a network process, on the target system (which could be the local system), instead of within a subprocess on the local system. Other than that the WASD DECnet CGI environment behaves identically to the standard subprocess CGI environment. CGIplus scripting is not supported and if CGIplus-only scripts are executed the behaviour is indeterminate.

Scripts that wish to use HTTP POST will need to read the request body from the NET$LINK stream, rather than from HTTP$INPUT as with subprocess based scripts. End of body is indicated by an empty record rather than EOF. Scripts may quite simply be made to function appropriately in both environments. The following C code fragment illustrates this.

```
if (getenv ("NET$LINK") == NULL)
{
   /* via subprocess CGI */
   while (ReadCount = fread (Buffer, 1, sizeof(Buffer), stdin))
   {
      /* processing, processing ... */
   }
}
else
{
   /* via DECnet CGI */
   if ((stdin = freopen ("NET$LINK", "r", stdin)) == NULL)
      exit (vaxc$errno);

   while (fgets (Buffer, sizeof(Buffer), stdin) != NULL)
   {
      /* check for end of stream */
      if (Buffer[0] == '\n' && Buffer[0] == '\0') break;

      /* processing, processing ... */
   }
}
```

An example of making the HELP database on a system other than that hosting the HTTP server (using the CONAN script) would be done using the mapping rules

```
map /FRODO/help /FRODO/help/
script /FRODO/help/* /FRODO::/cgi-bin/conan/*
```

and for the example DCL SHOW script

```
script /FRODO/show* /FRODO::/cgi-bin/show*
```

7.3 OSU (DECthreads) Emulation

The OSU, or *DECthreads* server is the most widely deployed VMS HTTP server environment, authored by David Jones and copyright the Ohio State University. See http://kcgl1.eng.ohio-state.edu/www/doc/serverinfo.html for more information.

The WASD HTTP server provides an emulation of the OSU scripting environment. This is provided so that OSU-based scripts (both CGI-style and OSU-specific) may be employed by WASD with no modification. As this emulation has been designed through examining OSU code and lots of trial and error it's behaviour may be incomplete or present errors. A list of OSU scripts known to work with WASD is provided at the end of this section, see Known Working Scripts.

Supported scripts include only those that depend on the OSU WWWEXEC object and dialog for all functionality. Any script that uses other OSU-specific functionality is not supported. Interactions between WASD's and OSU's authentication/authorization schemes may be expected.

OSU scripts expect to get the path information unmapped, whereas WASD always additionally maps any path after the script component has been derived from the request URI. It may be necessary to ensure OSU scripts are activated with the associated path SET to provide what they expect. For example:

```
set /htbin/* mapONCE
set /osu/* mapONCE
```

The author would like to know of any OSU scripts the WASD emulation *barfs* on, and will attempt to address the associated limitation(s) and/or problem(s).

OSU Setup

Software necessary for supporting the OSU scripting environment (e.g. WWWEXEC.COM) and selected OSU scripts (mainly for testing purposes) have been extracted from the OSU v3.4 package and included in the HT_ROOT:[SRC.OSU] directory. This has been done within the express OSU licensing conditions.

```
Copyright 1994,1997 The Ohio State University.
The Ohio State University will not assert copyright with respect
to reproduction, distribution, performance and/or modification
of this program by any person or entity that ensures that all
copies made, controlled or distributed by or for him or it bear
appropriate acknowlegement of the developers of this program.
```

An example DECnet and OSU scripting startup may be found in HT_ROOT:[EXAMPLE]. This should be called from or used within the HTTP server startup. It includes two logical definitions required for common OSU scripts. Other tailoring may be required for specific OSU scripts.

OSU - General Comments

David Jones, the author of the DECthreads (OSU) HTTP server, outlines his reasons for basing OSUs scripting on DECnet (reproduced from a USENET NEWS reply to a comment this author made about DECnet-based scripting).

```
------------------------------------------------------------------------

From           JONESD@er6.eng.ohio-state.edu (David Jones)
Organization   The Ohio State University
Date           12 Aug 1997 09:04:11 GMT
Newsgroups     vmsnet.sysmgt,comp.os.vms,comp.infosystems.www.servers.misc
Message-ID     <5sp8ub$brs$1@charm.magnus.acs.ohio-state.edu>

------------------------------------------------------------------------
```

. . . some text omitted

```
Since I was comfortable with DECnet, I based the scripting system
for the OSU server around it.    The key reasons to use netserver
processes rather than spawning sub-processes:

    1. DECnet automatically caches and re-uses netserver processes,
       whereas there were well-known performance problems with spawning
       sub-processes.

    2. DECnet processes are detached processes, so you don't worry about
       the effect of scripts consuming pooled quotas (e.g. bytlm) on
       the HTTP server process.

    3. Creation/connection with the DECnet server process is asynchronous
       with respect to the server so other operations can proceed concurrently.
       Spawning is done in supervisor mode, blocking the server's operation
       until the child process is completely initialized.
```

```
    4. With DECnet, scripts can be configured to run on different nodes
       for load balancing.

    5. In addition to the standard 'WWWEXEC' object, you can create
       other 'persistent' DECnet objects that the server communicates with
       as scripts. (this was implemented years before OpenMarket's FastCGI
       proposal).

    6. CGI is not the be-all end-all of scripting.  The dialog phase of
       OSU's scripting environment allows scripts to do things CGI
       is incapable of, such as ask the server to translate an arbitrary
       path and not just what followed the script name in the URL.

People grouse all the time about the installation difficulties caused by
it's reliance on DECnet,  the reason shown above were cited to show that it
wasn't made so capricously.

 . . .   some text omitted

David L. Jones                 |      Phone:    (614) 292-6929
Ohio State Unviversity         |      Internet:
2070 Neil Ave. Rm. 122         |                jonesd@kcgl1.eng.ohio-state.edu
Columbus, OH 43210             |                vman+@osu.edu

Disclaimer: Dogs can't tell it's not bacon.
```

The OSU server's DECnet scripting is not based on arbitrary considerations. This author does not disagree with any of the concerns, and as may be seen from WASD documentation the design of WASD also directly addresses points 1, 3 and 5 with the use of persistant subprocesses and CGIplus. Certainly DECnet-based scripting addresses the very legitimate point 4 (and also allows nodes with specific resources to participate without installing full HTTP server environments). Point 2 is not an issue with the use of detached scripting processes, or for all practical purposes addressed by adjusting account quotas to support the required number of subprocesses. Point 6 is only too true (possibly at least until Java servers and *servlets* become ubiquitous :^)

Known Working Scripts

The following is a list of OSU-specific scripts that the WASD implementation has either been developed or tested against, and any installation notes or other WASD specifics. The author would like to know of any OSU scripts the WASD emulation has problems or works successfully with.

- All of the scripts, etc. provided in the HT_ROOT:[SRC.OSU] directory. These include:
  - cgi_symbols
  - cgi-mailto
  - html_preproc
  - set_dcl_env
  - testcgi
  - testform
  - tmail
  - vmshelpgate

- webbook

- **helpgate**

  Comment out the *Conan The Librarian* mappings for the ‘‘/help’’ path and provide the following in HTTPD$MAP:

  ```
  # first make "/help" into a script specification
  map /help* /htbin/helpgate/help*
  # general rule mapping "/htbin" to OSU DECnet scripts
  exec /htbin/* /0::"0=wwwexec"/cgi-bin/*
  # map the non-script part of the path back to just "/help"
  pass /htbin/helpgate/help* /help*
  ```

  It is possible to support both HELP environments (although *helpgate* will not work without owning the ‘‘/help’’ path), merely provide another mapping for *Conan* with a slightly different path, for example:

  ```
  map /chelp /chelp/
  script /chelp/* /cgi-bin/conan/*
  ```

- **HTML pre-processor**

  Yes, backward compatibility can be provided for those old OSU .HTMLX files in your new WASD environment ;^) All that is needed is a file type mapping to the script in the HTTPD$CONFIG configuration file.

  ```
  [AddType]
  .HTMLX  text/html  /htbin/html_preproc  OSU SSI HTML
  ```

- showtime
- mgmt

7.4 User Scripts

The WASD DECnet environment provides a simple mechanism for executing scripts within accounts other than the server’s. This allows configured users to write and maintain scripts within their own areas and have them execute as themselves. Both standard CGI and OSU scripting may be provided for with this facility.

Of course there is always a down-side. Be careful to whom this capability is granted. User scripts are executed within a user network-mode process created by DECnet. Script actions cannot generally affect server behaviour, but they can access any WORLD-readable and modify any WORLD-writable resource in the system/cluster, opening a window for information leakage or mischievous/malicious actions. Script authors should be aware of any potential side-effects of their scripts and Web administrators vigilant against possible destructive behaviours of scripts they do not author.

User scripting is not enabled by default. To provide this facility mapping rules into the user area must be provided in much the same way as for user directories, see ‘‘Technical Overview, Mapping User Directories’’.

The ‘‘EXEC’’ rule provides a wildcard representation of users’ script paths. As part of this mapping a subdirectory specifically for the web scripts should always be included. **Never** map users’ top-level directories. For instance if a user’s account home directory was located in the area WWW_USER:[DANIEL] the following rule would potentially allow the user DANIEL to provide scripts from the home subdirectory [.WWW.CGI-BIN] using the accompanying rules (first for CGI, second for OSU scripts):

```
exec /~*/cgi-bin/* /0""::/www_user/*/www/cgi-bin/*
exec /~*/osu-bin/* /0""::"0=wwwexec"/www_user/*/www/cgi-bin/*
```

Scripts located in these directories are accessable via paths such as the following:

```
/~daniel/cgi-bin/test
```

Explicit User Account

Using mapping rules it is possible to explicitly specify the user account for a particular script or scripts to be executed within. This may be useful if an application has quota or other resource requirements that are desired to be withheld from the HTTP server account (i.e. it can provide a measure of isolation between the server and application accounts).

```
exec /whatever-bin/* /0"WHATEVER"::/whatever_root/cgi-bin/*
script /dowhatever/* /0"WHATEVER"::/whatever_root/cgi-bin/dowhatever/*
```

Proxy Access

For each user account permitted to execute local scripts proxy access to that account must be granted to the HTTP server account.

Caution!

If unsure of the security implications of this action consult the relevant VMS system management security documentation.

```
$ SET DEFAULT SYS$SYSTEM
$ MCR AUTHORIZE
UAF> ADD /PROXY <node>::HTTP$SERVER <account>
```

For example, the following would allow the HTTP server to execute scripts on behalf of the username DANIEL.

```
UAF> ADD /PROXY 0::HTTP$SERVER DANIEL
```

Chapter 8
Perl

WASD supports Perl scripting in the CGI, CGIplus and RTE environments. Generally no source changes are required to use standard CGI Perl scripts! Information in this section pertains specifically to VMS Perl 5.6 and following. Earlier versions may have some limitations. VMS Perl 5.6 is a relatively complete Perl implementation and standard distributions contain some VMS-specific functionality. In particular the VMS::DCLsym and VMS::Stdio can make life simpler for the VMS perl developer.

Users of VMS Perl are directed to the ''VMS Perl FAQ (Unofficial)'' http://w4.lns.cornell.edu/~pvhp/perl/VMS.html, an extensive and detailed resource, and to ''Perl on VMS'' at http://www.sidhe.org/vmsperl/, providing access to the latest release of Perl for VMS.

Please Note

The author is very much the Perl novice and this chapter probably reflects that. Additional material and improved code always gratefully received.

8.1 Activating Perl

There are a number of ways to activate a Perl script under VMS. Any of these may be used with the WASD server. If the script file is accessable via the *exec* or *script* rules of the HTTPD$MAP configuration file it can be activated by the server. The simplest example is to place the scripts somewhere in the CGI-BIN:[000000] path and execute via /cgi-bin/, although in common with other scripts it may be located anywhere a rule provides a path to access it (see Section 1.2).

Directly

When Perl is available from the command-line, either as a DCLTABLES defined verb, a DCL$PATH available verb, or as a *foreign* verb. The script (the file containg the Perl source) is provided to the Perl interpreter as a parameter to the Perl verb.

```
$ PERL perl-script-file-name
```

DCL Procedure Wrapped

If DCL pre-processing, or some other specific environment needs to be set up, the activation of the Perl script can be placed inside a DCL *wrapper* procedure. This is often used to allow the transparent activation of Perl scripts via the DCL$PATH mechanism.

```
$ PERL = "$PERL_ROOT:[000000]PERL.EXE"
$ DEFINE /USER PERL_ENV_TABLES CLISYM_GLOBAL,LNM$PROCESS
$ PERL perl-script-file-name
```

DCL Procedure Embedded

The Perl source is embedded as in-line data within a DCL procedure.

```
$ DEFINE /USER PERL_ENV_TABLES CLISYM_GLOBAL,LNM$PROCESS
$ PERL
$ DECK /DOLLARS="__END__"
# start of Perl script
print "Hello \"$ENV{'WWW_REMOTE_HOST'}\"\n";
__END__
```

8.2 CGI Environment

Due to changes in environment handling sometime between versions 5.0 and 5.6 it was impossible to access DCL symbols via the %ENV hash, making CGI-based scripts impossible to use under VMS Web servers without modification. Version 5.6 addresses this issue by providing a versatile mechanism for controlling where the environment variables are manipulated. The logical name PERL_ENV_TABLES specifies this location, or if defined as a search list, the locations.

Name	Location
CRTL_ENV	C run-time environment array (i.e. getenv( ))
CLISYM_LOCAL	get DCL symbols, set local
CLISYM_GLOBAL	get DCL symbols, set global
logical name table	any logical name table, including LNM$FILE_DEV

For WASD Perl scripting it is recommended that this be defined as CLISYM_GLOBAL,LNM$PROCESS. The CLISYM_GLOBAL allows access to the CGI variable environment, and LNM$PROCESS to significant logical name definitions for the subprocess (e.g. HTTP$INPUT and callout sequences). This can be done on a system-wide basis (i.e. for all Perl scripting) using

```
$ DEFINE /SYSTEM PERL_ENV_TABLES CLISYM_GLOBAL,LNM$PROCESS
```

during system startup, or by defining a user-mode logical in a DCL procedure *wrapper* immediately before activating the Perl interpreter (as show in the examples in this section).

8.3 POSTed Requests

Requests using the POST method contain all the content in the body of the request. In particular, requests generated via HTML <FORM> contructs do not deliver the form data via the request query string, it is provided in a URL-form-encoded body. This requires some explicit processing to recover the form elements. A number of Perl CGI modules exist to ease this chore, including the most popular CGI.pm. All of these should work in the VMS environment, and of course then with WASD.

For POSTed requests it is necessary for the script to have access to the request body. In Unix environments this is available via the <stdin> stream, and under Perl via STDIN, <>, etc. This equates to SYS$INPUT under VMS.

With WASD, when activating the .PL script file directly via a [DclScriptRunTime] entry (i.e. without a DCL procedure wrapper) STDIN is directly available without further issues.

If the script has a DCL wrapper procedure the DCL CLI has control of the SYS$INPUT stream and it becomes necessary to temporarily redirect this for the duration of the script. WASD provides the HTTP$INPUT process-level logical name to identify the script body stream (along with WWW_IN and APACHE$INPUT names for easing script portability). The redirection is simply done, as shown in the following example.

```
$ DEFINE /USER PERL_ENV_TABLES CLISYM_GLOBAL,LNM$PROCESS
$ DEFINE /USER SYS$INPUT HTTP$INPUT
$ PERL perl-script-file-name
```

If the script is embedded in a DCL procedure the DCL CLI is using SYS$INPUT to provide the script source to the Perl interpreter and so is completely unavailable for use. The request body is still available to the script however but must be explicitly read from HTTP$INPUT. This example provides the basics.

```
$ DEFINE /USER PERL_ENV_TABLES CLISYM_GLOBAL,LNM$PROCESS
$ PERL
$ DECK /DOLLARS="__END__"
# start of Perl script
print "HTTP method is \"$ENV{'WWW_REQUEST_METHOD'}\"\n";
# read POSTed body stream
open (HTTPIN, $ENV{"HTTP\$INPUT"})
   or die "Could not open $ENV{'HTTP\$INPUT'}\n";
while (<HTTPIN>)
{
  chop;  # remove trailing newline
  print "<HTTPIN> |$_|\n";
}
__END__
```

8.4 Reducing Latency

Perl is an interpreter, meaning scripts are provided and activated as source form, the interpreter processing the program ‘‘on-the-fly’’. Perl actually translates the entire script into an intermediate form before beginning execution. This has the advantage of discovering and reporting syntax errors before beginning any actual processing, and also improves the final run-time performance.

While having Perl an interpreter eases development and portability it does incur a performance penalty, particularly in activation latency, due to both interpreter image activation, and script and associated Perl module preprocessing. With standard CGI, where each request processed is handled as an autonomous activation, this becomes quite noticable and can have significant system impact.

WASD provides two solutions for this and other persistent scripting issues.

8.4.1 CGIplus

CGIplus substantially eliminates the overhead associated with CGI processing by allowing the subprocess and any associated image/application to continue executing between uses. For detailed information see Chapter 3. The good news is, CGIplus is relatively simple to support, even using Perl. The great news is, **CGIplus can reduce latency and improve performance by some twenty-fold!!**

With CGIplus the Perl script remains active for the life of the subprocess. That is it persists! Read the general philosphy and implementation details in the above reference. Note that it is still substantially CGI! The notable differences are two. CGI variables are obtained by reading a stream, not using the %ENV hash. The end-of-script is indicated by writing a special byte sequence (detected and used by the server). Of course the request body is still available via the usual stream.

Using the basic principles described in the CGIplus Chapter a Perl CGIplus script would be relatively simple to build from scratch. To assist in deploying CGIplus Perl scripting a CGIplus.pm Perl module has been provided. This allows a function reference to be passed to a module function and then using the module's own *environment* hash CGI variables may be simply accessed. The module also provides other functionality to ease the sending of binary response streams (which can be a problem with the VMS perl interpreter).

Examples for this may be found in the HT_ROOT:[SRC.PERL] directory.

8.4.2 Run-Time Environment

A Run-Time Environment (RTE) is almost identical to CGIplus. It allows an environment to persist between requests, substantially improving response latency and reducing system impact. For information see Chapter 4. There is a significant difference between RTE and CGIplus scripts. With CGIplus the script itself persists between uses, retaining all of it's state. With an RTE the script does not persist or retain state, only the RTE itself.

WASD provides an RTE Perl interpreter. This contains an embedded Perl engine and an associated Perl module that allows multiple scripts to be activated, preprocessed once and remain loaded *read-to-run*. This eliminates the overhead associated with activating the interpreter and Perl script with each request. This mechanism parallels the Apache *perl_mod* module and works on substantially unmodified CGI scripts. **The test-bench indicates an improvement of some twenty-five fold!**

Examples for RTE Perl may be found in the HT_ROOT:[SRC.PERL] directory.

Note

The Perl RTE is new to WASD 7.1 and to this extent has not received any wide and general deployment. There may still be issues to be resolved, particularly with script isolation within the interpreter. Please experiment with this environment - it's

potential for improving performance is substantial - and provide feedback to the author on any successes or problems in use.

8.5 Requirements

These are the basic configuration requirements for using Perl.

- HTTPD$CONFIG configuration file.

```
[DclScriptRunTime]
.PL PERL
.CGI PERL
```

- The following content types are configured, also in HTTPD$CONFIG.

```
[AddType]
.PL  text/plain  -  Perl source
.POD  text/plain  -  Perl documentation
.CGI  text/plain  -  Perl source
```

- Some mapping is required to use the Perl RTE interpreter. Something like the following will provide a Perl RTE path for development purposes.

```
exec /plrte/* (cgi-bin:[000000]perlrte.exe)/ht_root/src/perl/*
```

For further information see Section 4.2.

Chapter 9
Java

Java classes may be used to perform CGI scripting with WASD. They may be designed as standard CGI scripts (with the inevitable latency of the class loading) or as CGIplus scripts (with the attendant benefit of lower latency).

Note that Java scripts must always be mapped and executed using the CGIplus path, however some can behave as standard CGI scripts, exiting after responding to the request, while others can persist, responding to multiple requests (see Chapter 3). The CGIplus path is always necessary as Java does not have direct access to a process' general environment, the traditional way of passing CGI variables, so the WASD implementation uses the CGIplus data stream to provide CGI information.

WASD provides a class to allow a relatively simple interface to the CGI environment for both GET and POST method scripts. This and a collection of demonstration scripts may be found in the HT_ROOT:[SRC.JAVA] directory.

Developed using the first-release JDK1.1 beta kit for OpenVMS Alpha V7.1.

9.1 Requirements

- Ensure the Java class file type is mapped to the Java run-time in the HTTPD$CONFIG configuration file.

```
[DclScriptRunTime]
.CLASS  @CGI-BIN:[000000]JAVA.COM
```

- The following content types are configured, also in HTTPD$CONFIG.

```
[AddType]
.CLASS  application/octet-stream  -  Java class
.JAVA  text/plain  -  Java source
.JAR  application/octet-stream  -  Java archive
.PROPERTIES  text/plain  -  Java properties
```

- The CGI-BIN logical includes the HT_ROOT:[JAVA] class directory in the server startup.

```
$ JAVA_ROOT = F$TRNLNM("HT_ROOT") - ".]" + ".JAVA.]"
$ DEFINE /SYSTEM /TRANSLATION=(CONCEALED) -
         CGI-BIN 'EXE_ROOT','SCRIPT_LOCAL_ROOT', -
         'SCRIPT_ROOT','JAVA_ROOT'
```

Chapter 10
Raw TCP/IP Socket

For subprocess scripting the raw TCP/IP socket can be made available for scripts to transfer data directly to the client. The socket BG*nnnn*: device name is made available via the CGI variable WWW_GATEWAY_BG. This is enabled using the [DclGatewayBg] configuration directive. As it is a completely raw stream it cannot be used, and is not made available for SSL (''https:'') encrypted requests.

Note

Although one might imagine this direct transfer to be significantly more efficient than the standard script mailbox the test-bench indicates that to all purposes it provides a negligable improvement in throughput, even under high load. It probably only translates into measurable benefits for scripts producing large quantities of output (for instance hundreds of thousands or millions of bytes). For the average script the overhead of opening a stream to the raw TCP/IP device (which is not insignificant) and complications of the callout requirements isn't worth the effort. Still, it's there if someone wants or requires it.

The socket is created shareable between processes, and so a channel may be assigned by the script subprocess and have data written to it. The data is raw, in the sense the script must provide all carriage control, etc. All data transfered this way is outside of the server and so may not be WATCHed, etc.

The script must supply a full HTTP response. This means a NPH-style header (see Section 2.2.2) and body, with full carriage-control as required, etc. The server must be notified that the script is using the gateway device, by providing a CGI callout (see Chapter 5) before any script output and after output is concluded. The first callout provides the response HTTP status code, the second the number of bytes transfered. These are also required for correct logging of the request. If a channel to the BG: device is opened it should always be closed when it is finished with. Failure to do so could lead to resource starvation for the server.

The HT_ROOT:[SCRIPT]GATEWAY_BG_EXAMPLE.COM online demonstration example script demonstrates the use of the raw socket from DCL. The priciples can be applied to any scripting laguage.

The following code fragment shows the essential requirements using the C language.

```
int  byteCount;
char  *gatewayBg;

/* see if there's a raw socket available */
if (gatewayBg = getenv ("WWW_GATEWAY_BG"))
{
   /* yes, begin a callout */
   fputs (getenv("CGIPLUSESC"), stdout);
   fflush (stdout);

   /* notify of script response and HTTP status */
   fprintf (stdout, "GATEWAY-BEGIN: %d", 200);
   fflush (stdout);

   /* reopen <stdout> to the raw TCP/IP device */
   if ((stdout = freopen (gatewayBgPtr, "w", stdout, "ctx=bin")) == NULL)
      exit (vaxc$errno);
}

byteCount = fprintf (stdout,
"HTTP/1.0 200 OK\n\
Content-Type: text/plain\n\
\n");

 . . .  processing to <stdout>
   e.g. byteCount += fprintf (stdout, "Hello world!\n");

if (gatewayBg)
{
   /* reopen <stdout> so that it's communicating via the mailbox again */
   if ((stdout = freopen ("SYS$OUTPUT:", "w", stdout, "ctx=rec")) == NULL)
      exit (vaxc$errno);

   /* continue callout, notify of request data transfered */
   fprintf (stdout, "GATEWAY-END: %d", byteCount);
   fflush (stdout);

   /* end the callout */
   fputs (getenv("CGIPLUSEOT"), stdout);
   fflush (stdout);
}
```

Not Supported?

Not all vendor's TCP/IP package BG drivers, or not all older versions, may support the C_SHARE option when creating sockets. Symptoms may range from it being ignored (and the script being unable to open a channel to the BG*nnnn*: device) to an error being reported as the socket is being created (and the server being unable to start at all). If this occurs merely disable the [DclGatewayBg] configuration option. Script output is of course still available via the standard script output mailbox.

For portability scripts that use the raw socket for output should always use a construct similar to the above example code so only to redirect output when the GATEWAY_BG device is indicated as available.