WASD Web Services - Scripting

November 2018

For version 11.3 release of WASD VMS Web Services package.

Abstract

This document is an overview of scripting for the WASD VMS Web Services package.

For installation, update and detailed configuration information see ‘‘WASD Web Services - Install and Config’’

For configuration and use of other significant WASD capabilities see ‘‘WASD Web Services - Features and Facilities’’

And for a description of WASD Web document, SSI and directory listing behaviours and options, ‘‘WASD Web Services - Environment’’

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

Author

Mark G. Daniel

Mark.Daniel@wasd.vsm.com.au

A pox on the houses of all spammers. Make that two poxes.

Online Search

online search

Online PDF

This book is available in PDF for access and subsequent printing by suitable viewers (e.g. Ghostscript) from the location WASD_ROOT:[DOC.SCRIPTING]WASD_SCRIPTING.PDF

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.

Free Software Foundation

This package contains software made available by the Free Software Foundation under the GNU General Public License.

Ohio State University

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

OpenSSL Project

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

Paul E. Jones

This package uses SHA-1 hash code.

Contents

Chapter 2 CGI

Chapter 3 CGIplus

Chapter 4 Run-Time Environments

Chapter 5 WebSocket

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 is generally very much like any other implementation, VMS or otherwise (although there are always annoying idiosyncracies, see Section 1.12 for a partial solution to smoothing out some of these wrinkles for VMS environments).

Scripts are mechanisms for creating simple Web applications or Web services, sending data to (and often receiving data from) a client, extending the capabilities of the basic HTTPd. Scripts execute in processes and accounts separate from the actual HTTP server but under its control and interacting with it.

By default WASD manages a script's process environment in an independent detached process created by the HTTP server or as a network process created using DECnet. By configuration the server will use subprocesses in place of detached.

WASD scripting can deployed in a number of environments. Other chapters cover the specifics of these. Don't become bewildered or be put off by all these apparent options, they are basically variations on a CGI theme.

Chapter 2 - CGI
Chapter 3 - CGIplus
Chapter 4 - Run-Time Environments
Chapter 5 - WebSocket
Chapter 6 - CGI Callouts
Chapter 7 - ISAPI
Chapter 8 - DECnet & OSU
Chapter 9 - Java, Perl, PHP, Python, Tomcat
Chapter 10 - Request Redaction
Chapter 11 - Raw TCP/IP Socket

1.1 Scripting Accounts

It is strongly recommended to execute scripts in an account distinct from that executing the server. This minimises the risk of both unintentional and malicious interference with server operation through either Inter-Process Communication (IPC) or scripts manipulating files used by the server.

The default WASD installation creates two such accounts, with distinct UICs, usernames and default directory space. The UICs and home areas can be specified differently to the displayed defaults. Nothing should be assumed or read into the scripting account username - it's just a username.

Default Accounts

Username	UIC	Default	Description
HTTP$SERVER	[077,001]	WASD_ROOT:[HTTP$SERVER]	Server Account
HTTP$NOBODY	[076,001]	WASD_ROOT:[HTTP$NOBODY]	Scripting Account

During startup the server checks for the existence of the default scripting account and automatically configures itself to use this for scripting. If it is not present it falls-back to using the server account. Other account names can be used if the startup procedures are modified accordingly. The default scripting username may be overridden using the /SCRIPT=AS=<username> qualifier (see "WASD Web Services - Install and Config"). The default scripting account cannot be a member of the SYSTEM group and cannot have any privilege other than NETMBX and TMPMBX (Privileged User Scripting describes how to configure to allow this).

Scripting under a separate account is not available with subprocess scripting and is distinct from PERSONA scripting (even though it uses the same mechanism, see below).

1.2 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 "WASD Web Services - Features and Facilities") 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.

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 run-down (see below). This keeps process clutter on the system to a minimum.

1.2.1 Process Management

Scripting processes are created on-demand, within configuration limits and timeout periods. There are no arbitrary limits, only system resource limits, on the number of scripting processes. WASD_CONFIG_GLOBAL directives control the configuration limits of these (see “WASD Web Services - Install and Config”).

```
[DclHardLimit]  50
[DclSoftLimit]  40
[DclZombieLifeTime]  00:10:00
[DclCgiPlusLifeTime]  00:30:00
```

Other configuration directives are discussed later in this chapter.

Hard Limit

Scripting processes of all kinds (CGI, CGIplus and RTE) are created on-demand up until [DclHardLimit] is reached. If all scripting processes are busy with requests at that limit then the server provides a 503 (too busy) response.

Soft Limit

If there are more than [DclSoftLimit] scripting processes then the least-recently-used of any idle processes (those not currently processing a request) are proactively run-down until the soft-limit is reached. This provides *head-room* for the immediate creation of additional scripting processes for new requests that cannot be satisfied from currently instantiated processes. Soft-limit should of course be configured less than hard-limit (and if not WASD makes it that way).

Lifetimes

Idle scripting processes (those not having been given a request to process) are proactively run-down (see Section 1.2.6) after configured periods.

[DclZombieLifeTime] specifies the period in minutes a CGI scripting process can remain idle.

[DclCgiPlusLifeTime] specifies the period a CGIplus script (inside a CGIplus scripting process) or a RTE process can remain idle.

If requests being serviced by scripts drop to zero for a period (governed by the above lifetimes) then eventually all scripting processes should be run-down leaving only the server process.

1.2.2 Detached Process Scripting

The default is for WASD to execute scripts in detached processes created completely independently of the server process itself. This offers a significant number of advantages over spawned subprocesses

- pooled quotas are not 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
- 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.

Detached Process Management

With detached processes the server must explicitly ensure that each scripting process is removed from the system during server shutdown (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 the equivalent $STOP/ID= for instance), making it possible for "orphaned" scripting processes to remain - and potentially accumulate on the system!

To address this possibility the server scans the system for candidate processes during each startup. 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$NOBODY]
  Owner process ID        00000000    Dev Prot                S:RWPL,O:RWPL,G,W
  Reference count                1    Default buffer size               2048
  Device access control list:
    (IDENTIFIER=WASD_HTTPD_80,ACCESS=NONE)
    (IDENTIFIER=[WEB,HTTP$NOBODY],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).

SYLOGIN and LOGIN Procedures

Detached scripting processes are created through the full “LOGINOUT” life-cycle and execute all system and account LOGIN procedures. Although immune to the effects of most actions within these procedures, and absorbing any output generated during this phase of the process life-cycle, some consideration should be given to minimising the LOGIN procedure paths. This can noticably reduce initial script latency on less powerful platforms.

The usual recommendations for non-interactive LOGIN procedures apply for script environments as well. Avoid interactive-only commands and reduce unnecessary interactive process environment setup. This is usually accomplished though code structures such as the following

```
$ IF F$MODE() .EQS. "INTERACTIVE"
$ THEN
      . . .
$ ENDIF
```

```
$ IF F$MODE() .NES. "INTERACTIVE" THEN EXIT
```

WASD scripting processes can be specifically detected using DCL tests similar to the following. This checks the mode, that standard output is a mailbox, and the process name. These are fairly reliable (but not absolutely infallible) indicators.

```
$ IF F$MODE() .NES. "INTERACTIVE" .AND. -
     F$GETDVI("SYS$OUTPUT","MBX") .AND. -
     F$EXTRACT(0,4,F$PROCESS()) .EQS. "WASD" .AND. -
     F$EXTRACT(5,1,F$PROCESS()) .EQS. ":" .AND. -
     F$ELEMENT(1,"-",F$PROCESS()) .NES. "-"
$ THEN
$!   WASD scripting process!
      . . .
$ ENDIF
```

1.2.2.1 Persona Scripting

There are advantages in running a script under a non-server 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.

Enabling Persona Scripting

The $PERSONA functionality must be explicitly enabled at server startup using the /PERSONA qualifier (“Features and Facilities, 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 WASD_CONFIG_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
```

Access to package scripting directories (e.g. WASD_ROOT:[CGI-BIN]) is controlled by ACLs and possession of the rights identifier WASD_HTTP_NOBODY. If a non-server account requires access to these areas it too will need to be granted this identifier.

User Account Scripting

In some situations it may be desirable to allow the average Web user to experiment with or implement scripts. If the *set script=as=* mapping rule specifies a tilde character then for a user request the mapped SYSUAF username is substituted. Note that this requires the script to be colocated with the user account Web location and that the script is run under that account.

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/*
```

Where the site administrator has less than full control of the scripting environment it may be prudent to put some constraints on the quantity of resource that potentially can be consumed by non-core or errant scripting. The following WASD_CONFIG_MAP rule allows the "maximum" CPU time consumed by a single script to be constrained.

```
SET   /cgi-bin/cgi_process  script=CPU=00:00:05
```

Note that this is on a per-script basis, contrasted to the sort of limit a CPULM-type constraint would place on a scripting process.

The following WASD_CONFIG_GLOBAL rule specifies at which priority the scripting process executes. This can be used to provide the server and its infrastructure an advantage over user scripts.

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

See Section 1.2.2.3 for further detail.

Authenticated User Scripting

If the *set script=as=* mapping rule specifies a dollar then a request that has been SYSUAF authenticated has the SYSUAF username substituted. Note that the script itself can be located anywhere provided the user account has read and/or execute access to the area and file.

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

If the script has not been subject to SYSUAF authorization then this causes the script activation to fail. To allow authenticated requests to be executed under the corresponding VMS account and non-authenticated requests to script as the usual server/scripting account use the following variant.

```
SET   /cgi-bin/cgi_process  script=AS=$?
```

If the server startup included /PERSONA=AUTHORIZED then only requests that have been subject to HTTP authorization and authentication are allowed to script under non-server accounts.

Privileged User Scripting

By default a privileged account cannot be used for scripting. This is done to reduce the chance of unintended capabilities when executing scripts. With additional configuration it is possible to use such accounts. Great care should be exercised when undertaking this.

To allow the server to activate a script using a privileged account the keyword /PERSONA=RELAXED must be used with the persona startup qualifier. If the keywords /PERSONA=RELAXED=AUTHORIZED are used then privileged accounts are allowed for scripting but only if the request has been subject to HTTP authorization and authentication.

1.2.2.2 Restricting Persona 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 <u>all</u> accounts . . . including the server account!

Recommendation

The simplest solution might appear to be to just grant all required accounts the WASD_HTTP_NOBODY identifier described above. While this is certainly possible it does provide read access to all parts of the server package this identifier controls, and write access to the WASD_ROOT:[SCRATCH] default file scratch space (Section 1.10). If scripting outside of the site administrator's control is being deployed it may be better to create a separate identifier as just described.

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$NOBODY
```

Meaningful combinations of startup parameters are possible:

```
/PERSONA=(RELAXED)
/PERSONA=(RELAXED=AUTHORIZED)
/PERSONA=(AUTHORIZED,RELAXED)
/PERSONA=(ident-name,RELAXED)
/PERSONA=(ident-name,AUTHORIZED,RELAXED)
/PERSONA=(ident-name,RELAXED=AUTHORIZED)
```

1.2.2.3 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 WASD_CONFIG_GLOBAL [DclDetachProcessPriority] 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.

1.2.3 Subprocess Scripting

The WASD_CONFIG_GLOBAL directive [DclDetachProcess] can be used to disable the default detached process scripting.

```
[DclDetachProcess]  disabled
```

Note that other server configuration such as /PERSONA and/or /SCRIPT=AS= overrides this directive and so must also be disabled before subprocess scripting can be used.

Subprocess Scripting is Not Recommended

This section is included mainly for historical reference. There are so many advantages to detached process scripting and so many considerations with subprocess scripting that detached scripting is basically a “no-brainer” for production environments.

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 (Wrapping Local or Third-Party Scripts).
- Use the HTTPD$LOGIN procedure to establish a standard environment for all scripts (Section 1.9).

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.

When scripts are executed within unprivileged subprocesses created by the HTTP server, the processes are owned by the HTTP server account (HTTP$SERVER). Script actions could potentially affect server behaviour. For example it is possible for subprocesses to create or modify logical name values in the JOB table (e.g. change the value of LNM$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-accessible 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.

1.2.4 Script Process Default

For standard CGI and CGIplus script the script process' default device and directory is established using a SET DEFAULT command immediately before activating the script. This default is derived from the script file specification.

An alternative default location may be specified using the mapping rule shown in the following example.

```
set /cgi-bin/this-script* script=default=WEB:[THIS-SCRIPT]
```

The default may be specified in VMS or Unix file system syntax as appropriate. If in Unix syntax (beginning with a forward-slash) no SET DEFAULT is performed using DCL. The script itself must access this value using the SCRIPT_DEFAULT CGI variable and perform a *chdir()*.

1.2.5 Script Process Parse Type

On platforms where the Extended File Specification (EFS) is supported a SET PROCESS /PARSE=EXTENDED or SET PROCESS /PARSE=TRADITIONAL is executed by the scripting process before script activation depending on whether the script path is located on an ODS-2 or ODS-5 volume.

1.2.6 Script Process Run-Down

The server can stop a script process at any point, although this is generally done at a time and in such a way as to eliminate any disruption to request processing. Reasons for the server running-down a script process.

- Server script process limit reached. Less-used must be purged to allow execution of in-demand scripts.
- A script provides output that is not CGI or NPH compliant (i.e. the script is obviously in error).
- The administrator proactively purges scripts using the Administration Menu or command-line /DO=DCL=*PURGE* | *DELETE*.
- A client has cancelled its request against a long-running script and the [DclBitBucket-Timeout] period expires.
- The script itself exits or deletes its own process.

In running down a script process the server must both update its own internal data structures as well as manage the run-down of the script process environment and script process itself. These are the steps.

1. Exit handling.
    - If the script process is at DCL level no exit handling is possible.
    - If the script is executing an image a $FORCEX is issued against the process. This activates declared exit handlers (standard C library atexit( ), VMS system service $DCLEXH, etc.). An exit handler should always be declared for programs that need to cleanup after themselves or otherwise exit elegantly.

Generally CGIplus processes delete themselves immediately. With standard CGI scripts executing an image it may take from zero to a few seconds for the image run-down to be detected by the server. A script is allowed approximately one minute to complete the image run-down.

2. Input and output to all of the process' streams is cancelled. For scripts that may still be still processing this can result in I/O stream errors. The server waits for all queued I/O to disappear.
3. If the script process has not already deleted itself the server issues a $DELPRC against it.
4. The server receives the process termination AST and this completes the process run-down sequence.

1.2.7 Client Recalcitrance

If a client disconnects from a running script (by hitting the browser *Stop* button, or selecting another hyperlink) the loss of network connectivity is detected by the server at the next output write.

Generally it is necessary for there to be some mechanism for a client to stop long-running (and presumably resource consuming) scripts. Network disconnection is the only viable one. Experience would indicate however that most scripts are short running and most disconnections are due to clients changing their minds about waiting for a page to build or having seen the page superstructure moving on to something else.

With these considerations in mind there is significiant benefit in not running-down a script immediately the client disconnection is detected. A short wait will result in most scripts completing their output elegantly (the script itself unaware the output is not being transmitted on to the client), and in the case of persistent scripts remaining available for the next request, or for standard CGI the process remaining for use in the next CGI script.

The period allowing the script to complete its processing may be set using the WASD_CONFIG_GLOBAL configuration directive [DclBitBucketTimeout]. It should be set to say fifteen seconds, or whatever is appropriate to the local site.

```
  [DclBitBucketTimeout]  00:00:15
```

NB. "Bit-bucket" is a common term for the *place* discarded data is *stored*. :^)

1.3 Script Proctor

Script proctoring proactively creates and maintains the specified minimum number of scripting processes, configured persistent scripts, and scripting environments (RTEs). It is primarily intended for those environments that have significant startup latency but can also be used to maintain idle scripting processes ready for immediate use.

The script proctor initially instantiates configured items during server startup and before enabling request acceptance and processing.

Then during subsequent request processing, at each scripting process run-down it scans current DCL task list counting the number of instances of each configured item. The proctor facility can differentiate between idle and active instances of the script/RTE and will optionally maintain a specified number of idle processes in addition to any currently active. If fewer than the configured requirement(s) one or more new processes are instantiated.

It is possible (and probably likely) that a proctored script specification will at some stage fail to activate the script (activation specification error, script unavailable for some reason, etc.) which would lead to a runaway series of attempts to proctor with each process exit. To help avoid this situation proctored processes that exit before successfully completing initial startup are quickly suppressed from further proctoring action. This suppression then more slowly times out, again allowing proctoring for that item.

Proctored scripts and RTEs contain nothing of the usual request-related environment. No CGI variables to speak of, no service, no request method, nothing! This means that rules used for proctor activations must be outside all virtual service conditionals (i.e. outside of any specific [[*service*:*port*]] in the rules, can be inside [[*:*]]) and anything else that may be dependent on a request characteristic.

The easiest way for a script to detect if its been proctored into existence is to look for the absence of this or these. No REQUEST_METHOD is a fair indicator as it should exist with all “real” requests. Of course a proctored script is really just there to instantiate itself, not to do anything directly productive, and so a script/RTE can just recognise this and conclude with perhaps a 204 HTTP status (no content) and then remain quiescent (awaiting its first actual request). Any and all output from a proctored script goes to the bit-bucket.

Once proctored into existance the script process is then subject to the normal scripting process management and (for example) if idle for a period exceeding a lifetime value will be procactively removed. Of course, during that process rundown the proctor facility will effectively replace it with a new instance, maintaining the overall requirement.

The Server Admin, DCL Report includes a Proctor List with the currently configured proctor items and associated statistics.

Proctored script activation can be WATCHed just like any other script activation using the [x]CGI and [x]DCL items. To explicitly trigger such an event merely $STOP/ID=*pid* a proctored scripting process.

Proctor Configuration

Proctor global configuration is introduced with the WASD_CONFIG_GLOBAL [DclScriptProctor] item with each following line representing one script/RTE to be proctored. Each line contains three mandatory and one optional, space-separated components.

integer[+*integer*] *identification activation notepad*

which are, in order

1. the minimum *integer* number of instances of the item
   plus an optional minimum *integer* number of idle instances
2. an *identification string* used to match already running instances of the item
3. the *activation path* that can be used to activate the item

4. an optional *string* that is passed to the mapping notepad facility

The *zombie* form is

integer * [*activation*]

where the specified number of idle *processes* is maintained.

The minimum plus any idle requirement cannot exceed the [DclSoftLimit] configuration value (in order to minimise potential process thrashing).

The proctor facility works by matching the identification string to the script paths as present in the DCL task list (and as presented in the Server Admin, DCL Report). So it needs to contain something unique to that script or environment and often contains a wildcard specification.

The activation path used to activate the script/RTE is the same as if it was activated via a scripting request.

For an RTE the activation script specification does not actually need to exist. It must contain a minimum path to activate the desired environment but the script itself is not checked to exist by WASD and does not need to exist. If it does then it should do whatever is required to instantiate its required environment and return a 204 (no content) response. If it does not exist then the RTE itself should detect it's a proctored activation and return a default 204 response itself, instantiating only the RTE environment.

Remember

Rules for mapping proctored scripts and RTEs must be outside of any request-dependent conditionals including specific virtual services.

Proctored scripts can be detected during mapping using

```
if (request-method:)
```

or

```
if (!request-method:%)
```

(i.e. no request method) and specific data passed using the optional notepad string (see "WASD Web Services - Install and Config") and then conditionally processed using something like

```
if (notepad:blah)
```

Specific information can also be passed to the proctored script during mapping using such conditional processing in concert with the SET

```
script=param=name=value
```

mapping rule. This appears as a [WWW_]NAME CGI variable containing the value specified. Proctored scripts could then act according to any such data.

The combination of these allows some control of proctored scripting.

A proctor item with a minimum (and optionally idle) value of zero can be specified as a place-marker; the facility ignores zero valued items.

Proctor Example

This example illustrates a number of non-trivial proctoring scenarios. Only configuration items directly involved in the proctoring are shown; others would be involved in the general web-server infrastructure.

```
# WASD_CONFIG_GLOBAL
[DclScriptProctor]
2 /cgiplus-bin/mgd* /cgiplus-bin/mgd proctor=daniel
2 /apps/script /apps/script anyoldname=dothis
3+1 (*pyrte*)* /py-bin/proctor.py
2 *
3 * proctor=daniel
```

The [DclScriptProctor] contains five items. The first two specify that two scripts each be maintained, the third specifies four, the final two maintain zombie *processes*. The mapping rules (below) contain a conditional detecting the absence of a REQUEST_METHOD and processing the proctored scripts inside that decision structure. Proctor-specific mapping rules tend to be used only to supplement otherwise fundamental (but in this case proctored) scripting.

```
# WASD_CONFIG_MAP
if (request-method:)
   if (notepad:proctor=daniel) set * script=as=daniel
   if (notepad:anyoldname=dothis) set * script=param=DOTHIS=one
   # not a real script of course!
   script proctor=daniel proctor=daniel script=as=daniel
endif
```

Each of the five items explained in order:

1. Matches the CGIplus script "/cgiplus-bin/mgd" and the trailing wildcard any supplementary path provided to that script. The activation path is a straight-forward scripting path. An optional notepad datum is passed to the mapping facility. In the mapping rules the notepad datum supplied is detected ("if (notepad:proctor=daniel)") and the username under which the script is to be executed specified. A minimum of two instances of this script are maintained.
2. Matches the script "/apps/script" without trailing wildcard (as it is - a hypothetical - never used with a supplementary path). The activation path is again the straight-forward scripting path. An optional notepad datum is also passed from the proctor configuration to mapping which specifically detects it and set as CGI variable name and value that can subsequently be detected and acted upon by the proctored script. A minimum of two instances of this script are maintained.
3. Maintains a scripting environment (RTE) and is therefore a little less straight-forward. The intention is to maintain a minimum number of Python RTEs (a rather expensive-to-instantiate interpreter). The matching string is more focused on the underlying RTE. The RTE is not obvious in the activation path (as all RTE mapping is transparent to the script path). The RTE is the environment of interest though and so is the matching string of interest; "(*pyrte*)*", where the parentheses indicate an underlying RTE, the wildcards delimit the RTE name of interest, and the trailing wildcard matches any current script that the RTE may be executing. The Server Admin menu, DCL Report can be used to gain insight into any script or scripting environment strings to be matched. In this third case there is no supplementary mapping required. A minimum of three instances of this

RTE are maintained at least one of which must be idle or an additional instance will be created.

4. Maintains two idle scripting *processes* (under the default scripting account) available for scripting use without the latency of process creation.
5. Maintains three idle scripting *processes* with an associated mapping rule to activate them using the specified username. The *activation* string is arbitrary, should be unique, and is the "path" when being mapped. A *notepad* string can be specified.

1.4 Caching Script Output

The WASD cache was originally provided to reduce file-system access (a somewhat expensive activity under VMS). With the expansion in the use of dynamically generated page content (e.g. PHP, Perl, Python) there is an obvious need to reduce the system impact of some of these activities. While many such responses have content specific to the individual request a large number are also generated as general site pages, perhaps with simple time or date components, or other periodic information. Non-file caching is intended for this type of dynamic content.

Revalidation of non-file content is difficult to implement for a number of reasons, both by the server and by the scripts, and so is not provided. Instead the cache entry is flushed on expiry of the [CacheValidateSeconds], or as otherwise specified by path mapping, and the request is serviced by the content source (script, PHP, Perl, etc.) with the generated response being freshly cached. Browser requests specifying no-caching are honoured (within server configuration parameters) and will flush the entry, resulting in the content being reloaded.

Controlling Script Caching

Determining which script content is to be cached and which not, and how long before flushing, is done using mapping rules (described in detail in the "Features and Facilities"). The source of script cache content is specified using one or a combination of the following SET rules against general or specific paths in WASD_CONFIG_MAP. All mapping rules (script and non-script) are described here to put the script oriented ones into context. Those specific to script output caching are noted.

cache=[no]cgi from Common Gateway Interface (CGI) responses (**for script output**)
cache=[no]file from the file system (default and pre-8.4 cache behaviour)
cache=[no]net caches the full data stream irrespective of the source
cache=[no]nph full stream from Non-Parse Header (NPH) response (**for script output**)
cache=[no]query cache requests with query strings (**use with care**)
cache=[no]script both CGI and NPH responses (**for script output**)
cache=[no]ssi from Server-Side Includes (SSI) documents

A good understanding of site requirements and dynamic content sources, along with considerable care in specifying cache path SETings, is required to cache dynamic content effectively. It is especially important to get the content revalidation period appropriate to the content of the pages. This is specified using the following path SETings.

cache=expires=0 cancels any expiry
cache=expires=DAY expires when the day changes
cache=expires=HOUR when the hour changes
cache=expires=MINUTE when the minute changes

cache=expires=<hh:mm:ss> expires after the specified period in the cache

Examples

To cache the content of PHP-generated home pages that contain a time-of-day clock, resolving down to the minute, would require a mapping rule similar to the following.

```
set /**/index.php cache=cgi cache=expires=minute
```

To prevent requests from flushing a particular scripts output (say the main page of a site) using no-cache fields until the server determines that it needs reloading use the cache *guard* period.

```
set /index.py cache=script cache=expires=hour cache=guard=01:00:00
```

1.5 Enabling A Script

By default the server accesses scripts using the search list logical name CGI-BIN, although this can be significantly changed using mapping rules. CGI-BIN is defined to first search WASD_ROOT:[CGI-BIN] and then WASD_ROOT:[AXP-BIN], WASD_ROOT:[IA64-BIN], or WASD_ROOT:[VAX-BIN] depending on the platform. [CGI-BIN] is intended for architecture-neutral script files (.CLASS., COM, .PL, .PY, etc.) and the architecture specific directories for executables (.EXE, .DLL, etc.)

These directories are delivered empty and it is up to the site to populate them with the desired scripts. A script is made available by copying its file(s) into the appropriate directory. By default ACLs will be propagated to allow access by the default scripting account. Scripts can be made unavailable by deleting them from these directories.

Note

It is good security practice to deploy only those scripts a site is actually using. This minimises vulnerability by simply reducing the number of possibly problematic scripts. A periodic audit of script directories is a good policy.

WASD script executables are built into the WASD_ROOT:[AXP], WASD_ROOT:[IA64] or WASD_ROOT:[VAX] directories depending on the architecture. Other script files, such as DCL procedures, Perl examples, Java class examples, etc. are located in other directories in the WASD_ROOT:[SRC] tree. The procedure WASD_ROOT:[INSTALL]SCRIPTS.COM assists in the installation or deinstallation of groups of WASD scripts.

1.6 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 WASD_CONFIG_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) (Section 1.7).

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 accessible. 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 three requirements must be met.

1. The **server** account (HTTP$SERVER by default) must have read and execute access to the directory containing the scripts. Script files are searched for by the server before activation is attempted. This can be enabled using the SECHAN utility (see "Features and Facilities").

   ```
   $ SECHAN /ASIF=CGI-BIN device:[directory]script-directory.DIR
   ```

2. The **scripting** account (HTTP$NOBODY by default) must have read and execute access to any and all images and other resources required to use the application. There may be some consideration of file protections required when multiple accessors need to be accomodated (e.g. scripting and application accounts) so a specific solution may be required. If only the scripting account requires read access then the SECHAN utility could again be used to provide that to the directory (or directories) and contained files.

   ```
   $ SECHAN /ASIF=CGI-BIN device:[000000]directory.DIR
   $ SECHAN /ASIF=CGI-BIN device:[directory]*.*
   ```

3. Mapping rules must exist to make the script and any required resources accessible.

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
```

Note

When allowing the server and scripting account access into parts of the file system outside of the WASD package it is recommended to control the environment very carefully. Third-party scripting areas in particular should be modelled on those present in the package itself. The SECHAN utility described in the "Features and Facilities" may be of some assistance with this.

"Wrapping" Local or Third-Party Scripts

Sometimes it may be necessary to provide a particular non-WASD, local, or third-party script with a 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
```

1.7 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 process. 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 WASD_CONFIG_GLOBAL [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-accessible 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 accessible 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
```

1.8 Unix Syntax

CGI environment variables SCRIPT_FILENAME and PATH_TRANSLATED can be provided to any script (CGI, CGIplus, RTE) in Unix file-system syntax should that script require or prefer it using this format.

The path mapping rule “SET script=syntax=unix” changes the default syntax from VMS to Unix file-system. For example; by default using the URL

```
/cgi-bin/cgi_symbols/wasd_root/src/
```

would provide the request CGI data

```
WWW_PATH_INFO == "/wasd_root/src/"
WWW_PATH_TRANSLATED == "WASD_ROOT:[SRC]"
WWW_REQUEST_URI == "/cgi-bin/cgi_symbols/wasd_root/src/"
WWW_SCRIPT_FILENAME == "CGI-BIN:[000000]CGI_SYMBOLS.COM"
WWW_SCRIPT_NAME == "/cgi-bin/cgi_symbols"
```

If the script path had been specifically mapped using

```
set /cgi-bin/cgi_symbols* script=syntax=unix
```

the same CGI data would be provided as

```
WWW_PATH_INFO == "/wasd_root/src/"
WWW_PATH_TRANSLATED == "/wasd_root/SRC/"
WWW_REQUEST_URI == "/cgi-bin/cgi_symbols/wasd_root/src/"
WWW_SCRIPT_FILENAME == "/CGI-BIN/000000/CGI_SYMBOLS.COM"
WWW_SCRIPT_NAME == "/cgi-bin/cgi_symbols"
```

Note that the CGI or CGIplus script file is still activated using VMS file-system syntax, it is just the CGI representation that is changed. This can be particularly useful for environments ported from Unix expecting to manipulate paths using Unix syntax. This would most commonly occur with RTE engines such as PHP, Perl, etc.

1.9 Scripting Logicals

Two logicals provide some control of and input to the DCL process 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$LOGIN WASD_ROOT:[HTTP$NOBODY]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 process scripting environment. This shows the DCL commands used to support script activation. Intended for problem investigation.

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

  If the logical name value is a dotted-decimal specified IP address the verify is only applied to scripts associated with requests originating from that address. This is useful when trying to trouble-shoot scripts on a live server.

  ```
  $ DEFINE /SYSTEM HTTPD$VERIFY 192.168.0.2
  ```

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.

1.10 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

```
WASD_ROOT:[SCRATCH]
```

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

```
WASD_SCRATCH:
```

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

Note

Beware of shared scratch areas. They rely on cooperation between scripts for minimising potential interactions. They can also be a source of unintended or malicious information leakage.

Unique File Names - DCL

The “UNIQUE_ID” CGI variable provides a unique 19 character alpha-numeric string (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 = "WASD_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 is, say, only to be written, rewound and then re-read without closing first - but I’m sure you get the idea).

```
#define WASD_SCRATCH "WASD_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, WASD_SCRATCH SCRIPT_NAME ".%s", uniqueId);

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

1.11 DCL Processing of Requests

DCL is the native scripting environment for VMS and provides a rich set of constructs and capabilities for *ad hoc* and low usage scripting, and as a *glue* when several processing steps need to be undertaken for a particular script. In common with many interpreted environments care must be taken with effective exception handling and data validation. To assist with the processing of request content and response generation from within DCL procedures the CGIUTL utility is available in WASD_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.

Note

Never substitute the contents of CGI variables directly into the code stream using interpreters that will allows this (e.g. DCL, Perl). You run a very real risk of having unintended content maliciously change the intended function of the code. For example, never use comma substitution of a CGI variable at the DCL command line as in

```
$ COPY 'WWW_FORM_SRC' 'WWW_FORM_DST'
```

Always pre-process the content of the variable first, ensuring there has been nothing inserted that could subvert the intended purpose. The CGIUTL assists complying with this rule by providing an explicit, non-DCL substitution character for use on the command-line (see source code descriptive prologue).

1.12 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 WASD_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.

1.13 Script-Requested, Server-Generated Error Responses

Of course a script can generate any output it requires including non-success (non-200) pages (e.g. 400, 401, 302, etc.) For error pages a certain consistency results from making these substantially the same layout and content as those generated by the server itself. To this end, script response header output can contain one or more of several extension fields to indicate to the server that instead of sending the script response to the client it should internally generate an error response using the script-supplied information. These fields are listed in Script-Control: section of Section 2.2.1 and are available in any scripting environment.

If a "Script-Control: X-error-text="*text of error message*"" field occurs in the script response header the server stops processing further output and generates an error message. Other *error* fields can be used to provide additional or message-modifying information. A significant example is the "Script-Control: X-error-vms-status=*integer*" field which supplies a VMS status value for a more detailed, status-related error message explanation.

Essentially the script just generates a standard CGI "Status: *nnn*" response and includes at least the "X-error-text=" field before the header-terminating empty record (blank line). Some variations are shown in the following DCL examples.

```
$! vanilla error message
$ say = "write sys$output"
$ say "Status: 400"
$ say "Script-Control: X-error-text=""Confusing URL components!"""
$ say ""
```

```
$! VMS status error message
$ say = "write sys$output"
$! "status: 000" allows the server to select the HTTP status code
$ say "Status: 000"
$ say "Script-Control: X-error-text=""/a/file/name.txt"""
$ say "Script-Control: X-error-vms-status=%X00000910"
$ say "Script-Control: X-error-vms-text=""A:[FILE]NAME.TXT"""
$ say ""
```

```
$! add META source module name and line generating message
$ say = "write sys$output"
$ say "Status: 500"
$ say "Script-Control: X-error-text=""Don't know what to do now..."""
$ say "Script-Control: X-error-module=EXAMPLE; X-error-line=999"
$ say ""
```

Interestingly, because CGI environments should ignore response fields unknown to them, for scripts deployed across multiple server platforms it should be possible to have these WASD-specific elements in every header for WASD uses followed by other explicitly error page content for use in those other environments.

```
$! WASD error content, plus other platform content
$ say = "write sys$output"
$ say "Status: 404"
$ say "Script-Control: X-error-text=""Requested object not found."""
$ say "Content-Type: text/html"
$ say ""
$ say "<B>ERROR 404:</B>&nbsp; Requested object not found."
```

An example implemented using DCL is available WASD_ROOT:[SRC.OTHER]REQUEST_ERROR_MSG.COM

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, confirmed against the final RFC 3875, authored by David Robinson (drtr@apache.org) and Ken A.L.Coar (coar@apache.org), October 2004.

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 "Features and Facilities", 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).

There are 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.

NEVER, EVER SUBSTITUTE

the contents of CGI variables directly into the code stream using interpreters that will allows this (e.g. DCL, Perl). You run a very real risk of having unintended content maliciously change the intended function of the code. For example, never use comma substitution of a CGI variable at the DCL command line as in

```
$ COPY 'WWW_FORM_SRC' 'WWW_FORM_DST'
```

Always pre-process the content of the variable first, ensuring there has been nothing inserted that could subvert the intended purpose (repeated here to emphasize the significance of this rule).

CGI variable capacity now varies significantly with VMS version.

The total size 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 very large individual variables or a large number of form fields, etc., it may be possible to exhaust this quantity.

VMS V7.3-2 and later . . .

CGI variables may contain values in excess of 8000 characters (the full 8192 symbol capacity cannot be realized due to the way the symbols are created via the CLI). This is a significant increase on earlier capacities. Mailbox buffer [BufferSizeDclCommand] may need to be increased if this capacity is to be fully utilized.

VMS V7.3-1 and earlier . . .

Values may contain approximately 1000 characters minus the size of the variable name. This should still 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 with these versions is limited to 255 characters so the symbols for larger variables are built up over successive DCL commands with the limit determined by CLI behaviour.

Symbol Truncation

On VMS V7.3-2 and later symbol capacity should never be an issue (well, perhaps only with the most extraordinarily poorly designed script). With VMS V7.3-1 and earlier, with a symbol value that is too large, the server by default aborts the request, generating and returning a 500 HTTP status. Experience has shown that this occurs very rarely. If it does occur it is possible to instruct the server to instead truncate the CGI variable value and continue processing. Any CGI variable that is truncated in such a manner has its name placed in CGI variable SERVER_TRUNCATE, so that a script can check for, and take appropriate action on, any such truncation. To have the server truncate such variables instead of aborting processing SET the path using the *script=symbol=truncate* mapping rule. For example

```
set /cgi-bin/script-name* script=symbol=truncate
```

CGI Variables

Remember, by default all variables are prefixed by "WWW_" (though this may be modified using the *set CGIprefix=* mapping rule), and not all variables will be present for all requests. These CGI environment variables reflect a combination of HTTP/1.1 and HTTP/1.0 request parameters.

CGI Environment Variables

Name	Description	Origin
AUTH_ACCESS	“READ” or “READ+WRITE”	WASD
AUTH_AGENT	used by an authorization agent (specialized use)	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)	CGI
AUTH_USER	details of authenticated user	WASD
CONTENT_LENGTH	“Content-Length:” from request header	CGI
CONTENT_TYPE	“Content-Type:” from request header	CGI
DOCUMENT_ROOT	generally empty, configurable path setting	Apache
FORM_*field-name*	query string “&” separated form elements	WASD
GATEWAY_BG	device name of raw client socket (specialized use)	WASD
GATEWAY_EOF	End of request sentinal (specialized use)	WASD
GATEWAY_EOT	End of callout sentinal (specialized use)	WASD
GATEWAY_ESC	Callout escape sentinal (specialized use)	WASD
GATEWAY_INTERFACE	“CGI/1.1”	CGI
GATEWAY_MRS	maximum record size of script SYS$OUTPUT	WASD
HTTP2_PING	HTTP/2 server-client RTT ping in (real number) milliseconds	WASD
HTTP_ACCEPT	any list of browser-accepted content types	CGI
HTTP_ACCEPT_CHARSET	any list of browser-accepted character sets	CGI
HTTP_ACCEPT_LANGUAGE	any list of browser-accepted languages	CGI
HTTP_AUTHORIZATION	any from request header (specialized use)	CGI
HTTP_CACHE_CONTROL	cache control directive	CGI
HTTP_COOKIE	any cookie sent by the client	CGI
HTTP_CONNECTION	connection persistence request field	CGI
HTTP_FORWARDED	any proxy/gateway hosts that forwarded the request	CGI
HTTP_HOST	host and port request was sent to	CGI
HTTP_IF_MATCH	if-match request field	CGI
HTTP_IF_NONE_MATCH	if-none-match request field	CGI

Name	Description	Origin
HTTP_IF_MODIFIED_SINCE	any last modified GMT time string	CGI
HTTP_IF_UNMODIFIED_ SINCE	request field	CGI
HTTP_IF_RANGE	if-range request field	CGI
HTTP_KEEP_ALIVE	connection persistence request field	CGI
HTTP_PRAGMA	any pragma directive of request header	CGI
HTTP_REFERER	any source document URL for this request	CGI
HTTP_USER_AGENT	client/browser identification string	CGI
HTTP_X_FORWARDED_FOR	proxied client host name or address	Squid
HTTP_*field-name*	any other request header field	WASD
KEY_*n*	query string "+" separated elements	WASD
KEY_COUNT	number of "+" separated elements	WASD
PATH_INFO	virtual path of data requested in URL	CGI
PATH_TRANSLATED	VMS file path of data requested in URL	CGI
QUERY_STRING	un-URL-decoded string following "?" in URL	CGI
REMOTE_ADDR	IP host address of HTTP client	CGI
REMOTE_HOST	IP host name of HTTP client	CGI
REMOTE_USER	authenticated remote user name (or empty)	CGI
REQUEST_CHARSET	any server-determined request character set	WASD
REQUEST_METHOD	"GET", "PUT", etc.	CGI
REQUEST_PROTOCOL	"HTTP/2", "HTTP/1.1" or "HTTP/1.0"	WASD
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_DEFAULT	mapped default directory for script	WASD
SCRIPT_FILENAME	script file name (e.g. CGI-BIN:[000000]QUERY.COM)	Apache
SCRIPT_NAME	script being executed (e.g. "/query")	CGI
SERVER_ADDR	IP host name of server system	WASD
SERVER_ADMIN	email address for server administration	Apache

Name	Description	Origin
SERVER_CHARSET	server default character set	WASD
SERVER_GMT	offset from GMT (e.g. “+09:30”)	WASD
SERVER_NAME	IP host name of server	CGI
SERVER_PROTOCOL	HTTP protocol version (generally “HTTP/1.1”)	CGI
SERVER_PORT	IP port request was received on	CGI
SERVER_SIGNATURE	server ID, host name and port	Apache
SERVER_SOFTWARE	software ID of HTTP server	CGI
SERVER_TRUNCATE	CGI variable value the server was forced to truncate	WASD
UPSTREAM_ADDR	address of transparent proxy when client mapped	WASD
UNIQUE_ID	unique 19 character string	Apache
WATCH_SCRIPT	only present when script being WATCHed	WASD

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.

Query String Variables

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

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

Variables named WWW_KEY_*number* will be generated if the query string contains one or more plus (“+”) and no equate symbols (“=”).

Variables named WWW_FORM_*form-element-name* will be generated if the query string contains one or more equate symbols. Generally such a query string is used to encode form-URL-encoded (MIME type *x-www-form-urlencoded*) requests. By default the server will report an incorrect encoding with a 400 error response. However some scripts use malformed encodings and so this behaviour may be suppressed using the *set script=query=relaxed* mapping rule.

```
set /cgi-bin/script-name* script=query=relaxed
```

To suppress this decoding completely (and save a few CPU cycles) use the following rule.

```
set /cgi-bin/script-name* script=query=none
```

UNIQUE_ID Note

The UNIQUE_ID variable is a mostly Apache-compliant implementation (the "_" has been substituted for the "@" to allow its 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.

VMS Apache (CSWS) Compliance

WASD v7.0 had its CGI environment tailored slightly to ease portability between VMS Apache (Compaq Secure Web Server) 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 (Section 1.12) has also been made CSWS V1.0-1 and later (Apache 1.3.12 and higher) compliant.

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

2.2 Script Output

This information applies to all non-DECnet based scripting, CGI, CGIplus, RTE, ISAPI. WASD uses mailboxes for script 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 (also see IPC Tickler).

- **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 its 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 (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 (either full or internal)

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

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

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

And using the status field.

```
Status: 404 Not Found
Content-Type: text/plain

Huh?
```

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 Features and Facilities).

Output and HTTP/1.1

With HTTP/1.1 it is generally better to use CGI than NPH responses. A CGI response allows the server to parse the response header and from that make decisions about connection persistence and content-encoding. These can contribute significantly to reducing response latency and content transfer efficiency. It allows any policy established by server configuration for such characteristics to be employed.

WASD Specifics

This section describes how WASD deals with some particular output issues (also see IPC Tickler).

- **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 (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 (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.

C-RTL Features

Non-C Runtime Libraries (C-RTL) do not contend with records delimitted by embedded characters (the newlines and nulls, etc., of the C environment). They use VMS' and RMS' default record-oriented I/O. The C-RTL needs to accomodate the C environment's *bag-o'-bytes* paradigm for file content against RMS' record structures, and it's embedded terminator, stream-oriented I/O with unterminated, record-oriented I/O. Often this results in a number of issues particularly with code ported from *x environments.

The C-RTL behaviour can be modified in all sorts of ways, including some file and other I/O characteristics. The features available to such modification are incrementally increasing with each release of the C-RTL and/or C compiler. It is well advised to consult the latest release (or as appropriate for the local environment) of the Run-Time Library Reference Manual for OpenVMS Systems for the current set.

Behaviours are modified by setting various flags, either from within the program itself using thef using the decc$feature_set( ) and allied group of functions, or by defining an equivalent logical name, usually externally to and before executing the image. See C-RTL Reference Manual section Enabling C RTL Features Using Feature Logical Names. This is particularly useful if the source is unavailable or just as a simpler approach to modifying code.

An example of a useful feature and associated logical name is **DECC$STDIO_CTX_EOL** which when enabled "writing to stdout and stderr for stream access is deferred until a terminator is seen or the buffer is full" in contrast to the default behaviour of "each fwrite generates a separate write, which for mailbox and record files generates a separate record". For an application performing write( )s or fwrite( )s with a record-oriented <stdio> and generating inappropriate record boundaries the application could be wrapped as follows (a real-world example).

```
$ set noon
$ define/user/nolog sys$input http$input
$ define/user DECC$STDIO_CTX_EOL ENABLE
$ calcserver
$ exit(1)
```

IPC Tickler

The interactions between VMS' record-oriented I/O, various run-time libraries (in particular the C-RTL), the streaming character-oriented Web, and of course WASD, can be quite complex and result in unintended output or formatting. The CGI script Inter-Process Communication (IPC) tickler WASD_ROOT:[SRC.MISC]IPCTICKLER.C is designed to allow a script programmer to gain an appreciation of how these elements interact, how WASD attempts to accomodate them, what mechanisms a script can use to explicitly convey exact requirements to WASD ... and finally, how these affect output (in particular the carriage-control) delivered to the client. If installed use /cgi-bin/IPCtickler to obtain an HTML form allowing control of several parameters into the 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 its own termination. Caution, such scripts if problematic could easily accumulate and "clog up" a server or system.
- **X-buffer-records[=0 | 1] -** Buffer records written by the script until there is [BufferSizeDclOutput] bytes available then write it as a single block to the client. The optional zero returns the script to non-buffered records, a one enables record buffering.
- **X-content-encoding-gzip[=0 | 1] -** A script by specifying a zero can request that the server, by default encoding the response using GZIP compression, leaves the response unencoded. By specifying a one a script can request the server to encode the response using GZIP compression.
- **X-content-handler=*keyword* -** The output of the script is stripped of the CGI reponse header and the body is given to the specified content handler. Currently only the SSI engine is supported using "X-content-handler=SSI".
- **X-crlf-mode[=0 | 1] -** 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. The optional zero returns the script to record mode, a one enables CR-LF mode.
- **X-error-line=*integer* -** Source code module line number META in server generated error message (optional).
- **X-error-module=*string* -** Source code module name META in server generated error message (optional).
- **X-error-text="*string*" -** Text of error message (mandatory, to trigger server message generation). If without a VMS status value this is the entire error explanation. If with a VMS status it becomes the context of the status.
- **X-error-vms-status=*integer* -** VMS status value in decimal or VMS hexadecimal (e.g. %X0000002C, optional). Changes format of error message to detailed explanation of status.
- **X-error-vms-text="*string*" -** Context of a VMS status value (optional). Usually a VMS file specification of other text related to the status value (in commented content of error message).
- **X-http-status=*integer* -** Forces the HTTP response line status to this value regardless of the status in the CGI *Status:* response or the server status if being used as an error reporting script (see "WASD Web Services - Install and Config").
- **X-lifetime=*value* -** The number of minutes before the idle scripting process is deleted by the server. Zero sets it back to the default, "none" disables this functionality.

- **X-record-mode[=0 | 1] -** 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. The optional zero changes the script to stream mode, a one enables record mode.
- **X-record0-mode[=0 | 1] -** Some (C-RTL) carriage-control modes result in empty records being sent in lieu of carriage control (newlines). This (rather esoteric) mode only adds a newline to empty records.
- **X-stream-mode[=0 | 1] -** 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. The optional zero returns the script to record mode, a one enables stream mode.
- **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 as 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
```

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 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"));
```

2.2.3 Bulk Content Output

As described above, Section 2.2, the default script<->server IPC uses a mailbox. While versatile and sufficiently efficient for general use, when megabytes, tens of megabytes, and hundreds of megabytes need to be transferred, using a memory buffer shared between script and server can yield transfer **improvements of up to five times**.

YMMV

Of course, your mileage may vary with platform, O/S version and TCP/IP stack (i.e. as the relative bottlenecks shuffle about).

The script requests a memory-buffer using a CGI callout (Chapter 6). Buffer size is constrained by the usual VMS 32bit memory considerations, along with available process and system resources. The server creates and maps a non-permanent global section. If this is successful the script is advised of the global section name using the callout response. The script uses this to map the section name and can then populate the buffer. When the buffer is full or otherwise ready, the script issues a callout with the number of bytes to write, and then stalls. The complete memory buffer may be written at once or any subsection of that buffer. The write is accomplished asynchronously and may comprise multiple network $QIOs or TLS/SSL blocks. When complete, a callout response to the script is issued and the script can continue processing. Standard script mailbox I/O (SYS$OUTPUT, <stdout>) and memory-buffer I/O may be interleaved as required.

The callouts are as follows:

- **BUFFER-BEGIN:** *<integer>[k | M]*

  Create a temporary global section to act as a memory buffer shared between a script process and the server. The default is **M**egabytes.

- **BUFFER-END:**

  Dispose of the shared memory buffer created by callout BUFFER-BEGIN.

- **BUFFER-WRITE:** *<integer>*

  Instruct the server to write <integer> bytes from the shared memory buffer to the client.

See working examples in WASD_ROOT:[SRC.MISC]

online hypertext link

Actual data comparing standard mailbox IPC with memory-buffer generated using [SRC.MISC]MEMBUFDE on a HP rx2660 (1.40GHz/6.0MB) with 4 CPUs and 16383MB running VSI VMS V8.4-2L1 with Multinet UCX$IPC_SHR V55A-B147, OpenSSL 1.0.2k and WASD v11.2.0, with [BufferSizeDclOutput] 16384. In each case 250MB ("?250") is transfered via a either a 16.4kB mailbox (default) or 16.4kB memory buffer ("+b"). Significantly larger memory buffer may well improve throughput further.

```
$  wget "-O" nl: http://127.0.0.1/cgi-bin/membufdemo?250
--2017-10-14 03:19:05--  http://127.0.0.1/cgi-bin/membufdemo?250
Connecting to 127.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 262144000 (250M) [application/octet-stream]
Saving to: ’nl:’

nl:                 100%[=====================>] 250.00M  25.6MB/s   in 12s

2017-10-14 03:19:17 (20.5 MB/s) - ’nl:’ saved [262144000/262144000]

$  wget "-O" nl: http://127.0.0.1/cgi-bin/membufdemo?250+b
--2017-10-14 03:19:23--  http://127.0.0.1/cgi-bin/membufdemo?250+b
Connecting to 127.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 262144000 (250M) [application/octet-stream]
Saving to: ’nl:’

nl:                 100%[=====================>] 250.00M   105MB/s   in 2.4s

2017-10-14 03:19:26 (105 MB/s) - ’nl:’ saved [262144000/262144000]

$  wget "-O" nl: https://127.0.0.1/cgi-bin/membufdemo?250
--2017-10-14 03:19:50--  https://127.0.0.1/cgi-bin/membufdemo?250
Connecting to 127.0.0.1:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 262144000 (250M) [application/octet-stream]
Saving to: ’nl:’

nl:                 100%[=====================>] 250.00M  14.5MB/s   in 17s

2017-10-14 03:20:07 (14.5 MB/s) - ’nl:’ saved [262144000/262144000]

$  wget "-O" nl: https://127.0.0.1/cgi-bin/membufdemo?250+b
--2017-10-14 03:20:12--  https://127.0.0.1/cgi-bin/membufdemo?250+b
HTTP request sent, awaiting response... 200 OK
Length: 262144000 (250M) [application/octet-stream]
Saving to: ’nl:’

nl:                 100%[=====================>] 250.00M  16.6MB/s   in 15s

2017-10-14 03:20:27 (16.6 MB/s) - ’nl:’ saved [262144000/262144000]
```

It is obvious that memory-buffer provides significantly greater throughput than mailbox (from the http:// test) and that with TLS/SSL network transport the encryption becomes a significant overhead and choke-point. Nevertheless, there is still an approximate 15% dividend, plus the more efficient interface the script->memory-buffer->server provides. The VMS TLS/SSL implementation may improve with time, especially if TLS/SSL hardware engines become available with the port to x86_64.

POSTSCRIPT

The comparison also illustrates that the WASD environment can deliver significant bandwidth through its script->server->network pathways. On the demonstration class of system; ~200Mbps unencrypted and ~120Mbps encrypted using the standard mailbox IPC; with ~850Mbps unencrypted and ~130Mbps encrypted using the memory-buffer IPC.

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 = "$WASD_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.

```
if ((HttpInput = fopen ("HTTP$INPUT", "r", "ctx=bin")) == NULL)
   exit (vaxc$errno);
```

When scripting using the C programming language there is a tendency for the C-RTL to check for and/or add newline (0x10, <LF>) carriage-control on receipt of record (single write). While this can be useful in converting from VMS to C conventions it can also be counter-productive if the stream being received is already using C carriage-control. To prevent the C-RTL reinterpreting data passed to it it often, perhaps invariably, necessary to reopen the input stream as binary using a construct similar to following.

This, and its <stdin> equivalent (below), are used consistently in WASD scripts.

```
if ((stdin = freopen ("HTTP$INPUT", "r", stdin, "ctx=bin")) == NULL)
   exit (vaxc$errno);
```

```
if ((stdin = freopen ("SYS$INPUT", "r", stdin, "ctx=bin")) == NULL)
   exit (vaxc$errno);
```

The input stream should be read before generating any output. If an error occurs during the body processing it should be reported via a CGI response header indicating an error (i.e. non-200). With HTTP/1.1 request processing there is also a requirement (that CGILIB fulfills) to return a “100 Continue” interim response after receiving the client request header and before the client sends the request body. Output of anything before this “100 Continue” is delivered will cause it to be interleaved with the script response body.

2.4 CGI Function Library

A source code collection of C language functions useful for processing the more vexing aspects of CGI/CGIplus programming (Section 1.12).

2.5 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 (Section 1.11).

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 script process and then executing the image of a CGI script. It does this by allowing the script process and any associated image/application to continue executing between uses, eliminating 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 WASD_ROOT:[SRC.CGIPLUS]

online hypertext link

Instead of having the CGI variables available from the environment (generally accessed via the C Language *getenv()* standard library call, or via DCL symbols) a CGIplus script must read the CGI variables from an I/O stream named CGIPLUSIN. The variables can be supplied in one of two modes.

- **Record Mode -** Each CGI variable is supplied as an individual record (line). This is the default, and simplest method. Each contains a CGI variable name (in uppercase), 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.
- **Struct Mode -** All variables are supplied in a binary structure which must be parsed by the receiving script. This is the most efficient method for providing the CGIplus script with its CGI environment. Performance improvements in the order of 50-100% have been measured. This size of this structure is limited by the size of the CGIPLUSIN mailbox, which is in turn set as described above.

Record-Mode CGIplus

This default and simple record-oriented mode allows any environment that can read records from an input source to process CGIplus variables. This of course includes DCL (examples referenced below).

- 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. Record-mode can always be recognised by the single exclamation symbol comprising this record.
- The CGIplus variable stream **must always be completely read**, record-by-record up until the the first empty record (blank line, see below) before beginning any request processing.
- 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.

Struct-Mode CGIplus

This mode offers significantly lower system overheads and improves latency and performance at the cost of the additional complexity of recovering the variables from a binary structure. Code examples and CGILIB functions make this relatively trivial at the application level.

- The read will block between subsequent requests and so may be used to coordinate the application.
- The first record read in any request contains the size of the following binary record. It is also provided so that a script may be synchronized outside of the general CGIplus variable read loop. Struct-mode can always be recognised by the two, successive, leading exclamation marks preceding the variable structure record size integer.

- The CGIplus variables are read as a single, binary I/O.
- The contents of the binary structure must be parsed to obtain the individual CGI variables. Request processing may then commence.

Requirements when using CGIplus

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 script process. 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 WASD_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 process' 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 (Section 1.12).

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 its 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);
```

Record-Mode Code

The following C code fragment shows a basic CGIplus record-mode request loop, reading lines from CGIPLUSIN, and some basic processing to select required CGI variables for request processing. Generally this level of coding is not required as it is recommended to employ the functionality of something like the CGILIB functiona library.

```
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)
   }
}
```

Struct-Mode Code

This mode requires significantly more code than record-mode. A self-contained C language function, allowing CGI variable processing in standard CGI, CGIplus record-mode and CGIplus struct-mode, is available for inclusion in user applications. It automatically detects the environment and changes behaviour to suit. This or CGILIB is strongly recommended.

See source in WASD_ROOT:[SRC.CGIPLUS]CGIPLUS_CGIVAR.C

online hypertext link

CGIplus 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 ...)

   /* CGI response header */
   fprintf (stdout, "Content-Type: text/html\n\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 WASD_ROOT:[SRC.CGIPLUS]

online 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 processes may be purged or deleted using ("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 multiple processes 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 processes, monitoring those currently processing requests, creating new instances if and when necessary, and deleting the least-used, idle CGIplus processes 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 script process can be given a finite life-time set by configuration parameter (see "Features and Facilities, 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 process 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 run-down by the server at any time the script process soft-limit is reached, so resources should be largely quiescent when not actually processing (Section 1.2.6). Of course, in extreme situations, a CGIplus process 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 "Features and Facilities, 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 "Features and Facilities, 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*
```

Hint!

When changing CGIplus script mapping it is advised to purge execution of existing scripts then reloading the mapping rules. Some conflict is possible when using new rules while existing CGIplus scripts are executing.

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

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 its 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, that is for scripting engines, 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 its 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)/wasd_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
WASD_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 (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!

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

online hypertext link provides the basics of the environment.

A source code collection of C language functions useful for processing the more vexing aspects of CGI/CGIplus/RTE programming (Section 1.12). The example RTE implementation uses this library.

4.2 Server Configuration

The following configuration information uses the supplied Perl RTE as the example. Note that RTE scripting engines must always be mapped using the EXEC+ rules. The SCRIPT+ rule does not apply.

The following rule in WASD_CONFIG_MAP maps the /pl-bin/ location to where the site wishes to locate its CGI Perl scripts (not necessarily the same as in the example).

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

With this rule Perl scripts may be accessed using

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

This WASD_CONFIG_GLOBAL rule ensures Perl scripts could be activated via the Perl RTE even if the WASD_CONFIG_MAP rule did not exist (Section 1.7).

```
[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 accessible due to incorrent file protections the DCL of the scripting process 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 WASD_CONFIG_MAP path SETting “script=nofind” as in the following example.

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

CGI environment variables SCRIPT_FILENAME and PATH_TRANSLATED can be provided to any RTE script in Unix file-system syntax should that script require or prefer it using this format. See Section 1.8.

DCL procedure

If the RTE executable requires *wrapping* in a DCL procedure (perhaps to provide some command-line specific parameter or define a C-RTL logical name) this can be specified in place of an executable. Merely prefix the specification with a “@”. The default is to run an executable (this can explicitly be specified using a leading “$”) while the leading “@” provides a DCL procedure activation.

Chapter 5

WebSocket

WebSocket is a capability introduced with HTML5, providing an asynchronous, bidirectional, full-duplex connection over which messages can be sent between agents, commonly a browser client and a server application. Compatible browsers provide a JavaScript interface that allows connections to be set up, maintained, messages exchanged, and connections closed down, using a callable and event-based interface.

WASD provides a WebSocket compatible scripting environment, one that is activated in the same fashion as an equivalent CGI/CGIplus/RTE and has an identical CGI environment (variables, streams, etc.) but which uses a unique HTTP response and communicates with its client using the WebSocket protocol.

Client supplied data is available to the script via the WEBSOCKET_INPUT mailbox and data from the script supplied via the WEBSOCKET_OUTPUT mailbox (indicated via CGI variables). Communication using a WebSocket requires the use of a framing protocol while WEBSOCKET_INPUT and WEBSOCKET_OUTPUT are opaque octet-streams providing communication to and from the WebSocket application. CGI variables WEBSOCKET_INPUT_MRS and WEBSOCKET_OUTPUT_MRS indicate the respective mailbox capacity.

The WASD server largely acts as a conduit for the WebSocket octet-stream. It provides the upgrade from HTTP to WebSocket protocol handshake and then connects the bidirectional data stream to the WebSocket application activated in WASD's scripting environment which then is required to perform all of the protocol requirements, etc. The baseline WASD implementation is via the **wsLIB** library (see below). The complexity and potential extensibility of the WebSocket protocol means this decoupling of server infrastructure and protocol implementation offers a number of advantages, including more straight-forward updates and bug fixing (just the library), and alternate, concurrent implementations.

Long-lived WebSocket scripts by default have timeouts and other limits set to infinite. If control is required it must be exercised using the appropriate mapping SETings or DCL callouts.

A WASD *RawSocket* is an analogue to the WebSocket, providing a bidirectional, asynchronous, opaque data stream input and output on a per-service basis. See Section 5.8.

5.1 Multi-Client WebSocket Applications

A single WASD WebSocket server application (script) can support multiple clients by using some form of multi-threading such as AST-based I/O, POSIX Threads, multi-thread interpreter environment, etc. The WASD wsLIB library (Section 5.3) supports native AST concurrency.

A WebSocket connection to a script is maintained by the WEBSOCKET_INPUT and WEBSOCKET_OUTPUT channels remaining connected to the script. If the script closes them (or the image or process exits, etc.) the WebSocket connection is closed. WebSocket requests are maintained as long as the script maintains them, for a CGIplus script, until it exits. If a CGIplus script requires to disconnect from a WebSocket client without exiting it must do so explicitly (by using the wsLIB *close* function (and associated WebSocket protocol close handshake), closing C streams, deassigning channels, etc.)

Of course this is the underlying mechanism allowing a single CGIplus script to maintain connections with multiple WebSocket clients. Provided the script remains connected to the WebSocket IPC mailboxes and processes that I/O asynchronously a single script can concurrently handle multiple clients. The script just processes each request it is given, adding the new client to the existing group (and removing them as the IPC indicates they disconnect).

Obviously the script must remain resident via CGIplus or RTE.

BYTLM

WebSocket scripting environments have the potential to consume significantly more BYTLM than those for HTTP scripting. The potentially large number of mailboxes associated with each scripting process (two per WebSocket connection) means that server and scripting account(s) BYTLM and associated quotas will need to be increased appropriately.

The server will continue to provide requests to the script for as long as it appears idle (i.e. the CGIplus sentinal EOF is returned even though concurrent processing may continue). Obviously a single scripting process cannot accept an unlimited number of concurrent WebSockets. When a script decides it can process no more it should not return the sentinal EOF from the most recent request until it is in a position to process more, when it then provides the EOF and the server again will supply another request.

The original request is access logged at request run-down (when the WebSocket is finally closed either because the client disconnected or the script closed its connection to the WEBSOCKET_.. mailboxes). The access log status is 101 (Switching Protocols) and the bytes rx and tx reflect the total for the duration.

5.2 WebSocket Application

WebSocket server applications are essentially CGIplus scripts and so have similar programming considerations (see Chapter 3).

A WebSocket application however is typically long-lived and involves significant interaction between the participants. Either party can initiate independent communication with the other according to the required business logic.

A WASD WebSocket application relies on asynchronous I/O and other events to provide the communication granularity required for application interaction. The following pseudo-code shows the structure of one such hypothetical application. It accepts multiple, concurrent requests in it's main loop, creates the required WebSocket protocol supporting data structure, and then services application requirements in two event loops.

The first reads from the remote client and processes according to the business logic of client-initiated processing, asynchronously and/or synchronously writing data to the client. The second loop pushes data asynchronously to the client based on the application business logic providing those events. The close event occurs when the client or application close the WebSocket, or are otherwise disconnected, and finalises the request.

```
begin
{
   initialise
   loop
   {
      wait for next client request
      open the WebSocket
      begin asynchronous read from client
      signal ready for another request
   }
}
read event from client
{
   business logic
   asynchronous write to client
   next asynchronous read from client
}
push event to client
{
   business logic
   asynchronous write to client
}
close event
{
   business logic
   close the WebSocket
}
```

A second model for request synchronisation allows the initialisation to specify a routine to be called when another request is available, making all processing event-driven. In all other respects the processing is the same as above.

```
begin
{
   initialise
   set routine for request acceptance
   hibernate (or otherwise not exit)
}
request acceptance
{
   open the WebSocket
   business logic
   begin asynchronous read from client
   signal ready for another request
}
read event from client
{
   business logic
   asynchronous write to client
   next asynchronous read from client
}
push event to client
{
   business logic
   asynchronous write to client
}
close event
{
   business logic
   close the WebSocket
}
```

These basic structures are seen in all the WebSocket example applications.

5.3 WebSocket Library

wsLIB is a C code module that provides the basic infrastructure for building WebSocket applications to run as WASD scripting under both the models decribed above.

It abstracts much of the required functionality into a few callable functions using optional string descriptors so as to minimise dependency on the C language and on knowing the internals of the library data structure. The list of functions and associated parameters would unnecessarily clutter this document and so WebSocket application designers and programmers are referred to the descriptive prologue in the library code module itself (see below). While wsLIB usage is *relatively* straight-forward, the detail of any multi-threaded, asynchronous application can be daunting and so the example WebSocket applications (scripts) should be used as a wsLIB reference and tutorial.

wsLIB also contains routines for synchronising request acceptance and accessing CGI variables associated with that request. These variables are available for the period from request acceptance to the issuing of the CGIplus sentinal EOF indicating to the server the script is ready to accept another request. Any CGI variable values required during ongoing processing must be copied to request-specific storage. Again, the example WebSocket applications (scripts) should be used as a CGI variable processing reference and tutorial.

The library contains WATCH points. Network [x]Data and [x]Script provide a useful combination of traffic data. The library function WsLibWatchScript( ) allows WebSocket applications (scripts) to provide additional WATCHable information via the [x]Script item.

WASD_ROOT:[SRC.WEBSOCKET]WSLIB.C.

online hypertext link

5.4 WebSocket Application Examples

The WASD WebSocket implementation provides a number of scripting examples illustrating WebSocket programming basics and the use of the WASD wsLIB library. All of these illustrate multi-client support using asynchonous I/O. Each has a server component (the C code) and a client component (the HTML file containing the JavaScript code).

WASD_ROOT:[SRC.WEBSOCKET]

online hypertext link

The following examples concentrate on the server C code as this is WASD-specific. Any Web-Socket reference can adequitely cover the essentials of the client JavaScript implementation.

5.4.1 Chat

This almost has to be **the** classic example of asynchronous, bidirectional communications without HTTP kludges. Each connected client can enter a message and it is distributed to all connected clients.

WASD_ROOT:[SRC.WEBSOCKET]WS_CHAT.C.

online hypertext link

5.4.2 Echo

Each connected client can enter a message which is then returned to them.

WASD_ROOT:[SRC.WEBSOCKET]WS_ECHO.C.

online hypertext link

5.4.3 Mouse

The HTML/JavaScript/WebSocket client end connects to the script. Each mouse movement is then reported to the script. These data are distributed to all connected clients. This provides an asynchronous update facility from all clients to all clients.

The script is implemented using VMS I/O-driven ASTs. The code is also interesting because it implements all required functionality explicitly; no WebSocket library functions are employed.

WASD_ROOT:[SRC.WEBSOCKET]WS_MOUSE.C.

online hypertext link

5.5 WebSocket Configuration

WebSocket server applications are essentially CGIplus scripts and so are mapped and activated in the same fashion as any other CGIplus script (Section 3.3).

5.5.1 WebSocket Throttle

Throttle mapping rules may be applied to WebSocket requests. There is however, a **fundamental difference** between request throttling and WebSocket throttling though. HTTP request throttling applies control to the entire life of the response. WebSocket throttling applies only to establishing connection to the underlying server application. Once the script responds to accept the connection or reject it throttling is concluded.

Long-lived WebSocket connections are considered less suitable to full life-cycle throttling and should use internal mechanisms to control resource utilisation (i.e. using the delayed sentinal EOF mechanism described in Section 5.1). Essentially it is used to limit the impact concurrent requests have on the number of supporting script processes allowed to be instantiated to support the application.

For example, the rule

```
set /cgi-bin/ws_application throttle=1
```

will only allow one new request at a time attempt to connect to and/or create a WebSocket application script. This will effectively limit the number of supporting processes to one however many clients wish to connect.

To support concurrent requests distributed across multiple application scripts specify the throttle value as the number of separate scripts

```
set /cgi-bin/ws_application throttle=5
```

and if each script is to support a maximum number of individual connections then have it delay the EOF sentinal (described above) to block the server selecting it for the next request. Requests will be allocated until all processes have blocked after which they will be queued.

To return a "too busy" 503 to clients (almost) immediately upon all processes become full and blocking (maximum application concurrency has been reached) then set the "t/o-busy" value to 1 second.

```
set /cgi-bin/ws_application throttle=5,,,,,1
```

5.5.2 WebSocket Command-Line

Unconditionally disconnects all WebSocket applications.

```
$ HTTPD /DO=WEBSOCKET=DISCONNECT
```

For VMS V8.2 and later, more selective disconnects are possible. Disconnects WebSocket applications with connect number, with matching script names, and with matching scripting account usernames, respectively.

```
$ HTTPD /DO=WEBSOCKET=DISCONNECT=number
$ HTTPD /DO=WEBSOCKET=DISCONNECT=SCRIPT=pattern
$ HTTPD /DO=WEBSOCKET=DISCONNECT=USER=pattern
```

5.5.3 WebSocket Version

CGI variable WEBSOCKET_VERSION provides the WebSocket protocol version number negotiated by the server at connection establishment.

At the time of writing the WebSocket protocol has just gone to IETF Draft RFC and has during development been very volatile and may continue to be so as it evolves. WASD supports the current base protocol number and any higher. At some time in the future it may be necessary to constrain that to a supported version number or set of numbers. Defining the logical name WASD_WEBSOCKET_VERSION to be one or more comma-separated numbers will limit the supported protocol versions. For example

```
  $ DEFINE /TABLE=WASD_TABLE WASD_WEBSOCKET_VERSION "10, 9, 8"
```

limits requests to protocol version 10 (current), 9 (earlier) and 8 (earliest). Logical name is only tested once for each server startup (the first WebSocket request received). This logical name only controls server handshake support and behaviour. The underlying WebSocket library used by the application (e.g. wsLIB.c) supports version idiosyncracies for other aspects.

This string is also used as the list of versions reported in a 426 (upgrade required) response when a client makes a request using an unsupported version.

5.6 WebSocket Throughput

The raw WebSocket throughput of a platform (hardware plus VMS plus TCP/IP stack plus WASD and optionally network infrastructure) can be measured using the WSB utility. Measures of raw message and byte throughput for a series of messages of various sizes can provide useful information on the underlying maximum messaging characteristics of that platform.

The following example shows usage on an Alpha XP1000:

```
$ WSB == "$WASD_EXE:WSB"
$ WSB /DO=ECHO /THROUGHPUT /REPEAT=1000 /SIZE=0
%WSB-I-STATS, 1 total connections
Duration: 0.303 seconds
Tx: 1000 msgs at 3303/S, 0 bytes at 0 B/S
Rx: 1000 msgs at 3303/S, 0 bytes at 0 B/S
Total: 2000 msgs at 6607/S, 0 bytes at 0 B/S
$ WSB /DO=ECHO /THROUGHPUT /REPEAT=1000 /SIZE=16
%WSB-I-STATS, 1 total connections
Duration: 0.349 seconds
Tx: 1000 msgs at 2869/S, 16.0 kbytes at 45.9 kB/S
Rx: 1000 msgs at 2869/S, 16.0 kbytes at 45.9 kB/S
Total: 2000 msgs at 5737/S, 32.0 kbytes at 91.8 kB/S
$ WSB /DO=ECHO /THROUGHPUT /REPEAT=1000 /SIZE=64
%WSB-I-STATS, 1 total connections
Duration: 0.359 seconds
Tx: 1000 msgs at 2783/S, 64.0 kbytes at 178.1 kB/S
Rx: 1000 msgs at 2783/S, 64.0 kbytes at 178.1 kB/S
Total: 2000 msgs at 5566/S, 128.0 kbytes at 356.2 kB/S
$ WSB /DO=ECHO /THROUGHPUT /REPEAT=1000 /SIZE=256
%WSB-I-STATS, 1 total connections
Duration: 0.607 seconds
Tx: 1000 msgs at 1646/S, 256.0 kbytes at 421.5 kB/S
Rx: 1000 msgs at 1646/S, 256.0 kbytes at 421.5 kB/S
Total: 2000 msgs at 3293/S, 512.0 kbytes at 843.0 kB/S
$ WSB /DO=ECHO /THROUGHPUT /REPEAT=1000 /SIZE=1024
%WSB-I-STATS, 1 total connections
Duration: 0.659 seconds
Tx: 1000 msgs at 1517/S, 1.0 Mbytes at 1.6 MB/S
Rx: 1000 msgs at 1517/S, 1.0 Mbytes at 1.6 MB/S
Total: 2000 msgs at 3034/S, 2.0 Mbytes at 3.1 MB/S
$ WSB /DO=ECHO /THROUGHPUT /REPEAT=1000 /SIZE=65k
%WSB-I-STATS, 1 total connections
Duration: 10.279 seconds
Tx: 1000 msgs at 97/S, 65.0 Mbytes at 6.3 MB/S
Rx: 1000 msgs at 97/S, 65.0 Mbytes at 6.3 MB/S
Total: 2000 msgs at 195/S, 130.0 Mbytes at 12.6 MB/S
```

For more information see the description in the prologue of the program. (A zero-size message is legitimate with the WebSocket protocol.)

5.7 WebSocket References

- http://en.wikipedia.org/wiki/WebSockets
  Wikipedia overview of WebSockets
- http://www.websocket.org/index.html
  WebSocket.org - What is a Websocket?
- https://html.spec.whatwg.org/multipage/comms.html#network
  The WebSocket API (JavaScript)
- http://tools.ietf.org/html/rfc6455
  The WebSocket protocol
- https://trac.tools.ietf.org/wg/hybi/trac/wiki/FAQ
  The IETF WebSocket FAQ

- http://websocketstest.com/
  Real-Time Web[socket] Test

5.8 WASD “Raw”Socket

The WASD *RawSocket* is a variant of, and is heavily based on, the WASD WebSocket infrastructure. It allows a service (listening host and port) to accept a connection and *immediately* activate a configured WASD CGIplus script to service that connection. Full-duplex, asynchronous, bidirectional input/output is supported. While being referred to as a “socket”, of course it is not network socket communication (BSD or any other abstraction), but a read/write abstraction via mailbox I/O. Input is a stream of octets; output a stream of octets. The streams are opaque in this context and depend entirely on the abstraction (protocol) being implemented by the script. The "RawSocket" is just WASD nomenclature for WebSocket-style scripting without the WebSocket protocol.

Note

A RawSocket application is not intended to use a (standard web) browser as a client. Of course the application could (perhaps, also) behave as a web server if at activation it (also) expected to receive, parse and respond to an HTTP request, and therefore (also) be used by a web browser client.

The script when activated enters a CGIplus wait-service-wait loop using the standard CGIplus mechanism. When a request activates the script it issues a “100 Continue” CGI/HTTP header to signal the server that the request is accepted and now being processed. This header is NOT relayed to the client. The activation parameters (normally referred to as *request* parameters) are available via the usual CGI variables. The script can then read from and write to the RAWSOCKET_INPUT and RAWSOCKET_OUTPUT devices respectively. As with WebSocket applications, a single RawSocket application (script) can concurrently support multiple clients. The end-of-script is indicated by issuing an EOF to the mailbox. The script activation can be as long-lived as required and the server will not run a request down for any timeout reason.

5.8.1 RawSocket Application

RawSocket applications share the same lifecycle as described for WebSockets in Section 5.2.

5.8.2 RawSocket Library

rawLIB is a C code module that provides a similar API and functionality to the wsLIB library described in Section 5.3.

WASD_ROOT:[SRC.WEBSOCKET]RAWLIB.C.

online hypertext link

5.8.3 RawSocket Application Examples

The WASD RawSocket implementation provides a number of scripting examples illustrating programming basics and the use of the WASD rawLIB library. All of these illustrate multi-client support using asynchonouse I/O. All use *telnet* as a client interface although this is only for convenience. RawSocket is completely protocol agnostic.

WASD_ROOT:[SRC.WEBSOCKET]

online hypertext link

5.8.3.1 Chat

This almost has to be **the** classic example of asynchronous, bidirectional communications. Each connected client can enter a message and it is distributed to all connected clients.

WASD_ROOT:[SRC.WEBSOCKET]RAW_CHAT.C.

online hypertext link

5.8.3.2 Echo

Each connected client can enter a message which is then returned to them.

WASD_ROOT:[SRC.WEBSOCKET]RAW_ECHO.C.

online hypertext link

5.8.3.3 Terminal Server

This example implements a simple-minded telnet server. Definitely not intended for production.

WASD_ROOT:[SRC.WEBSOCKET]RAW_PTD.C.

online hypertext link

5.8.4 RawSocket Configuration

RawSocket server applications are essentially CGIplus scripts and so are mapped and activated in the same fashion as any other CGIplus script (Section 3.3). They are a little unique in that there is generally a one-to-one relationship between a script and a service. The service is flagged as implementing a *RawSocket* and that service is mapped directly to a script.

```
# WASD_CONFIG_SERVICE
[[http:*:1234]]
[ServiceRawSocket]  enabled

# WASD_CONFIG_MAP
[[*:1234]]
map * /cgiplus-bin/raw_script
```

It is possible to have conditional mapping based on the (rather limited) “request” parameters (WATCH is useful for understanding what request data is available). For example, the script activated may be varied on the client address.

```
# WASD_CONFIG_MAP
[[*:1234]]
if (remote-addr:131.185.10.0/24)
   map * /cgiplus-bin/raw_one
else
   map * /cgiplus-bin/raw_two
endif
```

Chapter 6
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 (“Features and Facilities, 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.

6.1 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 ("Features and Facilities, Security Profile") this can determine whether the specified file name is allowed access by the request's username.

- **BUFFER-BEGIN:** *<integer>[k | M]*

  Create a temporary global section to act as a memory buffer shared between a script process and the server. The default is **M**egabytes. See Section 2.2.3 for a description of the purpose and use of all the BUFFER-.. callouts.

- **BUFFER-END:**

  Dispose of the shared memory buffer created by callout BUFFER-BEGIN.

- **BUFFER-WRITE:** *<integer>*

  Instruct the server to write <integer> bytes from the shared memory buffer to the client.

- **CGIPLUS:** *string*

  This callout is used to indicate to the server that a CGIplus script can process the CGI variables in "struct" mode. By default each CGI variable is transfered to a CGIplus script one "record" at a time. In "struct" mode all variables are transfered in a single, binary I/O which must the be parsed by the the script. It is of course a much more efficient method for CGIplus (Struct-Mode CGIplus).

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

  When using the raw TCP/IP socket for output (Chapter 11) 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-CCL:** *integer*

  When using the raw TCP/IP socket for output (Chapter 11) this can be used to change the BG: device carriage-control. A value of 1 enables a <CR><LF> with each record (the default), while 0 disables it. This is analagous to the APACHE$SET_CCL utility.

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

  When using the raw TCP/IP socket for output (Chapter 11) 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 scripting process' lifetime which may be expressed as an integer number of minutes or in the format *hh:mm:ss*. 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 its 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.

- **NOTICED:** *string*

  Place the supplied string into the server process log. Used to report incidental processing or other errors.

- **OPCOM:** *string*

  Send the supplied string to OPCOM.

- **REDACT:**

  See Chapter 10.

- **REDACT-SIZE:**

  See Chapter 10.

- **SCRIPT-CONTROL:**

  Equivalent to the script issuing a "Script-Control:" response header field (although of course some script control directives will not apply after header generation).

- **TIMEOUT-BIT-BUCKET:** *integer*

  Specifies the period for which a script continues to execute if the client disconnects. Overrides the WASD_CONFIG_GLOBAL [DclBitBucketTimeout] confiuration directive.

- **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.

6.2 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 WASD_ROOT:[SRC.CGIPLUS]

online hypertext link

Chapter 7
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 its 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.

7.1 CGIsapi

WASD provides an environment for executing ISAPI based extensions. Unlike classic ISAPI the DLLs are not loaded into server space but into autonomous processes, in much the same way as CGIplus scripts are handled (Chapter 3). This still offers significantly improved performance through the persistance of the ISAPI extension between requests. Measurements show a potential five-fold, to in excess of ten-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 script process 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 script process may terminate.
2. Each process services only the one request at a time. This eliminates the threading issues.

WASD implements the ISAPI environment as an instance of its 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 *-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 "Features and Facilities, 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).

7.2 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 its build procedure in WASD_ROOT:[SRC.CGIPLUS].

The CGIsapi wrapper, WASD_ROOT:[SRC.CGIPLUS]CGISAPI.C, is relatively straightforward, 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 process' 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 process, 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 processes are idle they can be run-down 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 its 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 process 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 processes may be purged or deleted using ("Techncial Overview, Server Command Line Control"):

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

7.3 Server Configuration

Ensure the following are in the appropriate sections of WASD_CONFIG_GLOBAL.

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

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

Ensure this rule exists in the scripting section of WASD_CONFIG_MAP.

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

With this rule DLLs may be accessed using something like

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

Chapter 8
DECnet & OSU

"Imitation is the sincerest form of flattery" - proverb

Note

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

By default WASD executes scripts within detached processes, but can also provide scripting using DECnet for the process management. DECnet scripting is not provided to generally supplant the detached process-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

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. 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 process-based scripts, using the SCRIPT and EXEC rules ("Features and Facilities, 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 (Section 8.4).

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"/htbin/*
```

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

Scripting Account

By default the script process is created using the HTTPd scripting account (usually HTTP$NOBODY, although versions prior to 8.1 have used HTTP$SERVER). It is possible to specify alternate accounts for the scripts to be executed within.

The first examples are explicitly specifying an account in the script rule.

```
exec /frodo/* /FRODO"ACCOUNT"::"0=cgiwasd"/cgi-bin/*
script /frodo-whatever /FRODO"ACCOUNT"::/cgi-bin/whatever.com
```

It is also possible to have scripts that have been subject to SYSUAF authorization executed within the authenticated account. The dollar symbol in the following examples directs the server to substitute the authenticated username into the access string.

```
exec /frodo/* /FRODO"$"::"0=cgiwasd"/cgi-bin/*
script /frodo-whatever /FRODO"$"::/cgi-bin/whatever.com
```

The *set script=as=* rule used for PERSONA controlled process scripting can also be applied to DECnet scripts. This includes explicitly specified usernames as well as SYSUAF authenticated usernames. The server creates an appropriate access string when the script is activated.

```
set /frodo* script=as=$
exec /frodo/* /FRODO::"0=cgiwasd"/cgi-bin/*
script /frodo-whatever /FRODO::/cgi-bin/whatever.com
```

User scripts can also be activated using these rules, either explicitly specifying the "~" in an access string or using the *set script=as=* mapping rule.

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

See Section 8.4 for more detail.

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 (WASD_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/).

8.1 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.

8.1.1 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$NOBODY 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$SERVER HTTP$NOBODY /DEFAULT
```

It is necessary to ensure the account has permission to write into its 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!

8.1.2 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 WASD_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 WASD_ROOT:[EXAMPLE].

8.1.3 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$SERVERS_HTTP$NOBODY 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"
```

8.1.4 DECnet/OSU Startup

The example STARTUP.COM and STARTUP_DECNET.COM procedures found in the WASD_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.

8.2 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 process on the local system. Other than that the WASD DECnet CGI environment behaves identically to the standard (sub)process 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 (sub)process 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$LINK") == NULL)
{
   /* via process CGI */
   while (ReadCount = fread (Buffer, 1, sizeof(Buffer), stdin))
   {
      /* processing, processing ... */
   }
}
else
{
   /* via DECnet CGI */
   if ((stdin = freopen ("NET$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*
```

8.3 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 its 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 (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 WASD_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 WASD_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 processes 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 WASD_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 WASD_CONFIG_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 WASD_CONFIG_GLOBAL configuration file.

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

8.4 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 “Features and Facilities, 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 /~*/htbin/* /0"~"::"0=wwwexec"/www_user/*/www/htbin/*
```

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

```
/~daniel/cgi-bin/cgi_symbols
/~daniel/htbin/osu_testcgi
```

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$NOBODY <account>
```

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

```
UAF> ADD /PROXY 0::HTTP$NOBODY DANIEL
```

Chapter 9

Other Environments

WASD supports a number of scripting engines.

Java
Perl
PHP
Python
Tomcat (Java Server Pages)

Earlier releases of the WASD package included some of these in the basic package. Due to the growing number, greater complexity of the environments, and increasing version dependencies, these environments will be distributed independently of the main WASD package. Current releases may be found at the main WASD download site http://wasd.vsm.com.au/wasd/

Pages generated by scripting environments can optionally be cached by the server. For a certain class of script output this can offer reduced response latency and system impact. See Section 1.4.

9.1 Java

Java classes may be used to perform CGI/CGIplus scripting with WASD. This **is not** Java Server Pages, Tomcat, or anything of the like. The Java refered to here is a small, self-contained Java environment that may used with WASD "out-of-the-box". All you need is java installed on your VMS system. These 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).

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 WASD_ROOT:[SRC.JAVA] directory.

As the Java environment is constantly under development, both as a platform-independent environment and on the VMS platform in particular, it is possible that the latest VMS Java kit may not integrate well with the WASD Java environment. Of course every effort will be made to keep the WASD Java environment current.

9.1.1 CGIplus Only

Java CGI/CGIplus 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 (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.

9.1.2 Requirements

Ensure the Java class file type is mapped to the Java run-time in the WASD_CONFIG_GLOBAL configuration file.

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

The following content types are configured, also in WASD_CONFIG_GLOBAL.

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

Class files should be copied to the [CGI-BIN] directory (where all architecture neutral script files should be located).

9.1.3 Carriage Control

Getting carriage-control to make sense is often a challenge. System.out.print( ) only expresses carriage-control embedded in the string. System.out.println( ) the same but then issues an independent linefeed (the newline of the *ln*) which appears to WASD as an empty record. Choose your poison (and antidote). Using the "Script-Control: X-stream-mode", "Script-Control: X-record-mode" or "Script-Control: X-record0-mode" can assist WASD interpreting the output. See Chapter 2.

9.2 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.h
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.

9.2.1 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 accessible via the *exec* or *script* rules of the WASD_CONFIG_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 (Section 1.6).

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__
```

9.2.2 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( ))

Name	Location
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).

Note

Never substitute the contents of CGI variables directly into the code stream using interpreters that will allows this (e.g. DCL, Perl). You run a very real risk of having unintended content maliciously change the intended function of the code. Always pre-process the content of the variable first, ensuring there has been nothing inserted that could subvert the intended purpose. There are a number of security-related Perl scripting issues. It is suggested the reader consult one of the many Perl-CGI documents/books available.

9.2.3 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__
```

9.2.4 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. Both of these require the supplementary Perl package available from the WASD download page. Both are briefly described below.

9.2.4.1 CGIplus

CGIplus substantially eliminates the overhead associated with CGI processing by allowing the subprocess and any associated image/application to continue executing between uses (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 as part of the supplementary package.

9.2.4.2 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 (Chapter 4). There is a significant difference between RTE and CGIplus scripts. With CGIplus the script itself persists between uses, retaining all of its state. With an RTE the script does not persist or retain state, only the RTE itself.

The WASD RTE Perl interpreter 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!**

9.2.5 Requirements

These are the configuration requirements for using the basic CGI Perl.

- WASD_CONFIG_GLOBAL configuration file.

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

- The following content types are configured, also in WASD_CONFIG_GLOBAL.

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

Chapter 10

Request Redaction

Callout processing may redact (completely rewrite and restart) a request.

```
re-dact

  -verb (used with object)

  1.  to put into suitable literary form; revise; edit.
  2.  to draw up or frame (a statement, proclamation, etc.).

  [Origin: 13501400; ME < L redactus (ptp. of redigere to lead back), equiv.
  to red- red- + ctus, ptp. of agere to lead; see act]
```

REDACT: Callout

To do this a script must use the REDACT:<opaque> callout to send back to the server a completely new request header and body (if applicable) which the server then treats as if received from the client over the network. This allows a request to be partially or completely rewritten (as required) and restarted. The data supplied to this callout is treated as completely opaque and care must be taken to include all and no extra carriage-control, etc.

Request redaction may only be initiated (using the REDACT: callout) if the CGI response header has not been sent. Once request redaction has been initiated no CGI output subsequently can be generated. The server will generate an error if such a protocol error occurs.

REDACT-SIZE: Callout

The REDACT-SIZE:<integer> callout may be used prior to any REDACT: callout. By default the server allocates memory on demand to accomodate the redacted request. If the redacted request is large (more than [BufferSizeDclOutput]) and the total size of the redacted request known in advance there is some efficiency in requesting the server to preallocate this amount of space using the REDACT-SIZE: callout.

Code Example

An elementary (and somewhat contrived) example:

```
stdout = freopen ("SYS$OUTPUT:", "w", stdout, "ctx=bin", "ctx=xplct");
fputs (getenv("CGIPLUSESC"),stdout);
fflush (stdout);

fputs ("REDACT:HTTP/1.1 POST /an_example.php\r\n\
Host: example.com\r\n\
Content-Length: 26\r\n\
Content-Type: application/x-www-form-urlencoded\r\n\
\r\n",
       stdout);
fflush (stdout);

fwrite ("REDACT:one=two&three=four\n", 26, 1, stdout);
fflush (stdout);

fputs (getenv("CGIPLUSEOT"),stdout);
fflush (stdout);
```

Once the request has been redacted the script just finishes processing without other output and the server transparently restarts processing.

An actual usage example may be found in the WASD PAPI authentication agent (not a component of the standard WASD package).

Redact Rationale

This facility was originally incorporated to allow a PAPI

http://papi.rediris.es/
http://en.wikipedia.org/wiki/Point_of_Access_for_Providers_of_Information

authentication agent to store a request on-disk and then some time and several processing steps later restart the original request processing.

Chapter 11
Raw TCP/IP Socket

Deprecated and Discouraged

Unencrypted TCP/IP communication, especially for browser and other web connections, is increasingly considered poor practise, even for trivial purposes. Access to the raw socket will likely be disabled in the future. If this facility is in use for script->client data transfer reasons the consider using a shared-memory buffer as described in Section 2.2.3.

For detached and 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.

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 (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 (Chapter 6) 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 WASD_ROOT:[SRC.OTHER]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);
}
```

Carriage Control

By default the TCP/IP BG device driver supplies a <CR><LF> sequence as carriage control for each record. This supports record-oriented output such as DCL and various VMS utilities but is an issue when needing to output a binary object such as a large graphic. The CGI callout (Chapter 6) GATEWAY-CCL: directive allows the device carriage control to be set and reset programmatically. A value of 1 enables a <CR><LF> with each record, while 0 disables it. This is analagous to the APACHE$SET_CCL utility.

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.